Buildbot Documentation

Release 0.9.5

Brian Warner

Mar 20, 2017

Contents

This is the Buildbot documentation for Buildbot version `|version|`.

If you are evaluating Buildbot and would like to get started quickly, start with the *Tutorial*. Regular users of Buildbot should consult the *Manual*, and those wishing to modify Buildbot directly will want to be familiar with the *Developer's Documentation*.

CHAPTER 1

Buildbot Tutorial

Contents:

First Run

Goal

This tutorial will take you from zero to running your first buildbot master and worker as quickly as possible, without changing the default configuration.

This tutorial is all about instant gratification and the five minute experience: in five minutes we want to convince you that this project works, and that you should seriously consider spending time learning the system. In this tutorial no configuration or code changes are done.

This tutorial assumes that you are running Unix, but might be adaptable to Windows.

Thanks to virtualenv (https://pypi.python.org/pypi/virtualenv), installing buildbot in a standalone environment is very easy. For those more familiar with Docker (https://docker.com), there also exists a *docker version of these instructions*.

You should be able to cut and paste each shell block from this tutorial directly into a terminal.

Getting ready

There are many ways to get the code on your machine. We will use the easiest one: via `pip` in a virtualenv (https://pypi.python.org/pypi/virtualenv). It has the advantage of not polluting your operating system, as everything will be contained in the virtualenv.

To make this work, you will need the following installed:

- Python (https://www.python.org/) and the development packages for it
- virtualenv (https://pypi.python.org/pypi/virtualenv)

Preferably, use your distribution package manager to install these.

You will also need a working Internet connection, as virtualenv and pip will need to download other projects from the Internet.

Note: Buildbot does not require root access. Run the commands in this tutorial as a normal, unprivileged user.

Creating a master

The first necessary step is to create a virtualenv for our master. We will also use a separate directory to demonstrate the distinction between a master and worker:

```
cd
mkdir tmp
cd tmp
mkdir buildbot
cd buildbot
```

On Python 2:

```
virtualenv --no-site-packages sandbox
source sandbox/bin/activate
```

On Python 3:

```
python3 -m venv sandbox
source sandbox/bin/activate
```

Now that we are ready, we need to install buildbot:

```
pip install --upgrade pip
pip install 'buildbot[bundle]'
```

Now that buildbot is installed, it's time to create the master:

```
buildbot create-master master
```

Buildbot's activity is controlled by a configuration file. We will use the sample configuration file unchanged:

```
mv master/master.cfg.sample master/master.cfg
```

Finally, start the master:

```
buildbot start master
```

You will now see some log information from the master in this terminal. It should end with lines like these:

```
2014-11-01 15:52:55+0100 [-] BuildMaster is running
The buildmaster appears to have (re)started correctly.
```

From now on, feel free to visit the web status page running on the port 8010: http://localhost:8010/

Our master now needs (at least) a worker to execute its commands. For that, head on to the next section!

Creating a worker

The worker will be executing the commands sent by the master. In this tutorial, we are using the pyflakes project as an example. As a consequence of this, your worker will need access to the git (https://git-scm.com/) command in order to checkout some code. Be sure that it is installed, or the builds will fail.

Same as we did for our master, we will create a virtualenv for our worker next to the other one. It would however be completely ok to do this on another computer - as long as the *worker* computer is able to connect to the *master* one:

```
cd
mkdir tmp/bb-worker
cd tmp/bb-worker
```

On Python 2:

```
virtualenv --no-site-packages sandbox
source sandbox/bin/activate
```

On Python 3:

```
python3 -m venv sandbox
source sandbox/bin/activate
```

Install the `buildbot-worker` command:

```
pip install buildbot-worker
```

Now, create the worker:

```
buildbot-worker create-worker worker localhost example-worker pass
```

Note: If you decided to create this from another computer, you should replace `localhost` with the name of the computer where your master is running.

The username (`example-worker`), and password (`pass`) should be the same as those in `master/master.cfg`; verify this is the case by looking at the section for `c['workers']`:

```
cat ../bb-master/master/master.cfg
```

And finally, start the worker:

```
buildbot-worker start worker
```

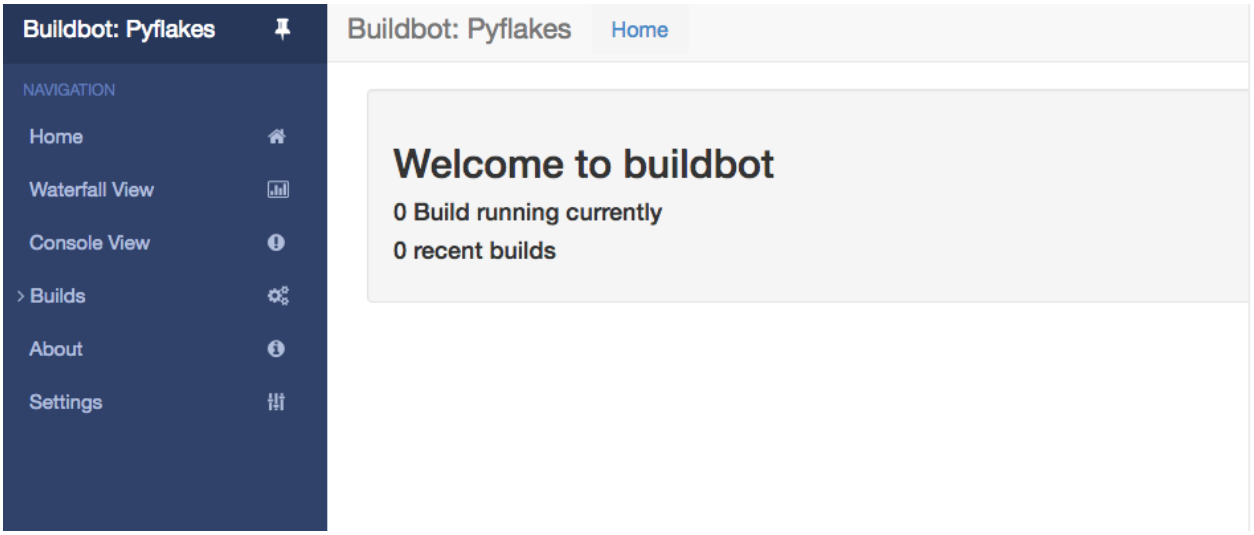
Check the worker's output. It should end with lines like these:

```
2014-11-01 15:56:51+0100 [-] Connecting to localhost:9989
2014-11-01 15:56:51+0100 [Broker,client] message from master: attached
The worker appears to have (re)started correctly.
```

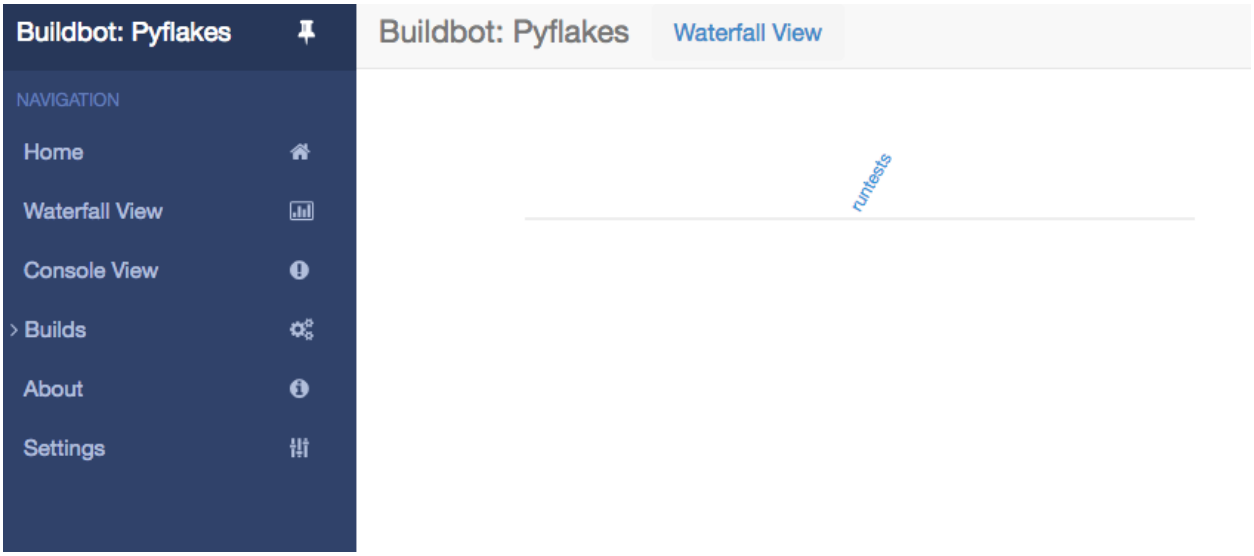
Meanwhile, from the other terminal, in the master log (`twisted.log` in the master directory), you should see lines like these:

```
2014-11-01 15:56:51+0100 [Broker,1,127.0.0.1] worker 'example-worker' attaching from␣
↪IPv4Address(TCP, '127.0.0.1', 54015)
2014-11-01 15:56:51+0100 [Broker,1,127.0.0.1] Got workerinfo from 'example-worker'
2014-11-01 15:56:51+0100 [-] bot attached
```

You should now be able to go to http://localhost:8010, where you will see a web page similar to:



Click on the Waterfall Display link (http://localhost:8010/waterfall) and you get this:



Your master is now quietly waiting for new commits to Pyflakes. This doesn't happen very often though. In the next section, we'll see how to manually start a build.

We just wanted to get you to dip your toes in the water. It's easy to take your first steps, but this is about as far as we can go without touching the configuration.

You've got a taste now, but you're probably curious for more. Let's step it up a little in the second tutorial by changing the configuration and doing an actual build. Continue on to *A Quick Tour*.

First Buildbot run with Docker

Note: Docker can be tricky to get working correctly if you haven't used it before. If you're having trouble, first determine whether it is a Buildbot issue or a Docker issue by running:

```
docker run ubuntu:12.04 apt-get update
```

If that fails, look for help with your Docker install. On the other hand, if that succeeds, then you may have better luck getting help from members of the Buildbot community.

Docker (https://www.docker.com) is a tool that makes building and deploying custom environments a breeze. It uses lightweight linux containers (LXC) and performs quickly, making it a great instrument for the testing community. The next section includes a Docker pre-flight check. If it takes more that 3 minutes to get the 'Success' message for you, try the Buildbot pip-based *first run* instead.

Current Docker dependencies

- Linux system, with at least kernel 3.8 and AUFS support. For example, Standard Ubuntu, Debian and Arch systems.
- Packages: lxc, iptables, ca-certificates, and bzip2 packages.
- Local clock on time or slightly in the future for proper SSL communication.
- This tutorial uses docker-compose to run a master, a worker, and a postgresql database server

Installation

- Use the Docker installation instructions (https://docs.docker.com/engine/installation/) for your operating system.
- Make sure you install docker-compose. As root or inside a virtualenv, run:

  ```
  pip install docker-compose
  ```

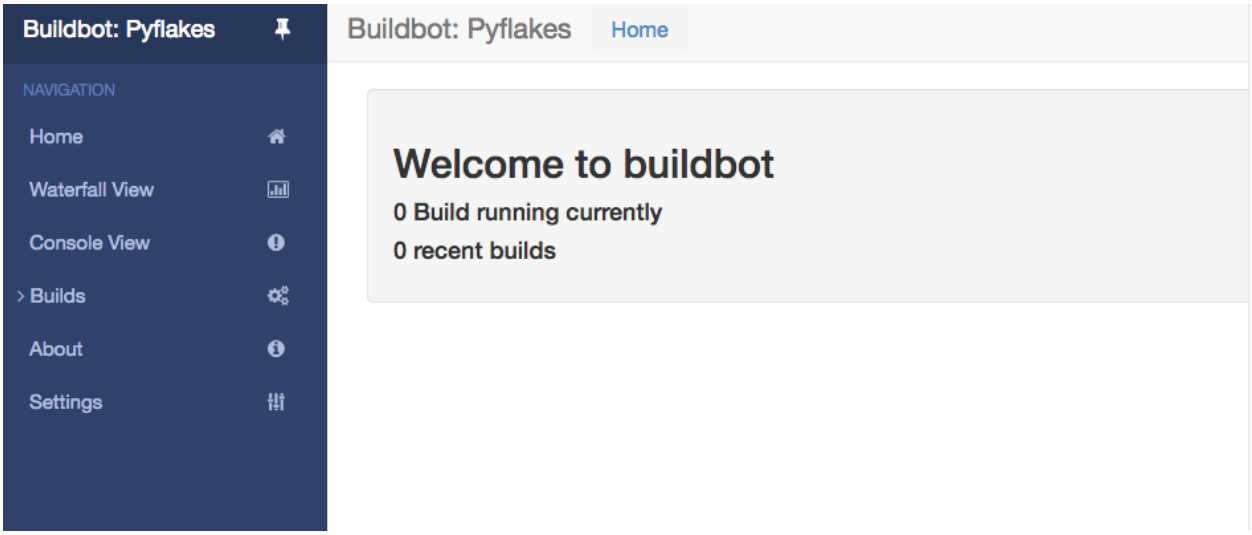
- Test docker is happy in your environment:

  ```
  sudo docker run -i busybox /bin/echo Success
  ```

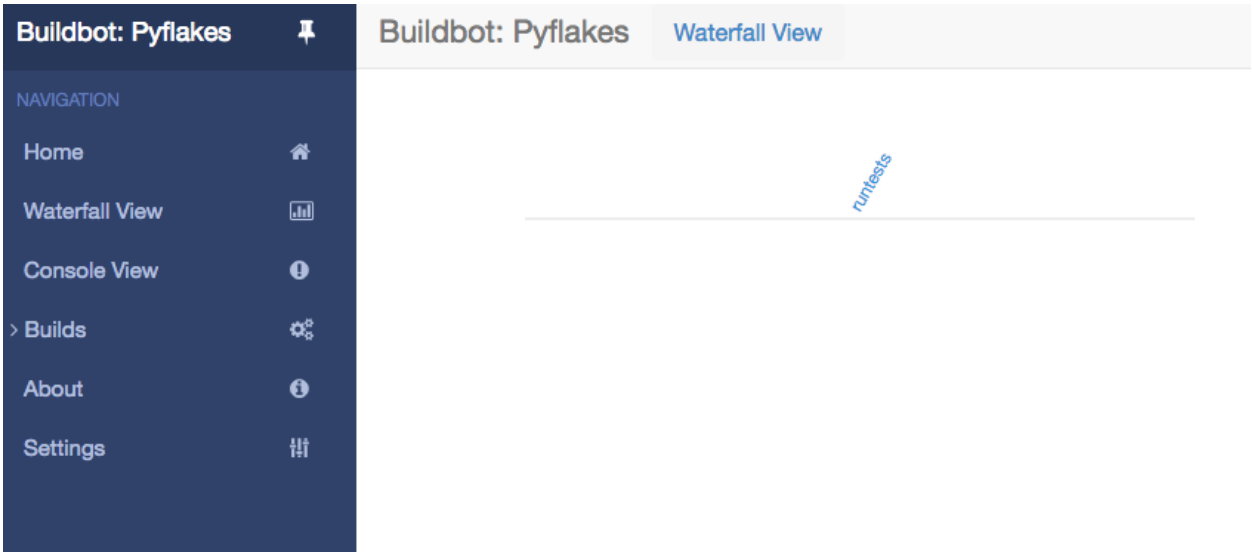
Building and running Buildbot

```
# clone the example repository
git clone --depth 1 https://github.com/buildbot/buildbot-docker-example-config

# Build the Buildbot container (it will take a few minutes to download packages)
cd buildbot-docker-example-config/simple
docker-compose up
```

You should now be able to go to http://localhost:8010 and see a web page similar to:



Click on the Waterfall Display link (http://localhost:8010/#/waterfall) and you get this:



Overview of the docker-compose configuration

This docker-compose configuration is made as a basis for what you would put in production

- Separated containers for each component
- A solid database backend with postgresql
- A buildbot master that exposes its configuration to the docker host
- A buildbot worker that can be cloned in order to add additional power
- Containers are linked together so that the only port exposed to external is the web server
- The default master container is based on Alpine linux for minimal footprint
- The default worker container is based on more widely known Ubuntu distribution, as this is the container you want to customize.
- Download the config from a tarball accessible via a web server

Playing with your Buildbot containers

If you've come this far, you have a Buildbot environment that you can freely experiment with.

In order to modify the configuration, you need to fork the project on github https://github.com/buildbot/buildbot-docker-example-config Then you can clone your own fork, and start the docker-compose again.

To modify your config, edit the master.cfg file, commit your changes, and push to your fork. You can use the command buildbot check-config in order to make sure the config is valid before the push. You will need to change `docker-compose.yml` the variable `BUILDBOT_CONFIG_URL` in order to point to your github fork.

The `BUILDBOT_CONFIG_URL` may point to a `.tar.gz` file accessible from HTTP. Several git servers like github can generate that tarball automatically from the master branch of a git repository If the `BUILDBOT_CONFIG_URL` does not end with `.tar.gz`, it is considered to be the URL to a `master.cfg` file accessible from HTTP.

Customize your Worker container

It is advised to customize you worker container in order to suit your project's build dependencies and need. An example DockerFile is available in the contrib directory of buildbot:

https://github.com/buildbot/buildbot/blob/master/master/contrib/docker/pythonnode_worker/Dockerfile

Multi-master

A multi-master environment can be setup using the `multimaster/docker-compose.yml` file in the example repository

> # Build the Buildbot container (it will take a few minutes to download packages) cd buildbot-docker-example-config/simple docker-compose up -d docker-compose scale buildbot=4

Going forward

You've got a taste now, but you're probably curious for more. Let's step it up a little in the second tutorial by changing the configuration and doing an actual build. Continue on to *A Quick Tour*.

A Quick Tour

Goal

This tutorial will expand on the *First Run* tutorial by taking a quick tour around some of the features of buildbot that are hinted at in the comments in the sample configuration. We will simply change parts of the default configuration and explain the activated features.

As a part of this tutorial, we will make buildbot do a few actual builds.

This section will teach you how to:

- make simple configuration changes and activate them
- deal with configuration errors
- force builds
- enable and control the IRC bot

- enable ssh debugging
- add a 'try' scheduler

Setting Project Name and URL

Let's start simple by looking at where you would customize the buildbot's project name and URL.

We continue where we left off in the *First Run* tutorial.

Open a new terminal, and first enter the same sandbox you created before (where `$EDITOR` is your editor of choice like vim, gedit, or emacs):

```
cd
cd tmp/buildbot
source sandbox/bin/activate
$EDITOR master/master.cfg
```

Now, look for the section marked *PROJECT IDENTITY* which reads:

```
####### PROJECT IDENTITY

# the 'title' string will appear at the top of this buildbot installation's
# home pages (linked to the 'titleURL').

c['title'] = "Pyflakes"
c['titleURL'] = "http://divmod.org/trac/wiki/DivmodPyflakes"
```

If you want, you can change either of these links to anything you want to see what happens when you change them.

After making a change go into the terminal and type:

```
buildbot reconfig master
```

You will see a handful of lines of output from the master log, much like this:

```
2011-12-04 10:11:09-0600 [-] loading configuration from /home/dustin/tmp/buildbot/
↪master/master.cfg
2011-12-04 10:11:09-0600 [-] configuration update started
2011-12-04 10:11:09-0600 [-] builder runtests is unchanged
2011-12-04 10:11:09-0600 [-] removing IStatusReceiver <WebStatus on port tcp:8010 at␣
↪0x2aee368>
2011-12-04 10:11:09-0600 [-] (TCP Port 8010 Closed)
2011-12-04 10:11:09-0600 [-] Stopping factory <buildbot.status.web.baseweb.
↪RotateLogSite instance at 0x2e36638>
2011-12-04 10:11:09-0600 [-] adding IStatusReceiver <WebStatus on port tcp:8010 at␣
↪0x2c2d950>
2011-12-04 10:11:09-0600 [-] RotateLogSite starting on 8010
2011-12-04 10:11:09-0600 [-] Starting factory <buildbot.status.web.baseweb.
↪RotateLogSite instance at 0x2e36e18>
2011-12-04 10:11:09-0600 [-] Setting up http.log rotating 10 files of 10000000 bytes␣
↪each
2011-12-04 10:11:09-0600 [-] WebStatus using (/home/dustin/tmp/buildbot/master/public_
↪html)
2011-12-04 10:11:09-0600 [-] removing 0 old schedulers, updating 0, and adding 0
2011-12-04 10:11:09-0600 [-] adding 1 new changesources, removing 1
2011-12-04 10:11:09-0600 [-] gitpoller: using workdir '/home/dustin/tmp/buildbot/
↪master/gitpoller-workdir'
2011-12-04 10:11:09-0600 [-] GitPoller repository already exists
```

```
2011-12-04 10:11:09-0600 [-] configuration update complete

Reconfiguration appears to have completed successfully.
```

The important lines are the ones telling you that it is loading the new configuration at the top, and the one at the bottom saying that the update is complete.

Now, if you go back to the waterfall page (http://localhost:8010/waterfall), you will see that the project's name is whatever you may have changed it to and when you click on the URL of the project name at the bottom of the page it should take you to the link you put in the configuration.

Configuration Errors

It is very common to make a mistake when configuring buildbot, so you might as well see now what happens in that case and what you can do to fix the error.

Open up the config again and introduce a syntax error by removing the first single quote in the two lines you changed, so they read:

```
c[title'] = "Pyflakes
c[titleURL'] = "http://divmod.org/trac/wiki/DivmodPyflakes"
```

This creates a Python `SyntaxError`. Now go ahead and reconfig the buildmaster:

```
buildbot reconfig master
```

This time, the output looks like:

```
2015-08-14 18:40:46+0000 [-] beginning configuration update
2015-08-14 18:40:46+0000 [-] Loading configuration from '/data/buildbot/master/master.
↪cfg'
2015-08-14 18:40:46+0000 [-] error while parsing config file:
        Traceback (most recent call last):
          File "/usr/local/lib/python2.7/dist-packages/buildbot/master.py", line 265,␣
↪in reconfig
            d = self.doReconfig()
          File "/usr/local/lib/python2.7/dist-packages/twisted/internet/defer.py",␣
↪line 1274, in unwindGenerator
            return _inlineCallbacks(None, gen, Deferred())
          File "/usr/local/lib/python2.7/dist-packages/twisted/internet/defer.py",␣
↪line 1128, in _inlineCallbacks
            result = g.send(result)
          File "/usr/local/lib/python2.7/dist-packages/buildbot/master.py", line 289,␣
↪in doReconfig
            self.configFileName)
        --- <exception caught here> ---
          File "/usr/local/lib/python2.7/dist-packages/buildbot/config.py", line 156,␣
↪in loadConfig
            exec f in localDict
        exceptions.SyntaxError: EOL while scanning string literal (master.cfg, line␣
↪103)

2015-08-14 18:40:46+0000 [-] error while parsing config file: EOL while scanning␣
↪string literal (master.cfg, line 103) (traceback in logfile)
2015-08-14 18:40:46+0000 [-] reconfig aborted without making any changes
```

```
Reconfiguration failed. Please inspect the master.cfg file for errors,
correct them, then try 'buildbot reconfig' again.
```

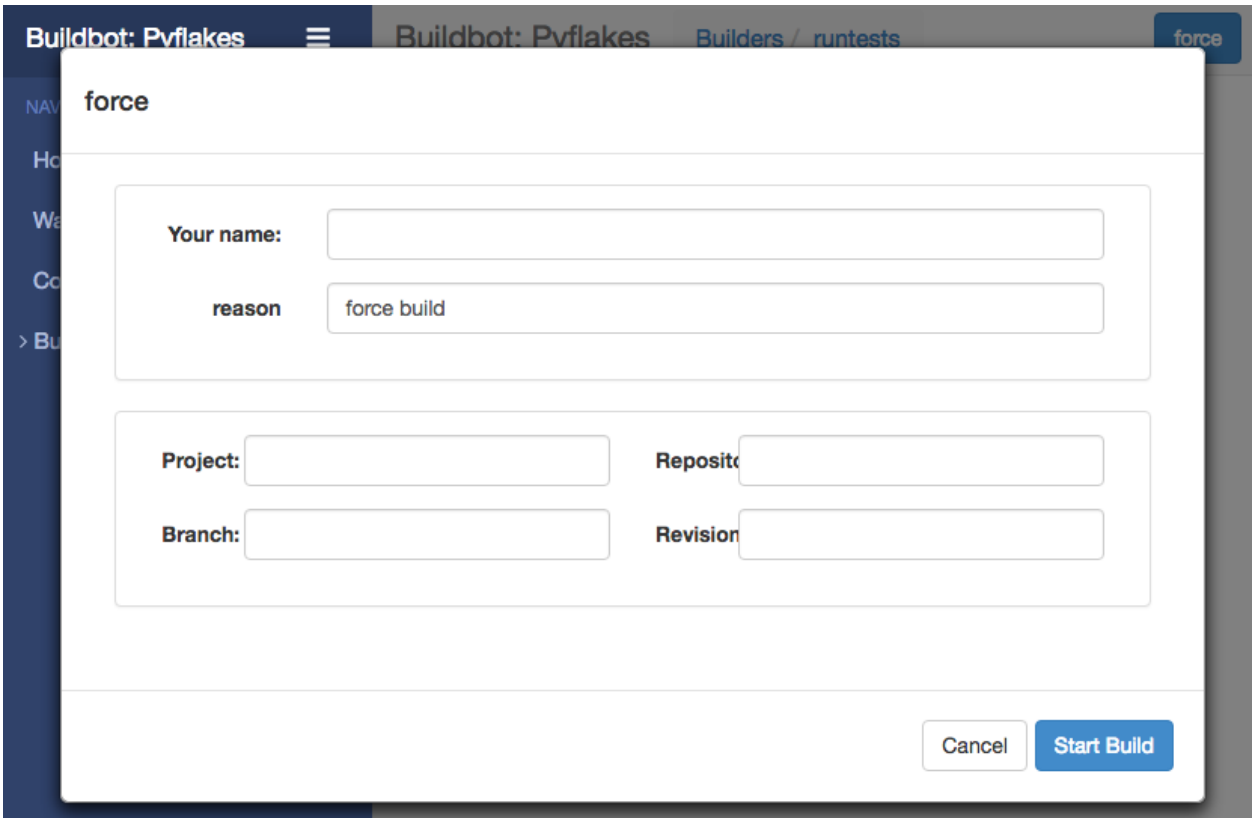
This time, it's clear that there was a mistake in the configuration. Luckily, the Buildbot master will ignore the wrong configuration and keep running with the previous configuration.

The message is clear enough, so open the configuration again, fix the error, and reconfig the master.

Your First Build

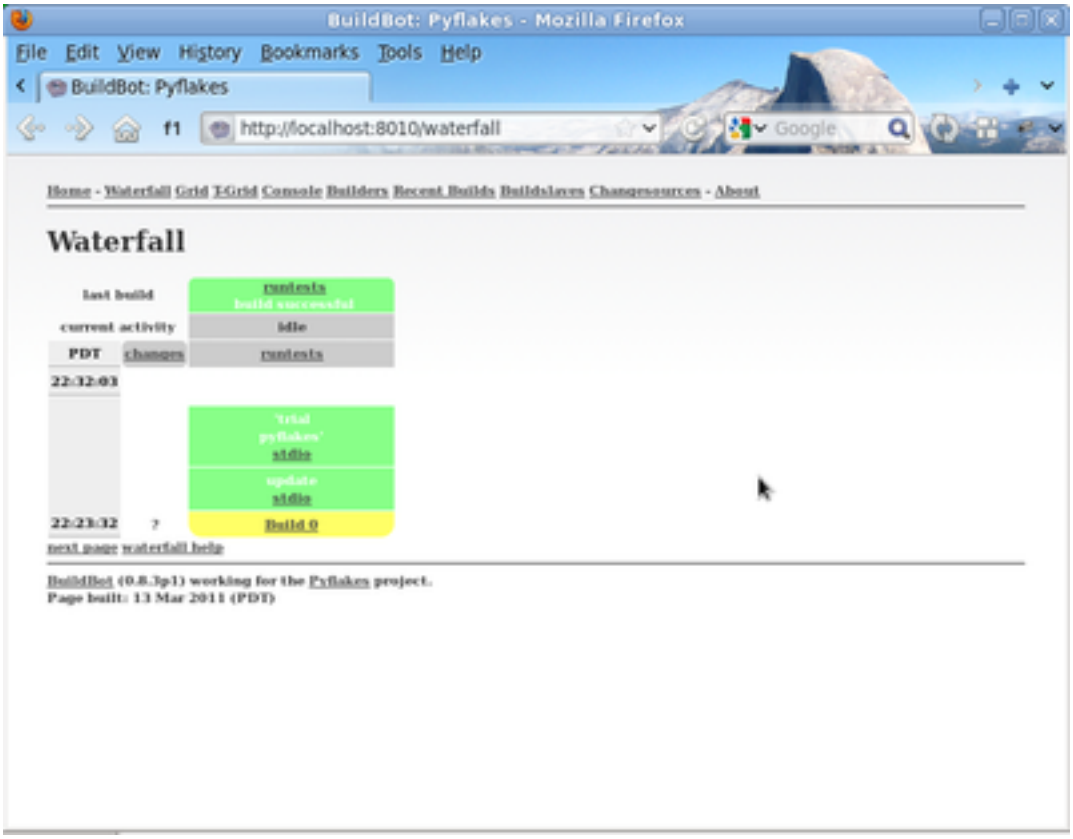
By now you're probably thinking: "All this time spent and still not done a single build? What was the name of this project again?"

On the waterfall (http://localhost:8010/waterfall) page, click on the runtests link. You'll see a builder page, and an option that allow you to force a build:



Click *Start Build* - there's no need to fill in any of the fields in this case. Next, click on view in waterfall (http://localhost:8010/waterfall?show=runtests).

You will now see:



Enabling the IRC Bot

Buildbot includes an IRC bot that you can tell to join a channel and control to report on the status of buildbot.

Note: Security Note

Please note that any user having access to your irc channel or can PM the bot will be able to create or stop builds bug #3377 (http://trac.buildbot.net/ticket/3377).

First, start an IRC client of your choice, connect to irc.freenode.net and join an empty channel. In this example we will use `#buildbot-test`, so go join that channel. (*Note: please do not join the main buildbot channel!*)

Edit `master.cfg` and look for the *BUILDBOT SERVICES* section. At the end of that section add the lines:

```
c['services'].append(reporters.IRC(host="irc.freenode.net", nick="bbtest",
                                   channels=["#buildbot-test"]))
```

Reconfigure the build master then do:

```
grep -i irc master/twistd.log
```

The log output should contain a line like this:

```
2016-11-13 15:53:06+0100 [-] Starting factory <buildbot.reporters.irc.
↪IrcStatusFactory instance at 0x7ff2b4b72710>
2016-11-13 15:53:19+0100 [IrcStatusBot,client] <buildbot.reporters.irc.IrcStatusBot␣
↪object at 0x7ff2b5075750>: I have joined #buildbot-test
```

You should see the bot now joining in your IRC client. In your IRC channel, type:

```
bbtest: commands
```

to get a list of the commands the bot supports.

Let's tell the bot to notify certain events, to learn which EVENTS we can notify on:

```
bbtest: help notify
```

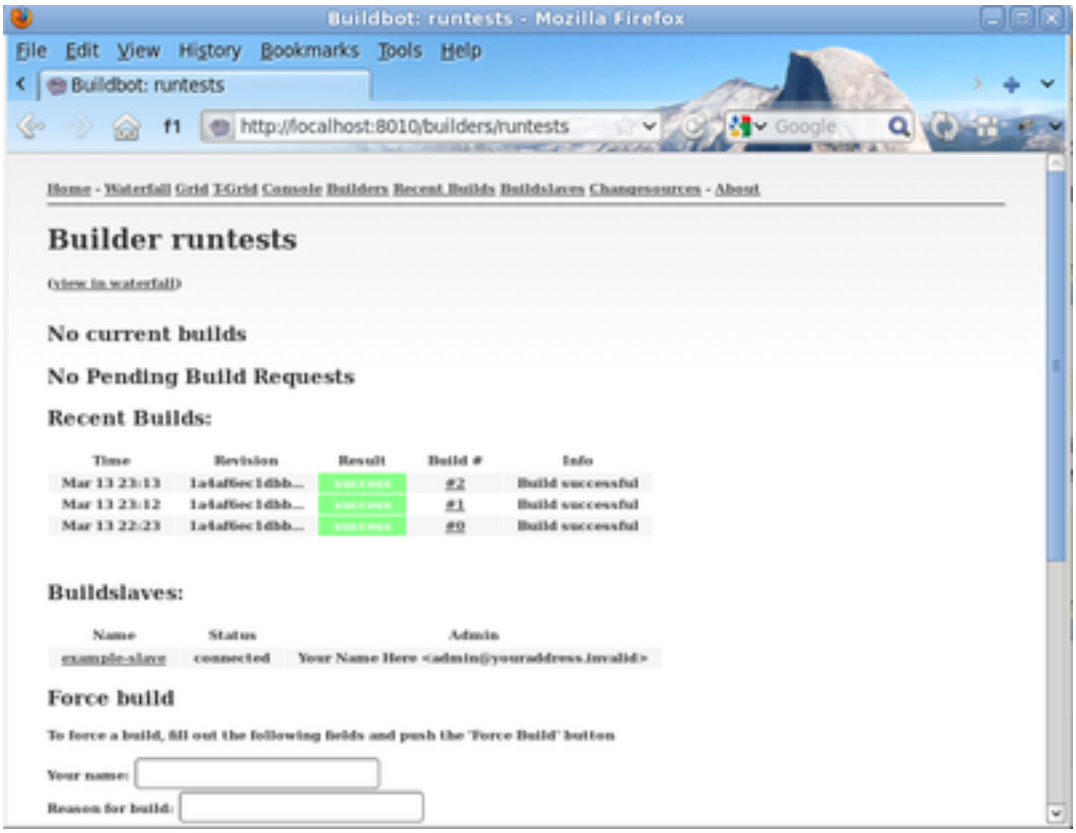
Now let's set some event notifications:

```
<@lsblakk> bbtest: notify on started finished failure
< bbtest> The following events are being notified: ['started', 'failure', 'finished']
```

Now, go back to the web interface and force another build. Alternatively, ask the bot to force a build:

```
<@lsblakk> bbtest: force build --codebase= runtests
< bbtest> build #1 of runtests started
< bbtest> Hey! build runtests #1 is complete: Success [finished]
```

You can also see the new builds in the web interface.



The full documentation is available at *IRC*.

Setting Authorized Web Users

The default configuration allows everyone to perform any task like creating or stopping builds via the web interface. To restrict this to a user, look for:

```
c['www'] = dict(port=8010,
                plugins=dict(waterfall_view={}, console_view={}))
```

and append:

```
c['www']['authz'] = util.Authz(
        allowRules = [
            util.AnyEndpointMatcher(role="admins")
        ],
        roleMatchers = [
            util.RolesFromUsername(roles=['admins'], usernames=['Alice'])
        ]
)
c['www']['auth'] = util.UserPasswordAuth([('Alice','Password1')])
```

For more details, see *Authentication plugins*.

Debugging with Manhole

You can do some debugging by using manhole, an interactive Python shell. It exposes full access to the buildmaster's account (including the ability to modify and delete files), so it should not be enabled with a weak or easily guessable password.

To use this you will need to install an additional package or two to your virtualenv:

```
cd
cd tmp/buildbot
source sandbox/bin/activate
easy_install pycrypto
easy_install pyasn1
```

In your master.cfg find:

```
c = BuildmasterConfig = {}
```

Insert the following to enable debugging mode with manhole:

```
####### DEBUGGING
from buildbot import manhole
c['manhole'] = manhole.PasswordManhole("tcp:1234:interface=127.0.0.1","admin","passwd
↪")
```

After restarting the master, you can ssh into the master and get an interactive Python shell:

```
ssh -p1234 admin@127.0.0.1
# enter passwd at prompt
```

Note: The pyasn1-0.1.1 release has a bug which results in an exception similar to this on startup:

```
exceptions.TypeError: argument 2 must be long, not int
```

If you see this, the temporary solution is to install the previous version of pyasn1:

```
pip install pyasn1-0.0.13b
```

If you wanted to check which workers are connected and what builders those workers are assigned to you could do:

```
>>> master.workers.workers
{'example-worker': <Worker 'example-worker', current builders: runtests>}
```

Objects can be explored in more depth using *dir(x)* or the helper function *show(x)*.

Adding a 'try' scheduler

Buildbot includes a way for developers to submit patches for testing without committing them to the source code control system. (This is really handy for projects that support several operating systems or architectures.)

To set this up, add the following lines to master.cfg:

```
from buildbot.scheduler import Try_Userpass
c['schedulers'] = []
c['schedulers'].append(Try_Userpass(
                                    name='try',
                                    builderNames=['runtests'],
                                    port=5555,
                                    userpass=[('sampleuser','samplepass')]))
```

Then you can submit changes using the *try* command.

Let's try this out by making a one-line change to pyflakes, say, to make it trace the tree by default:

```
git clone git://github.com/buildbot/pyflakes.git pyflakes-git
cd pyflakes-git/pyflakes
$EDITOR checker.py
# change "traceTree = False" on line 185 to "traceTree = True"
```

Then run buildbot's `try` command as follows:

```
source ~/tmp/buildbot/sandbox/bin/activate
buildbot try --connect=pb --master=127.0.0.1:5555 --username=sampleuser --
↪passwd=samplepass --vc=git
```

This will do `git diff` for you and send the resulting patch to the server for build and test against the latest sources from Git.

Now go back to the waterfall (http://localhost:8010/waterfall) page, click on the runtests link, and scroll down. You should see that another build has been started with your change (and stdout for the tests should be chock-full of parse trees as a result). The "Reason" for the job will be listed as "'try' job", and the blamelist will be empty.

To make yourself show up as the author of the change, use the `--who=emailaddr` option on `buildbot try` to pass your email address.

To make a description of the change show up, use the `--properties=comment="this is a comment"` option on `buildbot try`.

To use ssh instead of a private username/password database, see *Try_Jobdir*.

Further Reading

See the following user-contributed tutorials for other highlights and ideas:

Buildbot in 5 minutes - a user-contributed tutorial

(Ok, maybe 10.)

Buildbot is really an excellent piece of software, however it can be a bit confusing for a newcomer (like me when I first started looking at it). Typically, at first sight it looks like a bunch of complicated concepts that make no sense and whose relationships with each other are unclear. After some time and some reread, it all slowly starts to be more and more meaningful, until you finally say "oh!" and things start to make sense. Once you get there, you realize that the documentation is great, but only if you already know what it's about.

This is what happened to me, at least. Here I'm going to (try to) explain things in a way that would have helped me more as a newcomer. The approach I'm taking is more or less the reverse of that used by the documentation, that is, I'm going to start from the components that do the actual work (the builders) and go up the chain from there up to change sources. I hope purists will forgive this unorthodoxy. Here I'm trying to clarify the concepts only, and will not go into the details of each object or property; the documentation explains those quite well.

Installation

I won't cover the installation; both Buildbot master and worker are available as packages for the major distributions, and in any case the instructions in the official documentation are fine. This document will refer to Buildbot 0.8.5 which was current at the time of writing, but hopefully the concepts are not too different in other versions. All the code shown is of course python code, and has to be included in the master.cfg master configuration file.

We won't cover the basic things such as how to define the workers, project names, or other administrative information that is contained in that file; for that, again the official documentation is fine.

Builders: the workhorses

Since Buildbot is a tool whose goal is the automation of software builds, it makes sense to me to start from where we tell Buildbot how to build our software: the *builder* (or builders, since there can be more than one).

Simply put, a builder is an element that is in charge of performing some action or sequence of actions, normally something related to building software (for example, checking out the source, or `make all`), but it can also run arbitrary commands.

A builder is configured with a list of workers that it can use to carry out its task. The other fundamental piece of information that a builder needs is, of course, the list of things it has to do (which will normally run on the chosen worker). In Buildbot, this list of things is represented as a `BuildFactory` object, which is essentially a sequence of steps, each one defining a certain operation or command.

Enough talk, let's see an example. For this example, we are going to assume that our super software project can be built using a simple `make all`, and there is another target `make packages` that creates rpm, deb and tgz packages of the binaries. In the real world things are usually more complex (for example there may be a `configure` step, or multiple targets), but the concepts are the same; it will just be a matter of adding more steps to a builder, or creating multiple builders, although sometimes the resulting builders can be quite complex.

So to perform a manual build of our project we would type this from the command line (assuming we are at the root of the local copy of the repository):

```
$ make clean    # clean remnants of previous builds
...
$ svn update
...
$ make all
...
$ make packages
...
```

```
# optional but included in the example: copy packages to some central machine
$ scp packages/*.rpm packages/*.deb packages/*.tgz someuser@somehost:/repository
...
```

Here we're assuming the repository is SVN, but again the concepts are the same with git, mercurial or any other VCS.

Now, to automate this, we create a builder where each step is one of the commands we typed above. A step can be a shell command object, or a dedicated object that checks out the source code (there are various types for different repositories, see the docs for more info), or yet something else:

```
from buildbot.plugins import steps, util

# first, let's create the individual step objects

# step 1: make clean; this fails if the worker has no local copy, but
# is harmless and will only happen the first time
makeclean = steps.ShellCommand(name="make clean",
                               command=["make", "clean"],
                               description="make clean")

# step 2: svn update (here updates trunk, see the docs for more
# on how to update a branch, or make it more generic).
checkout = steps.SVN(baseURL='svn://myrepo/projects/coolproject/trunk',
                     mode="update",
                     username="foo",
                     password="bar",
                     haltOnFailure=True)

# step 3: make all
makeall = steps.ShellCommand(name="make all",
                             command=["make", "all"],
                             haltOnFailure=True,
                             description="make all")

# step 4: make packages
makepackages = steps.ShellCommand(name="make packages",
                                  command=["make", "packages"],
                                  haltOnFailure=True,
                                  description="make packages")

# step 5: upload packages to central server. This needs passwordless ssh
# from the worker to the server (set it up in advance as part of worker setup)
uploadpackages = steps.ShellCommand(name="upload packages",
                                    description="upload packages",
                                    command="scp packages/*.rpm packages/*.deb␣
↪packages/*.tgz someuser@somehost:/repository",
                                    haltOnFailure=True)

# create the build factory and add the steps to it
f_simplebuild = util.BuildFactory()
f_simplebuild.addStep(makeclean)
f_simplebuild.addStep(checkout)
f_simplebuild.addStep(makeall)
f_simplebuild.addStep(makepackages)
f_simplebuild.addStep(uploadpackages)

# finally, declare the list of builders. In this case, we only have one builder
c['builders'] = [
```

```
    util.BuilderConfig(name="simplebuild", workernames=['worker1', 'worker2', 'worker3
↪'], factory=f_simplebuild)
]
```

So our builder is called `simplebuild` and can run on either of `worker1`, `worker2` and `worker3`. If our repository has other branches besides trunk, we could create another one or more builders to build them; in the example, only the checkout step would be different, in that it would need to check out the specific branch. Depending on how exactly those branches have to be built, the shell commands may be recycled, or new ones would have to be created if they are different in the branch. You get the idea. The important thing is that all the builders be named differently and all be added to the `c['builders']` value (as can be seen above, it is a list of `BuilderConfig` objects).

Of course the type and number of steps will vary depending on the goal; for example, to just check that a commit doesn't break the build, we could include just up to the `make all` step. Or we could have a builder that performs a more thorough test by also doing `make test` or other targets. You get the idea. Note that at each step except the very first we use `haltOnFailure=True` because it would not make sense to execute a step if the previous one failed (ok, it wouldn't be needed for the last step, but it's harmless and protects us if one day we add another step after it).

Schedulers

Now this is all nice and dandy, but who tells the builder (or builders) to run, and when? This is the job of the *scheduler*, which is a fancy name for an element that waits for some event to happen, and when it does, based on that information decides whether and when to run a builder (and which one or ones). There can be more than one scheduler. I'm being purposely vague here because the possibilities are almost endless and highly dependent on the actual setup, build purposes, source repository layout and other elements.

So a scheduler needs to be configured with two main pieces of information: on one hand, which events to react to, and on the other hand, which builder or builders to trigger when those events are detected. (It's more complex than that, but if you understand this, you can get the rest of the details from the docs).

A simple type of scheduler may be a periodic scheduler: when a configurable amount of time has passed, run a certain builder (or builders). In our example, that's how we would trigger a build every hour:

```
from buildbot.plugins import schedulers

# define the periodic scheduler
hourlyscheduler = schedulers.Periodic(name="hourly",
                                      builderNames=["simplebuild"],
                                      periodicBuildTimer=3600)

# define the available schedulers
c['schedulers'] = [hourlyscheduler]
```

That's it. Every hour this `hourly` scheduler will run the `simplebuild` builder. If we have more than one builder that we want to run every hour, we can just add them to the `builderNames` list when defining the scheduler and they will all be run. Or since multiple scheduler are allowed, other schedulers can be defined and added to `c['schedulers']` in the same way.

Other types of schedulers exist; in particular, there are schedulers that can be more dynamic than the periodic one. The typical dynamic scheduler is one that learns about changes in a source repository (generally because some developer checks in some change), and triggers one or more builders in response to those changes. Let's assume for now that the scheduler "magically" learns about changes in the repository (more about this later); here's how we would define it:

```
from buildbot.plugins import schedulers

# define the dynamic scheduler
```

```
trunkchanged = schedulers.SingleBranchScheduler(name="trunkchanged",
                                                change_filter=util.
 ↪ChangeFilter(branch=None),
                                                treeStableTimer=300,
                                                builderNames=["simplebuild"])

# define the available schedulers
c['schedulers'] = [trunkchanged]
```

This scheduler receives changes happening to the repository, and among all of them, pays attention to those happening in "trunk" (that's what `branch=None` means). In other words, it filters the changes to react only to those it's interested in. When such changes are detected, and the tree has been quiet for 5 minutes (300 seconds), it runs the `simplebuild` builder. The `treeStableTimer` helps in those situations where commits tend to happen in bursts, which would otherwise result in multiple build requests queuing up.

What if we want to act on two branches (say, trunk and 7.2)? First we create two builders, one for each branch (see the builders paragraph above), then we create two dynamic schedulers:

```
from buildbot.plugins import schedulers

# define the dynamic scheduler for trunk
trunkchanged = schedulers.SingleBranchScheduler(name="trunkchanged",
                                                change_filter=util.
 ↪ChangeFilter(branch=None),
                                                treeStableTimer=300,
                                                builderNames=["simplebuild-trunk"])

# define the dynamic scheduler for the 7.2 branch
branch72changed = schedulers.SingleBranchScheduler(name="branch72changed",
                                                   change_filter=util.
 ↪ChangeFilter(branch='branches/7.2'),
                                                   treeStableTimer=300,
                                                   builderNames=["simplebuild-72"])

# define the available schedulers
c['schedulers'] = [trunkchanged, branch72changed]
```

The syntax of the change filter is VCS-dependent (above is for SVN), but again once the idea is clear, the documentation has all the details. Another feature of the scheduler is that is can be told which changes, within those it's paying attention to, are important and which are not. For example, there may be a documentation directory in the branch the scheduler is watching, but changes under that directory should not trigger a build of the binary. This finer filtering is implemented by means of the `fileIsImportant` argument to the scheduler (full details in the docs and - alas - in the sources).

Change sources

Earlier we said that a dynamic scheduler "magically" learns about changes; the final piece of the puzzle are *change sources*, which are precisely the elements in Buildbot whose task is to detect changes in the repository and communicate them to the schedulers. Note that periodic schedulers don't need a change source, since they only depend on elapsed time; dynamic schedulers, on the other hand, do need a change source.

A change source is generally configured with information about a source repository (which is where changes happen); a change source can watch changes at different levels in the hierarchy of the repository, so for example it is possible to watch the whole repository or a subset of it, or just a single branch. This determines the extent of the information that is passed down to the schedulers.

There are many ways a change source can learn about changes; it can periodically poll the repository for changes, or the VCS can be configured (for example through hook scripts triggered by commits) to push changes into the change source. While these two methods are probably the most common, they are not the only possibilities; it is possible for example to have a change source detect changes by parsing some email sent to a mailing list when a commit happens, and yet other methods exist. The manual again has the details.

To complete our example, here's a change source that polls a SVN repository every 2 minutes:

```
from buildbot.plugins import changes, util

svnpoller = changes.SVNPoller(repourl="svn://myrepo/projects/coolproject",
                              svnuser="foo",
                              svnpasswd="bar",
                              pollinterval=120,
                              split_file=util.svn.split_file_branches)

c['change_source'] = svnpoller
```

This poller watches the whole "coolproject" section of the repository, so it will detect changes in all the branches. We could have said:

```
repourl = "svn://myrepo/projects/coolproject/trunk"
```

or:

```
repourl = "svn://myrepo/projects/coolproject/branches/7.2"
```

to watch only a specific branch.

To watch another project, you need to create another change source – and you need to filter changes by project. For instance, when you add a change source watching project 'superproject' to the above example, you need to change:

```
trunkchanged = schedulers.SingleBranchScheduler(name="trunkchanged",
                                                change_filter=filter.
↪ChangeFilter(branch=None),
                                                # ...
                                                )
```

to e.g.:

```
trunkchanged = schedulers.SingleBranchScheduler(name="trunkchanged",
                                                change_filter=filter.
↪ChangeFilter(project="coolproject", branch=None),
                                                # ...
                                                )
```

else coolproject will be built when there's a change in superproject.

Since we're watching more than one branch, we need a method to tell in which branch the change occurred when we detect one. This is what the `split_file` argument does, it takes a callable that Buildbot will call to do the job. The split_file_branches function, which comes with Buildbot, is designed for exactly this purpose so that's what the example above uses.

And of course this is all SVN-specific, but there are pollers for all the popular VCSs.

But note: if you have many projects, branches, and builders it probably pays to not hardcode all the schedulers and builders in the configuration, but generate them dynamically starting from list of all projects, branches, targets etc. and using loops to generate all possible combinations (or only the needed ones, depending on the specific setup), as explained in the documentation chapter about *Customization*.

Status targets

Now that the basics are in place, let's go back to the builders, which is where the real work happens. *Status targets* are simply the means Buildbot uses to inform the world about what's happening, that is, how builders are doing. There are many status targets: a web interface, a mail notifier, an IRC notifier, and others. They are described fairly well in the manual.

One thing I've found useful is the ability to pass a domain name as the lookup argument to a `mailNotifier`, which allows you to take an unqualified username as it appears in the SVN change and create a valid email address by appending the given domain name to it:

```
from buildbot.plugins import status

# if jsmith commits a change, mail for the build is sent to jsmith@example.org
notifier = status.MailNotifier(fromaddr="buildbot@example.org",
                               sendToInterestedUsers=True,
                               lookup="example.org")
c['status'].append(notifier)
```

The mail notifier can be customized at will by means of the `messageFormatter` argument, which is a class that Buildbot calls to format the body of the email, and to which it makes available lots of information about the build. Here all the details.

Conclusion

Please note that this article has just scratched the surface; given the complexity of the task of build automation, the possibilities are almost endless. So there's much, much more to say about Buildbot. However, hopefully this is a preparation step before reading the official manual. Had I found an explanation as the one above when I was approaching Buildbot, I'd have had to read the manual just once, rather than multiple times. Hope this can help someone else.

(Thanks to Davide Brini for permission to include this tutorial, derived from one he originally posted at http://backreference.org .)

This is the Buildbot manual for Buildbot version `|version|`.

CHAPTER 2

Buildbot Manual

Introduction

Buildbot is a system to automate the compile/test cycle required by most software projects to validate code changes. By automatically rebuilding and testing the tree each time something has changed, build problems are pinpointed quickly, before other developers are inconvenienced by the failure. The guilty developer can be identified and harassed without human intervention. By running the builds on a variety of platforms, developers who do not have the facilities to test their changes everywhere before checkin will at least know shortly afterwards whether they have broken the build or not. Warning counts, lint checks, image size, compile time, and other build parameters can be tracked over time, are more visible, and are therefore easier to improve.

The overall goal is to reduce tree breakage and provide a platform to run tests or code-quality checks that are too annoying or pedantic for any human to waste their time with. Developers get immediate (and potentially public) feedback about their changes, encouraging them to be more careful about testing before checkin.

Features:

- run builds on a variety of worker platforms
- arbitrary build process: handles projects using C, Python, whatever
- minimal host requirements: Python and Twisted
- workers can be behind a firewall if they can still do checkout
- status delivery through web page, email, IRC, other protocols
- track builds in progress, provide estimated completion time
- flexible configuration by subclassing generic build process classes
- debug tools to force a new build, submit fake `Changes`, query worker status
- released under the GPL (https://opensource.org/licenses/gpl-2.0.php)

History and Philosophy

The Buildbot was inspired by a similar project built for a development team writing a cross-platform embedded system. The various components of the project were supposed to compile and run on several flavors of unix (linux, solaris, BSD), but individual developers had their own preferences and tended to stick to a single platform. From time to time, incompatibilities would sneak in (some unix platforms want to use `string.h`, some prefer `strings.h`), and then the tree would compile for some developers but not others. The buildbot was written to automate the human process of walking into the office, updating a tree, compiling (and discovering the breakage), finding the developer at fault, and complaining to them about the problem they had introduced. With multiple platforms it was difficult for developers to do the right thing (compile their potential change on all platforms); the buildbot offered a way to help.

Another problem was when programmers would change the behavior of a library without warning its users, or change internal aspects that other code was (unfortunately) depending upon. Adding unit tests to the codebase helps here: if an application's unit tests pass despite changes in the libraries it uses, you can have more confidence that the library changes haven't broken anything. Many developers complained that the unit tests were inconvenient or took too long to run: having the buildbot run them reduces the developer's workload to a minimum.

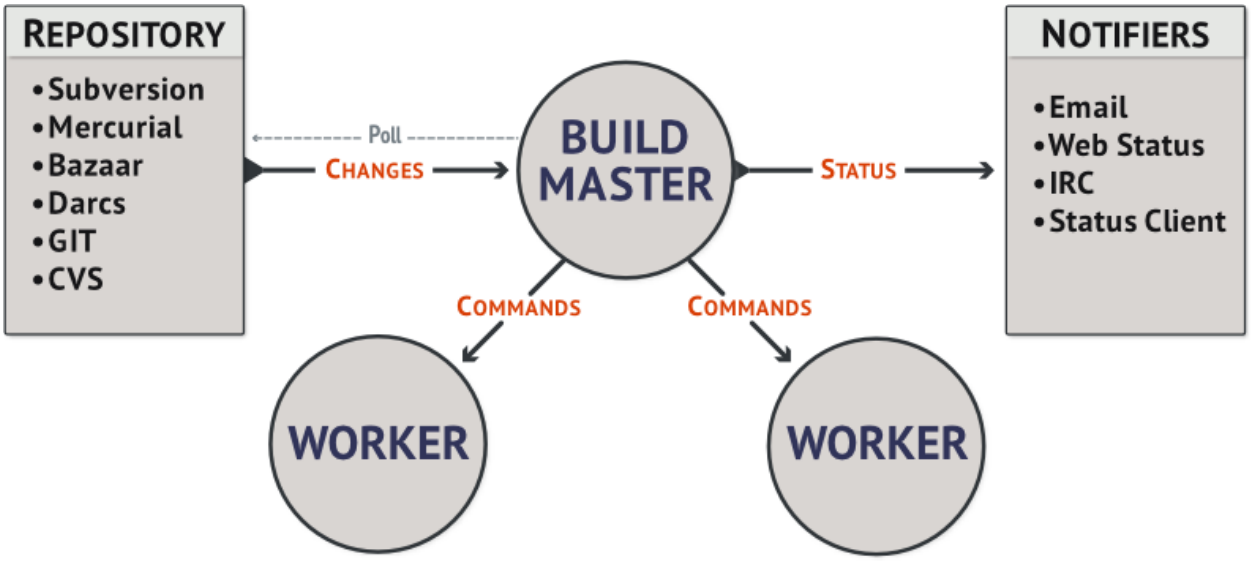
In general, having more visibility into the project is always good, and automation makes it easier for developers to do the right thing. When everyone can see the status of the project, developers are encouraged to keep the tree in good working order. Unit tests that aren't run on a regular basis tend to suffer from bitrot just like code does: exercising them on a regular basis helps to keep them functioning and useful.

The current version of the Buildbot is additionally targeted at distributed free-software projects, where resources and platforms are only available when provided by interested volunteers. The workers are designed to require an absolute minimum of configuration, reducing the effort a potential volunteer needs to expend to be able to contribute a new test environment to the project. The goal is for anyone who wishes that a given project would run on their favorite platform should be able to offer that project a worker, running on that platform, where they can verify that their portability code works, and keeps working.

System Architecture

The Buildbot consists of a single *buildmaster* and one or more *workers*, connected in a star topology. The buildmaster makes all decisions about what, when, and how to build. It sends commands to be run on the workers, which simply execute the commands and return the results. (certain steps involve more local decision making, where the overhead of sending a lot of commands back and forth would be inappropriate, but in general the buildmaster is responsible for everything).

The buildmaster is usually fed `Changes` by some sort of version control system (*Change Sources*), which may cause builds to be run. As the builds are performed, various status messages are produced, which are then sent to any registered *Reporters*.

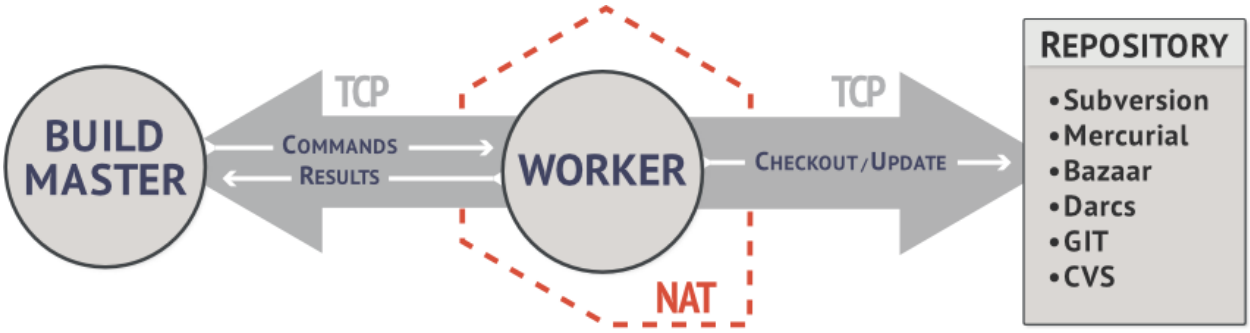


The buildmaster is configured and maintained by the *buildmaster admin*, who is generally the project team member responsible for build process issues. Each worker is maintained by a *worker admin*, who do not need to be quite as involved. Generally workers are run by anyone who has an interest in seeing the project work well on their favorite platform.

Worker Connections

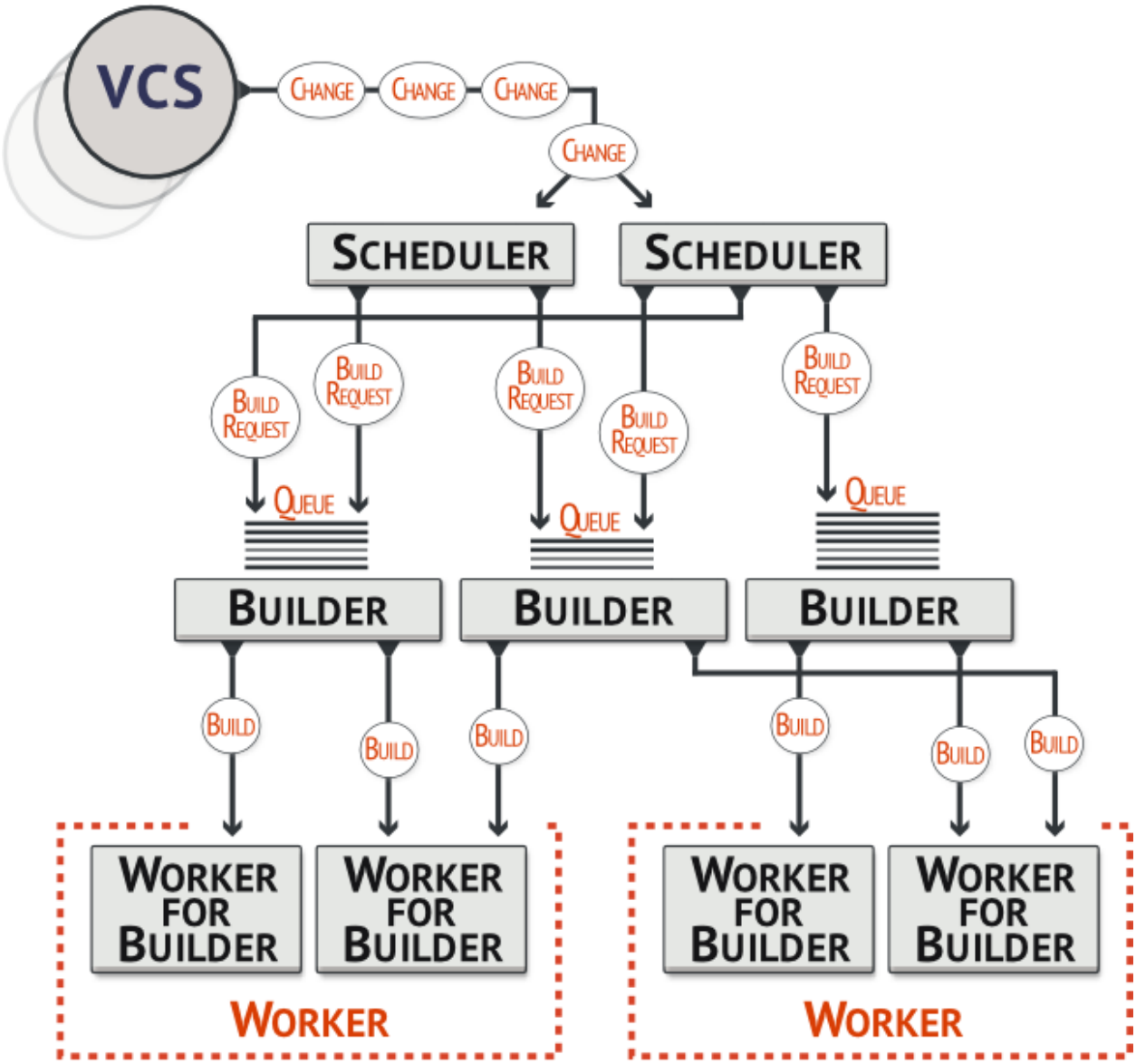
The workers are typically run on a variety of separate machines, at least one per platform of interest. These machines connect to the buildmaster over a TCP connection to a publically-visible port. As a result, the workers can live behind a NAT box or similar firewalls, as long as they can get to buildmaster. The TCP connections are initiated by the worker and accepted by the buildmaster, but commands and results travel both ways within this connection. The buildmaster is always in charge, so all commands travel exclusively from the buildmaster to the worker.

To perform builds, the workers must typically obtain source code from a CVS/SVN/etc repository. Therefore they must also be able to reach the repository. The buildmaster provides instructions for performing builds, but does not provide the source code itself.



Buildmaster Architecture

The buildmaster consists of several pieces:



Change Sources Which create a Change object each time something is modified in the VC repository. Most `ChangeSources` listen for messages from a hook script of some sort. Some sources actively poll the repository on a regular basis. All `Changes` are fed to the schedulers.

Schedulers Which decide when builds should be performed. They collect `Changes` into `BuildRequests`, which are then queued for delivery to `Builders` until a worker is available.

Builders Which control exactly *how* each build is performed (with a series of `BuildSteps`, configured in a `BuildFactory`). Each `Build` is run on a single worker.

Status plugins Which deliver information about the build results through protocols like HTTP, mail, and IRC.

Each `Builder` is configured with a list of `Workers` that it will use for its builds. These workers are expected to behave identically: the only reason to use multiple `Workers` for a single `Builder` is to provide a measure of load-balancing.

Within a single `Worker`, each `Builder` creates its own `WorkerForBuilder` instance. These `WorkerForBuilders` operate independently from each other. Each gets its own base directory to work in. It is quite common to have many `Builders` sharing the same worker. For example, there might be two workers: one

for i386, and a second for PowerPC. There may then be a pair of `Builders` that do a full compile/test run, one for each architecture, and a lone `Builder` that creates snapshot source tarballs if the full builders complete successfully. The full builders would each run on a single worker, whereas the tarball creation step might run on either worker (since the platform doesn't matter when creating source tarballs). In this case, the mapping would look like:

```
Builder(full-i386)  ->  Workers(worker-i386)
Builder(full-ppc)   ->  Workers(worker-ppc)
Builder(source-tarball) -> Workers(worker-i386, worker-ppc)
```

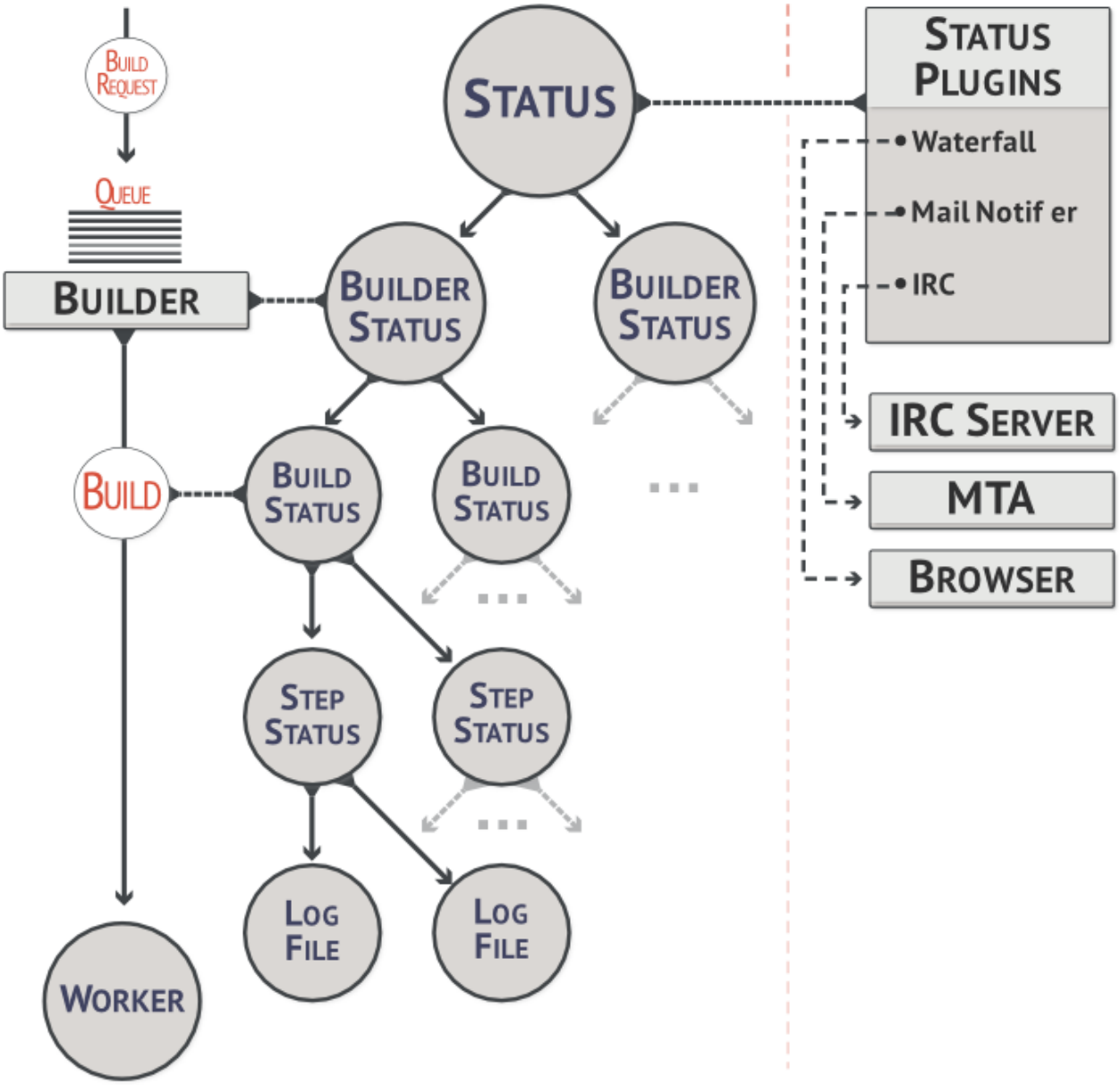
and each `Worker` would have two `WorkerForBuilders` inside it, one for a full builder, and a second for the source-tarball builder.

Once a `WorkerForBuilder` is available, the `Builder` pulls one or more `BuildRequests` off its incoming queue. (It may pull more than one if it determines that it can merge the requests together; for example, there may be multiple requests to build the current *HEAD* revision). These requests are merged into a single `Build` instance, which includes the `SourceStamp` that describes what exact version of the source code should be used for the build. The `Build` is then randomly assigned to a free `WorkerForBuilder` and the build begins.

The behaviour when `BuildRequests` are merged can be customized, *Collapsing Build Requests*.

Status Delivery Architecture

The buildmaster maintains a central `Status` object, to which various status plugins are connected. Through this `Status` object, a full hierarchy of build status objects can be obtained.



The configuration file controls which status plugins are active. Each status plugin gets a reference to the top-level `Status` object. From there they can request information on each `Builder`, `Build`, `Step`, and `LogFile`. This query-on-demand interface is used by the `html.Waterfall` plugin to create the main status page each time a web browser hits the main URL.

The status plugins can also subscribe to hear about new `Builds` as they occur: this is used by the `MailNotifier` to create new email messages for each recently-completed `Build`.

The `Status` object records the status of old builds on disk in the buildmaster's base directory. This allows it to return information about historical builds.

There are also status objects that correspond to `Schedulers` and `Workers`. These allow status plugins to report information about upcoming builds, and the online/offline status of each worker.

Control Flow

A day in the life of the buildbot:

- A developer commits some source code changes to the repository. A hook script or commit trigger of some sort sends information about this change to the buildmaster through one of its configured Change Sources. This notification might arrive via email, or over a network connection (either initiated by the buildmaster as it *subscribes* to changes, or by the commit trigger as it pushes `Changes` towards the buildmaster). The `Change` contains information about who made the change, what files were modified, which revision contains the change, and any checkin comments.
- The buildmaster distributes this change to all of its configured schedulers. Any `important` changes cause the `tree-stable-timer` to be started, and the `Change` is added to a list of those that will go into a new `Build`. When the timer expires, a `Build` is started on each of a set of configured Builders, all compiling/testing the same source code. Unless configured otherwise, all `Build`s run in parallel on the various workers.
- The `Build` consists of a series of `Step`s. Each `Step` causes some number of commands to be invoked on the remote worker associated with that `Builder`. The first step is almost always to perform a checkout of the appropriate revision from the same VC system that produced the `Change`. The rest generally perform a compile and run unit tests. As each `Step` runs, the worker reports back command output and return status to the buildmaster.
- As the `Build` runs, status messages like "Build Started", "Step Started", "Build Finished", etc, are published to a collection of Status Targets. One of these targets is usually the HTML `Waterfall` display, which shows a chronological list of events, and summarizes the results of the most recent build at the top of each column. Developers can periodically check this page to see how their changes have fared. If they see red, they know that they've made a mistake and need to fix it. If they see green, they know that they've done their duty and don't need to worry about their change breaking anything.
- If a `MailNotifier` status target is active, the completion of a build will cause email to be sent to any developers whose `Changes` were incorporated into this `Build`. The `MailNotifier` can be configured to only send mail upon failing builds, or for builds which have just transitioned from passing to failing. Other status targets can provide similar real-time notification via different communication channels, like IRC.

Installation

Buildbot Components

Buildbot is shipped in two components: the *buildmaster* (called `buildbot` for legacy reasons) and the *worker*. The worker component has far fewer requirements, and is more broadly compatible than the buildmaster. You will need to carefully pick the environment in which to run your buildmaster, but the worker should be able to run just about anywhere.

It is possible to install the buildmaster and worker on the same system, although for anything but the smallest installation this arrangement will not be very efficient.

Requirements

Common Requirements

At a bare minimum, you'll need the following for both the buildmaster and a worker:

Python: https://www.python.org

Buildbot master requires Python-2.7. Buildbot worker require Python-2.6, although Python-2.7 is recommended.

Note: This should be a "normal" build of Python. Builds of Python with debugging enabled or other unusual build parameters are likely to cause incorrect behavior.

Twisted: http://twistedmatrix.com

Buildbot requires Twisted-14.0.1 or later on the master, and Twisted-10.2.0 on the worker. In upcoming versions of Buildbot, a newer Twisted will also be required on the worker. As always, the most recent version is recommended. Note that Twisted requires ZopeInterface to be installed as well.

Future:

As part of ongoing (but as-yet incomplete) work to make Buildbot compatible with Python 3, the master requires the `future` module.

Of course, your project's build process will impose additional requirements on the workers. These hosts must have all the tools necessary to compile and test your project's source code.

Windows Support

Buildbot - both master and worker - runs well natively on Windows. The worker runs well on Cygwin, but because of problems with SQLite on Cygwin, the master does not.

Buildbot's windows testing is limited to the most recent Twisted and Python versions. For best results, use the most recent available versions of these libraries on Windows.

Pywin32: http://sourceforge.net/projects/pywin32/

Twisted requires PyWin32 in order to spawn processes on Windows.

Buildmaster Requirements

Note that all of these requirements aside from SQLite can easily be installed from the Python package repository, PyPI.

sqlite3: http://www.sqlite.org

Buildbot requires a database to store its state, and by default uses SQLite. Version 3.7.0 or higher is recommended, although Buildbot will run down to 3.6.16 – at the risk of "Database is locked" errors. The minimum version is 3.4.0, below which parallel database queries and schema introspection fail.

Please note that Python ships with sqlite3 by default since Python 2.6. Python2.6 for Windows ships with sqlite 3.6.2, thus you will not be able to run buildbot with sqlite on Windows and Python 2.6.

If you configure a different database engine, then SQLite is not required. however note that Buildbot's own unit tests require SQLite.

Jinja2: http://jinja.pocoo.org/

Buildbot requires Jinja version 2.1 or higher.

Jinja2 is a general purpose templating language and is used by Buildbot to generate the HTML output.

SQLAlchemy: http://www.sqlalchemy.org/

Buildbot requires SQLAlchemy version 0.8.0 or higher. SQLAlchemy allows Buildbot to build database schemas and queries for a wide variety of database systems.

SQLAlchemy-Migrate: https://sqlalchemy-migrate.readthedocs.io/en/latest/

Buildbot requires SQLAlchemy-Migrate version 0.9.0 or higher. Buildbot uses SQLAlchemy-Migrate to manage schema upgrades from version to version.

Python-Dateutil: http://labix.org/python-dateutil

Buildbot requires Python-Dateutil in version 1.5 or higher (the last version to support Python-2.x). This is a small, pure-Python library.

Autobahn:

The master requires Autobahn version 0.16.0 or higher with Python 2.7.

Installing the code

The Buildbot Packages

Buildbot comes in several parts: `buildbot` (the buildmaster), `buildbot-worker` (the worker), `buildbot-www`, and several web plugins such as `buildbot-waterfall-view`.

The worker and buildmaster can be installed individually or together. The base web (`buildbot.www`) and web plugins are required to run a master with a web interface (the common configuration).

Installation From PyPI

The preferred way to install Buildbot is using `pip`. For the master:

```
pip install buildbot
```

and for the worker:

```
pip install buildbot-worker
```

When using `pip` to install instead of distribution specific package manangers, e.g. via *apt-get* or *ports*, it is simpler to choose exactly which version one wants to use. It may however be easier to install via distribution specific package mangers but note that they may provide an earlier version than what is available via `pip`.

If you plan to use TLS or SSL in master configuration (e.g. to fetch resources over HTTPS using `twisted.web.client`), you need to install Buildbot with `tls` extras:

```
pip install buildbot[tls]
```

Installation From Tarballs

Buildbot master and `buildbot-worker` are installed using the standard Python distutils (http://docs.python.org/library/distutils.html) process. For either component, after unpacking the tarball, the process is:

```
python setup.py build
python setup.py install
```

where the install step may need to be done as root. This will put the bulk of the code in somewhere like `/usr/lib/pythonx.y/site-packages/buildbot`. It will also install the **`buildbot`** command-line tool in `/usr/bin/buildbot`.

If the environment variable `$NO_INSTALL_REQS` is set to `1`, then `setup.py` will not try to install Buildbot's requirements. This is usually only useful when building a Buildbot package.

To test this, shift to a different directory (like `/tmp`), and run:

```
buildbot --version
# or
buildbot-worker --version
```

If it shows you the versions of Buildbot and Twisted, the install went ok. If it says "no such command" or it gets an `ImportError` when it tries to load the libraries, then something went wrong. `pydoc buildbot` is another useful diagnostic tool.

Windows users will find these files in other places. You will need to make sure that Python can find the libraries, and will probably find it convenient to have **`buildbot`** on your `PATH`.

Installation in a Virtualenv

If you cannot or do not wish to install the buildbot into a site-wide location like `/usr` or `/usr/local`, you can also install it into the account's home directory or any other location using a tool like virtualenv (http://pypi.python.org/pypi/virtualenv).

Running Buildbot's Tests (optional)

If you wish, you can run the buildbot unit test suite. First, ensure you have the mock (http://pypi.python.org/pypi/mock) Python module installed from PyPI. You must not be using a Python wheels packaged version of Buildbot or have specified the bdist_wheel command when building. The test suite is not included with the PyPi packaged version. This module is not required for ordinary Buildbot operation - only to run the tests. Note that this is not the same as the Fedora `mock` package!

You can check with

```
python -mmock
```

Then, run the tests:

```
PYTHONPATH=. trial buildbot.test
# or
PYTHONPATH=. trial buildbot_worker.test
```

Nothing should fail, although a few might be skipped.

If any of the tests fail for reasons other than a missing `mock`, you should stop and investigate the cause before continuing the installation process, as it will probably be easier to track down the bug early. In most cases, the problem is incorrectly installed Python modules or a badly configured `PYTHONPATH`. This may be a good time to contact the Buildbot developers for help.

Upgrading to Nine

Upgrading a Buildbot instance from 0.8.x to 0.9.x may require a number of changes to the master configuration. Those changes are summarized here. If you are starting fresh with 0.9.0 or later, you can safely skip this section.

First important note is that Buildbot does not support an upgrade of a 0.8.x instance to 0.9.x. Notably the build data and logs will not be accessible anymore if you upgraded, thus the database migration scripts have been dropped.

You should not `pip upgrade -U buildbot`, but rather start from a clean virtualenv aside from your old master. You can keep your old master instance to serve the old build status.

Buildbot is now composed of several Python packages and Javascript UI, and the easiest way to install it is to run the following command within a virtualenv:

```
pip install 'buildbot[bundle]'
```

Config File Syntax

In preparation for compatibility with Python 3, Buildbot configuration files no longer allow the print statement:

```
print "foo"
```

To fix, simply enclose the print arguments in parentheses:

```
print("foo")
```

Plugins

Although plugin support was available in 0.8.12, its use is now highly recommended. Instead of importing modules directly in `master.cfg`, import the plugin kind from `buildbot.plugins`:

```
from buildbot.plugins import steps
```

Then access the plugin itself as an attribute:

```
steps.SetProperty(..)
```

See *Plugin Infrastructure in Buildbot* for more information.

Web Status

The most prominent change is that the existing `WebStatus` class is now gone, replaced by the new `www` functionality.

Thus an `html.WebStatus` entry in `c['status']` should be removed and replaced with configuration in `c['www']`. For example, replace:

```
from buildbot.status import html
c['status'].append(html.WebStatus(http_port=8010, allowForce=True)
```

with:

```
c['www'] = dict(port=8010,
                plugins=dict(waterfall_view={},
                console_view={}))
```

See *www* for more information.

Status Classes

Where in 0.8.x most of the data about a build was available synchronously, it must now be fetched dynamically using the *Data API*. All classes under the Python package `buildbot.status` should be considered deprecated. Many

have already been removed, and the remainder have limited functionality. Any custom code which refers to these classes must be rewritten to use the Data API. Avoid the temptation to reach into the Buildbot source code to find other useful-looking methods!

Common uses of the status API are:

- `getBuild` in a custom renderable
- `MailNotifier` message formatters (see below for upgrade hints)
- `doIf` functions on steps

Import paths for several classes under the `buildbot.status` package but which remain useful have changed. Most of these are now available as plugins (see above), but for the remainder, consult the source code.

BuildRequest Merging

Buildbot 0.9.x has replaced the old concept of request merging (`mergeRequests`) with a more flexible request-collapsing mechanism. See *`collapseRequests`* for more information.

Status Reporters

In fact, the whole `c['status']` configuration parameter is gone.

Many of the status listeners used in the status hierarchy in 0.8.x have been replaced with "reporters" that are available as buildbot plugins. However, note that not all status listeners have yet been ported. See the release notes for details.

Including the `"status"` key in the configuration object will cause a configuration error. All reporters should be included in `c['services']` as described in *Reporters*.

The available reporters as of 0.9.0 are

- *`MailNotifier`*
- *`IRC`*
- *`HttpStatusPush`*
- *`GerritStatusPush`*
- *`GitHubStatusPush`* (replaces `buildbot.status.github.GitHubStatus`)

See the reporter index for the full, current list.

A few notes on changes to the configuration of these reporters:

- *`MailNotifier`* argument `messageFormatter` should now be a `buildbot.reporters.message.MessageFormatter`, due to the removal of the status classes (see above), such formatters must be re-implemented using the Data API.
- *`MailNotifier`* argument `previousBuildGetter` is not supported anymore
- *`MailNotifier`* no longer forces SSL 3.0 when `useTls` is true.
- *`GerritStatusPush`* callbacks slightly changed signature, and include a master reference instead of a status reference.
- *`GitHubStatusPush`* now accepts a `context` parameter to be passed to the GitHub Status API.
- `buildbot.status.builder.Results` and the constants `buildbot.status.results.SUCCESS` should be imported from the *`buildbot.process.results`* module instead.

Steps

Buildbot-0.8.9 introduced "new-style steps", with an asynchronous `run` method. In the remaining 0.8.x releases, use of new-style and old-style steps were supported side-by-side. In 0.9.x, old-style steps are emulated using a collection of hacks to allow asynchronous calls to be called from synchronous code. This emulation is imperfect, and you are strongly encouraged to rewrite any custom steps as *New-Style Build Steps*.

Note that new-style steps now "push" their status when it changes, so the `describe` method no longer exists.

Identifiers

Many strings in Buildbot must now be identifiers. Identifiers are designed to fit easily and unambiguously into URLs, AMQP routes, and the like. An "identifier" is a nonempty unicode string of limited length, containing only ASCII alphanumeric characters along with - (dash) and _ (underscore), and not beginning with a digit

Unfortunately, many existing names do not fit this pattern.

The following fields are identifiers:

- worker name (50-character)
- builder name (20-character)
- step name (50-character)

Serving static files

Since version 0.9.0 Buildbot doesn't use and don't serve master's `public_html` directory. You need to use third-party HTTP server for serving static files.

Transition to "worker" terminology

Since version 0.9.0 of Buildbot "slave"-based terminology is deprecated in favor of "worker"-based terminology.

All identifiers, messages and documentation were updated to use "worker" instead of "slave". Old API names are still available, but deprecated.

For details about changed API and how to control generated warnings see *Transition to "worker" terminology*.

Other Config Settings

The default master.cfg file contains some new changes, which you should look over:

- `c['protocols'] = {'pb': {'port': 9989}}` (the default port used by the workers)
- Waterfall View: requires installation (`pip install buildbot-waterfall-view`) and configuration (`c['www'] = { ..., 'plugins': {'waterfall_view': {} }`).

Build History

There is no support for importing build history from 0.8.x (where the history was stored on-disk in pickle files) into 0.9.x (where it is stored in the database).

More Information

For minor changes not mentioned here, consult the release notes for the versions over which you are upgrading.

Buildbot-0.9.0 represents several years' work, and as such we may have missed potential migration issues. To find the latest "gotchas" and share with other users, see http://trac.buildbot.net/wiki/NineMigrationGuide.

Buildmaster Setup

Creating a buildmaster

As you learned earlier (*System Architecture*), the buildmaster runs on a central host (usually one that is publicly visible, so everybody can check on the status of the project), and controls all aspects of the buildbot system

You will probably wish to create a separate user account for the buildmaster, perhaps named `buildmaster`. Do not run the buildmaster as `root`!

You need to choose a directory for the buildmaster, called the `basedir`. This directory will be owned by the buildmaster. It will contain configuration, the database, and status information - including logfiles. On a large buildmaster this directory will see a lot of activity, so it should be on a disk with adequate space and speed.

Once you've picked a directory, use the `buildbot create-master` command to create the directory and populate it with startup files:

```
buildbot create-master -r basedir
```

You will need to create a *configuration file* before starting the buildmaster. Most of the rest of this manual is dedicated to explaining how to do this. A sample configuration file is placed in the working directory, named `master.cfg.sample`, which can be copied to `master.cfg` and edited to suit your purposes.

(Internal details: This command creates a file named `buildbot.tac` that contains all the state necessary to create the buildmaster. Twisted has a tool called `twistd` which can use this .tac file to create and launch a buildmaster instance. Twistd takes care of logging and daemonization (running the program in the background). `/usr/bin/buildbot` is a front end which runs *twistd* for you.)

Your master will need a database to store the various information about your builds, and its configuration. By default, the `sqlite3` backend will be used. This needs no configuration, neither extra software. All information will be stored in the file `state.sqlite`. Buildbot however supports multiple backends. See *Using A Database Server* for more options.

Buildmaster Options

This section lists options to the `create-master` command. You can also type `buildbot create-master --help` for an up-to-the-moment summary.

`--force`
: This option will allow to re-use an existing directory.

`--no-logrotate`
: This disables internal worker log management mechanism. With this option worker does not override the default logfile name and its behaviour giving a possibility to control those with command-line options of twistd daemon.

`--relocatable`
: This creates a "relocatable" `buildbot.tac`, which uses relative paths instead of absolute paths, so that the buildmaster directory can be moved about.

--config
The name of the configuration file to use. This configuration file need not reside in the buildmaster directory.

--log-size
This is the size in bytes when to rotate the Twisted log files. The default is 10MiB.

--log-count
This is the number of log rotations to keep around. You can either specify a number or `None` to keep all `twistd.log` files around. The default is 10.

--db
The database that the Buildmaster should use. Note that the same value must be added to the configuration file.

Upgrading an Existing Buildmaster

If you have just installed a new version of the Buildbot code, and you have buildmasters that were created using an older version, you'll need to upgrade these buildmasters before you can use them. The upgrade process adds and modifies files in the buildmaster's base directory to make it compatible with the new code.

```
buildbot upgrade-master basedir
```

This command will also scan your `master.cfg` file for incompatibilities (by loading it and printing any errors or deprecation warnings that occur). Each buildbot release tries to be compatible with configurations that worked cleanly (i.e. without deprecation warnings) on the previous release: any functions or classes that are to be removed will first be deprecated in a release, to give you a chance to start using the replacement.

The `upgrade-master` command is idempotent. It is safe to run it multiple times. After each upgrade of the buildbot code, you should use `upgrade-master` on all your buildmasters.

In general, Buildbot workers and masters can be upgraded independently, although some new features will not be available, depending on the master and worker versions.

Beyond this general information, read all of the sections below that apply to versions through which you are upgrading.

Version-specific Notes

Upgrading from Buildbot-0.8.x to Buildbot-0.9.x

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Upgrading a Buildmaster to Buildbot-0.7.6

The 0.7.6 release introduced the `public_html/` directory, which contains `index.html` and other files served by the `WebStatus` and `Waterfall` status displays. The `upgrade-master` command will create these files if they do not already exist. It will not modify existing copies, but it will write a new copy in e.g. `index.html.new` if the new version differs from the version that already exists.

Upgrading a Buildmaster to Buildbot-0.8.0

Buildbot-0.8.0 introduces a database backend, which is SQLite by default. The `upgrade-master` command will automatically create and populate this database with the changes the buildmaster has seen. Note that, as of this release, build history is *not* contained in the database, and is thus not migrated.

Upgrading into a non-SQLite database

If you are not using sqlite, you will need to add an entry into your `master.cfg` to reflect the database version you are using. The upgrade process does *not* edit your `master.cfg` for you. So something like:

```
# for using mysql:
c['db_url'] = 'mysql://bbuser:<password>@localhost/buildbot'
```

Once the parameter has been added, invoke `upgrade-master`. This will extract the DB url from your configuration file.

```
buildbot upgrade-master
```

See *Database Specification* for more options to specify a database.

Worker Setup

Creating a worker

Typically, you will be adding a worker to an existing buildmaster, to provide additional architecture coverage. The buildbot administrator will give you several pieces of information necessary to connect to the buildmaster. You should also be somewhat familiar with the project being tested, so you can troubleshoot build problems locally.

The buildbot exists to make sure that the project's stated `how to build it` process actually works. To this end, the worker should run in an environment just like that of your regular developers. Typically the project build process is documented somewhere (`README`, `INSTALL`, etc), in a document that should mention all library dependencies and contain a basic set of build instructions. This document will be useful as you configure the host and account in which the worker runs.

Here's a good checklist for setting up a worker:

1. Set up the account

   It is recommended (although not mandatory) to set up a separate user account for the worker. This account is frequently named `buildbot` or `worker`. This serves to isolate your personal working environment from that of the worker's, and helps to minimize the security threat posed by letting possibly-unknown contributors run arbitrary code on your system. The account should have a minimum of fancy init scripts.

2. Install the buildbot code

   Follow the instructions given earlier (*Installing the code*). If you use a separate worker account, and you didn't install the buildbot code to a shared location, then you will need to install it with `--home=~` for each account that needs it.

3. Set up the host

   Make sure the host can actually reach the buildmaster. Usually the buildmaster is running a status web-server on the same machine, so simply point your web browser at it and see if you can get there. Install whatever additional packages or libraries the project's INSTALL document advises. (or not: if your worker is supposed to make sure that building without optional libraries still works, then don't install those libraries.)

   Again, these libraries don't necessarily have to be installed to a site-wide shared location, but they must be available to your build process. Accomplishing this is usually very specific to the build process, so installing them to `/usr` or `/usr/local` is usually the best approach.

4. Test the build process

Follow the instructions in the `INSTALL` document, in the worker's account. Perform a full CVS (or whatever) checkout, configure, make, run tests, etc. Confirm that the build works without manual fussing. If it doesn't work when you do it by hand, it will be unlikely to work when the buildbot attempts to do it in an automated fashion.

5. Choose a base directory

   This should be somewhere in the worker's account, typically named after the project which is being tested. The worker will not touch any file outside of this directory. Something like `~/Buildbot` or `~/Workers/fooproject` is appropriate.

6. Get the buildmaster host/port, botname, and password

   When the buildbot admin configures the buildmaster to accept and use your worker, they will provide you with the following pieces of information:

   - your worker's name
   - the password assigned to your worker
   - the hostname and port number of the buildmaster, i.e. http://buildbot.example.org:8007

7. Create the worker

   Now run the 'worker' command as follows:

   ```
   buildbot-worker create-worker BASEDIR MASTERHOST:PORT
   WORKERNAME PASSWORD
   ```

   This will create the base directory and a collection of files inside, including the `buildbot.tac` file that contains all the information you passed to the **`buildbot`** command.

8. Fill in the hostinfo files

   When it first connects, the worker will send a few files up to the buildmaster which describe the host that it is running on. These files are presented on the web status display so that developers have more information to reproduce any test failures that are witnessed by the buildbot. There are sample files in the `info` subdirectory of the buildbot's base directory. You should edit these to correctly describe you and your host.

   BASEDIR`/info/admin` should contain your name and email address. This is the `worker admin address`, and will be visible from the build status page (so you may wish to munge it a bit if address-harvesting spambots are a concern).

   BASEDIR`/info/host` should be filled with a brief description of the host: OS, version, memory size, CPU speed, versions of relevant libraries installed, and finally the version of the buildbot code which is running the worker.

   The optional *BASEDIR*`/info/access_uri` can specify a URI which will connect a user to the machine. Many systems accept `ssh://hostname` URIs for this purpose.

   If you run many workers, you may want to create a single `~worker/info` file and share it among all the workers with symlinks.

Worker Options

There are a handful of options you might want to use when creating the worker with the `buildbot-worker create-worker <options> DIR <params>` command. You can type `buildbot-worker create-worker --help` for a summary. To use these, just include them on the `buildbot-worker create-worker` command line, like this

```
buildbot-worker create-worker --umask=022 ~/worker buildmaster.example.org:42012
↪{myworkername} {mypasswd}
```

--no-logrotate
: This disables internal worker log management mechanism. With this option worker does not override the default logfile name and its behaviour giving a possibility to control those with command-line options of twistd daemon.

--umask
: This is a string (generally an octal representation of an integer) which will cause the worker process' `umask` value to be set shortly after initialization. The `twistd` daemonization utility forces the umask to 077 at startup (which means that all files created by the worker or its child processes will be unreadable by any user other than the worker account). If you want build products to be readable by other accounts, you can add `--umask=022` to tell the worker to fix the umask after twistd clobbers it. If you want build products to be *writable* by other accounts too, use `--umask=000`, but this is likely to be a security problem.

--keepalive
: This is a number that indicates how frequently `keepalive` messages should be sent from the worker to the buildmaster, expressed in seconds. The default (600) causes a message to be sent to the buildmaster at least once every 10 minutes. To set this to a lower value, use e.g. `--keepalive=120`.

  If the worker is behind a NAT box or stateful firewall, these messages may help to keep the connection alive: some NAT boxes tend to forget about a connection if it has not been used in a while. When this happens, the buildmaster will think that the worker has disappeared, and builds will time out. Meanwhile the worker will not realize than anything is wrong.

--maxdelay
: This is a number that indicates the maximum amount of time the worker will wait between connection attempts, expressed in seconds. The default (300) causes the worker to wait at most 5 minutes before trying to connect to the buildmaster again.

--log-size
: This is the size in bytes when to rotate the Twisted log files.

--log-count
: This is the number of log rotations to keep around. You can either specify a number or `None` to keep all `twistd.log` files around. The default is 10.

--allow-shutdown
: Can also be passed directly to the Worker constructor in `buildbot.tac`. If set, it allows the worker to initiate a graceful shutdown, meaning that it will ask the master to shut down the worker when the current build, if any, is complete.

  Setting allow_shutdown to `file` will cause the worker to watch `shutdown.stamp` in basedir for updates to its mtime. When the mtime changes, the worker will request a graceful shutdown from the master. The file does not need to exist prior to starting the worker.

  Setting allow_shutdown to `signal` will set up a SIGHUP handler to start a graceful shutdown. When the signal is received, the worker will request a graceful shutdown from the master.

  The default value is `None`, in which case this feature will be disabled.

  Both master and worker must be at least version 0.8.3 for this feature to work.

Other Worker Configuration

unicode_encoding This represents the encoding that buildbot should use when converting unicode commandline arguments into byte strings in order to pass to the operating system when spawning new processes.

The default value is what Python's `sys.getfilesystemencoding` returns, which on Windows is 'mbcs', on Mac OSX is 'utf-8', and on Unix depends on your locale settings.

If you need a different encoding, this can be changed in your worker's `buildbot.tac` file by adding a `unicode_encoding` argument to the Worker constructor.

```
s = Worker(buildmaster_host, port, workername, passwd, basedir,
           keepalive, usepty, umask=umask, maxdelay=maxdelay,
           unicode_encoding='utf-8', allow_shutdown='signal')
```

Upgrading an Existing Worker

Version-specific Notes

During project lifetime worker has transitioned over few states:

1. Before Buildbot version 0.8.1 worker were integral part of `buildbot` package distribution.
2. Starting from Buildbot version 0.8.1 worker were extracted from `buildbot` package to `buildbot-slave` package.
3. Starting from Buildbot version 0.9.0 the `buildbot-slave` package was renamed to `buildbot-worker`.

Upgrading a Worker to buildbot-slave 0.8.1

Before Buildbot version 0.8.1, the Buildbot master and worker were part of the same distribution. As of version 0.8.1, the worker is a separate distribution.

As of this release, you will need to install `buildbot-slave` to run a worker.

Any automatic startup scripts that had run `buildbot start` for previous versions should be changed to run `buildslave start` instead.

If you are running a version later than 0.8.1, then you can skip the remainder of this section: the `upgrade-slave` command will take care of this. If you are upgrading directly to 0.8.1, read on.

The existing `buildbot.tac` for any workers running older versions will need to be edited or replaced. If the loss of cached worker state (e.g., for Source steps in copy mode) is not problematic, the easiest solution is to simply delete the worker directory and re-run `buildslave create-slave`.

If deleting the worker directory is problematic, the change to `buildbot.tac` is simple. On line 3, replace:

```
from buildbot.slave.bot import BuildSlave
```

with:

```
from buildslave.bot import BuildSlave
```

After this change, the worker should start as usual.

Upgrading from *0.8.1* to the latest `0.8.*` version of buildbot-slave

If you have just installed a new version of Buildbot-slave, you may need to take some steps to upgrade it. If you are upgrading to version 0.8.2 or later, you can run

```
buildslave upgrade-slave /path/to/worker/dir
```

Upgrading from the latest version of `buildbot-slave` to `buildbot-worker`

If the loss of cached worker state (e.g., for Source steps in copy mode) is not problematic, the easiest solution is to simply delete the worker directory and re-run `buildbot-worker create-worker`.

If deleting the worker directory is problematic, you can change `buildbot.tac` in the following way:

1. Replace:

   ```
   from buildslave.bot import BuildSlave
   ```

   with:

   ```
   from buildbot_worker.bot import Worker
   ```

2. Replace:

   ```
   application = service.Application('buildslave')
   ```

   with:

   ```
   application = service.Application('buildbot-worker')
   ```

3. Replace:

   ```
   s = BuildSlave(buildmaster_host, port, slavename, passwd, basedir,
                  keepalive, usepty, umask=umask, maxdelay=maxdelay,
                  numcpus=numcpus, allow_shutdown=allow_shutdown)
   ```

   with:

   ```
   s = Worker(buildmaster_host, port, slavename, passwd, basedir,
              keepalive, umask=umask, maxdelay=maxdelay,
              numcpus=numcpus, allow_shutdown=allow_shutdown)
   ```

See *Transition to "Worker" Terminology* for details of changes in version Buildbot `0.9.0`.

Next Steps

Launching the daemons

Both the buildmaster and the worker run as daemon programs. To launch them, pass the working directory to the **`buildbot`** and **`buildbot-worker`** commands, as appropriate:

```
# start a master
buildbot start [ BASEDIR ]
# start a worker
buildbot-worker start [ WORKER_BASEDIR ]
```

The *BASEDIR* is option and can be omitted if the current directory contains the buildbot configuration (the `buildbot.tac` file).

```
buildbot start
```

This command will start the daemon and then return, so normally it will not produce any output. To verify that the programs are indeed running, look for a pair of files named `twistd.log` and `twistd.pid` that should be created in the working directory. `twistd.pid` contains the process ID of the newly-spawned daemon.

When the worker connects to the buildmaster, new directories will start appearing in its base directory. The buildmaster tells the worker to create a directory for each Builder which will be using that worker. All build operations are performed within these directories: CVS checkouts, compiles, and tests.

Once you get everything running, you will want to arrange for the buildbot daemons to be started at boot time. One way is to use **cron**, by putting them in a `@reboot` crontab entry[1]

```
@reboot buildbot start [ BASEDIR ]
```

When you run **crontab** to set this up, remember to do it as the buildmaster or worker account! If you add this to your crontab when running as your regular account (or worse yet, root), then the daemon will run as the wrong user, quite possibly as one with more authority than you intended to provide.

It is important to remember that the environment provided to cron jobs and init scripts can be quite different that your normal runtime. There may be fewer environment variables specified, and the `PATH` may be shorter than usual. It is a good idea to test out this method of launching the worker by using a cron job with a time in the near future, with the same command, and then check `twistd.log` to make sure the worker actually started correctly. Common problems here are for `/usr/local` or `~/bin` to not be on your `PATH`, or for `PYTHONPATH` to not be set correctly. Sometimes `HOME` is messed up too.

Some distributions may include conveniences to make starting buildbot at boot time easy. For instance, with the default buildbot package in Debian-based distributions, you may only need to modify `/etc/default/buildbot` (see also `/etc/init.d/buildbot`, which reads the configuration in `/etc/default/buildbot`).

Buildbot also comes with its own init scripts that provide support for controlling multi-worker and multi-master setups (mostly because they are based on the init script from the Debian package). With a little modification these scripts can be used both on Debian and RHEL-based distributions and may thus prove helpful to package maintainers who are working on buildbot (or those that haven't yet split buildbot into master and worker packages).

```
# install as /etc/default/buildbot-worker
#         or /etc/sysconfig/buildbot-worker
worker/contrib/init-scripts/buildbot-worker.default

# install as /etc/default/buildmaster
#         or /etc/sysconfig/buildmaster
master/contrib/init-scripts/buildmaster.default

# install as /etc/init.d/buildbot-worker
worker/contrib/init-scripts/buildbot-worker.init.sh

# install as /etc/init.d/buildmaster
master/contrib/init-scripts/buildmaster.init.sh

# ... and tell sysvinit about them
chkconfig buildmaster reset
# ... or
update-rc.d buildmaster defaults
```

[1] This `@reboot` syntax is understood by Vixie cron, which is the flavor usually provided with Linux systems. Other unices may have a cron that doesn't understand `@reboot`

Logfiles

While a buildbot daemon runs, it emits text to a logfile, named `twistd.log`. A command like `tail -f twistd.log` is useful to watch the command output as it runs.

The buildmaster will announce any errors with its configuration file in the logfile, so it is a good idea to look at the log at startup time to check for any problems. Most buildmaster activities will cause lines to be added to the log.

Shutdown

To stop a buildmaster or worker manually, use:

```
buildbot stop [ BASEDIR ]
# or
buildbot-worker stop [ WORKER_BASEDIR ]
```

This simply looks for the `twistd.pid` file and kills whatever process is identified within.

At system shutdown, all processes are sent a `SIGKILL`. The buildmaster and worker will respond to this by shutting down normally.

The buildmaster will respond to a `SIGHUP` by re-reading its config file. Of course, this only works on Unix-like systems with signal support, and won't work on Windows. The following shortcut is available:

```
buildbot reconfig [ BASEDIR ]
```

When you update the Buildbot code to a new release, you will need to restart the buildmaster and/or worker before it can take advantage of the new code. You can do a `buildbot stop` *`BASEDIR`* and `buildbot start` *`BASEDIR`* in quick succession, or you can use the `restart` shortcut, which does both steps for you:

```
buildbot restart [ BASEDIR ]
```

Workers can similarly be restarted with:

```
buildbot-worker restart [ BASEDIR ]
```

There are certain configuration changes that are not handled cleanly by `buildbot reconfig`. If this occurs, `buildbot restart` is a more robust tool to fully switch over to the new configuration.

`buildbot restart` may also be used to start a stopped Buildbot instance. This behaviour is useful when writing scripts that stop, start and restart Buildbot.

A worker may also be gracefully shutdown from the web UI. This is useful to shutdown a worker without interrupting any current builds. The buildmaster will wait until the worker is finished all its current builds, and will then tell the worker to shutdown.

Concepts

This chapter defines some of the basic concepts that the Buildbot uses. You'll need to understand how the Buildbot sees the world to configure it properly.

Source Stamps

Source code comes from *repositories*, provided by version control systems. Repositories are generally identified by URLs, e.g., `git://github.com/buildbot/buildbot.git`.

In these days of distributed version control systems, the same *codebase* may appear in multiple repositories. For example, `https://github.com/mozilla/mozilla-central` and `http://hg.mozilla.org/mozilla-release` both contain the Firefox codebase, although not exactly the same code.

Many *projects* are built from multiple codebases. For example, a company may build several applications based on the same core library. The "app" codebase and the "core" codebase are in separate repositories, but are compiled together and constitute a single project. Changes to either codebase should cause a rebuild of the application.

Most version control systems define some sort of *revision* that can be used (sometimes in combination with a *branch*) to uniquely specify a particular version of the source code.

To build a project, Buildbot needs to know exactly which version of each codebase it should build. It uses a *source stamp* to do so for each codebase; the collection of sourcestamps required for a project is called a *source stamp set*.

Version Control Systems

Buildbot supports a significant number of version control systems, so it treats them abstractly.

For purposes of deciding when to perform builds, Buildbot's change sources monitor repositories, and represent any updates to those repositories as *changes*. These change sources fall broadly into two categories: pollers which periodically check the repository for updates; and hooks, where the repository is configured to notify Buildbot whenever an update occurs.

This concept does not map perfectly to every version control system. For example, for CVS Buildbot must guess that version updates made to multiple files within a short time represent a single change; for DVCS's like Git, Buildbot records a change when a commit is pushed to the monitored repository, not when it is initially committed. We assume that the `Changes` arrive at the master in the same order in which they are committed to the repository.

When it comes time to actually perform a build, a scheduler prepares a source stamp set, as described above, based on its configuration. When the build begins, one or more source steps use the information in the source stamp set to actually check out the source code, using the normal VCS commands.

Tree Stability

Changes tend to arrive at a buildmaster in bursts. In many cases, these bursts of changes are meant to be taken together. For example, a developer may have pushed multiple commits to a DVCS that comprise the same new feature or bugfix. To avoid trying to build every change, Buildbot supports the notion of *tree stability*, by waiting for a burst of changes to finish before starting to schedule builds. This is implemented as a timer, with builds not scheduled until no changes have occurred for the duration of the timer.

How Different VC Systems Specify Sources

For CVS, the static specifications are *repository* and *module*. In addition to those, each build uses a timestamp (or omits the timestamp to mean *the latest*) and *branch tag* (which defaults to `HEAD`). These parameters collectively specify a set of sources from which a build may be performed.

Subversion (http://subversion.tigris.org), combines the repository, module, and branch into a single *Subversion URL* parameter. Within that scope, source checkouts can be specified by a numeric *revision number* (a repository-wide monotonically-increasing marker, such that each transaction that changes the repository is indexed by a different revision number), or a revision timestamp. When branches are used, the repository and module form a static `baseURL`, while each build has a *revision number* and a *branch* (which defaults to a statically-specified `defaultBranch`). The `baseURL` and `branch` are simply concatenated together to derive the `repourl` to use for the checkout.

Perforce (http://www.perforce.com/) is similar. The server is specified through a `P4PORT` parameter. Module and branch are specified in a single depot path, and revisions are depot-wide. When branches are used, the `p4base` and `defaultBranch` are concatenated together to produce the depot path.

Bzr (http://bazaar-vcs.org) (which is a descendant of Arch/Bazaar, and is frequently referred to as "Bazaar") has the same sort of repository-vs-workspace model as Arch, but the repository data can either be stored inside the working directory or kept elsewhere (either on the same machine or on an entirely different machine). For the purposes of Buildbot (which never commits changes), the repository is specified with a URL and a revision number.

The most common way to obtain read-only access to a bzr tree is via HTTP, simply by making the repository visible through a web server like Apache. Bzr can also use FTP and SFTP servers, if the worker process has sufficient privileges to access them. Higher performance can be obtained by running a special Bazaar-specific server. None of these matter to the buildbot: the repository URL just has to match the kind of server being used. The `repoURL` argument provides the location of the repository.

Branches are expressed as subdirectories of the main central repository, which means that if branches are being used, the BZR step is given a `baseURL` and `defaultBranch` instead of getting the `repoURL` argument.

Darcs (http://darcs.net/) doesn't really have the notion of a single master repository. Nor does it really have branches. In Darcs, each working directory is also a repository, and there are operations to push and pull patches from one of these `repositories` to another. For the Buildbot's purposes, all you need to do is specify the URL of a repository that you want to build from. The worker will then pull the latest patches from that repository and build them. Multiple branches are implemented by using multiple repositories (possibly living on the same server).

Builders which use Darcs therefore have a static `repourl` which specifies the location of the repository. If branches are being used, the source Step is instead configured with a `baseURL` and a `defaultBranch`, and the two strings are simply concatenated together to obtain the repository's URL. Each build then has a specific branch which replaces `defaultBranch`, or just uses the default one. Instead of a revision number, each build can have a `context`, which is a string that records all the patches that are present in a given tree (this is the output of `darcs changes --context`, and is considerably less concise than, e.g. Subversion's revision number, but the patch-reordering flexibility of Darcs makes it impossible to provide a shorter useful specification).

Mercurial (https://www.mercurial-scm.org/) is like Darcs, in that each branch is stored in a separate repository. The `repourl`, `baseURL`, and `defaultBranch` arguments are all handled the same way as with Darcs. The *revision*, however, is the hash identifier returned by `hg identify`.

Git (http://git.or.cz/) also follows a decentralized model, and each repository can have several branches and tags. The source Step is configured with a static `repourl` which specifies the location of the repository. In addition, an optional `branch` parameter can be specified to check out code from a specific branch instead of the default *master* branch. The *revision* is specified as a SHA1 hash as returned by e.g. `git rev-parse`. No attempt is made to ensure that the specified revision is actually a subset of the specified branch.

Monotone (http://www.monotone.ca/) is another that follows a decentralized model where each repository can have several branches and tags. The source Step is configured with static `repourl` and `branch` parameters, which specifies the location of the repository and the branch to use. The *revision* is specified as a SHA1 hash as returned by e.g. `mtn automate select w:`. No attempt is made to ensure that the specified revision is actually a subset of the specified branch.

Changes

Who

Each `Change` has a `who` attribute, which specifies which developer is responsible for the change. This is a string which comes from a namespace controlled by the VC repository. Frequently this means it is a username on the host which runs the repository, but not all VC systems require this. Each `StatusNotifier` will map the `who` attribute into something appropriate for their particular means of communication: an email address, an IRC handle, etc.

This `who` attribute is also parsed and stored into Buildbot's database (see *User Objects*). Currently, only `who` attributes in Changes from `git` repositories are translated into user objects, but in the future all incoming Changes will have their `who` parsed and stored.

Files

It also has a list of `files`, which are just the tree-relative filenames of any files that were added, deleted, or modified for this `Change`. These filenames are used by the `fileIsImportant` function (in the scheduler) to decide whether it is worth triggering a new build or not, e.g. the function could use the following function to only run a build if a C file were checked in:

```
def has_C_files(change):
    for name in change.files:
        if name.endswith(".c"):
            return True
    return False
```

Certain `BuildSteps` can also use the list of changed files to run a more targeted series of tests, e.g. the `python_twisted.Trial` step can run just the unit tests that provide coverage for the modified .py files instead of running the full test suite.

Comments

The Change also has a `comments` attribute, which is a string containing any checkin comments.

Project

The `project` attribute of a change or source stamp describes the project to which it corresponds, as a short human-readable string. This is useful in cases where multiple independent projects are built on the same buildmaster. In such cases, it can be used to control which builds are scheduled for a given commit, and to limit status displays to only one project.

Repository

This attribute specifies the repository in which this change occurred. In the case of DVCS's, this information may be required to check out the committed source code. However, using the repository from a change has security risks: if Buildbot is configured to blindly trust this information, then it may easily be tricked into building arbitrary source code, potentially compromising the workers and the integrity of subsequent builds.

Codebase

This attribute specifies the codebase to which this change was made. As described *above*, multiple repositories may contain the same codebase. A change's codebase is usually determined by the *`codebaseGenerator`* configuration. By default the codebase is ''; this value is used automatically for single-codebase configurations.

Revision

Each Change can have a `revision` attribute, which describes how to get a tree with a specific state: a tree which includes this Change (and all that came before it) but none that come after it. If this information is unavailable, the `revision` attribute will be `None`. These revisions are provided by the `ChangeSource`.

Revisions are always strings.

CVS `revision` is the seconds since the epoch as an integer.

SVN `revision` is the revision number

Darcs `revision` is a large string, the output of **`darcs changes --context`**

Mercurial `revision` is a short string (a hash ID), the output of **`hg identify`**

P4 `revision` is the transaction number

Git `revision` is a short string (a SHA1 hash), the output of e.g. **`git rev-parse`**

Branches

The Change might also have a `branch` attribute. This indicates that all of the Change's files are in the same named branch. The schedulers get to decide whether the branch should be built or not.

For VC systems like CVS, Git and Monotone the `branch` name is unrelated to the filename. (That is, the branch name and the filename inhabit unrelated namespaces.) For SVN, branches are expressed as subdirectories of the repository, so the file's `repourl` is a combination of some base URL, the branch name, and the filename within the branch. (In a sense, the branch name and the filename inhabit the same namespace.) Darcs branches are subdirectories of a base URL just like SVN. Mercurial branches are the same as Darcs.

CVS branch='warner-newfeature', files=['src/foo.c']

SVN branch='branches/warner-newfeature', files=['src/foo.c']

Darcs branch='warner-newfeature', files=['src/foo.c']

Mercurial branch='warner-newfeature', files=['src/foo.c']

Git branch='warner-newfeature', files=['src/foo.c']

Monotone branch='warner-newfeature', files=['src/foo.c']

Change Properties

A Change may have one or more properties attached to it, usually specified through the Force Build form or `sendchange`. Properties are discussed in detail in the *Build Properties* section.

Scheduling Builds

Each Buildmaster has a set of scheduler objects, each of which gets a copy of every incoming `Change`. The Schedulers are responsible for deciding when `Build`s should be run. Some Buildbot installations might have a single scheduler, while others may have several, each for a different purpose.

For example, a *quick* scheduler might exist to give immediate feedback to developers, hoping to catch obvious problems in the code that can be detected quickly. These typically do not run the full test suite, nor do they run on a wide variety of platforms. They also usually do a VC update rather than performing a brand-new checkout each time.

A separate *full* scheduler might run more comprehensive tests, to catch more subtle problems. configured to run after the quick scheduler, to give developers time to commit fixes to bugs caught by the quick scheduler before running the comprehensive tests. This scheduler would also feed multiple `Builder`s.

Many schedulers can be configured to wait a while after seeing a source-code change - this is the *tree stable timer*. The timer allows multiple commits to be "batched" together. This is particularly useful in distributed version control systems, where a developer may push a long sequence of changes all at once. To save resources, it's often desirable only to test the most recent change.

Schedulers can also filter out the changes they are interested in, based on a number of criteria. For example, a scheduler that only builds documentation might skip any changes that do not affect the documentation. Schedulers can also filter on the branch to which a commit was made.

There is some support for configuring dependencies between builds - for example, you may want to build packages only for revisions which pass all of the unit tests. This support is under active development in Buildbot, and is referred to as "build coordination".

Periodic builds (those which are run every N seconds rather than after new Changes arrive) are triggered by a special *Periodic* scheduler.

Each scheduler creates and submits BuildSet objects to the BuildMaster, which is then responsible for making sure the individual BuildRequests are delivered to the target Builders.

Scheduler instances are activated by placing them in the *schedulers* list in the buildmaster config file. Each scheduler must have a unique name.

BuildSets

A BuildSet is the name given to a set of Builds that all compile/test the same version of the tree on multiple Builders. In general, all these component Builds will perform the same sequence of Steps, using the same source code, but on different platforms or against a different set of libraries.

The BuildSet is tracked as a single unit, which fails if any of the component Builds have failed, and therefore can succeed only if *all* of the component Builds have succeeded. There are two kinds of status notification messages that can be emitted for a BuildSet: the firstFailure type (which fires as soon as we know the BuildSet will fail), and the Finished type (which fires once the BuildSet has completely finished, regardless of whether the overall set passed or failed).

A BuildSet is created with set of one or more *source stamp* tuples of (branch, revision, changes, patch), some of which may be None, and a list of Builders on which it is to be run. They are then given to the BuildMaster, which is responsible for creating a separate BuildRequest for each Builder.

There are a couple of different likely values for the SourceStamp:

(revision=None, changes=*CHANGES*, patch=None) This is a SourceStamp used when a series of Changes have triggered a build. The VC step will attempt to check out a tree that contains *CHANGES* (and any changes that occurred before *CHANGES*, but not any that occurred after them.)

(revision=None, changes=None, patch=None) This builds the most recent code on the default branch. This is the sort of SourceStamp that would be used on a Build that was triggered by a user request, or a *Periodic* scheduler. It is also possible to configure the VC Source Step to always check out the latest sources rather than paying attention to the Changes in the SourceStamp, which will result in same behavior as this.

(branch=*BRANCH*, revision=None, changes=None, patch=None) This builds the most recent code on the given *BRANCH*. Again, this is generally triggered by a user request or a *Periodic* scheduler.

(revision=*REV*, changes=None, patch=(*LEVEL*, *DIFF*, *SUBDIR_ROOT*)) This checks out the tree at the given revision *REV*, then applies a patch (using patch -pLEVEL <DIFF) from inside the relative directory *SUBDIR_ROOT*. Item *SUBDIR_ROOT* is optional and defaults to the builder working directory. The *try* command creates this kind of SourceStamp. If patch is None, the patching step is bypassed.

The buildmaster is responsible for turning the BuildSet into a set of BuildRequest objects and queueing them on the appropriate Builders.

BuildRequests

A BuildRequest is a request to build a specific set of source code (specified by one ore more source stamps) on a single Builder. Each Builder runs the BuildRequest as soon as it can (i.e. when an associated worker becomes free). BuildRequests are prioritized from oldest to newest, so when a worker becomes free, the Builder with the oldest BuildRequest is run.

The BuildRequest contains one SourceStamp specification per codebase. The actual process of running the build (the series of Steps that will be executed) is implemented by the Build object. In the future this might be changed, to have the Build define *what* gets built, and a separate BuildProcess (provided by the Builder) to define *how* it gets built.

The BuildRequest may be mergeable with other compatible BuildRequests. Builds that are triggered by incoming Changes will generally be mergeable. Builds that are triggered by user requests are generally not, unless they are multiple requests to build the *latest sources* of the same branch. A merge of buildrequests is performed per codebase, thus on changes having the same codebase.

Builders

The Buildmaster runs a collection of Builders, each of which handles a single type of build (e.g. full versus quick), on one or more workers. Builders serve as a kind of queue for a particular type of build. Each Builder gets a separate column in the waterfall display. In general, each Builder runs independently (although various kinds of interlocks can cause one Builder to have an effect on another).

Each builder is a long-lived object which controls a sequence of Builds. Each Builder is created when the config file is first parsed, and lives forever (or rather until it is removed from the config file). It mediates the connections to the workers that do all the work, and is responsible for creating the Build objects - *Builds*.

Each builder gets a unique name, and the path name of a directory where it gets to do all its work (there is a buildmaster-side directory for keeping status information, as well as a worker-side directory where the actual checkout/compile/test commands are executed).

Build Factories

A builder also has a BuildFactory, which is responsible for creating new Build instances: because the Build instance is what actually performs each build, choosing the BuildFactory is the way to specify what happens each time a build is done (*Builds*).

Workers

Each builder is associated with one of more Workers. A builder which is used to perform Mac OS X builds (as opposed to Linux or Solaris builds) should naturally be associated with a Mac worker.

If multiple workers are available for any given builder, you will have some measure of redundancy: in case one worker goes offline, the others can still keep the Builder working. In addition, multiple workers will allow multiple simultaneous builds for the same Builder, which might be useful if you have a lot of forced or try builds taking place.

If you use this feature, it is important to make sure that the workers are all, in fact, capable of running the given build. The worker hosts should be configured similarly, otherwise you will spend a lot of time trying (unsuccessfully) to reproduce a failure that only occurs on some of the workers and not the others. Different platforms, operating systems, versions of major programs or libraries, all these things mean you should use separate Builders.

Builds

A build is a single compile or test run of a particular version of the source code, and is comprised of a series of steps. It is ultimately up to you what constitutes a build, but for compiled software it is generally the checkout, configure, make, and make check sequence. For interpreted projects like Python modules, a build is generally a checkout followed by an invocation of the bundled test suite.

A `BuildFactory` describes the steps a build will perform. The builder which starts a build uses its configured build factory to determine the build's steps.

Users

Buildbot has a somewhat limited awareness of *users*. It assumes the world consists of a set of developers, each of whom can be described by a couple of simple attributes. These developers make changes to the source code, causing builds which may succeed or fail.

Users also may have different levels of authorization when issuing Buildbot commands, such as forcing a build from the web interface or from an IRC channel.

Each developer is primarily known through the source control system. Each `Change` object that arrives is tagged with a `who` field that typically gives the account name (on the repository machine) of the user responsible for that change. This string is displayed on the HTML status pages and in each `Build`'s *blamelist*.

To do more with the User than just refer to them, this username needs to be mapped into an address of some sort. The responsibility for this mapping is left up to the status module which needs the address. In the future, the responsibility for managing users will be transferred to User Objects.

The `who` fields in `git` Changes are used to create *User Objects*, which allows for more control and flexibility in how Buildbot manages users.

User Objects

User Objects allow Buildbot to better manage users throughout its various interactions with users (see *Change Sources* and *Reporters*). The User Objects are stored in the Buildbot database and correlate the various attributes that a user might have: irc, Git, etc.

Changes

Incoming Changes all have a `who` attribute attached to them that specifies which developer is responsible for that Change. When a Change is first rendered, the `who` attribute is parsed and added to the database if it doesn't exist or checked against an existing user. The `who` attribute is formatted in different ways depending on the version control system that the Change came from.

`git` `who` attributes take the form `Full Name <Email>`.

`svn` `who` attributes are of the form `Username`.

`hg` `who` attributes are free-form strings, but usually adhere to similar conventions as `git` attributes (`Full Name <Email>`).

`cvs` `who` attributes are of the form `Username`.

`darcs` `who` attributes contain an `Email` and may also include a `Full Name` like `git` attributes.

`bzr` `who` attributes are free-form strings like `hg`, and can include a `Username`, `Email`, and/or `Full Name`.

Tools

For managing users manually, use the `buildbot user` command, which allows you to add, remove, update, and show various attributes of users in the Buildbot database (see *Command-line Tool*).

Uses

Correlating the various bits and pieces that Buildbot views as users also means that one attribute of a user can be translated into another. This provides a more complete view of users throughout Buildbot.

One such use is being able to find email addresses based on a set of Builds to notify users through the `MailNotifier`. This process is explained more clearly in *Email Addresses*.

Another way to utilize *User Objects* is through *UsersAuth* for web authentication. To use *UsersAuth*, you need to set a *bb_username* and *bb_password* via the `buildbot user` command line tool to check against. The password will be encrypted before storing in the database along with other user attributes.

Doing Things With Users

Each change has a single user who is responsible for it. Most builds have a set of changes: the build generally represents the first time these changes have been built and tested by the Buildbot. The build has a *blamelist* that is the union of the users responsible for all the build's changes. If the build was created by a *Try Schedulers* this list will include the submitter of the try job, if known.

The build provides a list of users who are interested in the build – the *interested users*. Usually this is equal to the blamelist, but may also be expanded, e.g., to include the current build sherrif or a module's maintainer.

If desired, the buildbot can notify the interested users until the problem is resolved.

Email Addresses

The *`MailNotifier`* is a status target which can send email about the results of each build. It accepts a static list of email addresses to which each message should be delivered, but it can also be configured to send mail to the `Build`'s Interested Users. To do this, it needs a way to convert User names into email addresses.

For many VC systems, the User Name is actually an account name on the system which hosts the repository. As such, turning the name into an email address is a simple matter of appending `@repositoryhost.com`. Some projects use other kinds of mappings (for example the preferred email address may be at `project.org` despite the repository host being named `cvs.project.org`), and some VC systems have full separation between the concept of a user and that of an account on the repository host (like Perforce). Some systems (like Git) put a full contact email address in every change.

To convert these names to addresses, the `MailNotifier` uses an `EmailLookup` object. This provides a `getAddress` method which accepts a name and (eventually) returns an address. The default `MailNotifier` module provides an `EmailLookup` which simply appends a static string, configurable when the notifier is created. To create more complex behaviors (perhaps using an LDAP lookup, or using `finger` on a central host to determine a preferred address for the developer), provide a different object as the `lookup` argument.

If an EmailLookup object isn't given to the MailNotifier, the MailNotifier will try to find emails through *User Objects*. This will work the same as if an EmailLookup object was used if every user in the Build's Interested Users list has an email in the database for them. If a user whose change led to a Build doesn't have an email attribute, that user will not receive an email. If `extraRecipients` is given, those users are still sent mail when the EmailLookup object is not specified.

In the future, when the Problem mechanism has been set up, the Buildbot will need to send mail to arbitrary Users. It will do this by locating a `MailNotifier`-like object among all the buildmaster's status targets, and asking it to send messages to various Users. This means the User-to-address mapping only has to be set up once, in your `MailNotifier`, and every email message the buildbot emits will take advantage of it.

IRC Nicknames

Like `MailNotifier`, the `buildbot.status.words.IRC` class provides a status target which can announce the results of each build. It also provides an interactive interface by responding to online queries posted in the channel or sent as private messages.

In the future, the buildbot can be configured map User names to IRC nicknames, to watch for the recent presence of these nicknames, and to deliver build status messages to the interested parties. Like `MailNotifier` does for email addresses, the `IRC` object will have an `IRCLookup` which is responsible for nicknames. The mapping can be set up statically, or it can be updated by online users themselves (by claiming a username with some kind of `buildbot: i am user warner` commands).

Once the mapping is established, the rest of the buildbot can ask the `IRC` object to send messages to various users. It can report on the likelihood that the user saw the given message (based upon how long the user has been inactive on the channel), which might prompt the Problem Hassler logic to send them an email message instead.

These operations and authentication of commands issued by particular nicknames will be implemented in *User Objects*.

Build Properties

Each build has a set of *Build Properties*, which can be used by its build steps to modify their actions. These properties, in the form of key-value pairs, provide a general framework for dynamically altering the behavior of a build based on its circumstances.

Properties form a simple kind of variable in a build. Some properties are set when the build starts, and properties can be changed as a build progresses – properties set or changed in one step may be accessed in subsequent steps. Property values can be numbers, strings, lists, or dictionaries - basically, anything that can be represented in JSON.

Properties are very flexible, and can be used to implement all manner of functionality. Here are some examples:

Most Source steps record the revision that they checked out in the `got_revision` property. A later step could use this property to specify the name of a fully-built tarball, dropped in an easily-accessible directory for later testing.

Note: In builds with more than one codebase, the `got_revision` property is a dictionary, keyed by codebase.

Some projects want to perform nightly builds as well as building in response to committed changes. Such a project would run two schedulers, both pointing to the same set of builders, but could provide an `is_nightly` property so that steps can distinguish the nightly builds, perhaps to run more resource-intensive tests.

Some projects have different build processes on different systems. Rather than create a build factory for each worker, the steps can use worker properties to identify the unique aspects of each worker and adapt the build process dynamically.

Multiple-Codebase Builds

What if an end-product is composed of code from several codebases? Changes may arrive from different repositories within the tree-stable-timer period. Buildbot will not only use the source-trees that contain changes but also needs the remaining source-trees to build the complete product.

For this reason a *Scheduler* can be configured to base a build on a set of several source-trees that can (partly) be overridden by the information from incoming `Changes`.

As described *above*, the source for each codebase is identified by a source stamp, containing its repository, branch and revision. A full build set will specify a source stamp set describing the source to use for each codebase.

Configuring all of this takes a coordinated approach. A complete multiple repository configuration consists of:

a *codebase generator*

> Every relevant change arriving from a VC must contain a codebase. This is done by a *codebaseGenerator* that is defined in the configuration. Most generators examine the repository of a change to determine its codebase, using project-specific rules.

some *schedulers*

> Each *scheduler* has to be configured with a set of all required `codebases` to build a product. These codebases indicate the set of required source-trees. In order for the scheduler to be able to produce a complete set for each build, the configuration can give a default repository, branch, and revision for each codebase. When a scheduler must generate a source stamp for a codebase that has received no changes, it applies these default values.

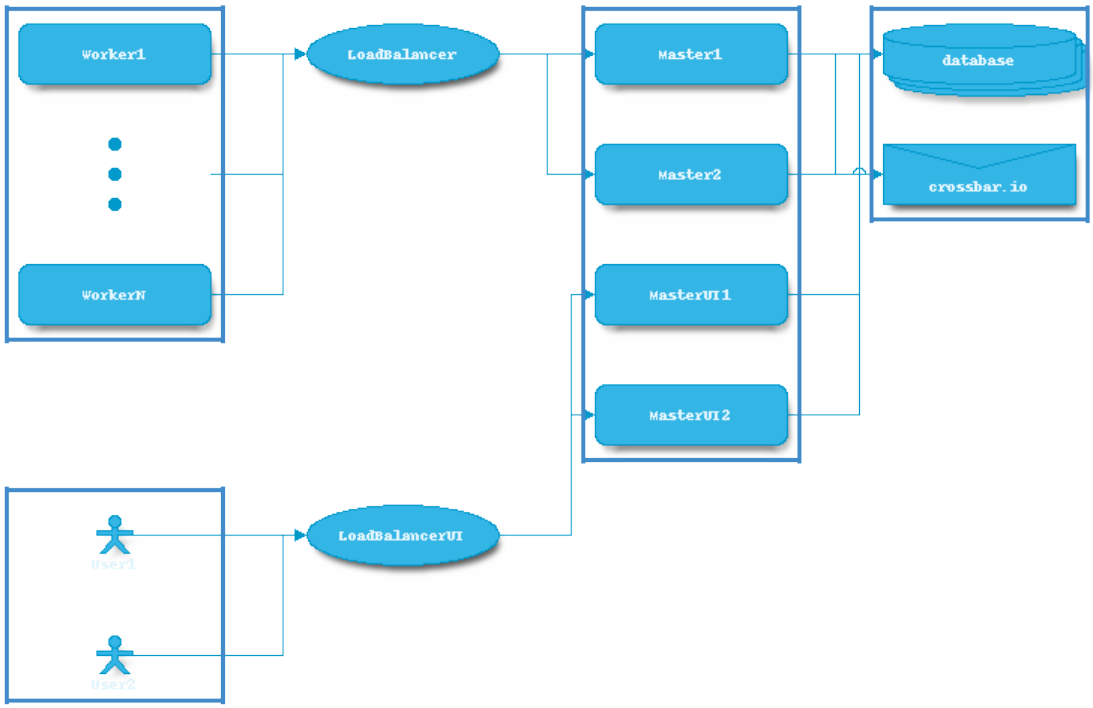
multiple *source steps* - one for each codebase

> A *Builders*'s build factory must include a *source step* for each codebase. Each of the source steps has a `codebase` attribute which is used to select an appropriate source stamp from the source stamp set for a build. This information comes from the arrived changes or from the scheduler's configured default values.
>
> ---
>
> **Note:** Each *source step* has to have its own `workdir` set in order for the checkout to be done for each codebase in its own directory.
>
> ---
>
> ---
>
> **Note:** Ensure you specify the codebase within your source step's Interpolate() calls (ex. `http://.../svn/%(src:codebase:branch)s)`. See *Interpolate* for details.
>
> ---

Warning: Defining a *codebaseGenerator* that returns non-empty (not `''`) codebases will change the behavior of all the schedulers.

Multimaster

Warning: Buildbot Multimaster is considered experimental. There are still some companies using it in production. Don't hesitate to use the mailing lists to share your experience.



Buildbot supports interconnection of several masters. This has to be done through a multi-master enabled message queue backend. As of now the only one supported is wamp and crossbar.io. see *wamp*

There are then several strategy for introducing multimaster in your buildbot infra. A simple way to say it is by adding the concept of symmetrics and asymmetrics multimaster (like there is SMP and AMP for multi core CPUs)

Symmetric multimaster is when each master share the exact same configuration. They run the same builders, same schedulers, same everything, the only difference is that workers are connected evenly between the masters (by any means (e.g. DNS load balancing, etc)) Symmetric multimaster is good to use to scale buildbot horizontally.

Asymmetric multimaster is when each master have different configuration. Each master may have a specific responsibility (e.g schedulers, set of builder, UI). This was more how you did in 0.8, also because of its own technical limitations. A nice feature of asymmetric multimaster is that you can have the UI only handled by some masters.

Separating the UI from the controlling will greatly help in the performance of the UI, because badly written BuildSteps?? can stall the reactor for several seconds.

The fanciest configuration would probably be a symmetric configuration for everything but the UI. You would scale the number of UI master according to your number of UI users, and scale the number of engine masters to the number of workers.

Depending on your workload and size of master host, it is probably a good idea to start thinking of multimaster starting from a hundred workers connected.

Multimaster can also be used for high availability, and seamless upgrade of configuration code. Complex configuration indeed requires sometimes to restart the master to reload custom steps or code, or just to upgrade the upstream buildbot version.

In this case, you will implement following procedure:

- Start new master(s) with new code and configuration.
- Send a graceful shutdown to the old master(s).

- New master(s) will start taking the new jobs, while old master(s) will just finish managing the running builds.
- As an old master is finishing the running builds, it will drop the connections from the workers, who will then reconnect automatically, and by the mean of load balancer will get connected to a new master to run new jobs.

As buildbot nine has been designed to allow such procedure, it has not been implemented in production yet as we know. There is probably a new REST api needed in order to graceful shutdown a master, and the details of gracefully dropping the connection to the workers to be sorted out.

Secret Management

Requirements

Buildbot steps might need secrets to execute their actions. Secrets are used to execute commands or to create authenticated network connections. Secrets may be a SSH key, a password, or a file content like a wgetrc file or a public SSH key. To preserve confidentiality, the secrets values must not be printed or logged in the twisted or steps logs. Secrets must not be stored in the Buildbot configuration (master.cfg), as the source code is usually shared in SCM like git.

How to use Buildbot Secret Management

Secrets and providers

Buildbot implements several providers for secrets retrieval:

- File system based: secrets are written in a file. This is a simple solution for example when secrets are managed by config management system like Ansible Vault.
- Third party backend based: secrets are stored by a specialized software. These solution are usually more secured.

Secrets providers are configured if needed in the master configuration. Multiple providers can be configured at once. The secret manager is a Buildbot service. The secret manager returns the specific provider results related to the providers registered in the configuration.

How to use secrets in Buildbot

The following example shows a basic usage of secrets in Buildbot.

```
# First we declare that the secrets are stored in a directory of the filesystem
# each file contain one secret identified by the filename
c['secretsProviders'] = [util.SecretInFile(directory="/path/toSecretsFiles"]

# then in a buildfactory:

# use a secret on a shell command via Interpolate
f1.addStep(ShellCommand(Interpolate("wget -u user -p %{secrets:userpassword}s %
↪{prop:urltofetch}s")))
```

Secrets are also interpolated in the build like properties are, and will be used in a command line for example.

Secrets storages

SecretInFile

```
c['secretsProviders'] = [util.SecretInFile(directory="/path/toSecretsFiles"]
```

In the passed directory, every file contains a secret identified by the filename.

e.g: a file `user` contains the text `pa$$w0rd`.

SecretInVault

```
c['secretsProviders'] = [util.SecretInVault(
                         vaultToken=open('VAULT_TOKEN').read(),
                         vaultServer="http://localhost:8200"
)]
```

Vault secures, stores, and tightly controls access to secrets. Vault presents a unified API to access multiple backends. To be authenticated in Vault, Buildbot need to send a token to the vault server. The token is generated when the Vault instance is initialized for the first time.

In the master configuration, the Vault provider is instantiated through the Buildbot service manager as a secret provider with the the Vault server address and the Vault token. The provider SecretInVault allows Buildbot to read secrets in Vault. For more informations about Vault please visit: Vault: https://www.vaultproject.io/

How to configure a Vault instance

Vault being a very generic system, it can be complex to install for the first time. Here is a simple tutorial to install the minimal Vault for use with Buildbot.

Use Docker to install Vault

A Docker image is available to help users installing Vault. Without any arguments, the command launches a Docker Vault developer instance, easy to use and test the functions. The developer version is already initialized and unsealed. To launch a Vault server please refer to the VaultDocker (https://hub.docker.com/_/vault/) documentation:

In a shell:

```
docker run vault
```

Starting the vault instance

Once the Docker image is created, launch a shell terminal on the Docker image:

```
docker exec -i -t ``docker_vault_image_name`` /bin/sh
```

Then, export the environment variable VAULT_ADDR needed to init Vault.

```
export VAULT_ADDR='vault.server.adress'
```

Writing secrets

By default Vault is initialized with a mount named secret. To add a new secret:

```
vault write secret/new_secret_key value=new_secret_value
```

Configuration

The following sections describe the configuration of the various Buildbot components. The information available here is sufficient to create basic build and test configurations, and does not assume great familiarity with Python.

In more advanced Buildbot configurations, Buildbot acts as a framework for a continuous-integration application. The next section, *Customization*, describes this approach, with frequent references into the *development documentation*.

Configuring Buildbot

The buildbot's behavior is defined by the *config file*, which normally lives in the `master.cfg` file in the buildmaster's base directory (but this can be changed with an option to the **`buildbot create-master`** command). This file completely specifies which `Builders` are to be run, which workers they should use, how `Changes` should be tracked, and where the status information is to be sent. The buildmaster's `buildbot.tac` file names the base directory; everything else comes from the config file.

A sample config file was installed for you when you created the buildmaster, but you will need to edit it before your buildbot will do anything useful.

This chapter gives an overview of the format of this file and the various sections in it. You will need to read the later chapters to understand how to fill in each section properly.

Config File Format

The config file is, fundamentally, just a piece of Python code which defines a dictionary named `BuildmasterConfig`, with a number of keys that are treated specially. You don't need to know Python to do basic configuration, though, you can just copy the syntax of the sample file. If you *are* comfortable writing Python code, however, you can use all the power of a full programming language to achieve more complicated configurations.

The `BuildmasterConfig` name is the only one which matters: all other names defined during the execution of the file are discarded. When parsing the config file, the Buildmaster generally compares the old configuration with the new one and performs the minimum set of actions necessary to bring the buildbot up to date: `Builders` which are not changed are left untouched, and `Builders` which are modified get to keep their old event history.

The beginning of the `master.cfg` file typically starts with something like:

```
BuildmasterConfig = c = {}
```

Therefore a config key like *`change_source`* will usually appear in `master.cfg` as `c['change_source']`.

See `cfg` for a full list of `BuildMasterConfig` keys.

Basic Python Syntax

The master configuration file is interpreted as Python, allowing the full flexibility of the language. For the configurations described in this section, a detailed knowledge of Python is not required, but the basic syntax is easily described.

Python comments start with a hash character #, tuples are defined with `(parenthesis, pairs)`, and lists (arrays) are defined with `[square, brackets]`. Tuples and lists are mostly interchangeable. Dictionaries (data structures which map *keys* to *values*) are defined with curly braces: `{'key1': value1, 'key2': value2}`. Function calls (and object instantiation) can use named parameters, like `steps.ShellCommand(command=["trial", "pyflakes"])`.

The config file starts with a series of `import` statements, which make various kinds of `Steps` and `Status` targets available for later use. The main `BuildmasterConfig` dictionary is created, then it is populated with a variety of keys, described section-by-section in subsequent chapters.

Predefined Config File Symbols

The following symbols are automatically available for use in the configuration file.

`basedir` the base directory for the buildmaster. This string has not been expanded, so it may start with a tilde. It needs to be expanded before use. The config file is located in:

```
os.path.expanduser(os.path.join(basedir, 'master.cfg'))
```

`__file__` the absolute path of the config file. The config file's directory is located in `os.path.dirname(__file__)`.

Testing the Config File

To verify that the config file is well-formed and contains no deprecated or invalid elements, use the `checkconfig` command, passing it either a master directory or a config file.

```
% buildbot checkconfig master.cfg
Config file is good!
# or
% buildbot checkconfig /tmp/masterdir
Config file is good!
```

If the config file has deprecated features (perhaps because you've upgraded the buildmaster and need to update the config file to match), they will be announced by checkconfig. In this case, the config file will work, but you should really remove the deprecated items and use the recommended replacements instead:

```
% buildbot checkconfig master.cfg
/usr/lib/python2.4/site-packages/buildbot/master.py:559: DeprecationWarning: c[
↪'sources'] is
deprecated as of 0.7.6 and will be removed by 0.8.0 . Please use c['change_source']␣
↪instead.
Config file is good!
```

If you have errors in your configuration file, checkconfig will let you know:

```
% buildbot checkconfig master.cfg
Configuration Errors:
c['workers'] must be a list of Worker instances
no workers are configured
builder 'smoketest' uses unknown workers 'linux-002'
```

If the config file is simply broken, that will be caught too:

```
% buildbot checkconfig master.cfg
error while parsing config file:
Traceback (most recent call last):
File "/home/buildbot/master/bin/buildbot", line 4, in <module>
    runner.run()
File "/home/buildbot/master/buildbot/scripts/runner.py", line 1358, in run
    if not doCheckConfig(so):
File "/home/buildbot/master/buildbot/scripts/runner.py", line 1079, in doCheckConfig
    return cl.load(quiet=quiet)
File "/home/buildbot/master/buildbot/scripts/checkconfig.py", line 29, in load
    self.basedir, self.configFileName)
--- <exception caught here> ---
File "/home/buildbot/master/buildbot/config.py", line 147, in loadConfig
    exec f in localDict
exceptions.SyntaxError: invalid syntax (master.cfg, line 52)
Configuration Errors:
error while parsing config file: invalid syntax (master.cfg, line 52) (traceback in␣
↪logfile)
```

Loading the Config File

The config file is only read at specific points in time. It is first read when the buildmaster is launched.

Note: If the configuration is invalid, the master will display the errors in the console output, but will not exit.

Reloading the Config File (reconfig)

If you are on the system hosting the buildmaster, you can send a `SIGHUP` signal to it: the **`buildbot`** tool has a shortcut for this:

```
buildbot reconfig BASEDIR
```

This command will show you all of the lines from `twistd.log` that relate to the reconfiguration. If there are any problems during the config-file reload, they will be displayed in these lines.

When reloading the config file, the buildmaster will endeavor to change as little as possible about the running system. For example, although old status targets may be shut down and new ones started up, any status targets that were not changed since the last time the config file was read will be left running and untouched. Likewise any `Builders` which have not been changed will be left running. If a `Builder` is modified (say, the build process is changed) while a `Build` is currently running, that `Build` will keep running with the old process until it completes. Any previously queued `Build`s (or `Build`s which get queued after the reconfig) will use the new process.

Warning: Buildbot's reconfiguration system is fragile for a few difficult-to-fix reasons:

- Any modules imported by the configuration file are not automatically reloaded. Python modules such as http://pypi.python.org/pypi/lazy-reload may help here, but reloading modules is fraught with subtleties and difficult-to-decipher failure cases.
- During the reconfiguration, active internal objects are divorced from the service hierarchy, leading to tracebacks in the web interface and other components. These are ordinarily transient, but with HTTP connection caching (either by the browser or an intervening proxy) they can last for a long time.

- If the new configuration file is invalid, it is possible for Buildbot's internal state to be corrupted, leading to undefined results. When this occurs, it is best to restart the master.
- For more advanced configurations, it is impossible for Buildbot to tell if the configuration for a `Builder` or `Scheduler` has changed, and thus the `Builder` or `Scheduler` will always be reloaded. This occurs most commonly when a callable is passed as a configuration parameter.

The bbproto project (at https://github.com/dabrahams/bbproto) may help to construct large (multi-file) configurations which can be effectively reloaded and reconfigured.

Global Configuration

The keys in this section affect the operations of the buildmaster globally.

Database Specification

Buildbot requires a connection to a database to maintain certain state information, such as tracking pending build requests. In the default configuration Buildbot uses a file-based SQLite database, stored in the `state.sqlite` file of the master's base directory. Override this configuration with the *db_url* parameter.

Buildbot accepts a database configuration in a dictionary named `db`. All keys are optional:

```
c['db'] = {
    'db_url' : 'sqlite:///state.sqlite',
}
```

The `db_url` key indicates the database engine to use. The format of this parameter is completely documented at http://www.sqlalchemy.org/docs/dialects/, but is generally of the form:

```
"driver://[username:password@]host:port/database[?args]"
```

These parameters can be specified directly in the configuration dictionary, as `c['db_url']` and `c['db_poll_interval']`, although this method is deprecated.

The following sections give additional information for particular database backends:

SQLite

For sqlite databases, since there is no host and port, relative paths are specified with `sqlite:///` and absolute paths with `sqlite:////`. Examples:

```
c['db_url'] = "sqlite:///state.sqlite"
```

SQLite requires no special configuration.

MySQL

```
c['db_url'] = "mysql://user:pass@example.com/database_name?max_idle=300"
```

The `max_idle` argument for MySQL connections is unique to Buildbot, and should be set to something less than the `wait_timeout` configured for your server. This controls the SQLAlchemy `pool_recycle` parameter, which defaults to no timeout. Setting this parameter ensures that connections are closed and re-opened after the configured amount of idle time. If you see errors such as `_mysql_exceptions.OperationalError: (2006, 'MySQL server has gone away')`, this means your `max_idle` setting is probably too high. `show global variables like 'wait_timeout';` will show what the currently configured `wait_timeout` is on your MySQL server.

Buildbot requires `use_unique=True` and `charset=utf8`, and will add them automatically, so they do not need to be specified in `db_url`.

MySQL defaults to the MyISAM storage engine, but this can be overridden with the `storage_engine` URL argument.

Postgres

```
c['db_url'] = "postgresql://username@hostname/dbname"
```

PosgreSQL requires no special configuration.

MQ Specification

Buildbot uses a message-queueing system to handle communication within the master. Messages are used to indicate events within the master, and components that are interested in those events arrange to receive them.

The message queueing implementation is configured as a dictionary in the `mq` option. The `type` key describes the type of MQ implemetation to be used. Note that the implementation type cannot be changed in a reconfig.

The available implemenetation types are described in the following sections.

Simple

```
c['mq'] = {
    'type' : 'simple',
    'debug' : False,
}
```

This is the default MQ implementation. Similar to SQLite, it has no additional software dependencies, but does not support multi-master mode.

Note that this implementation also does not support message persistence across a restart of the master. For example, if a change is received, but the master shuts down before the schedulers can create build requests for it, then those schedulers will not be notified of the change when the master starts again.

The `debug` key, which defaults to False, can be used to enable logging of every message produced on this master.

Wamp

Note: At the moment, wamp is the only message queue implementation for multimaster. It has been privileged as this is the only message queue that have very solid support for Twisted. Other more common message queue systems like `RabbitMQ` (using the `AMQP` protocol) do not have convincing driver for twisted, and this would require to run on threads, which will add an important performance overhead.

```
c['mq'] = {
    'type' : 'wamp',
    'router_url': 'ws://localhost:8080',
    'realm': 'realm1',
    'wamp_debug_level' : 'error' # valid are: none, critical, error, warn, info,␣
↪debug, trace
}
```

This is a MQ implementation using wamp (http://wamp.ws/) protocol. This implementation uses Python Autobahn (http://autobahn.ws) wamp client library, and is fully asynchronous (no use of threads). To use this implementation, you need a wamp router like Crossbar (http://crossbar.io).

Please refer to Crossbar documentation for more details, but the default Crossbar setup will just work with Buildbot, provided you use the example `mq` configuration above, and start Crossbar with:

```
# of course, you should work in a virtualenv...
pip install crossbar
crossbar init
crossbar start
```

The implementation does not yet support wamp authentication. This MQ allows buildbot to run in multi-master mode.

Note that this implementation also does not support message persistence across a restart of the master. For example, if a change is received, but the master shuts down before the schedulers can create build requests for it, then those schedulers will not be notified of the change when the master starts again.

router_url (mandatory): points to your router websocket url. Buildbot is only supporting wamp over websocket, which is a sub-protocol of http. SSL is supported using `wss://` instead of `ws://`.

`realm` (optional, defaults to `buildbot`): defines the wamp realm to use for your buildbot messages.

`wamp_debug_level` (optional, defaults to `error`): defines the log level of autobahn.

You must use a router with very reliable connection to the master. If for some reason, the wamp connection is lost, then the master will stop, and should be restarted via a process manager.

Multi-master mode

See *Multimaster* for details on the Multi-master mode in Buildbot Nine.

By default, Buildbot makes coherency checks that prevents typo in your `master.cfg` It make sure schedulers are not referencing unknown builders, and enforces there is at least one builder.

In the case of a asymmetric multimaster, those coherency checks can be harmful and prevent you to implement what you want. For example you might want to have one master dedicated to the UI, so that a big load generated by builds will not impact page load times.

To enable multi-master mode in this configuration, you will need to set the *multiMaster* option so that buildbot doesn't warn about missing schedulers or builders.

```
# Enable multiMaster mode; disables warnings about unknown builders and
# schedulers
c['multiMaster'] = True
c['db'] = {
    'db_url' : 'mysql://...',
}
c['mq'] = {  # Need to enable multimaster aware mq. Wamp is the only option for now.
    'type' : 'wamp',
    'router_url': 'ws://localhost:8080',
    'realm': 'realm1',
    'wamp_debug_level' : 'error' # valid are: none, critical, error, warn, info,␣
↪debug, trace
}
```

Site Definitions

Three basic settings describe the buildmaster in status reports:

```
c['title'] = "Buildbot"
c['titleURL'] = "http://buildbot.sourceforge.net/"
```

title is a short string that will appear at the top of this buildbot installation's home page (linked to the *titleURL*).

titleURL is a URL string that must end with a slash (`/`). HTML status displays will show `title` as a link to *titleURL*. This URL is often used to provide a link from buildbot HTML pages to your project's home page.

The *buildbotURL* string should point to the location where the buildbot's internal web server is visible. This URL must end with a slash (`/`).

When status notices are sent to users (e.g., by email or over IRC), *buildbotURL* will be used to create a URL to the specific build or problem that they are being notified about.

Log Handling

```
c['logCompressionMethod'] = 'gz'
c['logMaxSize'] = 1024*1024 # 1M
c['logMaxTailSize'] = 32768
c['logEncoding'] = 'utf-8'
```

The *logCompressionLimit* enables compression of build logs on disk for logs that are bigger than the given size, or disables that completely if set to `False`. The default value is 4096, which should be a reasonable default on most file systems. This setting has no impact on status plugins, and merely affects the required disk space on the master for build logs.

The *logCompressionMethod* controls what type of compression is used for build logs. The default is 'gz', and the other valid option are 'raw' (no compression), 'gz' or 'lz4' (required lz4 package).

Please find below some stats extracted from 50x "Pyflakes" runs (results may differ according to log type).

Table 2.1: Space saving details

compression	raw log size	compressed log size	space saving	compression speed
bz2	2.981 MB	0.603 MB	79.77%	3.433 MB/s
gz	2.981 MB	0.568 MB	80.95%	6.604 MB/s
lz4	2.981 MB	0.844 MB	71.68%	77.668 MB/s

The *logMaxSize* parameter sets an upper limit (in bytes) to how large logs from an individual build step can be. The default value is None, meaning no upper limit to the log size. Any output exceeding *logMaxSize* will be truncated, and a message to this effect will be added to the log's HEADER channel.

If *logMaxSize* is set, and the output from a step exceeds the maximum, the *logMaxTailSize* parameter controls how much of the end of the build log will be kept. The effect of setting this parameter is that the log will contain the first *logMaxSize* bytes and the last *logMaxTailSize* bytes of output. Don't set this value too high, as the the tail of the log is kept in memory.

The *logEncoding* parameter specifies the character encoding to use to decode bytestrings provided as logs. It defaults to `utf-8`, which should work in most cases, but can be overridden if necessary. In extreme cases, a callable can be specified for this parameter. It will be called with byte strings, and should return the corresponding Unicode string.

This setting can be overridden for a single build step with the `logEncoding` step parameter. It can also be overridden for a single log file by passing the `logEncoding` parameter to `addLog`.

Data Lifetime

Horizons

```
c['changeHorizon'] = 200
c['buildHorizon'] = 100
c['logHorizon'] = 40
c['buildCacheSize'] = 15
```

Buildbot stores historical information in its database. In a large installation, these can quickly consume disk space, yet in many cases developers never consult this historical information.

The *changeHorizon* key determines how many changes the master will keep a record of. One place these changes are displayed is on the waterfall page. This parameter defaults to 0, which means keep all changes indefinitely.

The *buildHorizon* specifies the minimum number of builds for each builder which should be kept. The *logHorizon* gives the minimum number of builds for which logs should be maintained; this parameter must be less than or equal to *buildHorizon*. Builds older than *logHorizon* but not older than *buildHorizon* will maintain their overall status and the status of each step, but the logfiles will be deleted.

Caches

```
c['caches'] = {
    'Changes' : 100,      # formerly c['changeCacheSize']
    'Builds' : 500,       # formerly c['buildCacheSize']
    'chdicts' : 100,
    'BuildRequests' : 10,
    'SourceStamps' : 20,
    'ssdicts' : 20,
    'objectids' : 10,
    'usdicts' : 100,
}
```

The *caches* configuration key contains the configuration for Buildbot's in-memory caches. These caches keep frequently-used objects in memory to avoid unnecessary trips to the database. Caches are divided by object type, and each has a configurable maximum size.

The default size for each cache is 1, except where noted below. A value of 1 allows Buildbot to make a number of optimizations without consuming much memory. Larger, busier installations will likely want to increase these values.

The available caches are:

Changes the number of change objects to cache in memory. This should be larger than the number of changes that typically arrive in the span of a few minutes, otherwise your schedulers will be reloading changes from the database every time they run. For distributed version control systems, like Git or Hg, several thousand changes may arrive at once, so setting this parameter to something like 10000 isn't unreasonable.

This parameter is the same as the deprecated global parameter *changeCacheSize*. Its default value is 10.

Builds The *buildCacheSize* parameter gives the number of builds for each builder which are cached in memory. This number should be larger than the number of builds required for commonly-used status displays (the waterfall or grid views), so that those displays do not miss the cache on a refresh.

This parameter is the same as the deprecated global parameter *buildCacheSize*. Its default value is 15.

chdicts The number of rows from the `changes` table to cache in memory. This value should be similar to the value for `Changes`.

BuildRequests The number of BuildRequest objects kept in memory. This number should be higher than the typical number of outstanding build requests. If the master ordinarily finds jobs for BuildRequests immediately, you may set a lower value.

SourceStamps the number of SourceStamp objects kept in memory. This number should generally be similar to the number `BuildRequesets`.

ssdicts The number of rows from the `sourcestamps` table to cache in memory. This value should be similar to the value for `SourceStamps`.

objectids The number of object IDs - a means to correlate an object in the Buildbot configuration with an identity in the database–to cache. In this version, object IDs are not looked up often during runtime, so a relatively low value such as 10 is fine.

usdicts The number of rows from the `users` table to cache in memory. Note that for a given user there will be a row for each attribute that user has.

c['buildCacheSize'] = 15

Merging Build Requests

```
c['collapseRequests'] = True
```

This is a global default value for builders' *collapseRequests* parameter, and controls the merging of build requests.

This parameter can be overridden on a per-builder basis. See *Collapsing Build Requests* for the allowed values for this parameter.

Prioritizing Builders

```
def prioritizeBuilders(buildmaster, builders):
    ...
c['prioritizeBuilders'] = prioritizeBuilders
```

By default, buildbot will attempt to start builds on builders in order, beginning with the builder with the oldest pending request. Customize this behavior with the *prioritizeBuilders* configuration key, which takes a callable. See *Builder Priority Functions* for details on this callable.

This parameter controls the order that the build master can start builds, and is useful in situations where there is resource contention between builders, e.g., for a test database. It does not affect the order in which a builder processes the build requests in its queue. For that purpose, see *Prioritizing Builds*.

Setting the PB Port for Workers

```
c['protocols'] = {"pb": {"port": 10000}}
```

The buildmaster will listen on a TCP port of your choosing for connections from workers. It can also use this port for connections from remote Change Sources, status clients, and debug tools. This port should be visible to the outside world, and you'll need to tell your worker admins about your choice.

It does not matter which port you pick, as long it is externally visible; however, you should probably use something larger than 1024, since most operating systems don't allow non-root processes to bind to low-numbered ports. If your buildmaster is behind a firewall or a NAT box of some sort, you may have to configure your firewall to permit inbound connections to this port.

`c['protocols']['pb']['port']` is a *strports* specification string, defined in the `twisted.application.strports` module (try `pydoc twisted.application.strports` to get documentation on the format).

This means that you can have the buildmaster listen on a localhost-only port by doing:

```
c['protocols'] = {"pb": {"port": "tcp:10000:interface=127.0.0.1"}}
```

This might be useful if you only run workers on the same machine, and they are all configured to contact the buildmaster at `localhost:10000`.

Note: In Buildbot versions <=0.8.8 you might see `slavePortnum` option. This option contains same value as `c['protocols']['pb']['port']` but not recomended to use.

Defining Global Properties

The *properties* configuration key defines a dictionary of properties that will be available to all builds started by the buildmaster:

```
c['properties'] = {
    'Widget-version' : '1.2',
    'release-stage' : 'alpha'
}
```

Manhole

If you set *manhole* to an instance of one of the classes in `buildbot.manhole`, you can telnet or ssh into the buildmaster and get an interactive Python shell, which may be useful for debugging buildbot internals. It is probably only useful for buildbot developers. It exposes full access to the buildmaster's account (including the ability to modify and delete files), so it should not be enabled with a weak or easily guessable password.

There are three separate `Manhole` classes. Two of them use SSH, one uses unencrypted telnet. Two of them use a username+password combination to grant access, one of them uses an SSH-style `authorized_keys` file which contains a list of ssh public keys.

Note: Using any Manhole requires that `pycrypto` and `pyasn1` be installed. These are not part of the normal Buildbot dependencies.

---

manhole.AuthorizedKeysManhole You construct this with the name of a file that contains one SSH public key per line, just like `~/.ssh/authorized_keys`. If you provide a non-absolute filename, it will be interpreted relative to the buildmaster's base directory.

manhole.PasswordManhole This one accepts SSH connections but asks for a username and password when authenticating. It accepts only one such pair.

manhole.TelnetManhole This accepts regular unencrypted telnet connections, and asks for a username/password pair before providing access. Because this username/password is transmitted in the clear, and because Manhole access to the buildmaster is equivalent to granting full shell privileges to both the buildmaster and all the workers (and to all accounts which then run code produced by the workers), it is highly recommended that you use one of the SSH manholes instead.

```
# some examples:
from buildbot.plugins import util
c['manhole'] = util.AuthorizedKeysManhole(1234, "authorized_keys")
c['manhole'] = util.PasswordManhole(1234, "alice", "mysecretpassword")
c['manhole'] = util.TelnetManhole(1234, "bob", "snoop_my_password_please")
```

The `Manhole` instance can be configured to listen on a specific port. You may wish to have this listening port bind to the loopback interface (sometimes known as *lo0*, *localhost*, or 127.0.0.1) to restrict access to clients which are running on the same host.

```
from buildbot.plugins import util
c['manhole'] = util.PasswordManhole("tcp:9999:interface=127.0.0.1","admin","passwd")
```

To have the `Manhole` listen on all interfaces, use `"tcp:9999"` or simply 9999. This port specification uses `twisted.application.strports`, so you can make it listen on SSL or even UNIX-domain sockets if you want.

Note that using any `Manhole` requires that the TwistedConch (http://twistedmatrix.com/trac/wiki/TwistedConch) package be installed.

The buildmaster's SSH server will use a different host key than the normal sshd running on a typical unix host. This will cause the ssh client to complain about a *host key mismatch*, because it does not realize there are two separate servers running on the same host. To avoid this, use a clause like the following in your `.ssh/config` file:

```
Host remotehost-buildbot
HostName remotehost
HostKeyAlias remotehost-buildbot
Port 9999
# use 'user' if you use PasswordManhole and your name is not 'admin'.
# if you use AuthorizedKeysManhole, this probably doesn't matter.
User admin
```

Using Manhole

After you have connected to a manhole instance, you will find yourself at a Python prompt. You have access to two objects: `master` (the BuildMaster) and `status` (the master's Status object). Most interesting objects on the master can be reached from these two objects.

To aid in navigation, the `show` method is defined. It displays the non-method attributes of an object.

A manhole session might look like:

```
>>> show(master)
data attributes of <buildbot.master.BuildMaster instance at 0x7f7a4ab7df38>
                      basedir : '/home/dustin/code/buildbot/t/buildbot/'...
                    botmaster : <type 'instance'>
                buildCacheSize : None
                  buildHorizon : None
                   buildbotURL : http://localhost:8010/
                changeCacheSize : None
                     change_svc : <type 'instance'>
                 configFileName : master.cfg
                             db : <class 'buildbot.db.connector.DBConnector'>
                         db_url : sqlite:///state.sqlite
                                  ...
>>> show(master.botmaster.builders['win32'])
data attributes of <Builder ''builder'' at 48963528>
                                  ...
>>> win32 = _
>>> win32.category = 'w32'
```

Metrics Options

```
c['metrics'] = dict(log_interval=10, periodic_interval=10)
```

`metrics` can be a dictionary that configures various aspects of the metrics subsystem. If *`metrics`* is `None`, then metrics collection, logging and reporting will be disabled.

`log_interval` determines how often metrics should be logged to twistd.log. It defaults to 60s. If set to 0 or `None`, then logging of metrics will be disabled. This value can be changed via a reconfig.

`periodic_interval` determines how often various non-event based metrics are collected, such as memory usage, uncollectable garbage, reactor delay. This defaults to 10s. If set to 0 or `None`, then periodic collection of this data is

disabled. This value can also be changed via a reconfig.

Read more about metrics in the *Metrics* section in the developer documentation.

Statistics Service

The Statistics Service (stats service for short) supports for collecting arbitrary data from within a running Buildbot instance and export it do a number of storage backends. Currently, only InfluxDB (https://influxdata.com/time-series-platform/influxdb/) is supported as a storage backend. Also, InfluxDB (or any other storage backend) is not a mandatory dependency. Buildbot can run without it although `StatsService` will be of no use in such a case. At present, `StatsService` can keep track of build properties, build times (start, end, duration) and arbitrary data produced inside Buildbot (more on this later).

Example usage:

```
captures = [stats.CaptureProperty('Builder1', 'tree-size-KiB'),
            stats.CaptureBuildDuration('Builder2')]
c['services'] = []
c['services'].append(stats.StatsService(
    storage_backends=[
        stats.InfluxStorageService('localhost', 8086, 'root', 'root', 'test',␣
↪captures)
    ], name="StatsService"))
```

The `services` configuration value should be initialized as a list and a `StatsService` instance should be appended to it as shown in the example above.

Statistics Service

class `buildbot.statistics.stats_service.`**`StatsService`**

This is the main class for statistics service. It is initialized in the master configuration as show in the example above. It takes two arguments:

`storage_backends` A list of storage backends (see *Storage Backends*). In the example above, `stats.InfluxStorageService` is an instance of a storage backend. Each storage backend is an instances of subclasses of `statsStorageBase`.

`name` The name of this service.

`yieldMetricsValue`: This method can be used to send arbitrary data for storage. (See *Using StatsService.yieldMetricsValue* for more information.)

Capture Classes

class `buildbot.statistics.capture.`**`CaptureProperty`**

Instance of this class declares which properties must be captured and sent to the *Storage Backends*. It takes the following arguments:

`builder_name` The name of builder in which the property is recorded.

`property_name` The name of property needed to be recorded as a statistic.

`callback=None` (Optional) A custom callback function for this class. This callback function should take in two arguments - *build_properties* (dict) and *property_name* (str) and return a string that will be sent for storage in the storage backends.

regex=False If this is set to `True`, then the property name can be a regular expression. All properties matching this regular expression will be sent for storage.

class `buildbot.statistics.capture.`**CapturePropertyAllBuilders**
Instance of this class declares which properties must be captured on all builders and sent to the *Storage Backends*. It takes the following arguments:

property_name The name of property needed to be recorded as a statistic.

callback=None (Optional) A custom callback function for this class. This callback function should take in two arguments - *build_properties* (dict) and *property_name* (str) and return a string that will be sent for storage in the storage backends.

regex=False If this is set to `True`, then the property name can be a regular expression. All properties matching this regular expression will be sent for storage.

class `buildbot.statistics.capture.`**CaptureBuildStartTime**
Instance of this class declares which builders' start times are to be captured and sent to *Storage Backends*. It takes the following arguments:

builder_name The name of builder whose times are to be recorded.

callback=None (Optional) A custom callback function for this class. This callback function should take in a Python datetime object and return a string that will be sent for storage in the storage backends.

class `buildbot.statistics.capture.`**CaptureBuildStartTimeAllBuilders**
Instance of this class declares start times of all builders to be captured and sent to *Storage Backends*. It takes the following arguments:

callback=None (Optional) A custom callback function for this class. This callback function should take in a Python datetime object and return a string that will be sent for storage in the storage backends.

class `buildbot.statistics.capture.`**CaptureBuildEndTime**
Exactly like `CaptureBuildStartTime` except it declares the builders whose end time is to be recorded. The arguments are same as `CaptureBuildStartTime`.

class `buildbot.statistics.capture.`**CaptureBuildEndTimeAllBuilders**
Exactly like `CaptureBuildStartTimeAllBuilders` except it declares all builders' end time to be recorded. The arguments are same as `CaptureBuildStartTimeAllBuilders`.

class `buildbot.statistics.capture.`**CaptureBuildDuration**
Instance of this class declares the builders whose build durations are to be recorded. It takes the following arguments:

builder_name The name of builder whose times are to be recorded.

report_in='seconds' Can be one of three: `'seconds'`, `'minutes'`, or `'hours'`. This is the units in which the build time will be reported.

callback=None (Optional) A custom callback function for this class. This callback function should take in two Python datetime objects - a `start_time` and an `end_time` and return a string that will be sent for storage in the storage backends.

class `buildbot.statistics.capture.`**CaptureBuildDurationAllBuilders**
Instance of this class declares build durations to be recorded for all builders. It takes the following arguments:

report_in='seconds' Can be one of three: `'seconds'`, `'minutes'`, or `'hours'`. This is the units in which the build time will be reported.

callback=None (Optional) A custom callback function for this class. This callback function should take in two Python datetime objects - a `start_time` and an `end_time` and return a string that will be sent for storage in the storage backends.

class buildbot.statistics.capture.**CaptureData**
Instance of this capture class is for capturing arbitrary data that is not stored as build-data. Needs to be used in conjunction with yieldMetricsValue (See *Using StatsService.yieldMetricsValue*). Takes the following arguments:

data_name The name of data to be captured. Same as in yieldMetricsValue.

builder_name The name of builder whose times are to be recorded.

callback=None The callback function for this class. This callback receives the data sent to yieldMetricsValue as post_data (See *Using StatsService.yieldMetricsValue*). It must return a string that is to be sent to the storage backends for storage.

class buildbot.statistics.capture.**CaptureDataAllBuilders**
Instance of this capture class for capturing arbitrary data that is not stored as build-data on all builders. Needs to be used in conjunction with yieldMetricsValue (See *Using StatsService.yieldMetricsValue*). Takes the following arguments:

data_name The name of data to be captured. Same as in yieldMetricsValue.

callback=None The callback function for this class. This callback receives the data sent to yieldMetricsValue as post_data (See *Using StatsService.yieldMetricsValue*). It must return a string that is to be sent to the storage backends for storage.

Using StatsService.yieldMetricsValue

Advanced users can modify BuildSteps to use StatsService.yieldMetricsValue which will send arbitrary data for storage to the StatsService. It takes the following arguments:

data_name The name of the data being sent or storage.

post_data A dictionary of key value pair that is sent for storage. The keys will act as columns in a database and the value is stored under that column.

buildid The integer build id of the current build. Obtainable in all BuildSteps.

Along with using yieldMetricsValue, the user will also need to use the CaptureData capture class. As an example, we can add the following to a build step:

```
yieldMetricsValue('test_data_name', {'some_data': 'some_value'}, buildid)
```

Then, we can add in the master configuration a capture class like this:

```
captures = [CaptureBuildData('test_data_name', 'Builder1')]
```

Pass this captures list to a storage backend (as shown in the example at the top of this section) for capturing this data.

Storage Backends

Storage backends are responsible for storing any statistics data sent to them. A storage backend will generally be some sort of a database-server running on a machine. (*Note*: This machine may be different from the one running BuildMaster)

Currently, only InfluxDB (https://influxdata.com/time-series-platform/influxdb/) is supported as a storage backend.

class buildbot.statistics.storage_backends.influxdb_client.**InfluxStorageService**
This class is a Buildbot client to the InfluxDB storage backend. InfluxDB (https://influxdata.com/time-series-platform/influxdb/) is a distributed, time series database that employs a key-value pair storage system.

It requires the following arguments:

url The URL where the service is running.

port The port on which the service is listening.

user Username of a InfluxDB user.

password Password for `user`.

db The name of database to be used.

captures A list of objects of *Capture Classes*. This tells which statistics are to be stored in this storage backend.

name=None (Optional) The name of this storage backend.

secretsProviders

see *Secret Management* for details on secret concepts.

Example usage:

```
c['secretsProviders'] = [ .. ]
```

`secretsProviders` is a list of secrets storage. See *Secret Management* to configure an available secret storage provider.

BuildbotNetUsageData

Since buildbot 0.9.0, buildbot has a simple feature which sends usage analysis info to buildbot.net. This is very important for buildbot developers to understand how the community is using the tools. This allows to better prioritize issues, and understand what plugins are actually being used. This will also be a tool to decide whether to keep support for very old tools. For example buildbot contains support for the venerable CVS, but we have no information whether it actually works beyond the unit tests. We rely on the community to test and report issues with the old features.

With BuildbotNetUsageData, we can know exactly what combination of plugins are working together, how much people are customizing plugins, what versions of the main dependencies people run.

We take your privacy very seriously.

BuildbotNetUsageData will never send information specific to your Code or Intellectual Property. No repository url, shell command values, host names, ip address or custom class names. If it does, then this is a bug, please report.

We still need to track unique number for installation. This is done via doing a sha1 hash of master's hostname, installation path and fqdn. Using a secure hash means there is no way of knowing hostname, path and fqdn given the hash, but still there is a different hash for each master.

You can see exactly what is sent in the master's twisted.log. Usage data is sent every time the master is started.

BuildbotNetUsageData can be configured with 4 values:

- `c['buildbotNetUsageData'] = None` disables the feature
- `c['buildbotNetUsageData'] = 'basic'` sends the basic information to buildbot including:
  - versions of buildbot, python and twisted
  - platform information (CPU, OS, distribution, python flavor (i.e CPython vs PyPy))
  - mq and database type (mysql or sqlite?)
  - www plugins usage

  - Plugins usages: This counts the number of time each class of buildbot is used in your configuration. This counts workers, builders, steps, schedulers, change sources. If the plugin is subclassed, then it will be prefixed with a >

example of basic report (for the metabuildbot):

```
{
'versions': {
    'Python': '2.7.6',
    'Twisted': '15.5.0',
    'Buildbot': '0.9.0rc2-176-g5fa9dbf'
},
'platform': {
    'machine': 'x86_64',
    'python_implementation': 'CPython',
    'version': '#140-Ubuntu SMP Mon Jul',
    'processor':
    'x86_64',
    'distro:': ('Ubuntu', '14.04', 'trusty')
    },
'db': 'sqlite',
'mq': 'simple',
'plugins': {
    'buildbot.schedulers.forcesched.ForceScheduler': 2,
    'buildbot.schedulers.triggerable.Triggerable': 1,
    'buildbot.config.BuilderConfig': 4,
    'buildbot.schedulers.basic.AnyBranchScheduler': 2,
    'buildbot.steps.source.git.Git': 4,
    '>>buildbot.steps.trigger.Trigger': 2,
    '>>>buildbot.worker.base.Worker': 4,
    'buildbot.reporters.irc.IRC': 1,
    '>>>buildbot.process.buildstep.LoggingBuildStep': 2},
'www_plugins': ['buildbot_travis', 'waterfall_view']
}
```

- `c['buildbotNetUsageData'] = 'full'` sends the basic information plus additional information:
  - configuration of each builders: how the steps are arranged together. for ex:

```
{
    'builders': [
        ['buildbot.steps.source.git.Git', '>>>buildbot.process.buildstep.
↪LoggingBuildStep'],
        ['buildbot.steps.source.git.Git', '>>buildbot.steps.trigger.
↪Trigger'],
        ['buildbot.steps.source.git.Git', '>>>buildbot.process.buildstep.
↪LoggingBuildStep'],
        ['buildbot.steps.source.git.Git', '>>buildbot.steps.trigger.
↪Trigger']]
}
```

- `c['buildbotNetUsageData'] = myCustomFunction`. You can also specify exactly what to send using a callback.

  The custom function will take the generated data from full report in the form of a dictionary, and return a customized report as a jsonable dictionary. You can use this to filter any information you don't want to disclose. You can use a custom http_proxy environment variable in order to not send any data while developing your callback.

Users Options

```
from buildbot.plugins import util
c['user_managers'] = []
c['user_managers'].append(util.CommandlineUserManager(username="user",
                                                      passwd="userpw",
                                                      port=9990))
```

user_managers contains a list of ways to manually manage User Objects within Buildbot (see *User Objects*). Currently implemented is a commandline tool *buildbot user*, described at length in *user*. In the future, a web client will also be able to manage User Objects and their attributes.

As shown above, to enable the *buildbot user* tool, you must initialize a *CommandlineUserManager* instance in your *master.cfg*. *CommandlineUserManager* instances require the following arguments:

username This is the *username* that will be registered on the PB connection and need to be used when calling *buildbot user*.

passwd This is the *passwd* that will be registered on the PB connection and need to be used when calling *buildbot user*.

port The PB connection *port* must be different than *c['protocols']['pb']['port']* and be specified when calling *buildbot user*

Input Validation

```
import re
c['validation'] = {
    'branch' : re.compile(r'^[\w.+/~-]*$'),
    'revision' : re.compile(r'^[ \w\.\-\/]*$'),
    'property_name' : re.compile(r'^[\w\.\-\/\~:]*$'),
    'property_value' : re.compile(r'^[\w\.\-\/\~:]*$'),
}
```

This option configures the validation applied to user inputs of various types. This validation is important since these values are often included in command-line arguments executed on workers. Allowing arbitrary input from untrusted users may raise security concerns.

The keys describe the type of input validated; the values are compiled regular expressions against which the input will be matched. The defaults for each type of input are those given in the example, above.

Revision Links

The *revlink* parameter is used to create links from revision IDs in the web status to a web-view of your source control system. The parameter's value must be a callable.

By default, Buildbot is configured to generate revlinks for a number of open source hosting platforms.

The callable takes the revision id and repository argument, and should return an URL to the revision. Note that the revision id may not always be in the form you expect, so code defensively. In particular, a revision of "??" may be supplied when no other information is available.

Note that `SourceStamps` that are not created from version-control changes (e.g., those created by a *Nightly* or *Periodic* scheduler) may have an empty repository string, if the repository is not known to the scheduler.

Revision Link Helpers

Buildbot provides two helpers for generating revision links. `buildbot.revlinks.RevlinkMatcher` takes a list of regular expressions, and replacement text. The regular expressions should all have the same number of capture groups. The replacement text should have sed-style references to that capture groups (i.e. '1' for the first capture group), and a single '%s' reference, for the revision ID. The repository given is tried against each regular expression in turn. The results are the substituted into the replacement text, along with the revision ID to obtain the revision link.

```
from buildbot.plugins import util
c['revlink'] = util.RevlinkMatch([r'git://notmuchmail.org/git/(.*)'],
                                 r'http://git.notmuchmail.org/git/\1/commit/%s')
```

`buildbot.revlinks.RevlinkMultiplexer` takes a list of revision link callables, and tries each in turn, returning the first successful match.

Codebase Generator

```
all_repositories = {
    r'https://hg/hg/mailsuite/mailclient': 'mailexe',
    r'https://hg/hg/mailsuite/mapilib': 'mapilib',
    r'https://hg/hg/mailsuite/imaplib': 'imaplib',
    r'https://github.com/mailinc/mailsuite/mailclient': 'mailexe',
    r'https://github.com/mailinc/mailsuite/mapilib': 'mapilib',
    r'https://github.com/mailinc/mailsuite/imaplib': 'imaplib',
}

def codebaseGenerator(chdict):
    return all_repositories[chdict['repository']]

c['codebaseGenerator'] = codebaseGenerator
```

For any incoming change, the *codebase* is set to ''. This codebase value is sufficient if all changes come from the same repository (or clones). If changes come from different repositories, extra processing will be needed to determine the codebase for the incoming change. This codebase will then be a logical name for the combination of repository and or branch etc.

The *codebaseGenerator* accepts a change dictionary as produced by the `buildbot.db.changes.ChangesConnectorComponent`, with a changeid equal to *None*.

Change Sources

A Version Control System maintains a source tree, and tells the buildmaster when it changes. The first step of each `Build` is typically to acquire a copy of some version of this tree.

This chapter describes how the Buildbot learns about what `Changes` have occurred. For more information on VC systems and `Changes`, see *Version Control Systems*.

`Changes` can be provided by a variety of `ChangeSource` types, although any given project will typically have only a single `ChangeSource` active. This section provides a description of all available `ChangeSource` types and explains how to set up each of them.

Choosing a Change Source

There are a variety of `ChangeSource` classes available, some of which are meant to be used in conjunction with other tools to deliver `Change` events from the VC repository to the buildmaster.

As a quick guide, here is a list of VC systems and the `ChangeSources` that might be useful with them. Note that some of these modules are in Buildbot's master/contrib (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib) directory, meaning that they have been offered by other users in hopes they may be useful, and might require some additional work to make them functional.

CVS

- `CVSMaildirSource` (watching mail sent by master/contrib/buildbot_cvs_mail.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/buildbot_cvs_mail.py) script)
- `PBChangeSource` (listening for connections from `buildbot sendchange` run in a loginfo script)
- `PBChangeSource` (listening for connections from a long-running master/contrib/viewcvspoll.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/viewcvspoll.py) polling process which examines the ViewCVS database directly)

- *Change Hooks* in WebStatus

SVN

- *PBChangeSource* (listening for connections from master/contrib/svn_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/svn_buildbot.py) run in a postcommit script)
- *PBChangeSource* (listening for connections from a long-running master/contrib/svn_watcher.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/svn_watcher.py) or master/contrib/svnpoller.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/svnpoller.py) polling process
- *SVNCommitEmailMaildirSource* (watching for email sent by `commit-email.pl`)
- *SVNPoller* (polling the SVN repository)
- *Change Hooks* in WebStatus

Darcs

- *PBChangeSource* (listening for connections from master/contrib/darcs_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/darcs_buildbot.py) in a commit script)
- *Change Hooks* in WebStatus

Mercurial

- *Change Hooks* in WebStatus (including master/contrib/hgbuildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/hgbuildbot.py), configurable in a `changegroup` hook)
- BitBucket change hook (specifically designed for BitBucket notifications, but requiring a publicly-accessible WebStatus)
- *HgPoller* (polling a remote Mercurial repository)
- *BitbucketPullrequestPoller* (polling Bitbucket for pull requests)
- *Mail-parsing ChangeSources*, though there are no ready-to-use recipes

Bzr (the newer Bazaar)

- *PBChangeSource* (listening for connections from master/contrib/bzr_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/bzr_buildbot.py) run in a post-change-branch-tip or commit hook)
- *BzrPoller* (polling the Bzr repository)
- *Change Hooks* in WebStatus

Git

- *PBChangeSource* (listening for connections from master/contrib/git_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/git_buildbot.py) run in the post-receive hook)
- *PBChangeSource* (listening for connections from master/contrib/github_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/github_buildbot.py), which listens for notifications from GitHub)
- *Change Hooks* in WebStatus
- *GitHub* change hook (specifically designed for GitHub notifications, but requiring a publicly-accessible WebStatus)

- *BitBucket* change hook (specifically designed for BitBucket notifications, but requiring a publicly-accessible WebStatus)
- *GitPoller* (polling a remote Git repository)
- *GitHubPullrequestPoller* (polling GitHub API for pull requests)
- *BitbucketPullrequestPoller* (polling Bitbucket for pull requests)

Repo/Gerrit

- *GerritChangeSource* connects to Gerrit via SSH to get a live stream of changes
- *GerritEventLogPoller* connects to Gerrit via HTTP with the help of the plugin events-log (https://gerrit.googlesource.com/plugins/events-log/)

Monotone

- *PBChangeSource* (listening for connections from `monotone-buildbot.lua`, which is available with Monotone)

All VC systems can be driven by a *PBChangeSource* and the `buildbot sendchange` tool run from some form of commit script. If you write an email parsing function, they can also all be driven by a suitable *mail-parsing source*. Additionally, handlers for web-based notification (i.e. from GitHub) can be used with WebStatus' change_hook module. The interface is simple, so adding your own handlers (and sharing!) should be a breeze.

See `chsrc` for a full list of change sources.

Configuring Change Sources

The *change_source* configuration key holds all active change sources for the configuration.

Most configurations have a single `ChangeSource`, watching only a single tree, e.g.,

```
from buildbot.plugins import changes

c['change_source'] = changes.PBChangeSource()
```

For more advanced configurations, the parameter can be a list of change sources:

```
source1 = ...
source2 = ...
c['change_source'] = [
    source1, source1
]
```

Repository and Project

`ChangeSources` will, in general, automatically provide the proper `repository` attribute for any changes they produce. For systems which operate on URL-like specifiers, this is a repository URL. Other `ChangeSources` adapt the concept as necessary.

Many `ChangeSources` allow you to specify a project, as well. This attribute is useful when building from several distinct codebases in the same buildmaster: the project string can serve to differentiate the different codebases. Schedulers can filter on project, so you can configure different builders to run for each project.

Mail-parsing ChangeSources

Many projects publish information about changes to their source tree by sending an email message out to a mailing list, frequently named *PROJECT*-commits or *PROJECT*-changes. Each message usually contains a description of the change (who made the change, which files were affected) and sometimes a copy of the diff. Humans can subscribe to this list to stay informed about what's happening to the source tree.

The Buildbot can also be subscribed to a *-commits* mailing list, and can trigger builds in response to Changes that it hears about. The buildmaster admin needs to arrange for these email messages to arrive in a place where the buildmaster can find them, and configure the buildmaster to parse the messages correctly. Once that is in place, the email parser will create Change objects and deliver them to the schedulers (see *Schedulers*) just like any other ChangeSource.

There are two components to setting up an email-based ChangeSource. The first is to route the email messages to the buildmaster, which is done by dropping them into a *maildir*. The second is to actually parse the messages, which is highly dependent upon the tool that was used to create them. Each VC system has a collection of favorite change-emailing tools, and each has a slightly different format, so each has a different parsing function. There is a separate ChangeSource variant for each parsing function.

Once you've chosen a maildir location and a parsing function, create the change source and put it in *change_source*:

```
from buildbot.plugins import changes

c['change_source'] = changes.CVSMaildirSource("~/maildir-buildbot",
                                               prefix="/trunk/")
```

Subscribing the Buildmaster

The recommended way to install the Buildbot is to create a dedicated account for the buildmaster. If you do this, the account will probably have a distinct email address (perhaps *buildmaster@example.org*). Then just arrange for this account's email to be delivered to a suitable maildir (described in the next section).

If the Buildbot does not have its own account, *extension addresses* can be used to distinguish between email intended for the buildmaster and email intended for the rest of the account. In most modern MTAs, the e.g. *foo@example.org* account has control over every email address at example.org which begins with "foo", such that email addressed to *account-foo@example.org* can be delivered to a different destination than *account-bar@example.org*. qmail does this by using separate `.qmail` files for the two destinations (`.qmail-foo` and `.qmail-bar`, with `.qmail` controlling the base address and `.qmail-default` controlling all other extensions). Other MTAs have similar mechanisms.

Thus you can assign an extension address like *foo-buildmaster@example.org* to the buildmaster, and retain *foo@example.org* for your own use.

Using Maildirs

A *maildir* is a simple directory structure originally developed for qmail that allows safe atomic update without locking. Create a base directory with three subdirectories: `new`, `tmp`, and `cur`. When messages arrive, they are put into a uniquely-named file (using pids, timestamps, and random numbers) in `tmp`. When the file is complete, it is atomically renamed into `new`. Eventually the buildmaster notices the file in `new`, reads and parses the contents, then moves it into `cur`. A cronjob can be used to delete files in `cur` at leisure.

Maildirs are frequently created with the **maildirmake** tool, but a simple `mkdir -p ~/MAILDIR/cur,new,tmp` is pretty much equivalent.

Many modern MTAs can deliver directly to maildirs. The usual `.forward` or `.procmailrc` syntax is to name the base directory with a trailing slash, so something like *~/MAILDIR/*. qmail and postfix are maildir-capable MTAs, and procmail is a maildir-capable MDA (Mail Delivery Agent).

Here is an example procmail config, located in `~/.procmailrc`:

```
# .procmailrc
# routes incoming mail to appropriate mailboxes
PATH=/usr/bin:/usr/local/bin
MAILDIR=$HOME/Mail
LOGFILE=.procmail_log
SHELL=/bin/sh

:0
*
new
```

If procmail is not setup on a system wide basis, then the following one-line `.forward` file will invoke it.

```
!/usr/bin/procmail
```

For MTAs which cannot put files into maildirs directly, the *safecat* tool can be executed from a `.forward` file to accomplish the same thing.

The Buildmaster uses the linux DNotify facility to receive immediate notification when the maildir's `new` directory has changed. When this facility is not available, it polls the directory for new messages, every 10 seconds by default.

Parsing Email Change Messages

The second component to setting up an email-based `ChangeSource` is to parse the actual notices. This is highly dependent upon the VC system and commit script in use.

A couple of common tools used to create these change emails, along with the Buildbot tools to parse them, are:

CVS

Buildbot CVS MailNotifier *CVSMaildirSource*

SVN

svnmailer http://opensource.perlig.de/en/svnmailer/

commit-email.pl *SVNCommitEmailMaildirSource*

Bzr

Launchpad *BzrLaunchpadEmailMaildirSource*

Mercurial

NotifyExtension https://www.mercurial-scm.org/wiki/NotifyExtension

Git

post-receive-email http://git.kernel.org/?p=git/git.git;a=blob;f=contrib/hooks/post-receive-email;hb=HEAD

The following sections describe the parsers available for each of these tools.

Most of these parsers accept a `prefix=` argument, which is used to limit the set of files that the buildmaster pays attention to. This is most useful for systems like CVS and SVN which put multiple projects in a single repository (or use repository names to indicate branches). Each filename that appears in the email is tested against the prefix: if

the filename does not start with the prefix, the file is ignored. If the filename *does* start with the prefix, that prefix is stripped from the filename before any further processing is done. Thus the prefix usually ends with a slash.

CVSMaildirSource

class buildbot.changes.mail.**CVSMaildirSource**

This parser works with the master/contrib/buildbot_cvs_mail.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/buildbot_cvs_mail.py) script.

The script sends an email containing all the files submitted in one directory. It is invoked by using the `CVSROOT/loginfo` facility.

The Buildbot's *CVSMaildirSource* knows how to parse these messages and turn them into Change objects. It takes the directory name of the maildir root. For example:

```
from buildbot.plugins import changes

c['change_source'] = changes.CVSMaildirSource("/home/buildbot/Mail")
```

Configuration of CVS and buildbot_cvs_mail.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/buildbot_cvs_mail.py)

CVS must be configured to invoke the buildbot_cvs_mail.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/buildbot_cvs_mail.py) script when files are checked in. This is done via the CVS loginfo configuration file.

To update this, first do:

```
cvs checkout CVSROOT
```

cd to the CVSROOT directory and edit the file loginfo, adding a line like:

```
SomeModule /cvsroot/CVSROOT/buildbot_cvs_mail.py --cvsroot :ext:example.com:/cvsroot -
↪e buildbot -P SomeModule %@{sVv@}
```

Note: For cvs version 1.12.x, the `--path %p` option is required. Version 1.11.x and 1.12.x report the directory path differently.

The above example you put the buildbot_cvs_mail.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/buildbot_cvs_mail.py) script under /cvsroot/CVSROOT. It can be anywhere. Run the script with `--help` to see all the options. At the very least, the options `-e` (email) and `-P` (project) should be specified. The line must end with `%{sVv}`. This is expanded to the files that were modified.

Additional entries can be added to support more modules.

See **`buildbot_cvs_mail.py --help`** for more information on the available options.

SVNCommitEmailMaildirSource

class buildbot.changes.mail.**SVNCommitEmailMaildirSource**

SVNCommitEmailMaildirSource parses message sent out by the `commit-email.pl` script, which is included in the Subversion distribution.

It does not currently handle branches: all of the Change objects that it creates will be associated with the default (i.e. trunk) branch.

```
from buildbot.plugins import changes

c['change_source'] = changes.SVNCommitEmailMaildirSource("~/maildir-buildbot")
```

BzrLaunchpadEmailMaildirSource

class `buildbot.changes.mail.`**BzrLaunchpadEmailMaildirSource**

BzrLaunchpadEmailMaildirSource parses the mails that are sent to addresses that subscribe to branch revision notifications for a bzr branch hosted on Launchpad.

The branch name defaults to `lp:`*Launchpad path*. For example `lp:~maria-captains/maria/5.1`.

If only a single branch is used, the default branch name can be changed by setting `defaultBranch`.

For multiple branches, pass a dictionary as the value of the `branchMap` option to map specific repository paths to specific branch names (see example below). The leading `lp:` prefix of the path is optional.

The `prefix` option is not supported (it is silently ignored). Use the `branchMap` and `defaultBranch` instead to assign changes to branches (and just do not subscribe the Buildbot to branches that are not of interest).

The revision number is obtained from the email text. The bzr revision id is not available in the mails sent by Launchpad. However, it is possible to set the bzr *append_revisions_only* option for public shared repositories to avoid new pushes of merges changing the meaning of old revision numbers.

```
from buildbot.plugins import changes

bm = {
    'lp:~maria-captains/maria/5.1': '5.1',
    'lp:~maria-captains/maria/6.0': '6.0'
}
c['change_source'] = changes.BzrLaunchpadEmailMaildirSource("~/maildir-buildbot",
                                                            branchMap=bm)
```

PBChangeSource

class `buildbot.changes.pb.`**PBChangeSource**

PBChangeSource actually listens on a TCP port for clients to connect and push change notices *into* the Buildmaster. This is used by the built-in `buildbot sendchange` notification tool, as well as several version-control hook scripts. This change is also useful for creating new kinds of change sources that work on a *push* model instead of some kind of subscription scheme, for example a script which is run out of an email `.forward` file. This ChangeSource always runs on the same TCP port as the workers. It shares the same protocol, and in fact shares the same space of "usernames", so you cannot configure a *PBChangeSource* with the same name as a worker.

If you have a publicly accessible worker port, and are using *PBChangeSource*, *you must establish a secure username and password for the change source*. If your sendchange credentials are known (e.g., the defaults), then your buildmaster is susceptible to injection of arbitrary changes, which (depending on the build factories) could lead to arbitrary code execution on workers.

The *PBChangeSource* is created with the following arguments.

port which port to listen on. If `None` (which is the default), it shares the port used for worker connections.

user The user account that the client program must use to connect. Defaults to `change`

passwd The password for the connection - defaults to `changepw`. Do not use this default on a publicly exposed port!

prefix The prefix to be found and stripped from filenames delivered over the connection, defaulting to `None`. Any filenames which do not start with this prefix will be removed. If all the filenames in a given Change are removed, the that whole Change will be dropped. This string should probably end with a directory separator.

This is useful for changes coming from version control systems that represent branches as parent directories within the repository (like SVN and Perforce). Use a prefix of `trunk/` or `project/branches/foobranch/` to only follow one branch and to get correct tree-relative filenames. Without a prefix, the *PBChangeSource* will probably deliver Changes with filenames like `trunk/foo.c` instead of just `foo.c`. Of course this also depends upon the tool sending the Changes in (like *buildbot sendchange*) and what filenames it is delivering: that tool may be filtering and stripping prefixes at the sending end.

For example:

```
from buildbot.plugins import changes

c['change_source'] = changes.PBChangeSource(port=9999, user='laura', passwd='fpga')
```

The following hooks are useful for sending changes to a *PBChangeSource*:

Bzr Hook

Bzr is also written in Python, and the Bzr hook depends on Twisted to send the changes.

To install, put master/contrib/bzr_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/bzr_buildbot.py) in one of your plugins locations a bzr plugins directory (e.g., `~/.bazaar/plugins`). Then, in one of your bazaar conf files (e.g., `~/.bazaar/locations.conf`), set the location you want to connect with Buildbot with these keys:

- `buildbot_on` one of 'commit', 'push, or 'change'. Turns the plugin on to report changes via commit, changes via push, or any changes to the trunk. 'change' is recommended.
- `buildbot_server` (required to send to a Buildbot master) the URL of the Buildbot master to which you will connect (as of this writing, the same server and port to which workers connect).
- `buildbot_port` (optional, defaults to 9989) the port of the Buildbot master to which you will connect (as of this writing, the same server and port to which workers connect)
- `buildbot_pqm` (optional, defaults to not pqm) Normally, the user that commits the revision is the user that is responsible for the change. When run in a pqm (Patch Queue Manager, see https://launchpad.net/pqm) environment, the user that commits is the Patch Queue Manager, and the user that committed the *parent* revision is responsible for the change. To turn on the pqm mode, set this value to any of (case-insensitive) "Yes", "Y", "True", or "T".
- `buildbot_dry_run` (optional, defaults to not a dry run) Normally, the post-commit hook will attempt to communicate with the configured Buildbot server and port. If this parameter is included and any of (case-insensitive) "Yes", "Y", "True", or "T", then the hook will simply print what it would have sent, but not attempt to contact the Buildbot master.
- `buildbot_send_branch_name` (optional, defaults to not sending the branch name) If your Buildbot's bzr source build step uses a repourl, do *not* turn this on. If your buildbot's bzr build step uses a baseURL, then you may set this value to any of (case-insensitive) "Yes", "Y", "True", or "T" to have the Buildbot master append the branch name to the baseURL.

Note: The bzr smart server (as of version 2.2.2) doesn't know how to resolve `bzr://` urls into absolute paths so any paths in `locations.conf` won't match, hence no change notifications will be sent to Buildbot. Setting configuration parameters globally or in-branch might still work. When Buildbot no longer has a hardcoded password, it will be a configuration option here as well.

Here's a simple example that you might have in your `~/.bazaar/locations.conf`.

```
[chroot-*:///var/local/myrepo/mybranch]
buildbot_on = change
buildbot_server = localhost
```

P4Source

The *`P4Source`* periodically polls a Perforce (http://www.perforce.com/) depot for changes. It accepts the following arguments:

`p4port` The Perforce server to connect to (as *`host:port`*).

`p4user` The Perforce user.

`p4passwd` The Perforce password.

`p4base` The base depot path to watch, without the trailing '/...'.

`p4bin` An optional string parameter. Specify the location of the perforce command line binary (p4). You only need to do this if the perforce binary is not in the path of the Buildbot user. Defaults to *p4*.

`split_file` A function that maps a pathname, without the leading `p4base`, to a (branch, filename) tuple. The default just returns `(None, branchfile)`, which effectively disables branch support. You should supply a function which understands your repository structure.

`pollInterval` How often to poll, in seconds. Defaults to 600 (10 minutes).

`project` Set the name of the project to be used for the *`P4Source`*. This will then be set in any changes generated by the `P4Source`, and can be used in a Change Filter for triggering particular builders.

`pollAtLaunch` Determines when the first poll occurs. True = immediately on launch, False = wait for one pollInterval (default).

`histmax` The maximum number of changes to inspect at a time. If more than this number occur since the last poll, older changes will be silently ignored.

`encoding` The character encoding of `p4`'s output. This defaults to "utf8", but if your commit messages are in another encoding, specify that here. For example, if you're using Perforce on Windows, you may need to use "cp437" as the encoding if "utf8" generates errors in your master log.

`server_tz` The timezone of the Perforce server, using the usual timezone format (e.g: `"Europe/Stockholm"`) in case it's not in UTC.

`use_tickets` Set to `True` to use ticket-based authentication, instead of passwords (but you still need to specify `p4passwd`).

`ticket_login_interval` How often to get a new ticket, in seconds, when `use_tickets` is enabled. Defaults to 86400 (24 hours).

Example

This configuration uses the P4PORT, P4USER, and P4PASSWD specified in the buildmaster's environment. It watches a project in which the branch name is simply the next path component, and the file is all path components after.

```
from buildbot.plugins import changes

s = changes.P4Source(p4base='//depot/project/',
                     split_file=lambda branchfile: branchfile.split('/',1))
c['change_source'] = s
```

SVNPoller

class buildbot.changes.svnpoller.**SVNPoller**

The *SVNPoller* is a ChangeSource which periodically polls a Subversion (http://subversion.tigris.org/) repository for new revisions, by running the `svn log` command in a subshell. It can watch a single branch or multiple branches.

SVNPoller accepts the following arguments:

repourl The base URL path to watch, like `svn://svn.twistedmatrix.com/svn/Twisted/trunk`, or `http://divmod.org/svn/Divmo/`, or even `file:///home/svn/Repository/ProjectA/branches/1.5/`. This must include the access scheme, the location of the repository (both the hostname for remote ones, and any additional directory names necessary to get to the repository), and the sub-path within the repository's virtual filesystem for the project and branch of interest.

The *SVNPoller* will only pay attention to files inside the subdirectory specified by the complete repourl.

split_file A function to convert pathnames into `(branch, relative_pathname)` tuples. Use this to explain your repository's branch-naming policy to *SVNPoller*. This function must accept a single string (the pathname relative to the repository) and return a two-entry tuple. Directory pathnames always end with a right slash to distinguish them from files, like `trunk/src/`, or `src/`. There are a few utility functions in `buildbot.changes.svnpoller` that can be used as a `split_file` function; see below for details.

For directories, the relative pathname returned by `split_file` should end with a right slash but an empty string is also accepted for the root, like `("branches/1.5.x", "")` being converted from `"branches/1.5.x/"`.

The default value always returns `(None, path)`, which indicates that all files are on the trunk.

Subclasses of *SVNPoller* can override the `split_file` method instead of using the `split_file=` argument.

project Set the name of the project to be used for the *SVNPoller*. This will then be set in any changes generated by the *SVNPoller*, and can be used in a *Change Filter* for triggering particular builders.

svnuser An optional string parameter. If set, the option *–user* argument will be added to all **svn** commands. Use this if you have to authenticate to the svn server before you can do **svn info** or **svn log** commands.

svnpasswd Like `svnuser`, this will cause a option *–password* argument to be passed to all **svn** commands.

pollInterval How often to poll, in seconds. Defaults to 600 (checking once every 10 minutes). Lower this if you want the Buildbot to notice changes faster, raise it if you want to reduce the network and CPU load on your svn server. Please be considerate of public SVN repositories by using a large interval when polling them.

pollAtLaunch Determines when the first poll occurs. True = immediately on launch, False = wait for one pollInterval (default).

histmax The maximum number of changes to inspect at a time. Every `pollInterval` seconds, the *SVNPoller* asks for the last `histmax` changes and looks through them for any revisions it does not already know about. If

more than `histmax` revisions have been committed since the last poll, older changes will be silently ignored. Larger values of `histmax` will cause more time and memory to be consumed on each poll attempt. `histmax` defaults to 100.

svnbin This controls the **svn** executable to use. If subversion is installed in a weird place on your system (outside of the buildmaster's `PATH`), use this to tell *SVNPoller* where to find it. The default value of *svn* will almost always be sufficient.

revlinktmpl This parameter is deprecated in favour of specifying a global revlink option. This parameter allows a link to be provided for each revision (for example, to websvn or viewvc). These links appear anywhere changes are shown, such as on build or change pages. The proper form for this parameter is an URL with the portion that will substitute for a revision number replaced by ''%s''. For example, `'http://myserver/websvn/revision.php?rev=%s'` could be used to cause revision links to be created to a websvn repository viewer.

cachepath If specified, this is a pathname of a cache file that *SVNPoller* will use to store its state between restarts of the master.

extra_args If specified, the extra arguments will be added to the svn command args.

Several split file functions are available for common SVN repository layouts. For a poller that is only monitoring trunk, the default split file function is available explicitly as `split_file_alwaystrunk`:

```
from buildbot.plugins import changes, util

c['change_source'] = changes.SVNPoller(
    repourl="svn://svn.twistedmatrix.com/svn/Twisted/trunk",
    split_file=util.svn.split_file_alwaystrunk)
```

For repositories with the `/trunk` and `/branches/`*BRANCH* layout, `split_file_branches` will do the job:

```
from buildbot.plugins import changes, util

c['change_source'] = changes.SVNPoller(
    repourl="https://amanda.svn.sourceforge.net/svnroot/amanda/amanda",
    split_file=util.svn.split_file_branches)
```

When using this splitter the poller will set the `project` attribute of any changes to the `project` attribute of the poller.

For repositories with the *PROJECT*`/trunk` and *PROJECT*`/branches/`*BRANCH* layout, `split_file_projects_branches` will do the job:

```
from buildbot.plugins import changes, util

c['change_source'] = changes.SVNPoller(
    repourl="https://amanda.svn.sourceforge.net/svnroot/amanda/",
    split_file=util.svn.split_file_projects_branches)
```

When using this splitter the poller will set the `project` attribute of any changes to the project determined by the splitter.

The *SVNPoller* is highly adaptable to various Subversion layouts. See *Customizing SVNPoller* for details and some common scenarios.

Bzr Poller

If you cannot insert a Bzr hook in the server, you can use the *BzrPoller*. To use it, put master/contrib/bzr_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/bzr_buildbot.py) somewhere that your Buildbot configuration can import it. Even putting it in the same directory as the `master.cfg`

should work. Install the poller in the Buildbot configuration as with any other change source. Minimally, provide a URL that you want to poll (`bzr://`, `bzr+ssh://`, or `lp:`), making sure the Buildbot user has necessary privileges.

```
# put bzr_buildbot.py file to the same directory as master.cfg
from bzr_buildbot import BzrPoller

c['change_source'] = BzrPoller(
    url='bzr://hostname/my_project',
    poll_interval=300)
```

The `BzrPoller` parameters are:

`url` The URL to poll.

`poll_interval` The number of seconds to wait between polls. Defaults to 10 minutes.

`branch_name` Any value to be used as the branch name. Defaults to None, or specify a string, or specify the constants from bzr_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/bzr_buildbot.py) `SHORT` or `FULL` to get the short branch name or full branch address.

`blame_merge_author` normally, the user that commits the revision is the user that is responsible for the change. When run in a pqm (Patch Queue Manager, see https://launchpad.net/pqm) environment, the user that commits is the Patch Queue Manager, and the user that committed the merged, *parent* revision is responsible for the change. Set this value to `True` if this is pointed against a PQM-managed branch.

GitPoller

If you cannot take advantage of post-receive hooks as provided by master/contrib/git_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/git_buildbot.py) for example, then you can use the *`GitPoller`*.

The *`GitPoller`* periodically fetches from a remote Git repository and processes any changes. It requires its own working directory for operation. The default should be adequate, but it can be overridden via the `workdir` property.

Note: There can only be a single *GitPoller* pointed at any given repository.

The *`GitPoller`* requires Git-1.7 and later. It accepts the following arguments:

`repourl` the git-url that describes the remote repository, e.g. `git@example.com:foobaz/myrepo.git` (see the **`git fetch`** help for more info on git-url formats)

`branches` One of the following:

- a list of the branches to fetch.
- `True` indicating that all branches should be fetched
- a callable which takes a single argument. It should take a remote refspec (such as `'refs/heads/master'`, and return a boolean indicating whether that branch should be fetched.

`branch` accepts a single branch name to fetch. Exists for backwards compatibility with old configurations.

`pollInterval` interval in seconds between polls, default is 10 minutes

`pollAtLaunch` Determines when the first poll occurs. True = immediately on launch, False = wait for one pollInterval (default).

`buildPushesWithNoCommits` Determine if a push on a new branch or update of an already known branch with already known commits should trigger a build. This is useful in case you have build steps depending on the name

of the branch and you use topic branches for development. When you merge your topic branch into "master" (for instance), a new build will be triggered. (defaults to False).

gitbin path to the Git binary, defaults to just `'git'`

category Set the category to be used for the changes produced by the *GitPoller*. This will then be set in any changes generated by the *GitPoller*, and can be used in a Change Filter for triggering particular builders.

project Set the name of the project to be used for the *GitPoller*. This will then be set in any changes generated by the `GitPoller`, and can be used in a Change Filter for triggering particular builders.

usetimestamps parse each revision's commit timestamp (default is `True`), or ignore it in favor of the current time (so recently processed commits appear together in the waterfall page)

encoding Set encoding will be used to parse author's name and commit message. Default encoding is `'utf-8'`. This will not be applied to file names since Git will translate non-ascii file names to unreadable escape sequences.

workdir the directory where the poller should keep its local repository. The default is `gitpoller_work`. If this is a relative path, it will be interpreted relative to the master's basedir. Multiple Git pollers can share the same directory.

only_tags Determines if the GitPoller should poll for new tags in the git repository.

A configuration for the Git poller might look like this:

```
from buildbot.plugins import changes

c['change_source'] = changes.GitPoller(repourl='git@example.com:foobaz/myrepo.git',
                                       branches=['master', 'great_new_feature'])
```

HgPoller

The *HgPoller* periodically pulls a named branch from a remote Mercurial repository and processes any changes. It requires its own working directory for operation, which must be specified via the `workdir` property.

The *HgPoller* requires a working `hg` executable, and at least a read-only access to the repository it polls (possibly through ssh keys or by tweaking the `hgrc` of the system user Buildbot runs as).

The *HgPoller* will not transmit any change if there are several heads on the watched named branch. This is similar (although not identical) to the Mercurial executable behaviour. This exceptional condition is usually the result of a developer mistake, and usually does not last for long. It is reported in logs. If fixed by a later merge, the buildmaster administrator does not have anything to do: that merge will be transmitted, together with the intermediate ones.

The *HgPoller* accepts the following arguments:

name the name of the poller. This must be unique, and defaults to the `repourl`.

repourl the url that describes the remote repository, e.g. `http://hg.example.com/projects/myrepo`. Any url suitable for `hg pull` can be specified.

branch the desired branch to pull, will default to `'default'`

workdir the directory where the poller should keep its local repository. It is mandatory for now, although later releases may provide a meaningful default.

It also serves to identify the poller in the buildmaster internal database. Changing it may result in re-processing all changes so far.

Several *HgPoller* instances may share the same `workdir` for mutualisation of the common history between two different branches, thus easing on local and remote system resources and bandwidth.

If relative, the `workdir` will be interpreted from the master directory.

pollInterval interval in seconds between polls, default is 10 minutes

pollAtLaunch Determines when the first poll occurs. True = immediately on launch, False = wait for one pollInterval (default).

hgbin path to the Mercurial binary, defaults to just `'hg'`

category Set the category to be used for the changes produced by the *HgPoller*. This will then be set in any changes generated by the *HgPoller*, and can be used in a Change Filter for triggering particular builders.

project Set the name of the project to be used for the *HgPoller*. This will then be set in any changes generated by the `HgPoller`, and can be used in a Change Filter for triggering particular builders.

usetimestamps parse each revision's commit timestamp (default is `True`), or ignore it in favor of the current time (so recently processed commits appear together in the waterfall page)

encoding Set encoding will be used to parse author's name and commit message. Default encoding is `'utf-8'`.

A configuration for the Mercurial poller might look like this:

```
from buildbot.plugins import changes

c['change_source'] = changes.HgPoller(repourl='http://hg.example.org/projects/myrepo',
                                       branch='great_new_feature',
                                       workdir='hg-myrepo')
```

GitHubPullrequestPoller

class `buildbot.changes.github.`**GitHubPullrequestPoller**

This *GitHubPullrequestPoller* periodically polls the GitHub API for new or updated pull requests. The *author*, *revision*, *revlink*, *branch* and *files* fields in the recorded changes are populated with information extracted from the pull request. This allows to filter for certain changes in files and create a blamelist based on the authors in the GitHub pull request.

The *GitHubPullrequestPoller* accepts the following arguments:

owner The owner of the GitHub repository. This argument is required.

repo The name of the GitHub repository. This argument is required.

branches List of branches to accept as base branch (e.g. master). Defaults to *None* and accepts all branches as base.

pollInterval Poll interval between polls in seconds. Default is 10 minutes.

pollAtLaunch Whether to poll on startup of the buildbot master. Default is *False* and first poll will occur *pollInterval* seconds after the master start.

category Set the category to be used for the changes produced by the *GitHubPullrequestPoller*. This will then be set in any changes generated by the *GitHubPullrequestPoller*, and can be used in a Change Filter for triggering particular builders.

project Set the name of the project to be used for the *GitHubPullrequestPoller*. This will then be set in any changes generated by the *GitHubPullrequestPoller*, and can be used in a Change Filter for triggering particular builders.

baseURL GitHub API endpoint. Default is `https://api.github.com`.

pullrequest_filter A callable which takes a *dict* which contains the decoded *JSON* object of the GitHub pull request as argument. All fields specified by the GitHub API are accessible. If the callable returns *False* the pull request is ignored. Default is *True* which does not filter any pull requests.

token A GitHub API token to execute all requests to the API authenticated. It is strongly recommended to use a API token since it increases GitHub API rate limits significantly.

repository_type Set which type of repository link will be in the *repository* property. Possible values `https`, `svn`, `git` or `svn`. This link can then be used in a Source Step to checkout the source.

magic_link Set to *True* if the changes should contain `refs/pulls/<PR #>/merge` in the *branch* property and a link to the base *repository* in the repository property. These properties can be used by the *GitHub* source to pull from the special branch in the base repository. Default is *False*.

BitbucketPullrequestPoller

class `buildbot.changes.bitbucket.`**`BitbucketPullrequestPoller`**

This *BitbucketPullrequestPoller* periodically polls Bitbucket for new or updated pull requests. It uses Bitbuckets powerful Pull Request REST API (https://confluence.atlassian.com/display/BITBUCKET/pullrequests+Resource) to gather the information needed.

The *BitbucketPullrequestPoller* accepts the following arguments:

owner The owner of the Bitbucket repository. All Bitbucket Urls are of the form `https://bitbucket.org/owner/slug/`.

slug The name of the Bitbucket repository.

branch A single branch or a list of branches which should be processed. If it is `None` (the default) all pull requests are used.

pollInterval Interval in seconds between polls, default is 10 minutes.

pollAtLaunch Determines when the first poll occurs. `True` = immediately on launch, `False` = wait for one `pollInterval` (default).

category Set the category to be used for the changes produced by the *BitbucketPullrequestPoller*. This will then be set in any changes generated by the *BitbucketPullrequestPoller*, and can be used in a Change Filter for triggering particular builders.

project Set the name of the project to be used for the *BitbucketPullrequestPoller*. This will then be set in any changes generated by the `BitbucketPullrequestPoller`, and can be used in a Change Filter for triggering particular builders.

pullrequest_filter A callable which takes one parameter, the decoded Python object of the pull request JSON. If the it returns `False` the pull request is ignored. It can be used to define custom filters based on the content of the pull request. See the Bitbucket documentation for more information about the format of the response. By default the filter always returns `True`.

usetimestamps parse each revision's commit timestamp (default is `True`), or ignore it in favor of the current time (so recently processed commits appear together in the waterfall page)

encoding Set encoding will be used to parse author's name and commit message. Default encoding is `'utf-8'`.

A minimal configuration for the Bitbucket pull request poller might look like this:

```
from buildbot.plugins import changes

c['change_source'] = changes.BitbucketPullrequestPoller(
    owner='myname',
    slug='myrepo',
  )
```

Here is a more complex configuration using a `pullrequest_filter`. The pull request is only processed if at least 3 people have already approved it:

```
def approve_filter(pr, threshold):
    approves = 0
    for participant in pr['participants']:
        if participant['approved']:
            approves = approves + 1

    if approves < threshold:
        return False
    return True

from buildbot.plugins import changes
c['change_source'] = changes.BitbucketPullrequestPoller(
    owner='myname',
    slug='myrepo',
    branch='mybranch',
    project='myproject',
    pullrequest_filter=lambda pr : approve_filter(pr,3),
    pollInterval=600,
)
```

Warning: Anyone who can create pull requests for the Bitbucket repository can initiate a change, potentially causing the buildmaster to run arbitrary code.

GerritChangeSource

class `buildbot.changes.gerritchangesource.`**`GerritChangeSource`**

The *`GerritChangeSource`* class connects to a Gerrit server by its SSH interface and uses its event source mechanism, gerrit stream-events (https://gerrit-documentation.storage.googleapis.com/Documentation/2.2.1/cmd-stream-events.html).

The *`GerritChangeSource`* accepts the following arguments:

`gerritserver` the dns or ip that host the Gerrit ssh server

`gerritport` the port of the Gerrit ssh server

`username` the username to use to connect to Gerrit

`identity_file` ssh identity file to for authentication (optional). Pay attention to the *ssh passphrase*

`handled_events` event to be handled (optional). By default processes *patchset-created* and *ref-updated*

`debug` Print Gerrit event in the log (default *False*). This allows to debug event content, but will eventually fill your logs with useless Gerrit event logs.

By default this class adds a change to the Buildbot system for each of the following events:

`patchset-created` A change is proposed for review. Automatic checks like `checkpatch.pl` can be automatically triggered. Beware of what kind of automatic task you trigger. At this point, no trusted human has reviewed the code, and a patch could be specially crafted by an attacker to compromise your workers.

`ref-updated` A change has been merged into the repository. Typically, this kind of event can lead to a complete rebuild of the project, and upload binaries to an incremental build results server.

But you can specify how to handle events:

- Any event with change and patchSet will be processed by universal collector by default.
- In case you've specified processing function for the given kind of events, all events of this kind will be processed only by this function, bypassing universal collector.

An example:

```
from buildbot.plugins import changes

class MyGerritChangeSource(changes.GerritChangeSource):
    """Custom GerritChangeSource
    """
    def eventReceived_patchset_created(self, properties, event):
        """Handler events without properties
        """
        properties = {}
        self.addChangeFromEvent(properties, event)
```

This class will populate the property list of the triggered build with the info received from Gerrit server in JSON format.

Warning: If you selected `GerritChangeSource`, you **must** use *`Gerrit`* source step: the `branch` property of the change will be *`target_branch/change_id`* and such a ref cannot be resolved, so the *`Git`* source step would fail.

In case of `patchset-created` event, these properties will be:

`event.change.branch` Branch of the Change

`event.change.id` Change's ID in the Gerrit system (the ChangeId: in commit comments)

`event.change.number` Change's number in Gerrit system

`event.change.owner.email` Change's owner email (owner is first uploader)

`event.change.owner.name` Change's owner name

`event.change.project` Project of the Change

`event.change.subject` Change's subject

`event.change.url` URL of the Change in the Gerrit's web interface

`event.patchSet.number` Patchset's version number

`event.patchSet.ref` Patchset's Gerrit "virtual branch"

`event.patchSet.revision` Patchset's Git commit ID

`event.patchSet.uploader.email` Patchset uploader's email (owner is first uploader)

`event.patchSet.uploader.name` Patchset uploader's name (owner is first uploader)

`event.type` Event type (`patchset-created`)

`event.uploader.email` Patchset uploader's email

`event.uploader.name` Patchset uploader's name

In case of `ref-updated` event, these properties will be:

`event.refUpdate.newRev` New Git commit ID (after merger)

`event.refUpdate.oldRev` Previous Git commit ID (before merger)

event.refUpdate.project Project that was updated

event.refUpdate.refName Branch that was updated

event.submitter.email Submitter's email (merger responsible)

event.submitter.name Submitter's name (merger responsible)

event.type Event type (`ref-updated`)

event.submitter.email Submitter's email (merger responsible)

event.submitter.name Submitter's name (merger responsible)

A configuration for this source might look like:

```
from buildbot.plugins import changes

c['change_source'] = changes.GerritChangeSource(
    "gerrit.example.com",
    "gerrit_user",
    handled_events=["patchset-created", "change-merged"])
```

See `master/docs/examples/git_gerrit.cfg` or `master/docs/examples/repo_gerrit.cfg` in the Buildbot distribution for a full example setup of Git+Gerrit or Repo+Gerrit of *GerritChangeSource*.

GerritEventLogPoller

class `buildbot.changes.gerritchangesource.`**GerritEventLogPoller**

The *GerritEventLogPoller* class is similar to *GerritChangeSource* but connects to the Gerrit server by its HTTP interface and uses the events-log (https://gerrit.googlesource.com/plugins/events-log/) plugin.

The *GerritEventLogPoller* accepts the following arguments:

baseURL the HTTP url where to find Gerrit

auth a requests authentication configuration. if Gerrit is configured with `BasicAuth`, then it shall be `('login', 'password')` if Gerrit is configured with `DigestAuth`, then it shall be `requests.auth.HTTPDigestAuth('login', 'password')` from the requests module.

handled_events event to be handled (optional). By default processes *patchset-created* and *ref-updated*

pollInterval interval in seconds between polls, default is 30 seconds

pollAtLaunch Determines when the first poll occurs. True = immediately on launch (default), False = wait for one pollInterval.

gitBaseURL The git URL where Gerrit is accessible via git+ssh protocol

debug Print Gerrit event in the log (default *False*). This allows to debug event content, but will eventually fill your logs with useless Gerrit event logs.

The same customization can be done as *GerritChangeSource* for handling special events.

GerritChangeFilter

class `buildbot.changes.gerritchangesource.`**GerritChangeFilter**

`GerritChangeFilter` is a ready to use `ChangeFilter` you can pass to *AnyBranchScheduler* in order to filter changes, to create pre-commit builders or post-commit schedulers. It has the same api as *Change Filter*, except it has additional *eventtype* set of filter (can as well be specified as value, list, regular expression or callable)

An example is following:

```
from buildbot.plugins import schedulers, util

# this scheduler will create builds when a patch is uploaded to gerrit
# but only if it is uploaded to the "main" branch
schedulers.AnyBranchScheduler(name="main-precommit",
                              change_filter=util.GerritChangeFilter(branch="main",
                                                                    eventtype=
↪"patchset-created"),
                              treeStableTimer=15*60,
                              builderNames=["main-precommit"])

# this scheduler will create builds when a patch is merged in the "main" branch
# for post-commit tests
schedulers.AnyBranchScheduler(name="main-postcommit",
                              change_filter=util.GerritChangeFilter("main", "ref-
↪updated"),
                              treeStableTimer=15*60,
                              builderNames=["main-postcommit"])
```

Change Hooks (HTTP Notifications)

Buildbot already provides a web frontend, and that frontend can easily be used to receive HTTP push notifications of commits from services like GitHub. See *Change Hooks* for more information.

Schedulers

Schedulers are responsible for initiating builds on builders.

Some schedulers listen for changes from ChangeSources and generate build sets in response to these changes. Others generate build sets without changes, based on other events in the buildmaster.

Configuring Schedulers

The *schedulers* configuration parameter gives a list of scheduler instances, each of which causes builds to be started on a particular set of Builders. The two basic scheduler classes you are likely to start with are *SingleBranchScheduler* and *Periodic*, but you can write a customized subclass to implement more complicated build scheduling.

Scheduler arguments should always be specified by name (as keyword arguments), to allow for future expansion:

```
sched = SingleBranchScheduler(name="quick", builderNames=['lin', 'win'])
```

There are several common arguments for schedulers, although not all are available with all schedulers.

name Each Scheduler must have a unique name. This is used in status displays, and is also available in the build property `scheduler`.

builderNames This is the set of builders which this scheduler should trigger, specified as a list of names (strings).

properties This is a dictionary specifying properties that will be transmitted to all builds started by this scheduler. The `owner` property may be of particular interest, as its contents (as a list) will be added to the list of "interested users" (*Doing Things With Users*) for each triggered build. For example

```
sched = Scheduler(...,
    properties = {
        'owner': ['zorro@example.com', 'silver@example.com']
    })
```

fileIsImportant A callable which takes one argument, a Change instance, and returns `True` if the change is worth building, and `False` if it is not. Unimportant Changes are accumulated until the build is triggered by an important change. The default value of None means that all Changes are important.

change_filter The change filter that will determine which changes are recognized by this scheduler; *Change Filters*. Note that this is different from `fileIsImportant`: if the change filter filters out a Change, then it is completely ignored by the scheduler. If a Change is allowed by the change filter, but is deemed unimportant, then it will not cause builds to start, but will be remembered and shown in status displays.

codebases When the scheduler processes data from more than one repository at the same time, a corresponding codebase definition should be passed for each repository.

This parameter can be specified either as a list of strings (simplest form; use if no special overrides are needed) or as a dictionary of dictionaries (where each dict is a codebase definition as described next).

Each codebase definition is a dictionary with any of the keys: `repository`, `branch`, `revision`. The codebase definitions are combined in a dictionary keyed by the name of the codebase.

```
codebases = {'codebase1': {'repository':'....',
                           'branch':'default',
                           'revision': None},
             'codebase2': {'repository':'....'} }
```

Important: The `codebases` parameter is only used to fill in missing details about a codebases when scheduling a build. For example, when a change to codebase `A` occurs, a scheduler must invent a sourcestamp for codebase `B`. The parameter does not act as a filter on incoming changes – use a change filter for that purpose.

Source steps can specify a codebase to which they will apply, and will use the sourcestamp for that codebase.

onlyImportant A boolean that, when `True`, only adds important changes to the buildset as specified in the `fileIsImportant` callable. This means that unimportant changes are ignored the same way a

`change_filter` filters changes. This defaults to `False` and only applies when `fileIsImportant` is given.

reason A string that will be used as the reason for the triggered build.

The remaining subsections represent a catalog of the available scheduler types. All these schedulers are defined in modules under `buildbot.schedulers`, and the docstrings there are the best source of documentation on the arguments taken by each one.

Scheduler Resiliency

In a multi-master configuration, schedulers with the same name can be configured on multiple masters. Only one instance of the scheduler will be active. If that instance becomes inactive, due to its master being shut down or failing, then another instance will become active after a short delay. This provides resiliency in scheduler configurations, so that schedulers are not a single point of failure in a Buildbot infrastructure.

The Data API and web UI display the master on which each scheduler is running.

There is currently no mechanism to control which master's scheduler instance becomes active. The behavior is non-deterministic, based on the timing of polling by inactive schedulers. The failover is non-revertive.

Change Filters

Several schedulers perform filtering on an incoming set of changes. The filter can most generically be specified as a `ChangeFilter`. Set up a `ChangeFilter` like this:

```
from buildbot.plugins import util
my_filter = util.ChangeFilter(project_re="^baseproduct/.*", branch="devel")
```

and then add it to a scheduler with the `change_filter` parameter:

```
sch = SomeSchedulerClass(...,
    change_filter=my_filter)
```

There are five attributes of changes on which you can filter:

project the project string, as defined by the ChangeSource.

repository the repository in which this change occurred.

branch the branch on which this change occurred. Note that 'trunk' or 'master' is often denoted by `None`.

category the category, again as defined by the ChangeSource.

codebase the change's codebase.

For each attribute, the filter can look for a single, specific value:

```
my_filter = util.ChangeFilter(project='myproject')
```

or accept any of a set of values:

```
my_filter = util.ChangeFilter(project=['myproject', 'jimsproject'])
```

or apply a regular expression, using the attribute name with a "`_re`" suffix:

```
my_filter = util.ChangeFilter(category_re='.*deve.*')
# or, to use regular expression flags:
import re
my_filter = util.ChangeFilter(category_re=re.compile('.*deve.*', re.I))
```

`buildbot.status.web.hooks.github.GitHubEventHandler` has a special `github_distinct` property that can be used to filter whether or not non-distinct changes should be considered. For example, if a commit is pushed to a branch that is not being watched and then later pushed to a watched branch, by default, this will be recorded as two separate Changes. In order to record a change only the first time the commit appears, you can install a custom `ChangeFilter` like this:

```
ChangeFilter(filter_fn = lambda c: c.properties.getProperty('github_distinct')
```

For anything more complicated, define a Python function to recognize the strings you want:

```
def my_branch_fn(branch):
    return branch in branches_to_build and branch not in branches_to_ignore
my_filter = util.ChangeFilter(branch_fn=my_branch_fn)
```

The special argument `filter_fn` can be used to specify a function that is given the entire Change object, and returns a boolean.

The entire set of allowed arguments, then, is

project	project_re	project_fn
repository	repository_re	repository_fn
branch	branch_re	branch_fn
category	category_re	category_fn
codebase	codebase_re	codebase_fn
filter_fn		

A Change passes the filter only if *all* arguments are satisfied. If no filter object is given to a scheduler, then all changes will be built (subject to any other restrictions the scheduler enforces).

Scheduler Types

The remaining subsections represent a catalog of the available Scheduler types. All these Schedulers are defined in modules under `buildbot.schedulers`, and the docstrings there are the best source of documentation on the arguments taken by each one.

SingleBranchScheduler

This is the original and still most popular scheduler class. It follows exactly one branch, and starts a configurable tree-stable-timer after each change on that branch. When the timer expires, it starts a build on some set of Builders. This scheduler accepts a `fileIsImportant` function which can be used to ignore some Changes if they do not affect any *important* files.

If `treeStableTimer` is not set, then this scheduler starts a build for every Change that matches its `change_filter` and statsfies `fileIsImportant`. If `treeStableTimer` is set, then a build is triggered for each set of Changes which arrive within the configured time, and match the filters.

Note: The behavior of this scheduler is undefined, if `treeStableTimer` is set, and changes from multiple branches, repositories or codebases are accepted by the filter.

Note: The `codebases` argument will filter out codebases not specified there, but *won't* filter based on the branches specified there.

The arguments to this scheduler are:

`name`

`builderNames`

`properties`

`fileIsImportant`

`change_filter`

`onlyImportant`

`reason`

`treeStableTimer` The scheduler will wait for this many seconds before starting the build. If new changes are made during this interval, the timer will be restarted, so really the build will be started after a change and then after this many seconds of inactivity.

If `treeStableTimer` is `None`, then a separate build is started immediately for each Change.

`fileIsImportant` A callable which takes one argument, a Change instance, and returns `True` if the change is worth building, and `False` if it is not. Unimportant Changes are accumulated until the build is triggered by an important change. The default value of None means that all Changes are important.

`categories` (deprecated; use change_filter) A list of categories of changes that this scheduler will respond to. If this is specified, then any non-matching changes are ignored.

`branch` (deprecated; use change_filter) The scheduler will pay attention to this branch, ignoring Changes that occur on other branches. Setting `branch` equal to the special value of `None` means it should only pay attention to the default branch.

Note: `None` is a keyword, not a string, so write `None` and not `"None"`.

Example:

```
from buildbot.plugins import schedulers, util
quick = schedulers.SingleBranchScheduler(
            name="quick",
            change_filter=util.ChangeFilter(branch='master'),
            treeStableTimer=60,
            builderNames=["quick-linux", "quick-netbsd"])
full = schedulers.SingleBranchScheduler(
            name="full",
            change_filter=util.ChangeFilter(branch='master'),
            treeStableTimer=5*60,
            builderNames=["full-linux", "full-netbsd", "full-OSX"])
c['schedulers'] = [quick, full]
```

In this example, the two *quick* builders are triggered 60 seconds after the tree has been changed. The *full* builds do not run quite so quickly (they wait 5 minutes), so hopefully if the quick builds fail due to a missing file or really simple typo, the developer can discover and fix the problem before the full builds are started. Both schedulers only pay attention to the default branch: any changes on other branches are ignored. Each scheduler triggers a different set of Builders, referenced by name.

Note: The old names for this scheduler, `buildbot.scheduler.Scheduler` and `buildbot.schedulers.basic.Scheduler`, are deprecated in favor of using `buildbot.plugins`:

```
from buildbot.plugins import schedulers
```

However if you must use a fully qualified name, it is `buildbot.schedulers.basic.SingleBranchScheduler`.

AnyBranchScheduler

This scheduler uses a tree-stable-timer like the default one, but uses a separate timer for each branch.

If `treeStableTimer` is not set, then this scheduler is indistinguishable from *`SingleBranchScheduler`*. If `treeStableTimer` is set, then a build is triggered for each set of Changes which arrive within the configured time, and match the filters.

The arguments to this scheduler are:

`name`

`builderNames`

`properties`

`fileIsImportant`

`change_filter`

`onlyImportant`

`reason` See *Configuring Schedulers*.

`treeStableTimer` The scheduler will wait for this many seconds before starting the build. If new changes are made *on the same branch* during this interval, the timer will be restarted.

`branches` (deprecated; use change_filter) Changes on branches not specified on this list will be ignored.

`categories` (deprecated; use change_filter) A list of categories of changes that this scheduler will respond to. If this is specified, then any non-matching changes are ignored.

Dependent Scheduler

It is common to wind up with one kind of build which should only be performed if the same source code was successfully handled by some other kind of build first. An example might be a packaging step: you might only want to produce .deb or RPM packages from a tree that was known to compile successfully and pass all unit tests. You could put the packaging step in the same Build as the compile and testing steps, but there might be other reasons to not do this (in particular you might have several Builders worth of compiles/tests, but only wish to do the packaging once). Another example is if you want to skip the *full* builds after a failing *quick* build of the same source code. Or, if one Build creates a product (like a compiled library) that is used by some other Builder, you'd want to make sure the consuming Build is run *after* the producing one.

You can use *dependencies* to express this relationship to the Buildbot. There is a special kind of scheduler named *`Dependent`* that will watch an *upstream* scheduler for builds to complete successfully (on all of its Builders). Each time that happens, the same source code (i.e. the same `SourceStamp`) will be used to start a new set of builds, on a different set of Builders. This *downstream* scheduler doesn't pay attention to Changes at all. It only pays attention to the upstream scheduler.

If the build fails on any of the Builders in the upstream set, the downstream builds will not fire. Note that, for SourceStamps generated by a *Dependent* scheduler, the `revision` is `None`, meaning HEAD. If any changes are committed between the time the upstream scheduler begins its build and the time the dependent scheduler begins its build, then those changes will be included in the downstream build. See the *Triggerable* scheduler for a more flexible dependency mechanism that can avoid this problem.

The keyword arguments to this scheduler are:

`name`

`builderNames`

properties See *Configuring Schedulers*.

upstream The upstream scheduler to watch. Note that this is an *instance*, not the name of the scheduler.

Example:

```
from buildbot.plugins import schedulers
tests = schedulers.SingleBranchScheduler(name="just-tests",
                                         treeStableTimer=5*60,
                                         builderNames=["full-linux",
                                                       "full-netbsd",
                                                       "full-OSX"])
package = schedulers.Dependent(name="build-package",
                               upstream=tests, # <- no quotes!
                               builderNames=["make-tarball", "make-deb",
                                             "make-rpm"])
c['schedulers'] = [tests, package]
```

Periodic Scheduler

This simple scheduler just triggers a build every *N* seconds.

The arguments to this scheduler are:

`name`

`builderNames`

`properties`

`onlyImportant`

createAbsoluteSourceStamps This option only has effect when using multiple codebases. When `True`, it uses the last seen revision for each codebase that does not have a change. When `False`, the default value, codebases without changes will use the revision from the `codebases` argument.

onlyIfChanged If this is true, then builds will not be scheduled at the designated time *unless* the specified branch has seen an important change since the previous build.

reason See *Configuring Schedulers*.

periodicBuildTimer The time, in seconds, after which to start a build.

Example:

```
from buildbot.plugins import schedulers
nightly = schedulers.Periodic(name="daily",
                              builderNames=["full-solaris"],
                              periodicBuildTimer=24*60*60)
c['schedulers'] = [nightly]
```

The scheduler in this example just runs the full solaris build once per day. Note that this scheduler only lets you control the time between builds, not the absolute time-of-day of each Build, so this could easily wind up an *evening* or *every afternoon* scheduler depending upon when it was first activated.

Nightly Scheduler

This is highly configurable periodic build scheduler, which triggers a build at particular times of day, week, month, or year. The configuration syntax is very similar to the well-known `crontab` format, in which you provide values for minute, hour, day, and month (some of which can be wildcards), and a build is triggered whenever the current time matches the given constraints. This can run a build every night, every morning, every weekend, alternate Thursdays, on your boss's birthday, etc.

Pass some subset of `minute`, `hour`, `dayOfMonth`, `month`, and `dayOfWeek`; each may be a single number or a list of valid values. The builds will be triggered whenever the current time matches these values. Wildcards are represented by a '*' string. All fields default to a wildcard except 'minute', so with no fields this defaults to a build every hour, on the hour. The full list of parameters is:

`name`

`builderNames`

`properties`

`fileIsImportant`

`change_filter`

`onlyImportant`

`reason`

`codebases`

`createAbsoluteSourceStamps` This option only has effect when using multiple codebases. When `True`, it uses the last seen revision for each codebase that does not have a change. When `False`, the default value, codebases without changes will use the revision from the `codebases` argument.

`onlyIfChanged` If this is true, then builds will not be scheduled at the designated time *unless* the change filter has accepted an important change since the previous build.

`branch` (deprecated; use `change_filter` and `codebases`) The branch to build when the time comes, and the branch to filter for if `change_filter` is not specified. Remember that a value of `None` here means the default branch, and will not match other branches!

`minute` The minute of the hour on which to start the build. This defaults to 0, meaning an hourly build.

`hour` The hour of the day on which to start the build, in 24-hour notation. This defaults to *, meaning every hour.

`dayOfMonth` The day of the month to start a build. This defaults to $\star$, meaning every day.

`month` The month in which to start the build, with January = 1. This defaults to $\star$, meaning every month.

`dayOfWeek` The day of the week to start a build, with Monday = 0. This defaults to $\star$, meaning every day of the week.

For example, the following `master.cfg` clause will cause a build to be started every night at 3:00am:

```
from buildbot.plugins import schedulers
c['schedulers'].append(
    schedulers.Nightly(name='nightly',
                       branch='master',
```

```
                        builderNames=['builder1', 'builder2'],
                        hour=3, minute=0))
```

This scheduler will perform a build each Monday morning at 6:23am and again at 8:23am, but only if someone has committed code in the interim:

```
c['schedulers'].append(
    schedulers.Nightly(name='BeforeWork',
                        branch=`default`,
                        builderNames=['builder1'],
                        dayOfWeek=0, hour=[6,8], minute=23,
                        onlyIfChanged=True))
```

The following runs a build every two hours, using Python's `range` function:

```
c.schedulers.append(
    timed.Nightly(name='every2hours',
        branch=None, # default branch
        builderNames=['builder1'],
        hour=range(0, 24, 2)))
```

Finally, this example will run only on December 24th:

```
c['schedulers'].append(
    timed.Nightly(name='SleighPreflightCheck',
        branch=None, # default branch
        builderNames=['flying_circuits', 'radar'],
        month=12,
        dayOfMonth=24,
        hour=12,
        minute=0))
```

Try Schedulers

This scheduler allows developers to use the **`buildbot try`** command to trigger builds of code they have not yet committed. See *`try`* for complete details.

Two implementations are available: *`Try_Jobdir`* and *`Try_Userpass`*. The former monitors a job directory, specified by the `jobdir` parameter, while the latter listens for PB connections on a specific `port`, and authenticates against `userport`.

The buildmaster must have a scheduler instance in the config file's *`schedulers`* list to receive try requests. This lets the administrator control who may initiate these *trial* builds, which branches are eligible for trial builds, and which Builders should be used for them.

The scheduler has various means to accept build requests. All of them enforce more security than the usual buildmaster ports do. Any source code being built can be used to compromise the worker accounts, but in general that code must be checked out from the VC repository first, so only people with commit privileges can get control of the workers. The usual force-build control channels can waste worker time but do not allow arbitrary commands to be executed by people who don't have those commit privileges. However, the source code patch that is provided with the trial build does not have to go through the VC system first, so it is important to make sure these builds cannot be abused by a non-committer to acquire as much control over the workers as a committer has. Ideally, only developers who have commit access to the VC repository would be able to start trial builds, but unfortunately the buildmaster does not, in general, have access to VC system's user list.

As a result, the try scheduler requires a bit more configuration. There are currently two ways to set this up:

jobdir (ssh) This approach creates a command queue directory, called the `jobdir`, in the buildmaster's working directory. The buildmaster admin sets the ownership and permissions of this directory to only grant write access to the desired set of developers, all of whom must have accounts on the machine. The **`buildbot try`** command creates a special file containing the source stamp information and drops it in the jobdir, just like a standard maildir. When the buildmaster notices the new file, it unpacks the information inside and starts the builds.

The config file entries used by 'buildbot try' either specify a local queuedir (for which write and mv are used) or a remote one (using scp and ssh).

The advantage of this scheme is that it is quite secure, the disadvantage is that it requires fiddling outside the buildmaster config (to set the permissions on the jobdir correctly). If the buildmaster machine happens to also house the VC repository, then it can be fairly easy to keep the VC userlist in sync with the trial-build userlist. If they are on different machines, this will be much more of a hassle. It may also involve granting developer accounts on a machine that would not otherwise require them.

To implement this, the worker invokes `ssh -l` *`username host`* `buildbot tryserver` *`ARGS`*, passing the patch contents over stdin. The arguments must include the inlet directory and the revision information.

user+password (PB) In this approach, each developer gets a username/password pair, which are all listed in the buildmaster's configuration file. When the developer runs **`buildbot try`**, their machine connects to the buildmaster via PB and authenticates themselves using that username and password, then sends a PB command to start the trial build.

The advantage of this scheme is that the entire configuration is performed inside the buildmaster's config file. The disadvantages are that it is less secure (while the *cred* authentication system does not expose the password in plaintext over the wire, it does not offer most of the other security properties that SSH does). In addition, the buildmaster admin is responsible for maintaining the username/password list, adding and deleting entries as developers come and go.

For example, to set up the *jobdir* style of trial build, using a command queue directory of *`MASTERDIR`*`/jobdir` (and assuming that all your project developers were members of the `developers` unix group), you would first set up that directory:

```
mkdir -p MASTERDIR/jobdir MASTERDIR/jobdir/new MASTERDIR/jobdir/cur MASTERDIR/jobdir/
↪tmp
chgrp developers MASTERDIR/jobdir MASTERDIR/jobdir/*
chmod g+rwx,o-rwx MASTERDIR/jobdir MASTERDIR/jobdir/*
```

and then use the following scheduler in the buildmaster's config file:

```
from buildbot.plugins import schedulers
s = schedulers.Try_Jobdir(name="try1",
                          builderNames=["full-linux", "full-netbsd",
                                        "full-OSX"],
                          jobdir="jobdir")
c['schedulers'] = [s]
```

Note that you must create the jobdir before telling the buildmaster to use this configuration, otherwise you will get an error. Also remember that the buildmaster must be able to read and write to the jobdir as well. Be sure to watch the `twistd.log` file (*Logfiles*) as you start using the jobdir, to make sure the buildmaster is happy with it.

Note: Patches in the jobdir are encoded using netstrings, which place an arbitrary upper limit on patch size of 99999 bytes. If your submitted try jobs are rejected with *BadJobfile*, try increasing this limit with a snippet like this in your *master.cfg*:

```
from twisted.protocols.basic import NetstringReceiver
NetstringReceiver.MAX_LENGTH = 1000000
```

To use the username/password form of authentication, create a `Try_Userpass` instance instead. It takes the same `builderNames` argument as the `Try_Jobdir` form, but accepts an additional `port` argument (to specify the TCP port to listen on) and a `userpass` list of username/password pairs to accept. Remember to use good passwords for this: the security of the worker accounts depends upon it:

```
from buildbot.plugins import schedulers
s = schedulers.Try_Userpass(name="try2",
                            builderNames=["full-linux", "full-netbsd",
                                          "full-OSX"],
                            port=8031,
                            userpass=[("alice","pw1"), ("bob", "pw2")])
c['schedulers'] = [s]
```

Like most places in the buildbot, the `port` argument takes a *strports* specification. See `twisted.application.strports` for details.

Triggerable Scheduler

The *Triggerable* scheduler waits to be triggered by a *Trigger* step (see *Triggering Schedulers*) in another build. That step can optionally wait for the scheduler's builds to complete. This provides two advantages over *Dependent* schedulers. First, the same scheduler can be triggered from multiple builds. Second, the ability to wait for *Triggerable*'s builds to complete provides a form of "subroutine call", where one or more builds can "call" a scheduler to perform some work for them, perhaps on other workers. The *Triggerable* scheduler supports multiple codebases. The scheduler filters out all codebases from *Trigger* steps that are not configured in the scheduler.

The parameters are just the basics:

`name`

`builderNames`

`properties`

`reason`

`codebases` See *Configuring Schedulers*.

This class is only useful in conjunction with the *Trigger* step. Here is a fully-worked example:

```
from buildbot.plugins import schedulers, util, steps

checkin = schedulers.SingleBranchScheduler(name="checkin",
                                           branch=None,
                                           treeStableTimer=5*60,
                                           builderNames=["checkin"])
nightly = schedulers.Nightly(name='nightly',
                             branch=None,
                             builderNames=['nightly'],
                             hour=3, minute=0)

mktarball = schedulers.Triggerable(name="mktarball", builderNames=["mktarball"])
build = schedulers.Triggerable(name="build-all-platforms",
                               builderNames=["build-all-platforms"])
test = schedulers.Triggerable(name="distributed-test",
                              builderNames=["distributed-test"])
package = schedulers.Triggerable(name="package-all-platforms",
```

```
                                    builderNames=["package-all-platforms"])
c['schedulers'] = [mktarball, checkin, nightly, build, test, package]

# on checkin, make a tarball, build it, and test it
checkin_factory = util.BuildFactory()
checkin_factory.addStep(steps.Trigger(schedulerNames=['mktarball'],
                                      waitForFinish=True))
checkin_factory.addStep(steps.Trigger(schedulerNames=['build-all-platforms'],
                                      waitForFinish=True))
checkin_factory.addStep(steps.Trigger(schedulerNames=['distributed-test'],
                                      waitForFinish=True))

# and every night, make a tarball, build it, and package it
nightly_factory = util.BuildFactory()
nightly_factory.addStep(steps.Trigger(schedulerNames=['mktarball'],
                                      waitForFinish=True))
nightly_factory.addStep(steps.Trigger(schedulerNames=['build-all-platforms'],
                                      waitForFinish=True))
nightly_factory.addStep(steps.Trigger(schedulerNames=['package-all-platforms'],
                                      waitForFinish=True))
```

NightlyTriggerable Scheduler

class buildbot.schedulers.timed.**NightlyTriggerable**

The *NightlyTriggerable* scheduler is a mix of the *Nightly* and *Triggerable* schedulers. This scheduler triggers builds at a particular time of day, week, or year, exactly as the *Nightly* scheduler. However, the source stamp set that is used that provided by the last *Trigger* step that targeted this scheduler.

The parameters are just the basics:

name

builderNames

properties

codebases See *Configuring Schedulers*.

minute

hour

dayOfMonth

month

dayOfWeek See *Nightly*.

This class is only useful in conjunction with the *Trigger* step. Note that waitForFinish is ignored by *Trigger* steps targeting this scheduler.

Here is a fully-worked example:

```
from buildbot.plugins import schedulers, util, steps

checkin = schedulers.SingleBranchScheduler(name="checkin",
                                           branch=None,
                                           treeStableTimer=5*60,
                                           builderNames=["checkin"])
nightly = schedulers.NightlyTriggerable(name='nightly',
```

```
                                        builderNames=['nightly'],
                                        hour=3, minute=0)
c['schedulers'] = [checkin, nightly]

# on checkin, run tests
checkin_factory = util.BuildFactory([
    steps.Test(),
    steps.Trigger(schedulerNames=['nightly'])
])

# and every night, package the latest successful build
nightly_factory = util.BuildFactory([
    steps.ShellCommand(command=['make', 'package'])
])
```

ForceScheduler Scheduler

The *ForceScheduler* scheduler is the way you can configure a force build form in the web UI.

In the `/#/builders/:builderid` web page, you will see, on the top right of the page, one button for each *ForceScheduler* scheduler that was configured for this builder. If you click on that button, a dialog will let you choose various parameters for requesting a new build.

The Buildbot framework allows you to customize exactly how the build form looks, which builders have a force build form (it might not make sense to force build every builder), and who is allowed to force builds on which builders.

How you do so is by configuring a *ForceScheduler*, and add it into the list *schedulers*.

The scheduler takes the following parameters:

`name`

Name of the scheduler (should be an *Identifier*).

`builderNames`

List of builders where the force button should appear. See *Configuring Schedulers*.

`reason`

A *parameter* allowing the user to specify the reason for the build. The default value is a string parameter with a default value "force build".

`reasonString`

A string that will be used to create the build reason for the forced build. This string can contain the placeholders `%(owner)s` and `%(reason)s`, which represents the value typed into the reason field.

`username`

A *parameter* specifying the username associated with the build (aka owner). The default value is a username parameter.

`codebases`

A list of strings or *CodebaseParameter* specifying the codebases that should be presented. The default is a single codebase with no name (i.e. *codebases=['']*).

`properties`

A list of *parameters*, one for each property. These can be arbitrary parameters, where the parameter's name is taken as the property name, or `AnyPropertyParameter`, which allows the web user to specify the property name. The default value is an empty list.

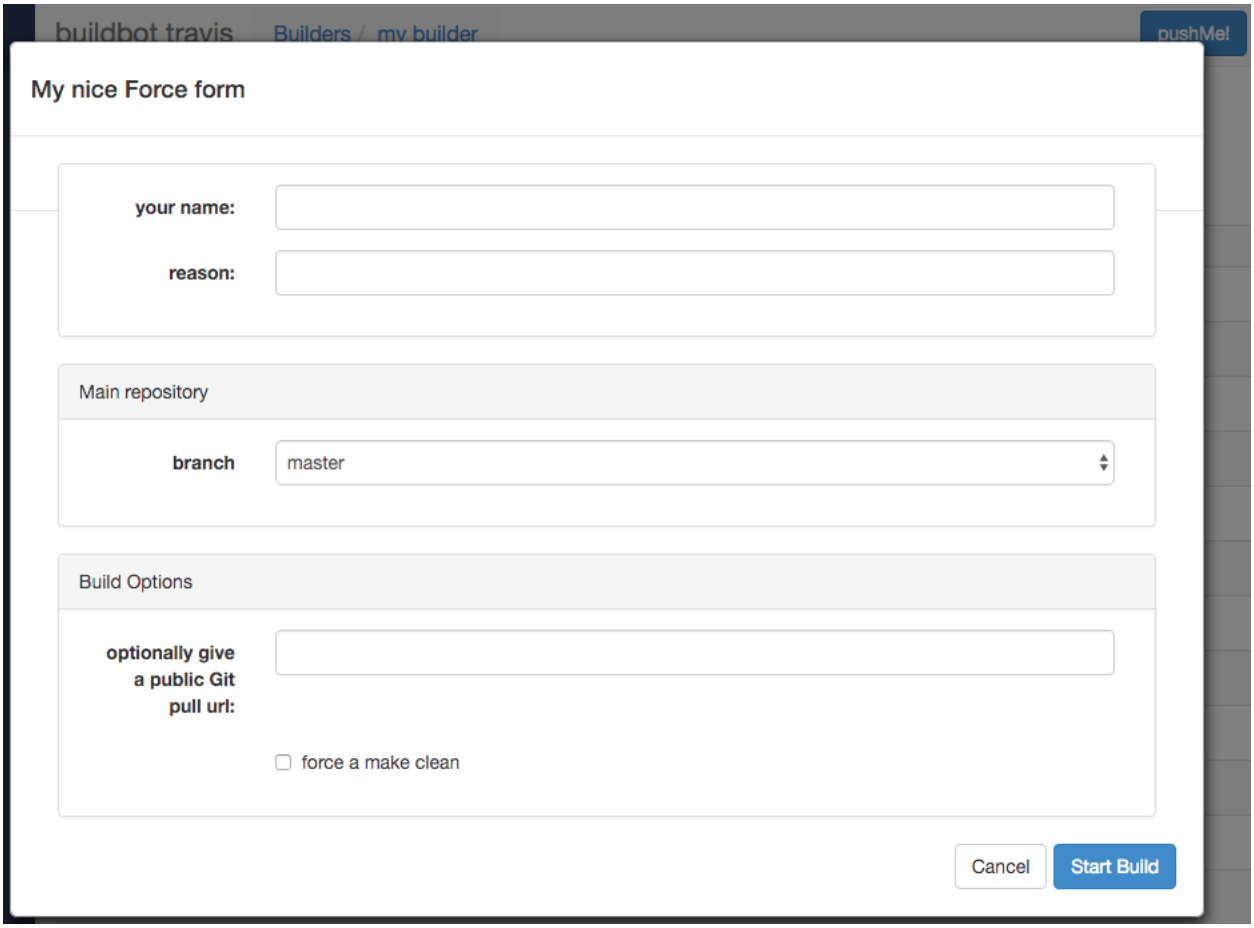
`buttonName`

The name of the "submit" button on the resulting force-build form. This defaults to the name of scheduler.

An example may be better than long explanation. What you need in your config file is something like:

```
from buildbot.plugins import schedulers, util

sch = schedulers.ForceScheduler(
    name="force",
    buttonName="pushMe!",
    label="My nice Force form",
    builderNames=["my-builder"],

    codebases=[
        util.CodebaseParameter(
            "",
            name="Main repository",
            # will generate a combo box
            branch=util.ChoiceStringParameter(
                name="branch",
                choices=["master", "hest"],
                default="master"),

            # will generate nothing in the form, but revision, repository,
            # and project are needed by buildbot scheduling system so we
            # need to pass a value ("")
            revision=util.FixedParameter(name="revision", default=""),
            repository=util.FixedParameter(name="repository", default=""),
            project=util.FixedParameter(name="project", default=""),
        ),
    ],

    # will generate a text input
    reason=util.StringParameter(name="reason",
                                label="reason:",
                                required=True, size=80),

    # in case you don't require authentication this will display
    # input for user to type his name
    username=util.UserNameParameter(label="your name:",
                                    size=80),
    # A completely customized property list.  The name of the
    # property is the name of the parameter
    properties=[
        util.NestedParameter(name="options", label="Build Options", layout="vertical",
↪ fields=[
            util.StringParameter(name="pull_url",
                                 label="optionally give a public Git pull url:",
                                 default="", size=80),
            util.BooleanParameter(name="force_build_clean",
                                  label="force a make clean",
                                  default=False)
        ])
    ])
```

This will result in the following UI:



Authorization

The force scheduler uses the web interface's authorization framework to determine which user has the right to force which build. Here is an example of code on how you can define which user has which right:

```
user_mapping = {
    re.compile("project1-builder"): ["project1-maintainer", "john"] ,
    re.compile("project2-builder"): ["project2-maintainer", "jack"],
    re.compile(".*"): ["root"]
}
def force_auth(user,  status):
    global user_mapping
    for r,users in user_mapping.items():
        if r.match(status.name):
            if user in users:
                    return True
    return False

# use authz_cfg in your WebStatus setup
authz_cfg=authz.Authz(
    auth=my_auth,
    forceBuild = force_auth,
)
```

ForceScheduler Parameters

Most of the arguments to *ForceScheduler* are "parameters". Several classes of parameters are available, each describing a different kind of input from a force-build form.

All parameter types have a few common arguments:

`name` (required)

The name of the parameter. For properties, this will correspond to the name of the property that your parameter will set. The name is also used internally as the identifier for in the HTML form.

`label` (optional; default is same as name)

The label of the parameter. This is what is displayed to the user.

`tablabel` (optional; default is same as label)

The label of the tab if this parameter is included into a tab layout NestedParameter. This is what is displayed to the user.

`default` (optional; default: "")

The default value for the parameter, that is used if there is no user input.

`required` (optional; default: False)

If this is true, then an error will be shown to user if there is no input in this field

The parameter types are:

NestedParameter

```
NestedParameter(name="options", label="Build options" layout="vertical", fields=[...
↪]),
```

This parameter type is a special parameter which contains other parameters. This can be used to group a set of parameters together, and define the layout of your form. You can recursively include NestedParameter into NestedParameter, to build very complex UI.

It adds the following arguments:

`layout` (optional, default: "vertical")

The layout defines how the fields are placed in the form.

The layouts implemented in the standard web application are:

- **`simple`: fields are displayed one by one without alignment.** They take the horizontal space that they need.
- `vertical`: all fields are displayed vertically, aligned in columns (as per the `column` attribute of the NestedParameter)
- **`tabs`: Each field gets its own tab (http://getbootstrap.com/components/#nav-tabs).** This can be used to declare complex build forms which won't fit into one screen. The children fields are usually other NestedParameters with vertical layout.

`columns` (optional, accepted values are 1,2,3,4)

The number of columns to use for a *vertical* layout. If omitted, it is set to 1 unless there are more than 3 visible child fields in which case it is set to 2.

FixedParameter

```
FixedParameter(name="branch", default="trunk"),
```

This parameter type will not be shown on the web form, and always generate a property with its default value.

StringParameter

```
StringParameter(name="pull_url",
    label="optionally give a public Git pull url:",
    default="", size=80)
```

This parameter type will show a single-line text-entry box, and allow the user to enter an arbitrary string. It adds the following arguments:

`regex` (optional)

A string that will be compiled as a regex, and used to validate the input of this parameter.

`size` (optional; default: 10)

The width of the input field (in characters).

TextParameter

```
StringParameter(name="comments",
    label="comments to be displayed to the user of the built binary",
    default="This is a development build", cols=60, rows=5)
```

This parameter type is similar to StringParameter, except that it is represented in the HTML form as a `textarea`, allowing multi-line input. It adds the StringParameter arguments, this type allows:

`cols` (optional; default: 80)

The number of columns the `textarea` will have.

`rows` (optional; default: 20)

The number of rows the `textarea` will have

This class could be subclassed in order to have more customization e.g.

- developer could send a list of Git branches to pull from
- developer could send a list of Gerrit changes to cherry-pick,
- developer could send a shell script to amend the build.

Beware of security issues anyway.

IntParameter

```
IntParameter(name="debug_level",
    label="debug level (1-10)", default=2)
```

This parameter type accepts an integer value using a text-entry box.

BooleanParameter

```
BooleanParameter(name="force_build_clean",
    label="force a make clean", default=False)
```

This type represents a boolean value. It will be presented as a checkbox.

UserNameParameter

```
UserNameParameter(label="your name:", size=80)
```

This parameter type accepts a username. If authentication is active, it will use the authenticated user instead of displaying a text-entry box.

`size` (optional; default: 10) The width of the input field (in characters).

`need_email` (optional; default True) If true, require a full email address rather than arbitrary text.

ChoiceStringParameter

```
ChoiceStringParameter(name="branch",
    choices=["main","devel"], default="main")
```

This parameter type lets the user choose between several choices (e.g the list of branches you are supporting, or the test campaign to run). If `multiple` is false, then its result is a string - one of the choices. If `multiple` is true, then the result is a list of strings from the choices.

Note that for some use cases, the choices need to be generated dynamically. This can be done via subclassing and overriding the 'getChoices' member function. An example of this is provided by the source for the `InheritBuildParameter` class.

Its arguments, in addition to the common options, are:

`choices`

The list of available choices.

`strict` (optional; default: True)

If true, verify that the user's input is from the list. Note that this only affects the validation of the form request; even if this argument is False, there is no HTML form component available to enter an arbitrary value.

`multiple`

If true, then the user may select multiple choices.

Example:

```
ChoiceStringParameter(name="forced_tests",
                      label="smoke test campaign to run",
                      default=default_tests,
                      multiple=True,
                      strict=True,
                      choices=["test_builder1", "test_builder2",
                               "test_builder3"])
```

```
# .. and later base the schedulers to trigger off this property:

# triggers the tests depending on the property forced_test
builder1.factory.addStep(Trigger(name="Trigger tests",
                                 schedulerNames=Property("forced_tests")))
```

CodebaseParameter

```
CodebaseParameter(codebase="myrepo")
```

This is a parameter group to specify a sourcestamp for a given codebase.

`codebase`

The name of the codebase.

`branch` (optional; default: StringParameter)

A *parameter* specifying the branch to build. The default value is a string parameter.

`revision` (optional; default: StringParameter)

A *parameter* specifying the revision to build. The default value is a string parameter.

`repository` (optional; default: StringParameter)

A *parameter* specifying the repository for the build. The default value is a string parameter.

`project` (optional; default: StringParameter)

A *parameter* specifying the project for the build. The default value is a string parameter.

InheritBuildParameter

Note: InheritBuildParameter is not yet ported to data API, and cannot be used with buildbot nine yet(bug #3521 (http://trac.buildbot.net/ticket/3521)).

This is a special parameter for inheriting force build properties from another build. The user is presented with a list of compatible builds from which to choose, and all forced-build parameters from the selected build are copied into the new build. The new parameter is:

`compatible_builds`

A function to find compatible builds in the build history. This function is given the master `Status` instance as first argument, and the current builder name as second argument, or None when forcing all builds.

Example:

```
def get_compatible_builds(status, builder):
    if builder is None: # this is the case for force_build_all
        return ["cannot generate build list here"]
    # find all successful builds in builder1 and builder2
    builds = []
    for builder in ["builder1","builder2"]:
        builder_status = status.getBuilder(builder)
        for num in range(1,40): # 40 last builds
```

```
            b = builder_status.getBuild(-num)
            if not b:
                continue
            if b.getResults() == FAILURE:
                continue
            builds.append(builder+"/"+str(b.getNumber()))
    return builds

# ...

sched = Scheduler(...,
    properties=[
        InheritBuildParameter(
            name="inherit",
            label="promote a build for merge",
            compatible_builds=get_compatible_builds,
            required = True),
            ])
```

WorkerChoiceParameter

Note: WorkerChoiceParameter is not yet ported to data API, and cannot be used with buildbot nine yet(bug #3521 (http://trac.buildbot.net/ticket/3521)).

This parameter allows a scheduler to require that a build is assigned to the chosen worker. The choice is assigned to the *workername* property for the build. The `enforceChosenWorker` functor must be assigned to the `canStartBuild` parameter for the `Builder`.

Example:

```
from buildbot.plugins import util

# schedulers:
ForceScheduler(
    # ...
    properties=[
        WorkerChoiceParameter(),
    ]
)

# builders:
BuilderConfig(
    # ...
    canStartBuild=util.enforceChosenWorker,
)
```

AnyPropertyParameter

This parameter type can only be used in `properties`, and allows the user to specify both the property name and value in the web form.

This Parameter is here to reimplement old Buildbot behavior, and should be avoided. Stricter parameter name and type should be preferred.

Workers

The `workers` configuration key specifies a list of known workers. In the common case, each worker is defined by an instance of the `Worker` class. It represents a standard, manually started machine that will try to connect to the buildbot master as a worker. Buildbot also supports "on-demand", or latent, workers, which allow buildbot to dynamically start and stop worker instances.

Defining Workers

A `Worker` instance is created with a `workername` and a `workerpassword`. These are the same two values that need to be provided to the worker administrator when they create the worker.

The workername must be unique, of course. The password exists to prevent evildoers from interfering with the buildbot by inserting their own (broken) workers into the system and thus displacing the real ones.

Workers with an unrecognized workername or a non-matching password will be rejected when they attempt to connect, and a message describing the problem will be written to the log file (see *Logfiles*).

A configuration for two workers would look like:

```
from buildbot.plugins import worker
c['workers'] = [
    worker.Worker('bot-solaris', 'solarispasswd'),
    worker.Worker('bot-bsd', 'bsdpasswd'),
]
```

Worker Options

`Worker` objects can also be created with an optional `properties` argument, a dictionary specifying properties that will be available to any builds performed on this worker. For example:

```
c['workers'] = [
    worker.Worker('bot-solaris', 'solarispasswd',
                  properties={ 'os':'solaris' }),
]
```

The `Worker` constructor can also take an optional `max_builds` parameter to limit the number of builds that it will execute simultaneously:

```
c['workers'] = [
    worker.Worker("bot-linux", "linuxpassword", max_builds=2)
]
```

Master-Worker TCP Keepalive

By default, the buildmaster sends a simple, non-blocking message to each worker every hour. These keepalives ensure that traffic is flowing over the underlying TCP connection, allowing the system's network stack to detect any problems before a build is started.

The interval can be modified by specifying the interval in seconds using the `keepalive_interval` parameter of `Worker`:

```
c['workers'] = [
    worker.Worker('bot-linux', 'linuxpasswd',
                  keepalive_interval=3600)
]
```

The interval can be set to `None` to disable this functionality altogether.

When Workers Go Missing

Sometimes, the workers go away. One very common reason for this is when the worker process is started once (manually) and left running, but then later the machine reboots and the process is not automatically restarted.

If you'd like to have the administrator of the worker (or other people) be notified by email when the worker has been missing for too long, just add the `notify_on_missing=` argument to the `Worker` definition. This value can be a single email address, or a list of addresses:

```
c['workers'] = [
    worker.Worker('bot-solaris', 'solarispasswd',
                  notify_on_missing="bob@example.com")
]
```

By default, this will send email when the worker has been disconnected for more than one hour. Only one email per connection-loss event will be sent. To change the timeout, use `missing_timeout=` and give it a number of seconds (the default is 3600).

You can have the buildmaster send email to multiple recipients: just provide a list of addresses instead of a single one:

```
c['workers'] = [
    worker.Worker('bot-solaris', 'solarispasswd',
                  notify_on_missing=["bob@example.com",
                                     "alice@example.org"],
                  missing_timeout=300)   # notify after 5 minutes
]
```

The email sent this way will use a `MailNotifier` (see *MailNotifier*) status target, if one is configured. This provides a way for you to control the *from* address of the email, as well as the relayhost (aka *smarthost*) to use as an SMTP server. If no `MailNotifier` is configured on this buildmaster, the worker-missing emails will be sent using a default configuration.

Note that if you want to have a `MailNotifier` for worker-missing emails but not for regular build emails, just create one with `builders=[]`, as follows:

```
from buildbot.plugins import status, worker
m = status.MailNotifier(fromaddr="buildbot@localhost", builders=[],
                        relayhost="smtp.example.org")
c['status'].append(m)

c['workers'] = [
```

```
        worker.Worker('bot-solaris', 'solarispasswd',
                      notify_on_missing="bob@example.com")
]
```

Local Workers

For smaller setups, you may want to just run the workers on the same machine as the master. To simplify the maintainance, you may even want to run them in the same process.

This is what LocalWorker is for. Instead of configuring a `worker.Worker`, you have to configure a `worker.LocalWorker`. As the worker is running on the same process, password is not necessary. You can run as many local workers as long as your machine CPU and memory is allowing.

A configuration for two workers would look like:

```
from buildbot.plugins import worker
c['workers'] = [
    worker.LocalWorker('bot1'),
    worker.LocalWorker('bot2'),
]
```

In order to use local workers you need to have `buildbot-worker` package installed.

Latent Workers

The standard buildbot model has workers started manually. The previous section described how to configure the master for this approach.

Another approach is to let the buildbot master start workers when builds are ready, on-demand. Thanks to services such as Amazon Web Services' Elastic Compute Cloud ("AWS EC2"), this is relatively easy to set up, and can be very useful for some situations.

The workers that are started on-demand are called "latent" workers. As of this writing, buildbot ships with an abstract base class for building latent workers, and a concrete implementation for AWS EC2 and for libvirt.

Common Options

The following options are available for all latent workers.

build_wait_timeout This option allows you to specify how long a latent worker should wait after a build for another build before it shuts down. It defaults to 10 minutes. If this is set to 0 then the worker will be shut down immediately. If it is less than 0 it will never automatically shutdown.

Supported Latent Workers

As of time of writing, Buildbot supports the following latent workers:

Amazon Web Services Elastic Compute Cloud ("AWS EC2")

class `buildbot.worker.ec2.`**`EC2LatentWorker`**

EC2 (http://aws.amazon.com/ec2/) is a web service that allows you to start virtual machines in an Amazon data center. Please see their website for details, including costs. Using the AWS EC2 latent workers involves getting an EC2 account with AWS and setting up payment; customizing one or more EC2 machine images ("AMIs") on your desired operating system(s) and publishing them (privately if needed); and configuring the buildbot master to know how to start your customized images for "substantiating" your latent workers.

This document will guide you through setup of a AWS EC2 latent worker:

Get an AWS EC2 Account

To start off, to use the AWS EC2 latent worker, you need to get an AWS developer account and sign up for EC2. Although Amazon often changes this process, these instructions should help you get started:

1. Go to http://aws.amazon.com/ and click to "Sign Up Now" for an AWS account.
2. Once you are logged into your account, you need to sign up for EC2. Instructions for how to do this have changed over time because Amazon changes their website, so the best advice is to hunt for it. After signing up for EC2, it may say it wants you to upload an x.509 cert. You will need this to create images (see below) but it is not technically necessary for the buildbot master configuration.
3. You must enter a valid credit card before you will be able to use EC2. Do that under 'Payment Method'.
4. Make sure you're signed up for EC2 by going to *Your Account* → *Account Activity* and verifying EC2 is listed.

Create an AMI

Now you need to create an AMI and configure the master. You may need to run through this cycle a few times to get it working, but these instructions should get you started.

Creating an AMI is out of the scope of this document. The EC2 Getting Started Guide (http://docs.amazonwebservices.com/AWSEC2/latest/GettingStartedGuide/) is a good resource for this task. Here are a few additional hints.

- When an instance of the image starts, it needs to automatically start a buildbot worker that connects to your master (to create a buildbot worker, *Creating a worker*; to make a daemon, *Launching the daemons*).
- You may want to make an instance of the buildbot worker, configure it as a standard worker in the master (i.e., not as a latent worker), and test and debug it that way before you turn it into an AMI and convert to a latent worker in the master.

Configure the Master with an `EC2LatentWorker`

Now let's assume you have an AMI that should work with the *`EC2LatentWorker`*. It's now time to set up your buildbot master configuration.

You will need some information from your AWS account: the *Access Key Id* and the *Secret Access Key*. If you've built the AMI yourself, you probably already are familiar with these values. If you have not, and someone has given you access to an AMI, these hints may help you find the necessary values:

- While logged into your AWS account, find the "Access Identifiers" link (either on the left, or via *Your Account* → *Access Identifiers*.
- On the page, you'll see alphanumeric values for "Your Access Key Id:" and "Your Secret Access Key:". Make a note of these. Later on, we'll call the first one your `identifier` and the second one your `secret_identifier`.

When creating an *EC2LatentWorker* in the buildbot master configuration, the first three arguments are required. The name and password are the first two arguments, and work the same as with normal workers. The next argument specifies the type of the EC2 virtual machine (available options as of this writing include `m1.small`, `m1.large`, `m1.xlarge`, `c1.medium`, and `c1.xlarge`; see the EC2 documentation for descriptions of these machines).

Here is the simplest example of configuring an EC2 latent worker. It specifies all necessary remaining values explicitly in the instantiation.

```
from buildbot.plugins import worker
c['workers'] = [
    worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                           ami='ami-12345',
                           identifier='publickey',
                           secret_identifier='privatekey'
                           keypair_name='latent_buildbot_worker',
                           security_name='latent_buildbot_worker',
                           )
]
```

The `ami` argument specifies the AMI that the master should start. The `identifier` argument specifies the AWS *Access Key Id*, and the `secret_identifier` specifies the AWS *Secret Access Key*. Both the AMI and the account information can be specified in alternate ways.

Note: Whoever has your `identifier` and `secret_identifier` values can request AWS work charged to your account, so these values need to be carefully protected. Another way to specify these access keys is to put them in a separate file. Buildbot supports the standard AWS credentials file. You can then make the access privileges stricter for this separate file, and potentially let more people read your main configuration file. If your master is running in EC2, you can also use IAM roles for EC2 to delegate permissions.

`keypair_name` and `security_name` allow you to specify different names for these AWS EC2 values.

You can make an `.aws` directory in the home folder of the user running the buildbot master. In that directory, create a file called `credentials`. The format of the file should be as follows, replacing `identifier` and `secret_identifier` with the credentials obtained before.

```
[default]
aws_access_key_id = identifier
aws_secret_access_key = secret_identifier
```

If you are using IAM roles, no config file is required. Then you can instantiate the worker as follows.

```
from buildbot.plugins import worker
c['workers'] = [
    worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                           ami='ami-12345',
                           keypair_name='latent_buildbot_worker',
```

```
                              security_name='latent_buildbot_worker',
                              )
]
```

Previous examples used a particular AMI. If the Buildbot master will be deployed in a process-controlled environment, it may be convenient to specify the AMI more flexibly. Rather than specifying an individual AMI, specify one or two AMI filters.

In all cases, the AMI that sorts last by its location (the S3 bucket and manifest name) will be preferred.

One available filter is to specify the acceptable AMI owners, by AWS account number (the 12 digit number, usually rendered in AWS with hyphens like "1234-5678-9012", should be entered as in integer).

```
from buildbot.plugins import worker
bot1 = worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                              valid_ami_owners=[11111111111,
                                                22222222222],
                              identifier='publickey',
                              secret_identifier='privatekey',
                              keypair_name='latent_buildbot_worker',
                              security_name='latent_buildbot_worker',
                              )
```

The other available filter is to provide a regular expression string that will be matched against each AMI's location (the S3 bucket and manifest name).

```
from buildbot.plugins import worker
bot1 = worker.EC2LatentWorker(
        'bot1', 'sekrit', 'm1.large',
        valid_ami_location_regex=r'buildbot\-.*/image.manifest.xml',
        identifier='publickey',
        secret_identifier='privatekey',
        keypair_name='latent_buildbot_worker',
        security_name='latent_buildbot_worker',
        )
```

The regular expression can specify a group, which will be preferred for the sorting. Only the first group is used; subsequent groups are ignored.

```
from buildbot.plugins import worker
bot1 = worker.EC2LatentWorker(
    'bot1', 'sekrit', 'm1.large',
    valid_ami_location_regex=r'buildbot\-.*\-(.*)/image.manifest.xml',
    identifier='publickey',
    secret_identifier='privatekey',
    keypair_name='latent_buildbot_worker',
    security_name='latent_buildbot_worker',
    )
```

If the group can be cast to an integer, it will be. This allows 10 to sort after 1, for instance.

```
from buildbot.plugins import worker
bot1 = worker.EC2LatentWorker(
        'bot1', 'sekrit', 'm1.large',
        valid_ami_location_regex=r'buildbot\-.*\-(\d+)/image.manifest.xml',
        identifier='publickey',
        secret_identifier='privatekey',
        keypair_name='latent_buildbot_worker',
```

```
        security_name='latent_buildbot_worker',
        )
```

In addition to using the password as a handshake between the master and the worker, you may want to use a firewall to assert that only machines from a specific IP can connect as workers. This is possible with AWS EC2 by using the Elastic IP feature. To configure, generate a Elastic IP in AWS, and then specify it in your configuration using the `elastic_ip` argument.

```
from buildbot.plugins import worker
c['workers'] = [
    worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                           'ami-12345',
                           identifier='publickey',
                           secret_identifier='privatekey',
                           elastic_ip='208.77.188.166',
                           keypair_name='latent_buildbot_worker',
                           security_name='latent_buildbot_worker',
                           )
]
```

One other way to configure a worker is by settings AWS tags. They can for example be used to have a more restrictive security IAM (http://aws.amazon.com/iam/) policy. To get Buildbot to tag the latent worker specify the tag keys and values in your configuration using the `tags` argument.

```
from buildbot.plugins import worker
c['workers'] = [
    worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                           'ami-12345',
                           identifier='publickey',
                           secret_identifier='privatekey',
                           keypair_name='latent_buildbot_worker',
                           security_name='latent_buildbot_worker',
                           tags={'SomeTag': 'foo'})
]
```

If the worker needs access to additional AWS resources, you can also enable your workers to access them via an EC2 instance profile. To use this capability, you must first create an instance profile separately in AWS. Then specify its name on EC2LatentWorker via instance_profile_name.

```
from buildbot.plugins import worker
c['workers'] = [
    worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                           ami='ami-12345',
                           keypair_name='latent_buildbot_worker',
                           security_name='latent_buildbot_worker',
                           instance_profile_name='my_profile'
                           )
]
```

You may also supply your own boto3.Session object to allow for more flexible session options (ex. cross-account) To use this capability, you must first create a boto3.Session object. Then provide it to EC2LatentWorker via `session` argument.

```
import boto3
from buildbot.plugins import worker

session = boto3.session.Session()
```

```
c['workers'] = [
    worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                           ami='ami-12345',
                           keypair_name='latent_buildbot_worker',
                           security_name='latent_buildbot_worker',
                           session=session
                           )
]
```

The *EC2LatentWorker* supports all other configuration from the standard `Worker`. The `missing_timeout` and `notify_on_missing` specify how long to wait for an EC2 instance to attach before considering the attempt to have failed, and email addresses to alert, respectively. `missing_timeout` defaults to 20 minutes.

Volumes

If you want to attach existing volumes to an ec2 latent worker, use the volumes attribute. This mechanism can be valuable if you want to maintain state on a conceptual worker across multiple start/terminate sequences. `volumes` expects a list of (volume_id, mount_point) tuples to attempt attaching when your instance has been created.

If you want to attach new ephemeral volumes, use the the block_device_map attribute. This follows the AWS API syntax, essentially acting as a passthrough. The only distinction is that the volumes default to deleting on termination to avoid leaking volume resources when workers are terminated. See boto documentation for further details.

```
from buildbot.plugins import worker
c['workers'] = [
    worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                           ami='ami-12345',
                           keypair_name='latent_buildbot_worker',
                           security_name='latent_buildbot_worker',
                           block_device_map= [
                             "DeviceName": "/dev/xvdb",
                             "Ebs" : {
                                "VolumeType": "io1",
                                "Iops": 1000,
                                "VolumeSize": 100
                              }
                           ]
                           )
]
```

VPC Support

If you are managing workers within a VPC, your worker configuration must be modified from above. You must specify the id of the subnet where you want your worker placed. You must also specify security groups created within your VPC as opposed to classic EC2 security groups. This can be done by passing the ids of the vpc security groups. Note, when using a VPC, you can not specify classic EC2 security groups (as specified by security_name).

```
from buildbot.plugins import worker
c['workers'] = [
    worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                           ami='ami-12345',
                           keypair_name='latent_buildbot_worker',
                           subnet_id='subnet-12345',
                           security_group_ids=['sg-12345','sg-67890']
```

```
                                )
]
```

Spot instances

If you would prefer to use spot instances for running your builds, you can accomplish that by passing in a True value to the `spot_instance` parameter to the *EC2LatentWorker* constructor. Additionally, you may want to specify `max_spot_price` and `price_multiplier` in order to limit your builds' budget consumption.

```
from buildbot.plugins import worker
c['workers'] = [
    worker.EC2LatentWorker('bot1', 'sekrit', 'm1.large',
                           'ami-12345', region='us-west-2',
                           identifier='publickey',
                           secret_identifier='privatekey',
                           elastic_ip='208.77.188.166',
                           keypair_name='latent_buildbot_worker',
                           security_name='latent_buildbot_worker',
                           placement='b', spot_instance=True,
                           max_spot_price=0.09,
                           price_multiplier=1.15,
                           product_description='Linux/UNIX')
]
```

This example would attempt to create a m1.large spot instance in the us-west-2b region costing no more than $0.09/hour. The spot prices for 'Linux/UNIX' spot instances in that region over the last 24 hours will be averaged and multiplied by the `price_multiplier` parameter, then a spot request will be sent to Amazon with the above details. If the multiple exceeds the `max_spot_price`, the bid price will be the `max_spot_price`.

Either `max_spot_price` or `price_multiplier`, but not both, may be None. If `price_multiplier` is None, then no historical price information is retrieved; the bid price is simply the specified `max_spot_price`. If the `max_spot_price` is None, then the multiple of the historical average spot prices is used as the bid price with no limit.

Libvirt

class `buildbot.worker.libvirt.`**`LibVirtWorker`**

libvirt (http://www.libvirt.org/) is a virtualization API for interacting with the virtualization capabilities of recent versions of Linux and other OSes. It is LGPL and comes with a stable C API, and Python bindings.

This means we know have an API which when tied to buildbot allows us to have workers that run under Xen, QEMU, KVM, LXC, OpenVZ, User Mode Linux, VirtualBox and VMWare.

The libvirt code in Buildbot was developed against libvirt 0.7.5 on Ubuntu Lucid. It is used with KVM to test Python code on Karmic VM's, but obviously isn't limited to that. Each build is run on a new VM, images are temporary and thrown away after each build.

This document will guide you through setup of a libvirt latent worker:

Setting up libvirt

We won't show you how to set up libvirt as it is quite different on each platform, but there are a few things you should keep in mind.

- If you are running on Ubuntu, your master should run Lucid. Libvirt and apparmor are buggy on Karmic.
- If you are using the system libvirt, your buildbot master user will need to be in the libvirtd group.
- If you are using KVM, your buildbot master user will need to be in the KVM group.
- You need to think carefully about your virtual network *first*. Will NAT be enough? What IP will my VM's need to connect to for connecting to the master?

Configuring your base image

You need to create a base image for your builds that has everything needed to build your software. You need to configure the base image with a buildbot worker that is configured to connect to the master on boot.

Because this image may need updating a lot, we strongly suggest scripting its creation.

If you want to have multiple workers using the same base image it can be annoying to duplicate the image just to change the buildbot credentials. One option is to use libvirt's DHCP server to allocate an identity to the worker: DHCP sets a hostname, and the worker takes its identity from that.

Doing all this is really beyond the scope of the manual, but there is a vmbuilder (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/libvirt/vmbuilder) script and a network.xml (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/libvirt/network.xml) file to create such a DHCP server in master/contrib/ (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/) (*Contrib Scripts*) that should get you started:

```
sudo apt-get install ubuntu-vm-builder
sudo contrib/libvirt/vmbuilder
```

Should create an `ubuntu/` folder with a suitable image in it.

```
virsh net-define contrib/libvirt/network.xml
virsh net-start buildbot-network
```

Should set up a KVM compatible libvirt network for your buildbot VM's to run on.

Configuring your Master

If you want to add a simple on demand VM to your setup, you only need the following. We set the username to `minion1`, the password to `sekrit`. The base image is called `base_image` and a copy of it will be made for the duration of the VM's life. That copy will be thrown away every time a build is complete.

```
from buildbot.plugins import worker, util
c['workers'] = [
    worker.LibVirtWorker('minion1', 'sekrit',
                         util.Connection("qemu:///session"),
                         '/home/buildbot/images/minion1',
```

```
                        '/home/buildbot/images/base_image')
]
```

You can use virt-manager to define `minion1` with the correct hardware. If you don't, buildbot won't be able to find a VM to start.

`LibVirtWorker` accepts the following arguments:

`name` Both a buildbot username and the name of the virtual machine.

`password` A password for the buildbot to login to the master with.

`connection` `Connection` instance wrapping connection to libvirt.

`hd_image` The path to a libvirt disk image, normally in qcow2 format when using KVM.

`base_image` If given a base image, buildbot will clone it every time it starts a VM. This means you always have a clean environment to do your build in.

`xml` If a VM isn't predefined in virt-manager, then you can instead provide XML like that used with `virsh define`. The VM will be created automatically when needed, and destroyed when not needed any longer.

OpenStack

class `buildbot.worker.openstack.`**`OpenStackLatentWorker`**

OpenStack (http://openstack.org/) is a series of interconnected components that facilitates managing compute, storage, and network resources in a data center. It is available under the Apache License and has a REST interface along with a Python client.

This document will guide you through setup of an OpenStack latent worker:

Install dependencies

OpenStackLatentWorker requires python-novaclient to work, you can install it with pip install python-novaclient.

Get an Account in an OpenStack cloud

Setting up OpenStack is outside the domain of this document. There are four account details necessary for the Buildbot master to interact with your OpenStack cloud: username, password, a tenant name, and the auth URL to use.

Create an Image

OpenStack supports a large number of image formats. OpenStack maintains a short list of prebuilt images; if the desired image is not listed, The OpenStack Compute Administration Manual (http://docs.openstack.org/trunk/openstack-

compute/admin/content/index.html) is a good resource for creating new images. You need to configure the image with a buildbot worker to connect to the master on boot.

Configure the Master with an OpenStackLatentWorker

With the configured image in hand, it is time to configure the buildbot master to create OpenStack instances of it. You will need the aforementioned account details. These are the same details set in either environment variables or passed as options to an OpenStack client.

`OpenStackLatentWorker` accepts the following arguments:

`name` The worker name.

`password` A password for the worker to login to the master with.

`flavor` The flavor ID to use for the instance.

`image` A string containing the image UUID to use for the instance. A callable may instead be passed. It will be passed the list of available images and must return the image to use.

`os_username`

`os_password`

`os_tenant_name`

`os_auth_url` The OpenStack authentication needed to create and delete instances. These are the same as the environment variables with uppercase names of the arguments.

`block_devices` A list of dictionaries. Each dictionary specifies a block device to set up during instance creation. The values support using properties from the build and will be rendered when the instance is started.

Supported keys

`uuid` (required): The image, snapshot, or volume UUID.

`volume_size` (optional): Size of the block device in GiB. If not specified, the minimum size in GiB to contain the source will be calculated and used.

`device_name` (optional): defaults to `vda`. The name of the device in the instance; e.g. vda or xda.

`source_type` (optional): defaults to `image`. The origin of the block device. Valid values are `image`, `snapshot`, or `volume`.

`destination_type` (optional): defaults to `volume`. Destination of block device: `volume` or `local`.

`delete_on_termination` (optional): defaults to `True`. Controls if the block device will be deleted when the instance terminates.

`boot_index` (optional): defaults to `0`. Integer used for boot order.

`meta` A dictionary of string key-value pairs to pass to the instance. These will be available under the `metadata` key from the metadata service.

`nova_args` (optional) A dict that will be appended to the arguments when creating a VM. Buildbot uses the OpenStack Nova version 2 API by default (see client_version).

`client_version` (optional) A string containing the Nova client version to use. Defaults to `2`. Supports using `2.X`, where X is a micro-version. Use `1.1` for the previous, deprecated, version. If using `1.1`, note that an older version of novaclient will be needed so it won't switch to using `2`.

Here is the simplest example of configuring an OpenStack latent worker.

```
from buildbot.plugins import worker
c['workers'] = [
    worker.OpenStackLatentWorker('bot2', 'sekrit',
                flavor=1, image='8ac9d4a4-5e03-48b0-acde-77a0345a9ab1',
                os_username='user', os_password='password',
                os_tenant_name='tenant',
                os_auth_url='http://127.0.0.1:35357/v2.0')
]
```

The `image` argument also supports being given a callable. The callable will be passed the list of available images and must return the image to use. The invocation happens in a separate thread to prevent blocking the build master when interacting with OpenStack.

```
from buildbot.plugins import worker

def find_image(images):
    # Sort oldest to newest.
    def key_fn(x):
        return x.created

    candidate_images = sorted(images, key=key_fn)
    # Return the oldest candiate image.
    return candidate_images[0]

c['workers'] = [
    worker.OpenStackLatentWorker('bot2', 'sekrit',
                flavor=1, image=find_image,
                os_username='user', os_password='password',
                os_tenant_name='tenant',
                os_auth_url='http://127.0.0.1:35357/v2.0')
]
```

The `block_devices` argument is minimally manipulated to provide some defaults and passed directly to novaclient. The simplest example is an image that is converted to a volume and the instance boots from that volume. When the instance is destroyed, the volume will be terminated as well.

```
from buildbot.plugins import worker
c['workers'] = [
    worker.OpenStackLatentWorker('bot2', 'sekrit',
                flavor=1, image='8ac9d4a4-5e03-48b0-acde-77a0345a9ab1',
                os_username='user', os_password='password',
                os_tenant_name='tenant',
                os_auth_url='http://127.0.0.1:35357/v2.0',
                block_devices=[
                    {'uuid': '3f0b8868-67e7-4a5b-b685-2824709bd486',
                    'volume_size': 10}])
]
```

The `nova_args` can be used to specify additional arguments for the novaclient. For example network mappings, which is required if your OpenStack tenancy has more than one network, and default cannot be determined. Please refer to your OpenStack manual whether it wants net-id or net-name.

Other useful parameters are `availability_zone`, `security_groups` and `config_drive`. Refer to Python bindings to the OpenStack Nova API (http://docs.openstack.org/developer/python-novaclient/) for more information. It is found on section Servers, method create.

```
from buildbot.plugins import worker
c['workers'] = [
```

```
    worker.OpenStackLatentWorker('bot2', 'sekrit',
                flavor=1, image='8ac9d4a4-5e03-48b0-acde-77a0345a9ab1',
                os_username='user', os_password='password',
                os_tenant_name='tenant',
                os_auth_url='http://127.0.0.1:35357/v2.0',
                nova_args={
                  'nics': [
                            {'net-id':'uid-of-network'}
                          ]})
]
```

`OpenStackLatentWorker` supports all other configuration from the standard `Worker`. The `missing_timeout` and `notify_on_missing` specify how long to wait for an OpenStack instance to attach before considering the attempt to have failed and email addresses to alert, respectively. `missing_timeout` defaults to 20 minutes.

Docker latent worker

class `buildbot.worker.docker.`**`DockerLatentWorker`**

class `buildbot.plugins.worker.`**`DockerLatentWorker`**

Docker (https://docker.com) is an open-source project that automates the deployment of applications inside software containers. The `DockerLatentWorker` attempts to instantiate a fresh image for each build to assure consistency of the environment between builds. Each image will be discarded once the worker finished processing the build queue (i.e. becomes `idle`). See *build_wait_timeout* to change this behavior.

This document will guide you through the setup of such workers.

Docker Installation

An easy way to try Docker is through installation of dedicated Virtual machines. Two of them stands out:

- CoreOS (https://coreos.com/)
- boot2docker (http://boot2docker.io/)

Beside, it is always possible to install Docker next to the buildmaster. Beware that in this case, overall performance will depend on how many builds the computer where you have your buildmaster can handle as everything will happen on the same one.

Note: It is not necessary to install Docker in the same environment as your master as we will make use to the Docker API through docker-py (https://pypi.python.org/pypi/docker-py). More in *master setup*.

CoreOS

CoreOS is targeted at building infrastructure and distributed systems. In order to get the latent worker working with CoreOS, it is necessary to expose the docker socket (https://coreos.com/docs/launching-containers/building/customizing-docker/) outside of the Virtual Machine. If you installed it via Vagrant (https://coreos.com/docs/running-coreos/platforms/vagrant/), it is also necessary to uncomment the following line in your `config.rb` file:

```
$expose_docker_tcp=2375
```

The following command should allow you to confirm that your Docker socket is now available via the network:

```
docker -H tcp://127.0.0.1:2375 ps
```

boot2docker

boot2docker is one of the fastest ways to boot to Docker. As it is meant to be used from outside of the Virtual Machine, the socket is already exposed. Please follow the installation instructions on how to find the address of your socket.

Image Creation

Our build master will need the name of an image to perform its builds. Each time a new build will be requested, the same base image will be used again and again, actually discarding the result of the previous build. If you need some persistant storage between builds, you can use Volumes.

Each Docker image has a single purpose. Our worker image will be running a buildbot worker.

Docker uses `Dockerfiles` to describe the steps necessary to build an image. The following example will build a minimal worker. This example is voluntarily simplistic, and should probably not be used in production, see next paragraph.

```
1  FROM debian:stable
2  RUN apt-get update && apt-get install -y \
3     python-dev \
4     python-pip
5  RUN pip install buildbot-worker
6  RUN groupadd -r buildbot && useradd -r -g buildbot buildbot
7  RUN mkdir /worker && chown buildbot:buildbot /worker
8  # Install your build-dependencies here ...
9  USER buildbot
10 WORKDIR /worker
11 RUN buildbot-worker create-worker . <master-hostname> <workername> <workerpassword>
12 ENTRYPOINT ["/usr/local/bin/buildbot-worker"]
13 CMD ["start", "--nodaemon"]
```

On line 11, the hostname for your master instance, as well as the worker name and password is setup. Don't forget to replace those values with some valid ones for your project.

It is a good practice to set the `ENTRYPOINT` to the worker executable, and the `CMD` to `["start", "--nodaemon"]`. This way, no parameter will be required when starting the image.

When your Dockerfile is ready, you can build your first image using the following command (replace *myworkername* with a relevant name for your case):

```
docker build -t myworkername - < Dockerfile
```

Reuse same image for different workers

Previous simple example hardcodes the worker name into the dockerfile, which will not work if you want to share your docker image between workers.

You can find in buildbot source code in master/contrib/docker (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/docker) one example configurations:

pythonnode_worker (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/docker/pythonnode_worker/)
a worker with Python and node installed, which demonstrate how to reuse the base worker to create variations of build environments. It is based on the official `buildbot/buildbot-worker` image.

The master setups several environment variables before starting the workers:

BUILDMASTER The address of the master the worker shall connect to

BUILDMASTER_PORT The port of the master's worker 'pb' protocol.

WORKERNAME The name the worker should use to connect to master

WORKERPASS The password the worker should use to connect to master

Master Setup

We will rely on docker-py to connect our master with docker. Now is the time to install it in your master environment.

Before adding the worker to your master configuration, it is possible to validate the previous steps by starting the newly created image interactively. To do this, enter the following lines in a Python prompt where docker-py is installed:

```
>>> import docker
>>> docker_socket = 'tcp://localhost:2375'
>>> client = docker.client.Client(base_url=docker_socket)
>>> worker_image = 'my_project_worker'
>>> container = client.create_container(worker_image)
>>> client.start(container['Id'])
>>> # Optionally examine the logs of the master
>>> client.stop(container['Id'])
>>> client.wait(container['Id'])
0
```

It is now time to add the new worker to the master configuration under *workers*.

The following example will add a Docker latent worker for docker running at the following address: `tcp://localhost:2375`, the worker name will be `docker`, its password: `password`, and the base image name will be `my_project_worker`:

```
from buildbot.plugins import worker
c['workers'] = [
    worker.DockerLatentWorker('docker', 'password',
                              docker_host='tcp://localhost:2375',
                              image='my_project_worker')
]
```

password (mandatory) The worker password part of the *Latent Workers* API. If the password is `None`, then it will be automatically generated from random number, and transmitted to the container via environment variable.

In addition to the arguments available for any *Latent Workers*, `DockerLatentWorker` will accept the following extra ones:

`docker_host` (mandatory) This is the address the master will use to connect with a running Docker instance.

`image`

This is the name of the image that will be started by the build master. It should start a worker. This option can be a renderable, like *Interpolate*, so that it generates from the build request properties.

`command` (optional) This will override the command setup during image creation.

`volumes` (optional) See *Setting up Volumes*

`dockerfile` (optional if `image` is given) This is the content of the Dockerfile that will be used to build the specified image if the image is not found by Docker. It should be a multiline string.

Note: In case `image` and `dockerfile` are given, no attempt is made to compare the image with the content of the Dockerfile parameter if the image is found.

`version` (optional, default to the highest version known by docker-py) This will indicates wich API version must be used to communicate with Docker.

`tls` (optional) This allow to use TLS when connecting with the Docker socket. This should be a `docker.tls.TLSConfig` object. See docker-py's own documentation (https://docker-py.readthedocs.io/en/1.10.4/tls/) for more details on how to initialise this object.

`followStartupLogs` (optional, defaults to false) This transfers docker container's log inside master logs during worker startup (before connection). This can be useful to debug worker startup. e.g network issues, etc.

`masterFQDN` (optional, defaults to socket.getfqdn()) Address of the master the worker should connect to. Use if you master machine does not have proper fqdn. This value is passed to the docker image via environment variable `BUILDMASTER`

`hostconfig` (optional) Extra host configuration parameters passed as a dictionary used to create HostConfig object. See docker-py's HostConfig documentation (https://docker-py.readthedocs.io/en/1.10.4/hostconfig/) for all the supported options.

Setting up Volumes

The `volume` parameter allows to share directory between containers, or between a container and the host system. Refer to Docker documentation for more information about Volumes.

The format of that variable has to be an array of string. Each string specify a volume in the following format: *`volumename`*`:`*`bindname`*. The volume name has to be appended with `:ro` if the volume should be mounted *read-only*.

Note: This is the same format as when specifying volumes on the command line for docker's own `-v` option.

Hyper latent worker

Hyper (https://hyper.sh) is a CaaS solution for hosting docker container in the cloud, billed to the second. It forms a very cost efficient solution to run your CI in the cloud.

Buildbot supports using Hyper (https://hyper.sh) to host your latent workers.

class buildbot.worker.hyper.**HyperLatentWorker**

class buildbot.plugins.worker.**HyperLatentWorker**

The HyperLatentWorker attempts to instantiate a fresh image for each build to assure consistency of the environment between builds. Each image will be discarded once the worker finished processing the build queue (i.e. becomes idle). See *build_wait_timeout* to change this behavior.

In addition to the arguments available for any *Latent Workers*, HyperLatentWorker will accept the following extra ones:

password (mandatory) The worker password part of the *Latent Workers* API. If the password is None, then it will be automatically generated from random number, and transmitted to the container via environment variable.

hyper_host (mandatory) This is the address the hyper infra endpoint will use to start docker containers.

image (mandatory) This is the name of the image that will be started by the build master. It should start a worker. This option can be a renderable, like *Interpolate*, so that it generates from the build request properties. Images are by default pulled from the public DockerHub (https://hub.docker.com/) docker registry. You can consult the hyper documentation to know how to configure a custom registry. HyperLatentWorker does not support starting a worker built from a Dockerfile.

masterFQDN (optional, defaults to socket.getfqdn()) Address of the master the worker should connect to. Use if you master machine does not have proper fqdn. This value is passed to the docker image via environment variable BUILDMASTER

If the value contains a colon (:), then BUILDMASTER and BUILDMASTER_PORT environment variables will be passed, following scheme: masterFQDN="$BUILDMASTER:$BUILDMASTER_PORT"

This feature is useful for testing behind a proxy using ngrok command like: ngrok tcp 9989 ngrok config can the be retrieved with following snippet:

```
from future.moves.urllib.parse import urlparse
import requests
r = requests.get("http://localhost:4040/api/tunnels/command_line").json()
masterFQDN = urlparse(r['public_url']).netloc
```

hyper_accesskey (mandatory) Access key to use as part of the creds to access hyper.

hyper_secretkey (mandatory) Secret key to use as part of the creds to access hyper.

hyper_size (optional, defaults to s3) Size of the container to use as per HyperPricing (https://hyper.sh/pricing.html)

Marathon latent worker

Marathon (https://mesosphere.github.io/marathon/) Marathon is a production-grade container orchestration platform for Mesosphere's Data-center Operating System (DC/OS) and Apache Mesos.

Buildbot supports using Marathon (https://mesosphere.github.io/marathon/) to host your latent workers. It requires either txrequests (https://pypi.python.org/pypi/txrequests) or treq (https://pypi.python.org/pypi/treq) to be installed to allow interaction with http server. See HTTPClientService for details.

class buildbot.worker.marathon.**MarathonLatentWorker**

class buildbot.plugins.worker.**MarathonLatentWorker**

The MarathonLatentWorker attempts to instantiate a fresh image for each build to assure consistency of the environment between builds. Each image will be discarded once the worker finished processing the build queue (i.e. becomes idle). See *build_wait_timeout* to change this behavior.

In addition to the arguments available for any *Latent Workers*, `MarathonLatentWorker` will accept the following extra ones:

`marathon_url` (mandatory) This is the URL to Marathon (https://mesosphere.github.io/marathon/) server. Its REST API will be used to start docker containers.

`marathon_auth` (optional) This is the optional `('userid', 'password')` `BasicAuth` credential. If txrequests (https://pypi.python.org/pypi/txrequests) is installed, this can be a requests authentication plugin (http://docs.python-requests.org/en/master/user/authentication/).

`image` (mandatory) This is the name of the image that will be started by the build master. It should start a worker. This option can be a renderable, like *Interpolate*, so that it generates from the build request properties. Images are by pulled from the default docker registry. MarathonLatentWorker does not support starting a worker built from a Dockerfile.

`masterFQDN` (optional, defaults to socket.getfqdn()) Address of the master the worker should connect to. Use if you master machine does not have proper fqdn. This value is passed to the docker image via environment variable `BUILDMASTER`

If the value contains a colon (`:`), then BUILDMASTER and BUILDMASTER_PORT environment variables will be passed, following scheme: `masterFQDN="$BUILDMASTER:$BUILDMASTER_PORT"`

`marathon_extra_config` (optional, defaults to `{}`) Extra configuration to be passed to Marathon API (http://mesosphere.github.io/marathon/docs/rest-api.html#post-v2-apps). This implementation will setup the minimal configuration to run a worker (docker image, `BRIDGED` network) It will let the default for everything else, including memory size, volume mounting, etc. This configuration is voluntarily very raw so that it is easy to use new marathon features. This dictionary will be merged into the Buildbot generated config, and recursively override it. See Marathon API (http://mesosphere.github.io/marathon/docs/rest-api.html#post-v2-apps) documentation to learn what to include in this config.

Dangers with Latent Workers

Any latent worker that interacts with a for-fee service, such as the *`EC2LatentWorker`*, brings significant risks. As already identified, the configuration will need access to account information that, if obtained by a criminal, can be used to charge services to your account. Also, bugs in the buildbot software may lead to unnecessary charges. In particular, if the master neglects to shut down an instance for some reason, a virtual machine may be running unnecessarily, charging against your account. Manual and/or automatic (e.g. nagios with a plugin using a library like boto) double-checking may be appropriate.

A comparatively trivial note is that currently if two instances try to attach to the same latent worker, it is likely that the system will become confused. This should not occur, unless, for instance, you configure a normal worker to connect with the authentication of a latent buildbot. If this situation does occurs, stop all attached instances and restart the master.

Builder Configuration

The *`builders`* configuration key is a list of objects giving configuration for the Builders. For more information on the function of Builders in Buildbot, see *the Concepts chapter*. The class definition for the builder configuration is in `buildbot.config`. However there is a much simpler way to use it, so in the configuration file, its use looks like:

```
from buildbot.plugins import util
c['builders'] = [
    util.BuilderConfig(name='quick', workernames=['bot1', 'bot2'], factory=f_quick),
    util.BuilderConfig(name='thorough', workername='bot1', factory=f_thorough),
]
```

`BuilderConfig` takes the following keyword arguments:

`name` This specifies the Builder's name, which is used in status reports.

`workername`

`workernames` These arguments specify the worker or workers that will be used by this Builder. All workers names must appear in the *`workers`* configuration parameter. Each worker can accommodate multiple builders. The `workernames` parameter can be a list of names, while `workername` can specify only one worker.

`factory` This is a `buildbot.process.factory.BuildFactory` instance which controls how the build is performed by defining the steps in the build. Full details appear in their own section, *Build Factories*.

Other optional keys may be set on each `BuilderConfig`:

`builddir` Specifies the name of a subdirectory of the master's basedir in which everything related to this builder will be stored. This holds build status information. If not set, this parameter defaults to the builder name, with some characters escaped. Each builder must have a unique build directory.

`workerbuilddir` Specifies the name of a subdirectory (under the worker's configured base directory) in which everything related to this builder will be placed on the worker. This is where checkouts, compiles, and tests are run. If not set, defaults to `builddir`. If a worker is connected to multiple builders that share the same `workerbuilddir`, make sure the worker is set to run one build at a time or ensure this is fine to run multiple builds from the same directory simultaneously.

`tags` If provided, this is a list of strings that identifies tags for the builder. Status clients can limit themselves to a subset of the available tags. A common use for this is to add new builders to your setup (for a new module, or for a new worker) that do not work correctly yet and allow you to integrate them with the active builders. You can tag these new builders with a `test` tag, make your main status clients ignore them, and have only private status clients pick them up. As soon as they work, you can move them over to the active tag.

`nextWorker` If provided, this is a function that controls which worker will be assigned future jobs. The function is passed three arguments, the `Builder` object which is assigning a new job, a list of `WorkerForBuilder` objects and the `BuildRequest`. The function should return one of the `WorkerForBuilder` objects, or `None` if none of the available workers should be used. As an example, for each `worker` in the list, `worker.worker` will be a `Worker` object, and `worker.worker.workername` is the worker's name. The function can optionally return a Deferred, which should fire with the same results.

`nextBuild` If provided, this is a function that controls which build request will be handled next. The function is passed two arguments, the `Builder` object which is assigning a new job, and a list of `BuildRequest` objects of pending builds. The function should return one of the `BuildRequest` objects, or `None` if none of the pending builds should be started. This function can optionally return a Deferred which should fire with the same results.

`canStartBuild` If provided, this is a function that can veto whether a particular worker should be used for a given build request. The function is passed three arguments: the `Builder`, a `Worker`, and a `BuildRequest`. The function should return `True` if the combination is acceptable, or `False` otherwise. This function can optionally return a Deferred which should fire with the same results.

`locks` This argument specifies a list of locks that apply to this builder; see *Interlocks*.

`env` A Builder may be given a dictionary of environment variables in this parameter. The variables are used in *`ShellCommand`* steps in builds created by this builder. The environment variables will override anything in

the worker's environment. Variables passed directly to a `ShellCommand` will override variables of the same name passed to the Builder.

For example, if you have a pool of identical workers it is often easier to manage variables like `PATH` from Buildbot rather than manually editing it inside of the workers' environment.

```
f = factory.BuildFactory
f.addStep(ShellCommand(
              command=['bash', './configure']))
f.addStep(Compile())

c['builders'] = [
  BuilderConfig(name='test', factory=f,
        workernames=['worker1', 'worker2', 'worker3', 'worker4'],
        env={'PATH': '/opt/local/bin:/opt/app/bin:/usr/local/bin:/usr/bin'}),
]
```

Unlike most builder configuration arguments, this argument can contain renderables.

`collapseRequests` Specifies how build requests for this builder should be collapsed. See *Collapsing Build Requests*, below.

`properties` A builder may be given a dictionary of *Build Properties* specific for this builder in this parameter. Those values can be used later on like other properties. *Interpolate*.

`description` A builder may be given an arbitrary description, which will show up in the web status on the builder's page.

Collapsing Build Requests

When more than one build request is available for a builder, Buildbot can "collapse" the requests into a single build. This is desirable when build requests arrive more quickly than the available workers can satisfy them, but has the drawback that separate results for each build are not available.

Requests are only candidated for a merge if both requests have exactly the same *codebases*.

This behavior can be controlled globally, using the *`collapseRequests`* parameter, and on a per-`Builder` basis, using the `collapseRequests` argument to the `Builder` configuration. If `collapseRequests` is given, it completely overrides the global configuration.

Possible values for both `collapseRequests` configurations are:

`True` Requests will be collapsed if their sourcestamp are compatible (see below for definition of compatible).

`False` Requests will never be collapsed.

`callable(builder, req1, req2)` Requests will be collapsed if the callable returns true. See *Collapse Request Functions* for detailed example.

Sourcestamps are compatible if all of the below conditions are met:

- Their codebase, branch, project, and repository attributes match exactly
- Neither source stamp has a patch (e.g., from a try scheduler)
- Either both source stamps are associated with changes, or neither are associated with changes but they have matching revisions.

Prioritizing Builds

The `BuilderConfig` parameter `nextBuild` can be use to prioritize build requests within a builder. Note that this is orthogonal to *Prioritizing Builders*, which controls the order in which builders are called on to start their builds. The details of writing such a function are in *Build Priority Functions*.

Such a function can be provided to the BuilderConfig as follows:

```
def pickNextBuild(builder, requests):
    ...
c['builders'] = [
    BuilderConfig(name='test', factory=f,
        nextBuild=pickNextBuild,
        workernames=['worker1', 'worker2', 'worker3', 'worker4']),
]
```

Build Factories

Each Builder is equipped with a `build factory`, which defines the steps used to perform that particular type of build. This factory is created in the configuration file, and attached to a Builder through the `factory` element of its dictionary.

The steps used by these builds are defined in the next section, *Build Steps*.

Note: Build factories are used with builders, and are not added directly to the buildmaster configuration dictionary.

Defining a Build Factory

A `BuildFactory` defines the steps that every build will follow. Think of it as a glorified script. For example, a build factory which consists of an SVN checkout followed by a `make build` would be configured as follows:

```
from buildbot.plugins import util, steps

f = util.BuildFactory()
f.addStep(steps.SVN(repourl="http://..", mode="incremental"))
f.addStep(steps.Compile(command=["make", "build"]))
```

This factory would then be attached to one builder (or several, if desired):

```
c['builders'].append(
    BuilderConfig(name='quick', workernames=['bot1', 'bot2'], factory=f))
```

It is also possible to pass a list of steps into the `BuildFactory` when it is created. Using `addStep` is usually simpler, but there are cases where it is more convenient to create the list of steps ahead of time, perhaps using some Python tricks to generate the steps.

```
from buildbot.plugins import steps, util

all_steps = [
    steps.CVS(cvsroot=CVSROOT, cvsmodule="project", mode="update"),
    steps.Compile(command=["make", "build"]),
]
f = util.BuildFactory(all_steps)
```

Finally, you can also add a sequence of steps all at once:

```
f.addSteps(all_steps)
```

Attributes

The following attributes can be set on a build factory after it is created, e.g.,

```
f = util.BuildFactory()
f.useProgress = False
```

useProgress (defaults to `True`): if `True`, the buildmaster keeps track of how long each step takes, so it can provide estimates of how long future builds will take. If builds are not expected to take a consistent amount of time (such as incremental builds in which a random set of files are recompiled or tested each time), this should be set to `False` to inhibit progress-tracking.

workdir (defaults to 'build'): workdir given to every build step created by this factory as default. The workdir can be overridden in a build step definition.

If this attribute is set to a string, that string will be used for constructing the workdir (worker base + builder builddir + workdir). The attribute can also be a Python callable, for more complex cases, as described in *Factory Workdir Functions*.

Dynamic Build Factories

In some cases you may not know what commands to run until after you checkout the source tree. For those cases you can dynamically add steps during a build from other steps.

The `Build` object provides 2 functions to do this:

addStepsAfterCurrentStep(self, step_factories) This adds the steps after the step that is currently executing.

addStepsAfterLastStep(self, step_factories) This adds the steps onto the end of the build.

Both functions only accept as an argument a list of steps to add to the build.

For example lets say you have a script checked in into your source tree called build.sh. When this script is called with the argument `--list-stages` it outputs a newline separated list of stage names. This can be used to generate at runtime a step for each stage in the build. Each stage is then run in this example using `./build.sh --run-stage <stage name>`.

```
from buildbot.plugins import util, steps
from buildbot.process import buildstep, logobserver
from twisted.internet import defer

class GenerateStagesCommand(buildstep.ShellMixin, steps.BuildStep):

```

```
    def __init__(self, **kwargs):
        kwargs = self.setupShellMixin(kwargs)
        steps.BuildStep.__init__(self, **kwargs)
        self.observer = logobserver.BufferLogObserver()
        self.addLogObserver('stdio', self.observer)

    def extract_stages(self, stdout):
        stages = []
        for line in stdout.split('\n'):
            stage = str(line.strip())
            if stage:
                stages.append(stage)
        return stages

    @defer.inlineCallbacks
    def run(self):
        # run './build.sh --list-stages' to generate the list of stages
        cmd = yield self.makeRemoteShellCommand()
        yield self.runCommand(cmd)

        # if the command passes extract the list of stages
        result = cmd.results()
        if result == util.SUCCESS:
            # create a ShellCommand for each stage and add them to the build
            self.build.addStepsAfterCurrentStep([
                steps.ShellCommand(name=stage, command=["./build.sh", "--run-stage",␣
↪stage])
                for stage in self.extract_stages(self.observer.getStdout())
            ])

        defer.returnValue(result)

f = util.BuildFactory()
f.addStep(steps.Git(repourl=repourl))
f.addStep(GenerateStagesCommand(
    name="Generate build stages",
    command=["./build.sh", "--list-stages"],
    haltOnFailure=True))
```

Predefined Build Factories

Buildbot includes a few predefined build factories that perform common build sequences. In practice, these are rarely used, as every site has slightly different requirements, but the source for these factories may provide examples for implementation of those requirements.

GNUAutoconf

class buildbot.process.factory.**GNUAutoconf**

GNU Autoconf (http://www.gnu.org/software/autoconf/) is a software portability tool, intended to make it possible to write programs in C (and other languages) which will run on a variety of UNIX-like systems. Most GNU software is built using autoconf. It is frequently used in combination with GNU automake. These tools both encourage a build process which usually looks like this:

```
% CONFIG_ENV=foo ./configure --with-flags
% make all
% make check
# make install
```

(except of course the Buildbot always skips the `make install` part).

The Buildbot's *buildbot.process.factory.GNUAutoconf* factory is designed to build projects which use GNU autoconf and/or automake. The configuration environment variables, the configure flags, and command lines used for the compile and test are all configurable, in general the default values will be suitable.

Example:

```
f = util.GNUAutoconf(source=source.SVN(repourl=URL, mode="copy"),
                     flags=["--disable-nls"])
```

Required Arguments:

source This argument must be a step specification tuple that provides a BuildStep to generate the source tree.

Optional Arguments:

configure The command used to configure the tree. Defaults to **./configure**. Accepts either a string or a list of shell argv elements.

configureEnv The environment used for the initial configuration step. This accepts a dictionary which will be merged into the worker's normal environment. This is commonly used to provide things like `CFLAGS="-O2 -g"` (to turn off debug symbols during the compile). Defaults to an empty dictionary.

configureFlags A list of flags to be appended to the argument list of the configure command. This is commonly used to enable or disable specific features of the autoconf-controlled package, like `["--without-x"]` to disable windowing support. Defaults to an empty list.

reconf use autoreconf to generate the ./configure file, set to True to use a buildbot default autoreconf command, or define the command for the ShellCommand.

compile this is a shell command or list of argv values which is used to actually compile the tree. It defaults to `make all`. If set to `None`, the compile step is skipped.

test this is a shell command or list of argv values which is used to run the tree's self-tests. It defaults to `make check`. If set to None, the test step is skipped.

distcheck this is a shell command or list of argv values which is used to run the packaging test. It defaults to `make distcheck`. If set to None, the test step is skipped.

BasicBuildFactory

class `buildbot.process.factory.`**BasicBuildFactory**

This is a subclass of `GNUAutoconf` which assumes the source is in CVS, and uses `mode='full'` and `method='clobber'` to always build from a clean working copy.

BasicSVN

class `buildbot.process.factory.`**BasicSVN**

This class is similar to `QuickBuildFactory`, but uses SVN instead of CVS.

QuickBuildFactory

class buildbot.process.factory.**QuickBuildFactory**

The QuickBuildFactory class is a subclass of GNUAutoconf which assumes the source is in CVS, and uses mode='incremental' to get incremental updates.

The difference between a *full build* and a *quick build* is that quick builds are generally done incrementally, starting with the tree where the previous build was performed. That simply means that the source-checkout step should be given a mode='incremental' flag, to do the source update in-place.

In addition to that, this class sets the useProgress flag to False. Incremental builds will (or at least the ought to) compile as few files as necessary, so they will take an unpredictable amount of time to run. Therefore it would be misleading to claim to predict how long the build will take.

This class is probably not of use to new projects.

CPAN

class buildbot.process.factory.**CPAN**

Most Perl modules available from the CPAN (http://www.cpan.org/) archive use the MakeMaker module to provide configuration, build, and test services. The standard build routine for these modules looks like:

```
% perl Makefile.PL
% make
% make test
# make install
```

(except again Buildbot skips the install step)

Buildbot provides a CPAN factory to compile and test these projects.

Arguments:

source (required): A step specification tuple, like that used by GNUAutoconf.

perl A string which specifies the **perl** executable to use. Defaults to just **perl**.

Distutils

class buildbot.process.factory.**Distutils**

Most Python modules use the distutils package to provide configuration and build services. The standard build process looks like:

```
% python ./setup.py build
% python ./setup.py install
```

Unfortunately, although Python provides a standard unit-test framework named unittest, to the best of my knowledge distutils does not provide a standardized target to run such unit tests. (Please let me know if I'm wrong, and I will update this factory.)

The Distutils factory provides support for running the build part of this process. It accepts the same source= parameter as the other build factories.

Arguments:

source (required): A step specification tuple, like that used by GNUAutoconf.

python A string which specifies the **python** executable to use. Defaults to just **python**.

test Provides a shell command which runs unit tests. This accepts either a string or a list. The default value is None, which disables the test step (since there is no common default command to run unit tests in distutils modules).

Trial

class buildbot.process.factory.**Trial**

Twisted provides a unit test tool named **trial** which provides a few improvements over Python's built-in unittest module. Many Python projects which use Twisted for their networking or application services also use trial for their unit tests. These modules are usually built and tested with something like the following:

```
% python ./setup.py build
% PYTHONPATH=build/lib.linux-i686-2.3 trial -v PROJECTNAME.test
% python ./setup.py install
```

Unfortunately, the build/lib directory into which the built/copied .py files are placed is actually architecture-dependent, and I do not yet know of a simple way to calculate its value. For many projects it is sufficient to import their libraries *in place* from the tree's base directory (PYTHONPATH=.).

In addition, the *PROJECTNAME* value where the test files are located is project-dependent: it is usually just the project's top-level library directory, as common practice suggests the unit test files are put in the test sub-module. This value cannot be guessed, the Trial class must be told where to find the test files.

The Trial class provides support for building and testing projects which use distutils and trial. If the test module name is specified, trial will be invoked. The library path used for testing can also be set.

One advantage of trial is that the Buildbot happens to know how to parse trial output, letting it identify which tests passed and which ones failed. The Buildbot can then provide fine-grained reports about how many tests have failed, when individual tests fail when they had been passing previously, etc.

Another feature of trial is that you can give it a series of source .py files, and it will search them for special test-case-name tags that indicate which test cases provide coverage for that file. Trial can then run just the appropriate tests. This is useful for quick builds, where you want to only run the test cases that cover the changed functionality.

Arguments:

testpath Provides a directory to add to PYTHONPATH when running the unit tests, if tests are being run. Defaults to . to include the project files in-place. The generated build library is frequently architecture-dependent, but may simply be build/lib for pure-Python modules.

python which Python executable to use. This list will form the start of the *argv* array that will launch trial. If you use this, you should set trial to an explicit path (like /usr/bin/trial or ./bin/trial). The parameter defaults to None, which leaves it out entirely (running trial args instead of python ./bin/trial args). Likely values are ['python'], ['python2.2'], or ['python', '-Wall'].

trial provides the name of the **trial** command. It is occasionally useful to use an alternate executable, such as **trial2.2** which might run the tests under an older version of Python. Defaults to **trial**.

trialMode a list of arguments to pass to trial, specifically to set the reporting mode. This defaults to ['--reporter=bwverbose'], which only works for Twisted-2.1.0 and later.

trialArgs a list of arguments to pass to trial, available to turn on any extra flags you like. Defaults to [].

tests Provides a module name or names which contain the unit tests for this project. Accepts a string, typically *PROJECTNAME*.test, or a list of strings. Defaults to None, indicating that no tests should be run. You must either set this or testChanges.

testChanges if `True`, ignore the `tests` parameter and instead ask the Build for all the files that make up the Changes going into this build. Pass these filenames to trial and ask it to look for test-case-name tags, running just the tests necessary to cover the changes.

recurse If `True`, tells Trial (with the `--recurse` argument) to look in all subdirectories for additional test cases.

reactor which reactor to use, like 'gtk' or 'java'. If not provided, the Twisted's usual platform-dependent default is used.

randomly If `True`, tells Trial (with the `--random=0` argument) to run the test cases in random order, which sometimes catches subtle inter-test dependency bugs. Defaults to `False`.

The step can also take any of the `ShellCommand` arguments, e.g., `haltOnFailure`.

Unless one of `tests` or `testChanges` are set, the step will generate an exception.

Properties

Build properties are a generalized way to provide configuration information to build steps; see *Build Properties* for the conceptual overview of properties.

Some build properties come from external sources and are set before the build begins; others are set during the build, and available for later steps. The sources for properties are:

global configuration These properties apply to all builds.

schedulers A scheduler can specify properties that become available to all builds it starts.

changes A change can have properties attached to it, supplying extra information gathered by the change source. This is most commonly used with the *sendchange* command.

forced builds The "Force Build" form allows users to specify properties

workers A worker can pass properties on to the builds it performs.

builds A build automatically sets a number of properties on itself.

builders A builder can set properties on all the builds it runs.

steps The steps of a build can set properties that are available to subsequent steps. In particular, source steps set the *got_revision* property.

If the same property is supplied in multiple places, the final appearance takes precedence. For example, a property set in a builder configuration will override one supplied by a scheduler.

Properties are stored internally in JSON format, so they are limited to basic types of data: numbers, strings, lists, and dictionaries.

Common Build Properties

The following build properties are set when the build is started, and are available to all steps.

got_revision This property is set when a `Source` step checks out the source tree, and provides the revision that was actually obtained from the VC system. In general this should be the same as `revision`, except for non-absolute sourcestamps, where `got_revision` indicates what revision was current when the checkout was performed. This can be used to rebuild the same source code later.

Note: For some VC systems (Darcs in particular), the revision is a large string containing newlines, and is not suitable for interpolation into a filename.

For multi-codebase builds (where codebase is not the default ''), this property is a dictionary, keyed by codebase.

buildername This is a string that indicates which `Builder` the build was a part of. The combination of buildername and buildnumber uniquely identify a build.

buildnumber Each build gets a number, scoped to the `Builder` (so the first build performed on any given `Builder` will have a build number of 0). This integer property contains the build's number.

workername This is a string which identifies which worker the build is running on.

scheduler If the build was started from a scheduler, then this property will contain the name of that scheduler.

builddir The absolute path of the base working directory on the worker, of the current builder.

For single codebase builds, where the codebase is '', the following *Source Stamp Attributes* are also available as properties: `branch`, `revision`, `repository`, and `project` .

Source Stamp Attributes

`branch` `revision` `repository` `project` `codebase`

For details of these attributes see *Concepts*.

`changes`

This attribute is a list of dictionaries representing the changes that make up this sourcestamp.

Using Properties in Steps

For the most part, properties are used to alter the behavior of build steps during a build. This is done by using renderables (objects implementing the `IRenderable` interface) as step parameters. When the step is started, each such object is rendered using the current values of the build properties, and the resultant rendering is substituted as the actual value of the step parameter.

Buildbot offers several renderable object types covering common cases. It's also possible to *create custom renderables*.

Note: Properties are defined while a build is in progress; their values are not available when the configuration file is parsed. This can sometimes confuse newcomers to Buildbot! In particular, the following is a common error:

```
if Property('release_train') == 'alpha':
    f.addStep(...)
```

This does not work because the value of the property is not available when the `if` statement is executed. However, Python will not detect this as an error - you will just never see the step added to the factory.

You can use renderables in most step parameters. Please file bugs for any parameters which do not accept renderables.

Property

The simplest renderable is `Property`, which renders to the value of the property named by its argument:

```
from buildbot.plugins import steps, util

f.addStep(steps.ShellCommand(command=['echo', 'buildername:',
                             util.Property('buildername')]))
```

You can specify a default value by passing a `default` keyword argument:

```
f.addStep(steps.ShellCommand(command=['echo', 'warnings:',
                             util.Property('warnings', default='none')]))
```

The default value is used when the property doesn't exist, or when the value is something Python regards as `False`. The `defaultWhenFalse` argument can be set to `False` to force buildbot to use the default argument only if the parameter is not set:

```
f.addStep(steps.ShellCommand(command=['echo', 'warnings:',
                             util.Property('warnings', default='none',
                                           defaultWhenFalse=False)]))
```

The default value can be a renderable itself, e.g.,

```
command=util.Property('command', default=util.Property('default-command'))
```

Interpolate

`Property` can only be used to replace an entire argument: in the example above, it replaces an argument to `echo`. Often, properties need to be interpolated into strings, instead. The tool for that job is *Interpolate*.

The more common pattern is to use Python dictionary-style string interpolation by using the `%(prop:<propname>)s` syntax. In this form, the property name goes in the parentheses, as above. A common mistake is to omit the trailing "s", leading to a rather obscure error from Python ("ValueError: unsupported format character").

```
from buildbot.plugins import steps, util
f.addStep(steps.ShellCommand(command=['make',
                                      util.Interpolate('REVISION=%(prop:got_revision)s
↪'),
                                      'dist']))
```

This example will result in a `make` command with an argument like `REVISION=12098`. The syntax of dictionary-style interpolation is a selector, followed by a colon, followed by a selector specific key, optionally followed by a colon and a string indicating how to interpret the value produced by the key.

The following selectors are supported.

prop The key is the name of a property.

src The key is a codebase and source stamp attribute, separated by a colon. Note, it is `%(src:<codebase>:<ssattr>)s` syntax, which differs from other selectors.

kw The key refers to a keyword argument passed to `Interpolate`. Those keyword arguments may be ordinary values or renderables.

secrets The key refers to a secret provided by a provider declared in *secretsProviders*.

The following ways of interpreting the value are available.

-replacement If the key exists, substitute its value; otherwise, substitute `replacement`. `replacement` may be empty (`%(prop:propname:-)s`). This is the default.

~replacement Like `-replacement`, but only substitutes the value of the key if it is something Python regards as `True`. Python considers `None`, 0, empty lists, and the empty string to be false, so such values will be replaced by `replacement`.

+replacement If the key exists, substitute `replacement`; otherwise, substitute an empty string.

`?|sub_if_exists|sub_if_missing`

#?|sub_if_true|sub_if_false Ternary substitution, depending on either the key being present (with `?`, similar to `+`) or being `True` (with `#?`, like `~`). Notice that there is a pipe immediately following the question mark *and* between the two substitution alternatives. The character that follows the question mark is used as the delimiter between the two alternatives. In the above examples, it is a pipe, but any character other than `(` can be used.

Note: Although these are similar to shell substitutions, no other substitutions are currently supported.

Example:

```
from buildbot.plugins import steps, util
f.addStep(steps.ShellCommand(
    command=[
        'save-build-artifacts-script.sh',
        util.Interpolate('-r %(prop:repository)s'),
        util.Interpolate('-b %(src::branch)s'),
        util.Interpolate('-d %(kw:data)s', data="some extra needed data")
    ]))
```

Note: We use `%(src::branch)s` in most of examples, because `codebase` is empty by default.

Example:

```
from buildbot.plugins import steps, util
f.addStep(steps.ShellCommand(
    command=[
        'make',
        util.Interpolate('REVISION=%(prop:got_revision:-%(src::revision:-unknown)s)s
↪'),
        'dist'
    ]))
```

In addition, `Interpolate` supports using positional string interpolation. Here, `%s` is used as a placeholder, and the substitutions (which may be renderables), are given as subsequent arguments:

```
TODO
```

Note: Like Python, you can use either positional interpolation *or* dictionary-style interpolation, not both. Thus you cannot use a string like `Interpolate("foo-%(src::revision)s-%s", "branch")`.

Renderer

While Interpolate can handle many simple cases, and even some common conditionals, more complex cases are best handled with Python code. The `renderer` decorator creates a renderable object whose rendering is obtained by calling the decorated function when the step it's passed to begins. The function receives an `IProperties` object, which it can use to examine the values of any and all properties. For example:

```
from buildbot.plugins import steps, util

@util.renderer
def makeCommand(props):
    command = ['make']
    cpus = props.getProperty('CPUs')
    if cpus:
        command.extend(['-j', str(cpus+1)])
    else:
        command.extend(['-j', '2'])
    command.extend([Interpolate('%(prop:MAKETARGET)s')])
    return command

f.addStep(steps.ShellCommand(command=makeCommand))
```

You can think of `renderer` as saying "call this function when the step starts".

Note: Since 0.9.3, renderer can itself return `IRenderable` objects or containers containing `IRenderable`.

Note: Config errors with Renderables may not always be caught via checkconfig

Transform

`Transform` is an alternative to `renderer`. While `renderer` is useful for creating new renderables, `Transform` is easier to use when you want to transform or combine the renderings of preexisting ones.

`Transform` takes a function and any number of positional and keyword arguments. The function must either be a callable object or a renderable producing one. When rendered, a `Transform` first replaces all of its arguments that are renderables with their renderings, then calls the function, passing it the positional and keyword arguments, and returns the result as its own rendering.

For example, suppose `my_path` is a path on the worker, and you want to get it relative to the build directory. You can do it like this:

```
import os.path
from buildbot.plugins import util

my_path_rel = util.Transform(os.path.relpath, my_path, start=util.Property('builddir
↪'))
```

This works whether `my_path` is an ordinary string or a renderable. `my_path_rel` will be a renderable in either case, however.

FlattenList

If nested list should be flatten for some renderables, FlattenList could be used. For example:

```
f.addStep(ShellCommand(command=[ 'make' ], descriptionDone=FlattenList([ 'make ', [
↪'done' ]])))
```

`descriptionDone` would be set to `[ 'make', 'done' ]` when the `ShellCommand` executes. This is useful when a list-returning property is used in renderables.

Note: ShellCommand automatically flattens nested lists in its `command` argument, so there is no need to use `FlattenList` for it.

WithProperties

Warning: This class is deprecated. It is an older version of *Interpolate*. It exists for compatibility with older configs.

The simplest use of this class is with positional string interpolation. Here, `%s` is used as a placeholder, and property names are given as subsequent arguments:

```
from buildbot.plugins import steps, util
f.addStep(steps.ShellCommand(
    command=["tar", "czf",
             util.WithProperties("build-%s-%s.tar.gz", "branch", "revision"),
             "source"]))
```

If this `BuildStep` were used in a tree obtained from Git, it would create a tarball with a name like `build-master-a7d3a333db708e786edb34b6af646edd8d4d3ad9.tar.gz`.

The more common pattern is to use Python dictionary-style string interpolation by using the `%(propname)s` syntax. In this form, the property name goes in the parentheses, as above. A common mistake is to omit the trailing "s", leading to a rather obscure error from Python ("ValueError: unsupported format character").

```
from buildbot.plugins import steps, util
f.addStep(steps.ShellCommand(command=['make',
                                      util.WithProperties('REVISION=%(got_revision)s
↪'),
                                      'dist']))
```

This example will result in a `make` command with an argument like `REVISION=12098`. The dictionary-style interpolation supports a number of more advanced syntaxes in the parentheses.

`propname:-replacement` If `propname` exists, substitute its value; otherwise, substitute `replacement`. `replacement` may be empty (`%(propname:-)s`)

`propname:~replacement` Like `propname:-replacement`, but only substitutes the value of property `propname` if it is something Python regards as `True`. Python considers `None`, 0, empty lists, and the empty string to be false, so such values will be replaced by `replacement`.

`propname:+replacement` If `propname` exists, substitute `replacement`; otherwise, substitute an empty string.

Although these are similar to shell substitutions, no other substitutions are currently supported, and `replacement` in the above cannot contain more substitutions.

Note: like Python, you can use either positional interpolation *or* dictionary-style interpolation, not both. Thus you cannot use a string like `WithProperties("foo-%(revision)s-%s", "branch")`.

Custom Renderables

If the options described above are not sufficient, more complex substitutions can be achieved by writing custom renderables.

The `IRenderable` interface is simple - objects must provide a *getRenderingFor* method. The method should take one argument - an `IProperties` provider - and should return the rendered value or a deferred firing with one. Pass instances of the class anywhere other renderables are accepted. For example:

```
@implementer(IRenderable)
class DetermineFoo(object):
    def getRenderingFor(self, props):
        if props.hasProperty('bar'):
            return props['bar']
        elif props.hasProperty('baz'):
            return props['baz']
        return 'qux'
ShellCommand(command=['echo', DetermineFoo()])
```

or, more practically,

```
@implementer(IRenderable)
class Now(object):
    def getRenderingFor(self, props):
        return time.clock()
ShellCommand(command=['make', Interpolate('TIME=%(kw:now)s', now=Now())])
```

This is equivalent to:

```
@renderer
def now(props):
    return time.clock()
ShellCommand(command=['make', Interpolate('TIME=%(kw:now)s', now=now)])
```

Note that a custom renderable must be instantiated (and its constructor can take whatever arguments you'd like), whereas a function decorated with `renderer` can be used directly.

URL for build

Its common to need to use the URL for the build in a step. For this you can use a special custom renderer as following:

```
from buildbot.plugins import *

ShellCommand(command=['make', Interpolate('BUILDURL=%(kw:url)s', url=util.
↪URLForBuild)])
```

Build Steps

`BuildSteps` are usually specified in the buildmaster's configuration file, in a list that goes into the `BuildFactory`.

The BuildStep instances in this list are used as templates to construct new independent copies for each build (so that state can be kept on the BuildStep in one build without affecting a later build). Each BuildFactory can be created with a list of steps, or the factory can be created empty and then steps added to it using the addStep method:

```
from buildbot.plugins import util, steps

f = util.BuildFactory()
f.addSteps([
    steps.SVN(repourl="http://svn.example.org/Trunk/"),
    steps.ShellCommand(command=["make", "all"]),
    steps.ShellCommand(command=["make", "test"])
])
```

The basic behavior for a BuildStep is to:

- run for a while, then stop
- possibly invoke some RemoteCommands on the attached worker
- possibly produce a set of log files
- finish with a status described by one of four values defined in buildbot.status.builder: SUCCESS, WARNINGS, FAILURE, SKIPPED
- provide a list of short strings to describe the step

The rest of this section describes all the standard BuildStep objects available for use in a Build, and the parameters which can be used to control each. A full list of build steps is available in the step.

Common Parameters

All `BuildSteps` accept some common parameters. Some of these control how their individual status affects the overall build. Others are used to specify which *Locks* (see *Interlocks*) should be acquired before allowing the step to run.

Arguments common to all `BuildStep` subclasses:

`name` the name used to describe the step on the status display. It is also used to give a name to any `LogFiles` created by this step.

`haltOnFailure` if `True`, a `FAILURE` of this build step will cause the build to halt immediately. Steps with `alwaysRun=True` are still run. Generally speaking, `haltOnFailure` implies `flunkOnFailure` (the default for most `BuildSteps`). In some cases, particularly series of tests, it makes sense to `haltOnFailure` if something fails early on but not `flunkOnFailure`. This can be achieved with `haltOnFailure=True`, `flunkOnFailure=False`.

`flunkOnWarnings` when `True`, a `WARNINGS` or `FAILURE` of this build step will mark the overall build as `FAILURE`. The remaining steps will still be executed.

`flunkOnFailure` when `True`, a `FAILURE` of this build step will mark the overall build as a `FAILURE`. The remaining steps will still be executed.

`warnOnWarnings` when `True`, a `WARNINGS` or `FAILURE` of this build step will mark the overall build as having `WARNINGS`. The remaining steps will still be executed.

`warnOnFailure` when `True`, a `FAILURE` of this build step will mark the overall build as having `WARNINGS`. The remaining steps will still be executed.

`alwaysRun` if `True`, this build step will always be run, even if a previous buildstep with `haltOnFailure=True` has failed.

description This will be used to describe the command (on the Waterfall display) while the command is still running. It should be a single imperfect-tense verb, like *compiling* or *testing*. The preferred form is a single, short string, but for historical reasons a list of strings is also acceptable.

descriptionDone This will be used to describe the command once it has finished. A simple noun like *compile* or *tests* should be used. Like `description`, this may either be a string or a list of short strings.

If neither `description` nor `descriptionDone` are set, the actual command arguments will be used to construct the description. This may be a bit too wide to fit comfortably on the Waterfall display.

All subclasses of `BuildStep` will contain the description attributes. Consequently, you could add a *`ShellCommand`* step like so:

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(command=["make", "test"],
                             description="testing",
                             descriptionDone="tests"))
```

descriptionSuffix This is an optional suffix appended to the end of the description (ie, after `description` and `descriptionDone`). This can be used to distinguish between build steps that would display the same descriptions in the waterfall. This parameter may be a string, a list of short strings or `None`.

For example, a builder might use the *`Compile`* step to build two different codebases. The `descriptionSuffix` could be set to *projectFoo* and *projectBar*, respectively for each step, which will result in the full descriptions *compiling projectFoo* and *compiling projectBar* to be shown in the waterfall.

doStepIf A step can be configured to only run under certain conditions. To do this, set the step's `doStepIf` to a boolean value, or to a function that returns a boolean value or Deferred. If the value or function result is false, then the step will return `SKIPPED` without doing anything. Otherwise, the step will be executed normally. If you set `doStepIf` to a function, that function should accept one parameter, which will be the `Step` object itself.

hideStepIf A step can be optionally hidden from the waterfall and build details web pages. To do this, set the step's `hideStepIf` to a boolean value, or to a function that takes two parameters – the results and the `BuildStep` – and returns a boolean value. Steps are always shown while they execute, however after the step as finished, this parameter is evaluated (if a function) and if the value is True, the step is hidden. For example, in order to hide the step if the step has been skipped:

```
factory.addStep(Foo(..., hideStepIf=lambda results, s: results==SKIPPED))
```

locks a list of `Locks` (instances of `buildbot.locks.WorkerLock` or `buildbot.locks.MasterLock`) that should be acquired before starting this `BuildStep`. Alternatively this could be a renderable that returns this list during build execution. This lets you defer picking the locks to acquire until the build step is about to start running. The `Locks` will be released when the step is complete. Note that this is a list of actual `Lock` instances, not names. Also note that all Locks must have unique names. See *Interlocks*.

logEncoding The character encoding to use to decode logs produced during the execution of this step. This overrides the default *`logEncoding`*; see *Log Handling*.

Source Checkout

> **Caution:** Support for the old worker-side source checkout steps was removed in Buildbot-0.9.0.
>
> The old source steps used to be imported like this:

```
from buildbot.steps.source.oldsource import Git

... Git ...
```

or:

```
from buildbot.steps.source import Git
```

while new source steps are in separate Python modules for each version-control system and, using the plugin infrastructure are available as:

```
from buildbot.plugins import steps

... steps.Git ...
```

Common Parameters

All source checkout steps accept some common parameters to control how they get the sources and where they should be placed. The remaining per-VC-system parameters are mostly to specify where exactly the sources are coming from.

`mode` `method`

These two parameters specify the means by which the source is checked out. `mode` specifies the type of checkout and `method` tells about the way to implement it.

```
from buildbot.plugins import steps

factory = BuildFactory()
factory.addStep(steps.Mercurial(repourl='path/to/repo', mode='full',
                                method='fresh'))
```

The `mode` parameter a string describing the kind of VC operation that is desired, defaulting to `incremental`. The options are

`incremental` Update the source to the desired revision, but do not remove any other files generated by previous builds. This allows compilers to take advantage of object files from previous builds. This mode is exactly same as the old `update` mode.

`full` Update the source, but delete remnants of previous builds. Build steps that follow will need to regenerate all object files.

Methods are specific to the version-control system in question, as they may take advantage of special behaviors in that version-control system that can make checkouts more efficient or reliable.

`workdir` like all Steps, this indicates the directory where the build will take place. Source Steps are special in that they perform some operations outside of the workdir (like creating the workdir itself).

`alwaysUseLatest` if True, bypass the usual behavior of checking out the revision in the source stamp, and always update to the latest revision in the repository instead.

`retry` If set, this specifies a tuple of `(delay, repeats)` which means that when a full VC checkout fails, it should be retried up to `repeats` times, waiting `delay` seconds between attempts. If you don't provide this, it defaults to `None`, which means VC operations should not be retried. This is provided to make life easier for workers which are stuck behind poor network connections.

`repository` The name of this parameter might vary depending on the Source step you are running. The concept explained here is common to all steps and applies to `repourl` as well as for `baseURL` (when applicable).

A common idiom is to pass `Property('repository', 'url://default/repo/path')` as repository. This grabs the repository from the source stamp of the build. This can be a security issue, if you allow force builds from the web, or have the `WebStatus` change hooks enabled; as the worker will download code from an arbitrary repository.

codebase This specifies which codebase the source step should use to select the right source stamp. The default codebase value is `''`. The codebase must correspond to a codebase assigned by the *codebaseGenerator*. If there is no codebaseGenerator defined in the master then codebase doesn't need to be set, the default value will then match all changes.

timeout Specifies the timeout for worker-side operations, in seconds. If your repositories are particularly large, then you may need to increase this value from its default of 1200 (20 minutes).

logEnviron If this option is true (the default), then the step's logfile will describe the environment variables on the worker. In situations where the environment is not relevant and is long, it may be easier to set `logEnviron=False`.

env a dictionary of environment strings which will be added to the child command's environment. The usual property interpolations can be used in environment variable names and values - see *Properties*.

Mercurial

class `buildbot.steps.source.mercurial.`**`Mercurial`**

The *Mercurial* build step performs a Mercurial (https://www.mercurial-scm.org/) (aka `hg`) checkout or update.

Branches are available in two modes: `dirname`, where the name of the branch is a suffix of the name of the repository, or `inrepo`, which uses Hg's named-branches support. Make sure this setting matches your changehook, if you have that installed.

```
from buildbot.plugins import steps

factory.addStep(steps.Mercurial(repourl='path/to/repo', mode='full',
                                method='fresh', branchType='inrepo'))
```

The Mercurial step takes the following arguments:

repourl where the Mercurial source repository is available.

defaultBranch this specifies the name of the branch to use when a Build does not provide one of its own. This will be appended to `repourl` to create the string that will be passed to the `hg clone` command.

branchType either 'dirname' (default) or 'inrepo' depending on whether the branch name should be appended to the `repourl` or the branch is a Mercurial named branch and can be found within the `repourl`.

clobberOnBranchChange boolean, defaults to `True`. If set and using inrepos branches, clobber the tree at each branch change. Otherwise, just update to the branch.

`mode` `method`

Mercurial's incremental mode does not require a method. The full mode has three methods defined:

clobber It removes the build directory entirely then makes full clone from repo. This can be slow as it need to clone whole repository

fresh This remove all other files except those tracked by VCS. First it does **`hg purge --all`** then pull/update

clean All the files which are tracked by Mercurial and listed ignore files are not deleted. Remaining all other files will be deleted before pull/update. This is equivalent to **`hg purge`** then pull/update.

Git

class buildbot.steps.source.git.**Git**

The *Git* build step clones or updates a Git (http://git.or.cz/) repository and checks out the specified branch or revision.

Note: The Buildbot supports Git version 1.2.0 and later: earlier versions (such as the one shipped in Ubuntu 'Dapper') do not support the **git init** command that the Buildbot uses.

```
from buildbot.plugins import steps

factory.addStep(steps.Git(repourl='git://path/to/repo', mode='full',
                          method='clobber', submodules=True))
```

The Git step takes the following arguments:

repourl (required): the URL of the upstream Git repository.

branch (optional): this specifies the name of the branch to use when a Build does not provide one of its own. If this parameter is not specified, and the Build does not provide a branch, the default branch of the remote repository will be used.

submodules (optional): when initializing/updating a Git repository, this tells Buildbot whether to handle Git submodules. Default: `False`.

shallow (optional): instructs git to attempt shallow clones (`--depth 1`). The depth defaults to 1 and can be changed by passing an integer instead of `True`. This option can be used only in full builds with clobber method.

reference (optional): use the specified string as a path to a reference repository on the local machine. Git will try to grab objects from this path first instead of the main repository, if they exist.

origin (optional): By default, any clone will use the name "origin" as the remote repository (eg, "origin/master"). This renderable option allows that to be configured to an alternate name.

progress (optional): passes the (`--progress`) flag to (**git fetch**). This solves issues of long fetches being killed due to lack of output, but requires Git 1.7.2 or later.

retryFetch (optional): defaults to `False`. If true, if the `git fetch` fails then buildbot retries to fetch again instead of failing the entire source checkout.

clobberOnFailure (optional): defaults to `False`. If a fetch or full clone fails we can checkout source removing everything. This way new repository will be cloned. If retry fails it fails the source checkout step.

`mode`

(optional): defaults to `'incremental'`. Specifies whether to clean the build tree or not.

incremental The source is update, but any built files are left untouched.

full The build tree is clean of any built files. The exact method for doing this is controlled by the `method` argument.

`method`

(optional): defaults to `fresh` when mode is `full`. Git's incremental mode does not require a method. The full mode has four methods defined:

clobber It removes the build directory entirely then makes full clone from repo. This can be slow as it need to clone whole repository. To make faster clones enable `shallow` option. If shallow options is enabled and build request have unknown revision value, then this step fails.

fresh This remove all other files except those tracked by Git. First it does **git clean -d -f -f -x** then fetch/checkout to a specified revision(if any). This option is equal to update mode with `ignore_ignores=True` in old steps.

clean All the files which are tracked by Git and listed ignore files are not deleted. Remaining all other files will be deleted before fetch/checkout. This is equivalent to **git clean -d -f -f** then fetch. This is equivalent to `ignore_ignores=False` in old steps.

copy This first checkout source into source directory then copy the `source` directory to `build` directory then performs the build operation in the copied directory. This way we make fresh builds with very less bandwidth to download source. The behavior of source checkout follows exactly same as incremental. It performs all the incremental checkout behavior in `source` directory.

`getDescription`

(optional) After checkout, invoke a *git describe* on the revision and save the result in a property; the property's name is either `commit-description` or `commit-description-foo`, depending on whether the `codebase` argument was also provided. The argument should either be a `bool` or `dict`, and will change how *git describe* is called:

- `getDescription=False`: disables this feature explicitly
- `getDescription=True` or empty `dict()`: Run *git describe* with no args
- `getDescription={...}`: a dict with keys named the same as the Git option. Each key's value can be `False` or `None` to explicitly skip that argument.

  For the following keys, a value of `True` appends the same-named Git argument:

  - `all`: *–all*
  - `always`: *–always*
  - `contains`: *–contains*
  - `debug`: *–debug*
  - `long`: *–long'*
  - `exact-match`: *–exact-match*
  - `tags`: *–tags*
  - `dirty`: *–dirty*

  For the following keys, an integer or string value (depending on what Git expects) will set the argument's parameter appropriately. Examples show the key-value pair:

  - `match=foo`: *–match foo*
  - `abbrev=7`: *–abbrev=7*
  - `candidates=7`: *–candidates=7*
  - `dirty=foo`: *–dirty=foo*

`config`

(optional) A dict of git configuration settings to pass to the remote git commands.

SVN

class `buildbot.steps.source.svn.`**SVN**

The *SVN* build step performs a Subversion (http://subversion.tigris.org) checkout or update. There are two basic ways of setting up the checkout step, depending upon whether you are using multiple branches or not.

The *SVN* step should be created with the `repourl` argument:

repourl (required): this specifies the `URL` argument that will be given to the **svn checkout** command. It dictates both where the repository is located and which sub-tree should be extracted. One way to specify the branch is to use `Interpolate`. For example, if you wanted to check out the trunk repository, you could use `repourl=Interpolate("http://svn.example.com/repos/%(src::branch)s")`. Alternatively, if you are using a remote Subversion repository which is accessible through HTTP at a URL of `http://svn.example.com/repos`, and you wanted to check out the `trunk/calc` sub-tree, you would directly use `repourl="http://svn.example.com/repos/trunk/calc"` as an argument to your *SVN* step.

If you are building from multiple branches, then you should create the *SVN* step with the `repourl` and provide branch information with *Interpolate*:

```
from buildbot.plugins import steps, util

factory.addStep(steps.SVN(mode='incremental',
                  repourl=util.Interpolate('svn://svn.example.org/svn/%(src::branch)s/
↪myproject')))
```

Alternatively, the `repourl` argument can be used to create the *SVN* step without *Interpolate*:

```
from buildbot.plugins import steps

factory.addStep(steps.SVN(mode='full',
                  repourl='svn://svn.example.org/svn/myproject/trunk'))
```

username (optional): if specified, this will be passed to the `svn` binary with a `--username` option.

password (optional): if specified, this will be passed to the `svn` binary with a `--password` option.

extra_args (optional): if specified, an array of strings that will be passed as extra arguments to the `svn` binary.

keep_on_purge (optional): specific files or directories to keep between purges, like some build outputs that can be reused between builds.

depth (optional): Specify depth argument to achieve sparse checkout. Only available if worker has Subversion 1.5 or higher.

If set to `empty` updates will not pull in any files or subdirectories not already present. If set to `files`, updates will pull in any files not already present, but not directories. If set to `immediates`, updates will pull in any files or subdirectories not already present, the new subdirectories will have depth: empty. If set to `infinity`, updates will pull in any files or subdirectories not already present; the new subdirectories will have depth-infinity. Infinity is equivalent to SVN default update behavior, without specifying any depth argument.

preferLastChangedRev (optional): By default, the `got_revision` property is set to the repository's global revision ("Revision" in the *svn info* output). Set this parameter to `True` to have it set to the "Last Changed Rev" instead.

`mode` `method`

SVN's incremental mode does not require a method. The full mode has five methods defined:

clobber It removes the working directory for each build then makes full checkout.

fresh This always always purges local changes before updating. This deletes unversioned files and reverts everything that would appear in a **svn status --no-ignore**. This is equivalent to the old update mode with `always_purge`.

clean This is same as fresh except that it deletes all unversioned files generated by **svn status**.

copy This first checkout source into source directory then copy the `source` directory to `build` directory then performs the build operation in the copied directory. This way we make fresh builds with very less bandwidth to download source. The behavior of source checkout follows exactly same as incremental. It performs all the incremental checkout behavior in `source` directory.

export Similar to `method='copy'`, except using `svn export` to create build directory so that there are no `.svn` directories in the build directory.

If you are using branches, you must also make sure your `ChangeSource` will report the correct branch names.

CVS

class `buildbot.steps.source.cvs.`**CVS**

The *CVS* build step performs a CVS (http://www.nongnu.org/cvs/) checkout or update.

```
from buildbot.plugins import steps

factory.addStep(steps.CVS(mode='incremental',
                cvsroot=':pserver:me@cvs.example.net:/cvsroot/myproj',
                cvsmodule='buildbot'))
```

This step takes the following arguments:

cvsroot (required): specify the CVSROOT value, which points to a CVS repository, probably on a remote machine. For example, if Buildbot was hosted in CVS then the CVSROOT value you would use to get a copy of the Buildbot source code might be `:pserver:anonymous@cvs.example.net:/cvsroot/buildbot`.

cvsmodule (required): specify the cvs `module`, which is generally a subdirectory of the `CVSROOT`. The cvsmodule for the Buildbot source code is `buildbot`.

branch a string which will be used in a `-r` argument. This is most useful for specifying a branch to work on. Defaults to `HEAD`.

global_options a list of flags to be put before the argument `checkout` in the CVS command.

extra_options a list of flags to be put after the `checkout` in the CVS command.

`mode` `method`

No method is needed for incremental mode. For full mode, `method` can take the values shown below. If no value is given, it defaults to `fresh`.

clobber This specifies to remove the `workdir` and make a full checkout.

fresh This method first runs `cvsdisard` in the build directory, then updates it. This requires `cvsdiscard` which is a part of the cvsutil package.

clean This method is the same as `method='fresh'`, but it runs `cvsdiscard --ignore` instead of `cvsdiscard`.

copy This maintains a `source` directory for source, which it updates copies to the build directory. This allows Buildbot to start with a fresh directory, without downloading the entire repository on every build.

login Password to use while performing login to the remote CVS server. Default is `None` meaning that no login needs to be peformed.

Bzr

class buildbot.steps.source.bzr.**Bzr**

bzr (http://bazaar.canonical.com/en/) is a descendant of Arch/Baz, and is frequently referred to as simply *Bazaar*. The repository-vs-workspace model is similar to Darcs, but it uses a strictly linear sequence of revisions (one history per branch) like Arch. Branches are put in subdirectories. This makes it look very much like Mercurial.

```
from buildbot.plugins import steps

factory.addStep(steps.Bzr(mode='incremental',
                          repourl='lp:~knielsen/maria/tmp-buildbot-test'))
```

The step takes the following arguments:

repourl (required unless baseURL is provided): the URL at which the Bzr source repository is available.

baseURL (required unless repourl is provided): the base repository URL, to which a branch name will be appended. It should probably end in a slash.

defaultBranch (allowed if and only if baseURL is provided): this specifies the name of the branch to use when a Build does not provide one of its own. This will be appended to baseURL to create the string that will be passed to the bzr checkout command.

mode method

No method is needed for incremental mode. For full mode, method can take the values shown below. If no value is given, it defaults to fresh.

clobber This specifies to remove the workdir and make a full checkout.

fresh This method first runs bzr clean-tree to remove all the unversioned files then update the repo. This remove all unversioned files including those in .bzrignore.

clean This is same as fresh except that it doesn't remove the files mentioned in .bzrginore i.e, by running bzr clean-tree --ignore.

copy A local bzr repository is maintained and the repo is copied to build directory for each build. Before each build the local bzr repo is updated then copied to build for next steps.

P4

class buildbot.steps.source.p4.**P4**

The *P4* build step creates a Perforce (http://www.perforce.com/) client specification and performs an update.

```
from buildbot.plugins import steps, util

factory.addStep(steps.P4(p4port=p4port,
                         p4client=util.WithProperties('%(P4USER)s-%(workername)s-
↪%(buildername)s'),
                         p4user=p4user,
                         p4base='//depot',
                         p4viewspec=p4viewspec,
                         mode='incremental'))
```

You can specify the client spec in two different ways. You can use the p4base, p4branch, and (optionally) p4extra_views to build up the viewspec, or you can utilize the p4viewspec to specify the whole viewspec as a set of tuples.

Using `p4viewspec` will allow you to add lines such as:

```
//depot/branch/mybranch/...              //<p4client>/...
-//depot/branch/mybranch/notthisdir/... //<p4client>/notthisdir/...
```

If you specify `p4viewspec` and any of `p4base`, `p4branch`, and/or `p4extra_views` you will receive a configuration error exception.

p4base A view into the Perforce depot without branch name or trailing `/...`. Typically `//depot/proj`.

p4branch (optional): A single string, which is appended to the p4base as follows `<p4base>/<p4branch>/...` to form the first line in the viewspec

p4extra_views (optional): a list of `(depotpath, clientpath)` tuples containing extra views to be mapped into the client specification. Both will have `/...` appended automatically. The client name and source directory will be prepended to the client path.

p4viewspec This will override any p4branch, p4base, and/or p4extra_views specified. The viewspec will be an array of tuples as follows:

```
[('//depot/main/','')]
```

It yields a viewspec with just:

```
//depot/main/... //<p4client>/...
```

p4viewspec_suffix (optional): The `p4viewspec` lets you customize the client spec for a builder but, as the previous example shows, it automatically adds `...` at the end of each line. If you need to also specify file-level remappings, you can set the `p4viewspec_suffix` to `None` so that nothing is added to your viewspec:

```
[('//depot/main/...', '...'),
 ('-//depot/main/config.xml', 'config.xml'),
 ('//depot/main/config.vancouver.xml', 'config.xml')]
```

It yields a viewspec with:

```
//depot/main/...                   //<p4client>/...
-//depot/main/config.xml           //<p4client/main/config.xml
//depot/main/config.vancouver.xml //<p4client>/main/config.xml
```

Note how, with `p4viewspec_suffix` set to `None`, you need to manually add `...` where you need it.

p4client_spec_options (optional): By default, clients are created with the `allwrite rmdir` options. This string lets you change that.

p4port (optional): the *`host:port`* string describing how to get to the P4 Depot (repository), used as the option *-p* argument for all p4 commands.

p4user (optional): the Perforce user, used as the option *-u* argument to all p4 commands.

p4passwd (optional): the Perforce password, used as the option *-p* argument to all p4 commands.

p4client (optional): The name of the client to use. In `mode='full'` and `mode='incremental'`, it's particularly important that a unique name is used for each checkout directory to avoid incorrect synchronization. For this reason, Python percent substitution will be performed on this value to replace `%(prop:workername)s` with the worker name and `%(prop:buildername)s` with the builder name. The default is `buildbot_%(prop:workername)s_%(prop:buildername)s`.

p4line_end (optional): The type of line ending handling P4 should use. This is added directly to the client spec's `LineEnd` property. The default is `local`.

p4extra_args (optional): Extra arguments to be added to the P4 command-line for the `sync` command. So for instance if you want to sync only to populate a Perforce proxy (without actually syncing files to disk), you can do:

```
P4(p4extra_args=['-Zproxyload'], ...)
```

use_tickets Set to `True` to use ticket-based authentication, instead of passwords (but you still need to specify `p4passwd`).

Repo

class `buildbot.steps.source.repo.`**Repo**

The *Repo* build step performs a Repo (http://lwn.net/Articles/304488/) init and sync.

The Repo step takes the following arguments:

manifestURL (required): the URL at which the Repo's manifests source repository is available.

manifestBranch (optional, defaults to `master`): the manifest repository branch on which repo will take its manifest. Corresponds to the `-b` argument to the **repo init** command.

manifestFile (optional, defaults to `default.xml`): the manifest filename. Corresponds to the `-m` argument to the **repo init** command.

tarball (optional, defaults to `None`): the repo tarball used for fast bootstrap. If not present the tarball will be created automatically after first sync. It is a copy of the `.repo` directory which contains all the Git objects. This feature helps to minimize network usage on very big projects with lots of workers.

jobs (optional, defaults to `None`): Number of projects to fetch simultaneously while syncing. Passed to repo sync subcommand with "-j".

syncAllBranches (optional, defaults to `False`): renderable boolean to control whether `repo` syncs all branches. I.e. `repo sync -c`

depth (optional, defaults to 0): Depth argument passed to repo init. Specifies the amount of git history to store. A depth of 1 is useful for shallow clones. This can save considerable disk space on very large projects.

updateTarballAge (optional, defaults to "one week"): renderable to control the policy of updating of the tarball given properties. Returns: max age of tarball in seconds, or `None`, if we want to skip tarball update. The default value should be good trade off on size of the tarball, and update frequency compared to cost of tarball creation

repoDownloads (optional, defaults to None): list of `repo download` commands to perform at the end of the Repo step each string in the list will be prefixed `repo download`, and run as is. This means you can include parameter in the string. For example:

- `["-c project 1234/4"]` will cherry-pick patchset 4 of patch 1234 in project `project`
- `["-f project 1234/4"]` will enforce fast-forward on patchset 4 of patch 1234 in project `project`

class `buildbot.steps.source.repo.`**RepoDownloadsFromProperties**

`util.repo.DownloadsFromProperties` can be used as a renderable of the `repoDownload` parameter it will look in passed properties for string with following possible format:

- `repo download project change_number/patchset_number`
- `project change_number/patchset_number`
- `project/change_number/patchset_number`

All of these properties will be translated into a **repo download**. This feature allows integrators to build with several pending interdependent changes, which at the moment cannot be described properly in Gerrit, and can only be described by humans.

class buildbot.steps.source.repo.**RepoDownloadsFromChangeSource**

util.repo.DownloadsFromChangeSource can be used as a renderable of the repoDownload parameter

This rendereable integrates with *GerritChangeSource*, and will automatically use the **repo download** command of repo to download the additional changes introduced by a pending changeset.

Note: You can use the two above Rendereable in conjunction by using the class buildbot.process.properties.FlattenList

---

For example:

```
from buildbot.plugins import steps, util

factory.addStep(steps.Repo(manifestURL='git://gerrit.example.org/manifest.git',
                           repoDownloads=util.FlattenList([
                                util.RepoDownloadsFromChangeSource(),
                                util.RepoDownloadsFromProperties("repo_downloads")
                           ])))
```

Gerrit

class buildbot.steps.source.gerrit.**Gerrit**

Gerrit step is exactly like the *Git* step, except that it integrates with *GerritChangeSource*, and will automatically checkout the additional changes.

Gerrit integration can be also triggered using forced build with property named gerrit_change with values in format change_number/patchset_number. This property will be translated into a branch name. This feature allows integrators to build with several pending interdependent changes, which at the moment cannot be described properly in Gerrit, and can only be described by humans.

GitHub

class buildbot.steps.source.github.**GitHub**

GitHub step is exactly like the *Git* step, except that it will ignore the revision sent by *GitHub* change hook, and rather take the branch if the branch ends with /merge.

This allows to test github pull requests merged directly into the mainline.

GitHub indeed provides refs/origin/pull/NNN/merge on top of refs/origin/pull/NNN/head which is a magic ref that always create a merge commit to the latest version of the mainline (i.e. the target branch for the pull request).

The revision in the GitHub event points to /head is important for the GitHub reporter as this is the revision that will be tagged with a CI status when the build is finished.

If you want to use *Trigger* to create sub tests and want to have the GitHub reporter still update the original revision, make sure you set updateSourceStamp=False in the *Trigger* configuration.

Darcs

class buildbot.steps.source.darcs.**Darcs**

The *Darcs* build step performs a Darcs (http://darcs.net/) checkout or update.

```
from buildbot.plugins import steps

factory.addStep(steps.Darcs(repourl='http://path/to/repo',
                            mode='full', method='clobber', retry=(10, 1)))
```

Darcs step takes the following arguments:

repourl (required): The URL at which the Darcs source repository is available.

mode

(optional): defaults to `'incremental'`. Specifies whether to clean the build tree or not.

incremental The source is update, but any built files are left untouched.

full The build tree is clean of any built files. The exact method for doing this is controlled by the `method` argument.

method (optional): defaults to `copy` when mode is `full`. Darcs' incremental mode does not require a method. The full mode has two methods defined:

clobber It removes the working directory for each build then makes full checkout.

copy This first checkout source into source directory then copy the `source` directory to `build` directory then performs the build operation in the copied directory. This way we make fresh builds with very less bandwidth to download source. The behavior of source checkout follows exactly same as incremental. It performs all the incremental checkout behavior in `source` directory.

Monotone

class buildbot.steps.source.mtn.**Monotone**

The *Monotone* build step performs a Monotone (http://www.monotone.ca/) checkout or update.

```
from buildbot.plugins import steps

factory.addStep(steps.Monotone(repourl='http://path/to/repo',
                               mode='full', method='clobber',
                               branch='some.branch.name', retry=(10, 1)))
```

Monotone step takes the following arguments:

repourl the URL at which the Monotone source repository is available.

branch this specifies the name of the branch to use when a Build does not provide one of its own.

progress this is a boolean that has a pull from the repository use `--ticker=dot` instead of the default `--ticker=none`.

mode

(optional): defaults to `'incremental'`. Specifies whether to clean the build tree or not. In any case, the worker first pulls from the given remote repository to synchronize (or possibly initialize) its local database. The mode and method only affect how the build tree is checked-out or updated from the local database.

incremental The source is update, but any built files are left untouched.

full The build tree is clean of any built files. The exact method for doing this is controlled by the `method` argument. Even in this mode, the revisions already pulled remain in the database and a fresh pull is rarely needed.

`method`

(optional): defaults to `copy` when mode is `full`. Monotone's incremental mode does not require a method. The full mode has four methods defined:

clobber It removes the build directory entirely then makes fresh checkout from the database.

clean This remove all other files except those tracked and ignored by Monotone. It will remove all the files that appear in **mtn ls unknown**. Then it will pull from remote and update the working directory.

fresh This remove all other files except those tracked by Monotone. It will remove all the files that appear in **mtn ls ignored** and **mtn ls unknows**. Then pull and update similar to `clean`

copy This first checkout source into source directory then copy the `source` directory to `build` directory then performs the build operation in the copied directory. This way we make fresh builds with very less bandwidth to download source. The behavior of source checkout follows exactly same as incremental. It performs all the incremental checkout behavior in `source` directory.

ShellCommand

Most interesting steps involve executing a process of some sort on the worker. The *ShellCommand* class handles this activity.

Several subclasses of *ShellCommand* are provided as starting points for common build steps.

Using ShellCommands

class `buildbot.steps.shell.`**ShellCommand**

This is a useful base class for just about everything you might want to do during a build (except for the initial source checkout). It runs a single command in a child shell on the worker. All stdout/stderr is recorded into a `LogFile`. The step usually finishes with a status of `FAILURE` if the command's exit code is non-zero, otherwise it has a status of `SUCCESS`.

The preferred way to specify the command is with a list of argv strings, since this allows for spaces in filenames and avoids doing any fragile shell-escaping. You can also specify the command with a single string, in which case the string is given to `/bin/sh -c` *COMMAND* for parsing.

On Windows, commands are run via `cmd.exe /c` which works well. However, if you're running a batch file, the error level does not get propagated correctly unless you add 'call' before your batch file's name: `cmd=['call', 'myfile.bat', ...]`.

The *ShellCommand* arguments are:

command a list of strings (preferred) or single string (discouraged) which specifies the command to be run. A list of strings is preferred because it can be used directly as an argv array. Using a single string (with embedded spaces) requires the worker to pass the string to **/bin/sh** for interpretation, which raises all sorts of difficult questions about how to escape or interpret shell metacharacters.

If `command` contains nested lists (for example, from a properties substitution), then that list will be flattened before it is executed.

workdir All ShellCommands are run by default in the `workdir`, which defaults to the `build` subdirectory of the worker builder's base directory. The absolute path of the workdir will thus be the worker's basedir (set as an option to `buildbot-worker create-worker`, *Creating a worker*) plus the builder's basedir (set in the builder's `builddir` key in `master.cfg`) plus the workdir itself (a class-level attribute of the BuildFactory, defaults to `build`).

For example:

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(command=["make", "test"],
                             workdir="build/tests"))
```

env a dictionary of environment strings which will be added to the child command's environment. For example, to run tests with a different i18n language setting, you might use:

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(command=["make", "test"],
                             env={'LANG': 'fr_FR'}))
```

These variable settings will override any existing ones in the worker's environment or the environment specified in the `Builder`. The exception is `PYTHONPATH`, which is merged with (actually prepended to) any existing `PYTHONPATH` setting. The following example will prepend `/home/buildbot/lib/python` to any existing `PYTHONPATH`:

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(
              command=["make", "test"],
              env={'PYTHONPATH': "/home/buildbot/lib/python"}))
```

To avoid the need of concatenating path together in the master config file, if the value is a list, it will be joined together using the right platform dependent separator.

Those variables support expansion so that if you just want to prepend `/home/buildbot/bin` to the `PATH` environment variable, you can do it by putting the value `${PATH}` at the end of the value like in the example below. Variables that don't exist on the worker will be replaced by `""`.

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(
              command=["make", "test"],
              env={'PATH': ["/home/buildbot/bin",
                            "${PATH}"]}))
```

Note that environment values must be strings (or lists that are turned into strings). In particular, numeric properties such as `buildnumber` must be substituted using *Interpolate*.

want_stdout if `False`, stdout from the child process is discarded rather than being sent to the buildmaster for inclusion in the step's `LogFile`.

want_stderr like `want_stdout` but for `stderr`. Note that commands run through a PTY do not have separate `stdout`/`stderr` streams: both are merged into `stdout`.

usePTY Should this command be run in a `pty`? `False` by default. This option is not available on Windows.

In general, you do not want to use a pseudo-terminal. This is is *only* useful for running commands that require a terminal - for example, testing a command-line application that will only accept passwords read from a terminal.

Using a pseudo-terminal brings lots of compatibility problems, and prevents Buildbot from distinguishing the standard error (red) and standard output (black) streams.

In previous versions, the advantage of using a pseudo-terminal was that `grandchild` processes were more likely to be cleaned up if the build was interrupted or times out. This occurred because using a pseudo-terminal incidentally puts the command into its own process group.

As of Buildbot-0.8.4, all commands are placed in process groups, and thus grandchild processes will be cleaned up properly.

logfiles Sometimes commands will log interesting data to a local file, rather than emitting everything to stdout or stderr. For example, Twisted's **trial** command (which runs unit tests) only presents summary information to stdout, and puts the rest into a file named `_trial_temp/test.log`. It is often useful to watch these files as the command runs, rather than using **/bin/cat** to dump their contents afterwards.

The `logfiles=` argument allows you to collect data from these secondary logfiles in near-real-time, as the step is running. It accepts a dictionary which maps from a local Log name (which is how the log data is presented in the build results) to either a remote filename (interpreted relative to the build's working directory), or a dictionary of options. Each named file will be polled on a regular basis (every couple of seconds) as the build runs, and any new text will be sent over to the buildmaster.

If you provide a dictionary of options instead of a string, you must specify the `filename` key. You can optionally provide a `follow` key which is a boolean controlling whether a logfile is followed or concatenated in its entirety. Following is appropriate for logfiles to which the build step will append, where the pre-existing contents are not interesting. The default value for `follow` is `False`, which gives the same behavior as just providing a string filename.

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(
                    command=["make", "test"],
                    logfiles={"triallog": "_trial_temp/test.log"}))
```

The above example will add a log named 'triallog' on the master, based on `_trial_temp/test.log` on the worker.

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(command=["make", "test"],
                             logfiles={
                                 "triallog": {
                                    "filename": "_trial_temp/test.log",
                                    "follow": True
                                 }
                             }))
```

lazylogfiles If set to `True`, logfiles will be tracked lazily, meaning that they will only be added when and if something is written to them. This can be used to suppress the display of empty or missing log files. The default is `False`.

timeout if the command fails to produce any output for this many seconds, it is assumed to be locked up and will be killed. This defaults to 1200 seconds. Pass `None` to disable.

maxTime if the command takes longer than this many seconds, it will be killed. This is disabled by default.

logEnviron If this option is `True` (the default), then the step's logfile will describe the environment variables on the worker. In situations where the environment is not relevant and is long, it may be easier to set `logEnviron=False`.

interruptSignal If the command should be interrupted (either by buildmaster or timeout etc.), what signal should be sent to the process, specified by name. By default this is "KILL" (9). Specify "TERM" (15) to give the process a chance to cleanup. This functionality requires a 0.8.6 worker or newer.

`sigtermTime`

If set, when interrupting, try to kill the command with SIGTERM and wait for sigtermTime seconds before firing `interuptSignal`. If None, `interruptSignal` will be fired immediately on interrupt.

initialStdin If the command expects input on stdin, that can be supplied a a string with this parameter. This value should not be excessively large, as it is handled as a single string throughout Buildbot – for example, do not pass the contents of a tarball with this parameter.

decodeRC This is a dictionary that decodes exit codes into results value. For example, `{0:SUCCESS, 1:FAILURE,2:WARNINGS}`, will treat the exit code `2` as WARNINGS. The default is to treat just 0 as successful. (`{0:SUCCESS}`) any exit code not present in the dictionary will be treated as `FAILURE`

Shell Sequence

Some steps have a specific purpose, but require multiple shell commands to implement them. For example, a build is often `configure; make; make install`. We have two ways to handle that:

- Create one shell command with all these. To put the logs of each commands in separate logfiles, we need to re-write the script as `configure 1> configure_log; ...` and to add these `configure_log` files as `logfiles` argument of the buildstep. This has the drawback of complicating the shell script, and making it harder to maintain as the logfile name is put in different places.
- Create three *ShellCommand* instances, but this loads the build UI unnecessarily.

ShellSequence is a class to execute not one but a sequence of shell commands during a build. It takes as argument a renderable, or list of commands which are *ShellArg* objects. Each such object represents a shell invocation.

The single *ShellSequence* argument aside from the common parameters is:

`commands`

A list of *ShellArg* objects or a renderable the returns a list of *ShellArg* objects.

```
from buildbot.plugins import steps, util

f.addStep(steps.ShellSequence(
    commands=[
        util.ShellArg(command=['configure']),
        util.ShellArg(command=['make'], logfile='make'),
        util.ShellArg(command=['make', 'check_warning'], logfile='warning',␣
↪warnOnFailure=True),
        util.ShellArg(command=['make', 'install'], logfile='make install')
    ]))
```

All these commands share the same configuration of `environment`, `workdir` and `pty` usage that can be setup the same way as in *ShellCommand*.

class `buildbot.steps.shellsequence.`**ShellArg**(*self*, *command=None*, *logfile=None*, *haltOnFailure=False*, *flunkOnWarnings=False*, *flunkOnFailure=False*, *warnOnWarnings=False*, *warnOnFailure=False*)

Parameters

- **command** – (see the *ShellCommand* `command` argument),

- **logfile** – optional log file name, used as the stdio log of the command

The `haltOnFailure`, `flunkOnWarnings`, `flunkOnFailure`, `warnOnWarnings`, `warnOnFailure` parameters drive the execution of the sequence, the same way steps are scheduled in the build. They have the same default values as for buildsteps - see *Common Parameters*.

Any of the arguments to this class can be renderable.

Note that if `logfile` name does not start with the prefix `stdio`, that prefix will be set like `stdio <logfile>`.

The two *`ShellSequence`* methods below tune the behavior of how the list of shell commands are executed, and can be overridden in subclasses.

class `buildbot.steps.shellsequence.`**`ShellSequence`**

`shouldRunTheCommand`(*oneCmd*)

Parameters **`oneCmd`** – a string or a list of strings, as rendered from a *`ShellArg`* instance's `command` argument.

Determine whether the command `oneCmd` should be executed. If `shouldRunTheCommand` returns `False`, the result of the command will be recorded as SKIPPED. The default methods skips all empty strings and empty lists.

`getFinalState`()

Return the status text of the step in the end. The default value is to set the text describing the execution of the last shell command.

`runShellSequence(commands)`:

Parameters **`commands`** – list of shell args

This method actually runs the shell sequence. The default `run` method calls `runShellSequence`, but subclasses can override `run` to perform other operations, if desired.

Configure

class `buildbot.steps.shell.`**`Configure`**

This is intended to handle the **`./configure`** step from autoconf-style projects, or the `perl Makefile.PL` step from perl `MakeMaker.pm`-style modules. The default command is **`./configure`** but you can change this by providing a `command=` parameter. The arguments are identical to *`ShellCommand`*.

```
from buildbot.plugins import steps

f.addStep(steps.Configure())
```

CMake

class `buildbot.steps.cmake.`**`CMake`**

This is intended to handle the **`cmake`** step for projects that use CMake-based build systems (http://cmake.org).

Note: Links below point to the latest CMake documentation. Make sure that you check the documentation for the CMake you use.

In addition to the parameters *ShellCommand* supports, this step accepts the following parameters:

path Either a path to a source directory to (re-)generate a build system for it in the current working directory. Or an existing build directory to re-generate its build system.

generator A build system generator. See cmake-generators(7) (https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html) for available options.

definitions A dictionary that contains parameters that will be converted to `-D{name}={value}` when passed to CMake. Refer to cmake(1) (https://cmake.org/cmake/help/latest/manual/cmake.1.html) for more information.

options A list or a tuple that contains options that will be passed to CMake as is. Refer to cmake(1) (https://cmake.org/cmake/help/latest/manual/cmake.1.html) for more information.

cmake Path to the CMake binary. Default is **cmake**

```
from buildbot.plugins import steps

...

factory.addStep(
    steps.CMake(
        generator='Ninja',
        definitions={
            'CMAKE_BUILD_TYPE': Property('BUILD_TYPE')
        },
        options=[
            '-Wno-dev'
        ]
    )
)

...
```

Compile

This is meant to handle compiling or building a project written in C. The default command is `make all`. When the compilation is finished, the log file is scanned for GCC warning messages, a summary log is created with any problems that were seen, and the step is marked as WARNINGS if any were discovered. Through the `WarningCountingShellCommand` superclass, the number of warnings is stored in a Build Property named *warnings-count*, which is accumulated over all *Compile* steps (so if two warnings are found in one step, and three are found in another step, the overall build will have a *warnings-count* property of 5). Each step can be optionally given a maximum number of warnings via the maxWarnCount parameter. If this limit is exceeded, the step will be marked as a failure.

The default regular expression used to detect a warning is `'.*warning[: ].*'` , which is fairly liberal and may cause false-positives. To use a different regexp, provide a `warningPattern=` argument, or use a subclass which sets the `warningPattern` attribute:

```
from buildbot.plugins import steps

f.addStep(steps.Compile(command=["make", "test"],
                        warningPattern="^Warning: "))
```

The `warningPattern=` can also be a pre-compiled Python regexp object: this makes it possible to add flags like `re.I` (to use case-insensitive matching).

Note that the compiled `warningPattern` will have its `match` method called, which is subtly different from a `search`. Your regular expression must match the from the beginning of the line. This means that to look for the word "warning" in the middle of a line, you will need to prepend `'.*'` to your regular expression.

The `suppressionFile=` argument can be specified as the (relative) path of a file inside the workdir defining warnings to be suppressed from the warning counting and log file. The file will be uploaded to the master from the worker before compiling, and any warning matched by a line in the suppression file will be ignored. This is useful to accept certain warnings (e.g. in some special module of the source tree or in cases where the compiler is being particularly stupid), yet still be able to easily detect and fix the introduction of new warnings.

The file must contain one line per pattern of warnings to ignore. Empty lines and lines beginning with `#` are ignored. Other lines must consist of a regexp matching the file name, followed by a colon (`:`), followed by a regexp matching the text of the warning. Optionally this may be followed by another colon and a line number range. For example:

```
# Sample warning suppression file

mi_packrec.c : .*result of 32-bit shift implicitly converted to 64 bits.* : 560-600
DictTabInfo.cpp : .*invalid access to non-static.*
kernel_types.h : .*only defines private constructors and has no friends.* : 51
```

If no line number range is specified, the pattern matches the whole file; if only one number is given it matches only on that line.

The default warningPattern regexp only matches the warning text, so line numbers and file names are ignored. To enable line number and file name matching, provide a different regexp and provide a function (callable) as the argument of `warningExtractor=`. The function is called with three arguments: the `BuildStep` object, the line in the log file with the warning, and the `SRE_Match` object of the regexp search for `warningPattern`. It should return a tuple `(filename, linenumber, warning_test)`. For example:

```
f.addStep(Compile(command=["make"],
                  warningPattern="^(.\*?):([0-9]+): [Ww]arning: (.\*)$",
                  warningExtractor=Compile.warnExtractFromRegexpGroups,
                  suppressionFile="support-files/compiler_warnings.supp"))
```

(`Compile.warnExtractFromRegexpGroups` is a pre-defined function that returns the filename, linenumber, and text from groups (1,2,3) of the regexp match).

In projects with source files in multiple directories, it is possible to get full path names for file names matched in the suppression file, as long as the build command outputs the names of directories as they are entered into and left again. For this, specify regexps for the arguments `directoryEnterPattern=` and `directoryLeavePattern=`. The `directoryEnterPattern=` regexp should return the name of the directory entered into in the first matched group. The defaults, which are suitable for GNU Make, are these:

```
directoryEnterPattern="make.*: Entering directory [\"`'](.*)[`'\"]"
directoryLeavePattern="make.*: Leaving directory"
```

(TODO: this step needs to be extended to look for GCC error messages as well, and collect them into a separate logfile, along with the source code filenames involved).

Visual C++

These steps are meant to handle compilation using Microsoft compilers. VC++ 6-14 (aka Visual Studio 2003-2015 and VCExpress9) are supported via calling `devenv`. Msbuild as well as Windows Driver Kit 8 are supported via the `MsBuild4`, `MsBuild12`, and `MsBuild14` steps. These steps will take care of setting up a clean compilation environment, parsing the generated output in real time, and delivering as detailed as possible information about the compilation executed.

All of the classes are in `buildbot.steps.vstudio`. The available classes are:

- `VC6`
- `VC7`
- `VC8`
- `VC9`
- `VC10`
- `VC11`
- `VC12`
- `VC14`
- `VS2003`
- `VS2005`
- `VS2008`
- `VS2010`
- `VS2012`
- `VS2013`
- `VS2015`
- `VCExpress9`
- `MsBuild4`
- `MsBuild12`
- `MsBuild14`

The available constructor arguments are

`mode` The mode default to `rebuild`, which means that first all the remaining object files will be cleaned by the compiler. The alternate values are `build`, where only the updated files will be recompiled, and `clean`, where the current build files are removed and no compilation occurs.

`projectfile` This is a mandatory argument which specifies the project file to be used during the compilation.

`config` This argument defaults to `release` an gives to the compiler the configuration to use.

`installdir` This is the place where the compiler is installed. The default value is compiler specific and is the default place where the compiler is installed.

`useenv` This boolean parameter, defaulting to `False` instruct the compiler to use its own settings or the one defined through the environment variables `PATH`, `INCLUDE`, and `LIB`. If any of the `INCLUDE` or `LIB` parameter is defined, this parameter automatically switches to `True`.

`PATH` This is a list of path to be added to the `PATH` environment variable. The default value is the one defined in the compiler options.

`INCLUDE` This is a list of path where the compiler will first look for include files. Then comes the default paths defined in the compiler options.

`LIB` This is a list of path where the compiler will first look for libraries. Then comes the default path defined in the compiler options.

`arch` That one is only available with the class VS2005 (VC8). It gives the target architecture of the built artifact. It defaults to `x86` and does not apply to `MsBuild4` or `MsBuild12`. Please see `platform` below.

project This gives the specific project to build from within a workspace. It defaults to building all projects. This is useful for building cmake generate projects.

platform This is a mandatory argument for `MsBuild4` and `MsBuild12` specifying the target platform such as 'Win32', 'x64' or 'Vista Debug'. The last one is an example of driver targets that appear once Windows Driver Kit 8 is installed.

Here is an example on how to drive compilation with Visual Studio 2013:

```
from buildbot.plugins import steps

f.addStep(
    steps.VS2013(projectfile="project.sln", config="release",
        arch="x64", mode="build",
           INCLUDE=[r'C:\3rd-party\libmagic\include'],
           LIB=[r'C:\3rd-party\libmagic\lib-x64']))
```

Here is a similar example using "MsBuild12":

```
from buildbot.plugins import steps

# Build one project in Release mode for Win32
f.addStep(
    steps.MsBuild12(projectfile="trunk.sln", config="Release", platform="Win32",
            workdir="trunk",
            project="tools\\protoc"))

# Build the entire solution in Debug mode for x64
f.addStep(
    steps.MsBuild12(projectfile="trunk.sln", config='Debug', platform='x64',
            workdir="trunk"))
```

Cppcheck

This step runs `cppcheck`, analyse its output, and set the outcome in *Properties*.

```
from buildbot.plugins import steps

f.addStep(steps.Cppcheck(enable=['all'], inconclusive=True))
```

This class adds the following arguments:

binary (Optional, default to `cppcheck`) Use this if you need to give the full path to the cppcheck binary or if your binary is called differently.

source (Optional, default to `['.']`) This is the list of paths for the sources to be checked by this step.

enable (Optional) Use this to give a list of the message classes that should be in cppcheck report. See the cppcheck man page for more information.

inconclusive (Optional) Set this to `True` if you want cppcheck to also report inconclusive results. See the cppcheck man page for more information.

extra_args (Optional) This is the list of extra arguments to be given to the cppcheck command.

All other arguments are identical to *ShellCommand*.

Robocopy

class `buildbot.steps.mswin.`**Robocopy**

This step runs `robocopy` on Windows.

Robocopy (https://technet.microsoft.com/en-us/library/cc733145.aspx) is available in versions of Windows starting with Windows Vista and Windows Server 2008. For previous versions of Windows, it's available as part of the Windows Server 2003 Resource Kit Tools (https://www.microsoft.com/en-us/download/details.aspx?id=17657).

```
from buildbot.plugins import steps, util

f.addStep(
    steps.Robocopy(
        name='deploy_binaries',
        description='Deploying binaries...',
        descriptionDone='Deployed binaries.',
        source=util.Interpolate('Build\\Bin\\%(prop:configuration)s'),
        destination=util.Interpolate('%(prop:deploy_dir)\\Bin\\%(prop:configuration)s
↪'),
        mirror=True
    )
)
```

Available constructor arguments are:

source The path to the source directory (mandatory).

destination The path to the destination directory (mandatory).

files An array of file names or patterns to copy.

recursive Copy files and directories recursively (`/E` parameter).

mirror Mirror the source directory in the destination directory, including removing files that don't exist anymore (`/MIR` parameter).

move Delete the source directory after the copy is complete (`/MOVE` parameter).

exclude_files An array of file names or patterns to exclude from the copy (`/XF` parameter).

exclude_dirs An array of directory names or patterns to exclude from the copy (`/XD` parameter).

custom_opts An array of custom parameters to pass directly to the `robocopy` command.

verbose Whether to output verbose information (`/V /TS /TP` parameters).

Note that parameters `/TEE /NP` will always be appended to the command to signify, respectively, to output logging to the console, use Unicode logging, and not print any percentage progress information for each file.

Test

```
from buildbot.plugins import steps

f.addStep(steps.Test())
```

This is meant to handle unit tests. The default command is **`make test`**, and the `warnOnFailure` flag is set. The other arguments are identical to *`ShellCommand`*.

TreeSize

```
from buildbot.plugins import steps

f.addStep(steps.TreeSize())
```

This is a simple command that uses the **du** tool to measure the size of the code tree. It puts the size (as a count of 1024-byte blocks, aka 'KiB' or 'kibibytes') on the step's status text, and sets a build property named `tree-size-KiB` with the same value. All arguments are identical to *ShellCommand*.

PerlModuleTest

```
from buildbot.plugins import steps

f.addStep(steps.PerlModuleTest())
```

This is a simple command that knows how to run tests of perl modules. It parses the output to determine the number of tests passed and failed and total number executed, saving the results for later query. The command is `prove --lib lib -r t`, although this can be overridden with the `command` argument. All other arguments are identical to those for *ShellCommand*.

MTR (mysql-test-run)

The *MTR* class is a subclass of *Test*. It is used to run test suites using the mysql-test-run program, as used in MySQL, Drizzle, MariaDB, and MySQL storage engine plugins.

The shell command to run the test suite is specified in the same way as for the *Test* class. The *MTR* class will parse the output of running the test suite, and use the count of tests executed so far to provide more accurate completion time estimates. Any test failures that occur during the test are summarized on the Waterfall Display.

Server error logs are added as additional log files, useful to debug test failures.

Optionally, data about the test run and any test failures can be inserted into a database for further analysis and report generation. To use this facility, create an instance of `twisted.enterprise.adbapi.ConnectionPool` with connections to the database. The necessary tables can be created automatically by setting `autoCreateTables` to `True`, or manually using the SQL found in the mtrlogobserver.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/steps/mtrlogobserver.py) source file.

One problem with specifying a database is that each reload of the configuration will get a new instance of `ConnectionPool` (even if the connection parameters are the same). To avoid that Buildbot thinks the builder configuration has changed because of this, use the `steps.mtrlogobserver.EqConnectionPool` subclass of `ConnectionPool`, which implements an equality operation that avoids this problem.

Example use:

```
from buildbot.plugins import steps, util

myPool = util.EqConnectionPool("MySQLdb", "host", "buildbot", "password", "db")
myFactory.addStep(steps.MTR(workdir="mysql-test", dbpool=myPool,
                            command=["perl", "mysql-test-run.pl", "--force"]))
```

The *MTR* step's arguments are:

textLimit Maximum number of test failures to show on the waterfall page (to not flood the page in case of a large number of test failures. Defaults to 5.

testNameLimit Maximum length of test names to show unabbreviated in the waterfall page, to avoid excessive column width. Defaults to 16.

parallel Value of option *–parallel* option used for `mysql-test-run.pl` (number of processes used to run the test suite in parallel). Defaults to 4. This is used to determine the number of server error log files to download from the worker. Specifying a too high value does not hurt (as nonexisting error logs will be ignored), however if using option *–parallel* value greater than the default it needs to be specified, or some server error logs will be missing.

dbpool An instance of `twisted.enterprise.adbapi.ConnectionPool`, or `None`. Defaults to `None`. If specified, results are inserted into the database using the `ConnectionPool`.

autoCreateTables Boolean, defaults to `False`. If `True` (and `dbpool` is specified), the necessary database tables will be created automatically if they do not exist already. Alternatively, the tables can be created manually from the SQL statements found in the mtrlogobserver.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/steps/mtrlogobserver.py) source file.

test_type Short string that will be inserted into the database in the row for the test run. Defaults to the empty string, but can be specified to identify different types of test runs.

test_info Descriptive string that will be inserted into the database in the row for the test run. Defaults to the empty string, but can be specified as a user-readable description of this particular test run.

mtr_subdir The subdirectory in which to look for server error log files. Defaults to `mysql-test`, which is usually correct. *Interpolate* is supported.

SubunitShellCommand

class `buildbot.steps.subunit.`**SubunitShellCommand**

This buildstep is similar to *ShellCommand*, except that it runs the log content through a subunit filter to extract test and failure counts.

```
from buildbot.plugins import steps

f.addStep(steps.SubunitShellCommand(command="make test"))
```

This runs `make test` and filters it through subunit. The 'tests' and 'test failed' progress metrics will now accumulate test data from the test run.

If `failureOnNoTests` is `True`, this step will fail if no test is run. By default `failureOnNoTests` is False.

Worker Filesystem Steps

Here are some buildsteps for manipulating the worker's filesystem.

FileExists

This step will assert that a given file exists, failing if it does not. The filename can be specified with a property.

```
from buildbot.plugins import steps

f.addStep(steps.FileExists(file='test_data'))
```

This step requires worker version 0.8.4 or later.

CopyDirectory

This command copies a directory on the worker.

```
from buildbot.plugins import steps

f.addStep(steps.CopyDirectory(src="build/data", dest="tmp/data"))
```

This step requires worker version 0.8.5 or later.

The CopyDirectory step takes the following arguments:

timeout if the copy command fails to produce any output for this many seconds, it is assumed to be locked up and will be killed. This defaults to 120 seconds. Pass `None` to disable.

maxTime if the command takes longer than this many seconds, it will be killed. This is disabled by default.

RemoveDirectory

This command recursively deletes a directory on the worker.

```
from buildbot.plugins import steps

f.addStep(steps.RemoveDirectory(dir="build/build"))
```

This step requires worker version 0.8.4 or later.

MakeDirectory

This command creates a directory on the worker.

```
from buildbot.plugins import steps

f.addStep(steps.MakeDirectory(dir="build/build"))
```

This step requires worker version 0.8.5 or later.

Python BuildSteps

Here are some `BuildSteps` that are specifically useful for projects implemented in Python.

BuildEPYDoc

class `buildbot.steps.python.`**BuildEPYDoc**

epydoc (http://epydoc.sourceforge.net/) is a tool for generating API documentation for Python modules from their docstrings. It reads all the `.py` files from your source tree, processes the docstrings therein, and creates a large tree of `.html` files (or a single `.pdf` file).

The *`BuildEPYDoc`* step will run **epydoc** to produce this API documentation, and will count the errors and warnings from its output.

You must supply the command line to be used. The default is `make epydocs`, which assumes that your project has a `Makefile` with an *epydocs* target. You might wish to use something like `epydoc -o apiref source/PKGNAME` instead. You might also want to add option *–pdf* to generate a PDF file instead of a large tree of HTML files.

The API docs are generated in-place in the build tree (under the workdir, in the subdirectory controlled by the option *-o* argument). To make them useful, you will probably have to copy them to somewhere they can be read. For example if you have server with configured nginx web server, you can place generated docs to it's public folder with command like `rsync -ad apiref/ dev.example.com:~/usr/share/nginx/www/current-apiref/`. You might instead want to bundle them into a tarball and publish it in the same place where the generated install tarball is placed.

```
from buildbot.plugins import steps

f.addStep(steps.BuildEPYDoc(command=["epydoc", "-o", "apiref", "source/mypkg"]))
```

PyFlakes

class `buildbot.steps.python.`**`PyFlakes`**

PyFlakes (https://launchpad.net/pyflakes) is a tool to perform basic static analysis of Python code to look for simple errors, like missing imports and references of undefined names. It is like a fast and simple form of the C **`lint`** program. Other tools (like pychecker (http://pychecker.sourceforge.net/)) provide more detailed results but take longer to run.

The *`PyFlakes`* step will run pyflakes and count the various kinds of errors and warnings it detects.

You must supply the command line to be used. The default is `make pyflakes`, which assumes you have a top-level `Makefile` with a `pyflakes` target. You might want to use something like `pyflakes .` or `pyflakes src`.

```
from buildbot.plugins import steps

f.addStep(steps.PyFlakes(command=["pyflakes", "src"]))
```

Sphinx

class `buildbot.steps.python.`**`Sphinx`**

Sphinx (http://sphinx.pocoo.org/) is the Python Documentation Generator. It uses RestructuredText (http://docutils.sourceforge.net/rst.html) as input format.

The *`Sphinx`* step will run **`sphinx-build`** or any other program specified in its `sphinx` argument and count the various warnings and error it detects.

```
from buildbot.plugins import steps

f.addStep(steps.Sphinx(sphinx_builddir="_build"))
```

This step takes the following arguments:

`sphinx_builddir` (required) Name of the directory where the documentation will be generated.

`sphinx_sourcedir` (optional, defaulting to `.`), Name the directory where the `conf.py` file will be found

`sphinx_builder` (optional) Indicates the builder to use.

`sphinx` (optional, defaulting to **`sphinx-build`**) Indicates the executable to run.

tags (optional) List of `tags` to pass to **sphinx-build**

defines (optional) Dictionary of defines to overwrite values of the `conf.py` file.

mode (optional) String, one of `full` or `incremental` (the default). If set to `full`, indicates to Sphinx to rebuild everything without re-using the previous build results.

PyLint

Similarly, the *PyLint* step will run **pylint** and analyze the results.

You must supply the command line to be used. There is no default.

```
from buildbot.plugins import steps

f.addStep(steps.PyLint(command=["pylint", "src"]))
```

Trial

class `buildbot.steps.python_twisted.`**Trial**

This step runs a unit test suite using **trial**, a unittest-like testing framework that is a component of Twisted Python. Trial is used to implement Twisted's own unit tests, and is the unittest-framework of choice for many projects that use Twisted internally.

Projects that use trial typically have all their test cases in a 'test' subdirectory of their top-level library directory. For example, for a package `petmail`, the tests might be in `petmail/test/test_*.py`. More complicated packages (like Twisted itself) may have multiple test directories, like `twisted/test/test_*.py` for the core functionality and `twisted/mail/test/test_*.py` for the email-specific tests.

To run trial tests manually, you run the **trial** executable and tell it where the test cases are located. The most common way of doing this is with a module name. For petmail, this might look like **trial petmail.test**, which would locate all the `test_*.py` files under `petmail/test/`, running every test case it could find in them. Unlike the `unittest.py` that comes with Python, it is not necessary to run the `test_foo.py` as a script; you always let trial do the importing and running. The step's `tests` parameter controls which tests trial will run: it can be a string or a list of strings.

To find the test cases, the Python search path must allow something like `import petmail.test` to work. For packages that don't use a separate top-level `lib` directory, `PYTHONPATH=.` will work, and will use the test cases (and the code they are testing) in-place. `PYTHONPATH=build/lib` or `PYTHONPATH=build/lib.somearch` are also useful when you do a `python setup.py build` step first. The `testpath` attribute of this class controls what `PYTHONPATH` is set to before running **trial**.

Trial has the ability, through the `--testmodule` flag, to run only the set of test cases named by special `test-case-name` tags in source files. We can get the list of changed source files from our parent Build and provide them to trial, thus running the minimal set of test cases needed to cover the Changes. This is useful for quick builds, especially in trees with a lot of test cases. The `testChanges` parameter controls this feature: if set, it will override `tests`.

The trial executable itself is typically just **trial**, and is typically found in the shell search path. It can be overridden with the `trial` parameter. This is useful for Twisted's own unittests, which want to use the copy of bin/trial that comes with the sources.

To influence the version of Python being used for the tests, or to add flags to the command, set the `python` parameter. This can be a string (like `python2.2`) or a list (like `['python2.3', '-Wall']`).

Trial creates and switches into a directory named `_trial_temp/` before running the tests, and sends the twisted log (which includes all exceptions) to a file named `test.log`. This file will be pulled up to the master where it can be seen as part of the status output.

```
from buildbot.plugins import steps

f.addStep(steps.Trial(tests='petmail.test'))
```

Trial has the ability to run tests on several workers in parallel (beginning with Twisted 12.3.0). Set `jobs` to the number of workers you want to run. Note that running **`trial`** in this way will create multiple log files (named `test.N.log`, `err.N.log` and `out.N.log` starting with `N=0`) rather than a single `test.log`.

This step takes the following arguments:

`jobs` (optional) Number of worker-resident trial workers to use when running the tests. Defaults to 1 worker. Only works with Twisted>=12.3.0.

RemovePYCs

class `buildbot.steps.python_twisted.`**`RemovePYCs`**

This is a simple built-in step that will remove `.pyc` files from the workdir. This is useful in builds that update their source (and thus do not automatically delete `.pyc` files) but where some part of the build process is dynamically searching for Python modules. Notably, trial has a bad habit of finding old test modules.

```
from buildbot.plugins import steps

f.addStep(steps.RemovePYCs())
```

Transferring Files

class `buildbot.steps.transfer.`**`FileUpload`**

class `buildbot.steps.transfer.`**`FileDownload`**

Most of the work involved in a build will take place on the worker. But occasionally it is useful to do some work on the buildmaster side. The most basic way to involve the buildmaster is simply to move a file from the worker to the master, or vice versa. There are a pair of steps named *`FileUpload`* and *`FileDownload`* to provide this functionality. *`FileUpload`* moves a file *up to* the master, while *`FileDownload`* moves a file *down from* the master.

As an example, let's assume that there is a step which produces an HTML file within the source tree that contains some sort of generated project documentation. And let's assume that we run nginx web server on buildmaster host for serving static files. We want to move this file to the buildmaster, into a `/usr/share/nginx/www/` directory, so it can be visible to developers. This file will wind up in the worker-side working directory under the name `docs/reference.html`. We want to put it into the master-side `/usr/share/nginx/www/ref.html`, and add a link to the HTML status to the uploaded file.

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(command=["make", "docs"]))
f.addStep(steps.FileUpload(workersrc="docs/reference.html",
                           masterdest="/usr/share/nginx/www/ref.html",
                           url="http://somesite/~buildbot/ref.html"))
```

The masterdest= argument will be passed to os.path.expanduser, so things like ~ will be expanded properly. Non-absolute paths will be interpreted relative to the buildmaster's base directory. Likewise, the workersrc= argument will be expanded and interpreted relative to the builder's working directory.

Note: The copied file will have the same permissions on the master as on the worker, look at the mode= parameter to set it differently.

To move a file from the master to the worker, use the *FileDownload* command. For example, let's assume that some step requires a configuration file that, for whatever reason, could not be recorded in the source code repository or generated on the worker side:

```
from buildbot.plugins import steps

f.addStep(steps.FileDownload(mastersrc="~/todays_build_config.txt",
                             workerdest="build_config.txt"))
f.addStep(steps.ShellCommand(command=["make", "config"]))
```

Like *FileUpload*, the mastersrc= argument is interpreted relative to the buildmaster's base directory, and the workerdest= argument is relative to the builder's working directory. If the worker is running in ~worker, and the builder's builddir is something like tests-i386, then the workdir is going to be ~worker/tests-i386/build, and a workerdest= of foo/bar.html will get put in ~worker/tests-i386/build/foo/bar.html. Both of these commands will create any missing intervening directories.

Other Parameters

The maxsize= argument lets you set a maximum size for the file to be transferred. This may help to avoid surprises: transferring a 100MB coredump when you were expecting to move a 10kB status file might take an awfully long time. The blocksize= argument controls how the file is sent over the network: larger blocksizes are slightly more efficient but also consume more memory on each end, and there is a hard-coded limit of about 640kB.

The mode= argument allows you to control the access permissions of the target file, traditionally expressed as an octal integer. The most common value is probably 0755, which sets the *x* executable bit on the file (useful for shell scripts and the like). The default value for mode= is None, which means the permission bits will default to whatever the umask of the writing process is. The default umask tends to be fairly restrictive, but at least on the worker you can make it less restrictive with a –umask command-line option at creation time (*Worker Options*).

The keepstamp= argument is a boolean that, when True, forces the modified and accessed time of the destination file to match the times of the source file. When False (the default), the modified and accessed times of the destination file are set to the current time on the buildmaster.

The url= argument allows you to specify an url that will be displayed in the HTML status. The title of the url will be the name of the item transferred (directory for DirectoryUpload or file for FileUpload). This allows the user to add a link to the uploaded item if that one is uploaded to an accessible place.

For *FileUpload*, the urlText= argument allows you to specify the url title that will be displayed in the web UI.

Transfering Directories

class buildbot.steps.transfer.**DirectoryUpload**

To transfer complete directories from the worker to the master, there is a BuildStep named *DirectoryUpload*. It works like *FileUpload*, just for directories. However it does not support the maxsize, blocksize and mode arguments. As an example, let's assume an generated project documentation, which consists of many files (like the output of **doxygen** or **epydoc**). And let's assume that we run nginx web server on buildmaster host for serving

static files. We want to move the entire documentation to the buildmaster, into a `/usr/share/nginx/www/docs` directory, and add a link to the uploaded documentation on the HTML status page. On the worker-side the directory can be found under `docs`:

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(command=["make", "docs"]))
f.addStep(steps.DirectoryUpload(workersrc="docs",
                                masterdest="/usr/share/nginx/www/docs",
                                url="~buildbot/docs"))
```

The *DirectoryUpload* step will create all necessary directories and transfers empty directories, too.

The `maxsize` and `blocksize` parameters are the same as for *FileUpload*, although note that the size of the transferred data is implementation-dependent, and probably much larger than you expect due to the encoding used (currently tar).

The optional `compress` argument can be given as `'gz'` or `'bz2'` to compress the datastream.

Note: The permissions on the copied files will be the same on the master as originally on the worker, see option `buildbot-worker create-worker --umask` to change the default one.

Transferring Multiple Files At Once

class `buildbot.steps.transfer.`**MultipleFileUpload**

In addition to the *FileUpload* and *DirectoryUpload* steps there is the *MultipleFileUpload* step for uploading a bunch of files (and directories) in a single `BuildStep`. The step supports all arguments that are supported by *FileUpload* and *DirectoryUpload*, but instead of a the single `workersrc` parameter it takes a (plural) `workersrcs` parameter. This parameter should either be a list, or something that can be rendered as a list. Additionally it supports the `glob` parameter if this parameter is set to `True` all arguments in `workersrcs` will be parsed through `glob` and the results will be uploaded to `masterdest`.:

```
from buildbot.plugins import steps

f.addStep(steps.ShellCommand(command=["make", "test"]))
f.addStep(steps.ShellCommand(command=["make", "docs"]))
f.addStep(steps.MultipleFileUpload(workersrcs=["docs", "test-results.html"],
                                   masterdest="/usr/share/nginx/www/",
                                   url="~buildbot"))
```

The `url=` parameter, can be used to specify a link to be displayed in the HTML status of the step.

The way URLs are added to the step can be customized by extending the *MultipleFileUpload* class. The *allUploadsDone* method is called after all files have been uploaded and sets the URL. The *uploadDone* method is called once for each uploaded file and can be used to create file-specific links.

```
import os

from buildbot.plugins import steps

class CustomFileUpload(steps.MultipleFileUpload):
    linkTypes = ('.html', '.txt')

    def linkFile(self, basename):
        name, ext = os.path.splitext(basename)
```

```
        return ext in self.linkTypes

    def uploadDone(self, result, source, masterdest):
        if self.url:
            basename = os.path.basename(source)
            if self.linkFile(basename):
                self.addURL(self.url + '/' + basename, basename)

    def allUploadsDone(self, result, sources, masterdest):
        if self.url:
            notLinked = [src for src in sources if not self.linkFile(src)]
            numFiles = len(notLinked)
            if numFiles:
                self.addURL(self.url, '... %d more' % numFiles)
```

Transfering Strings

class buildbot.steps.transfer.**StringDownload**

class buildbot.steps.transfer.**JSONStringDownload**

class buildbot.steps.transfer.**JSONPropertiesDownload**

Sometimes it is useful to transfer a calculated value from the master to the worker. Instead of having to create a temporary file and then use FileDownload, you can use one of the string download steps.

```
from buildbot.plugins import steps, util

f.addStep(steps.StringDownload(util.Interpolate("%(src::branch)s-%(prop:got_
↪revision)s\n"),
        workerdest="buildid.txt"))
```

StringDownload works just like *FileDownload* except it takes a single argument, `s`, representing the string to download instead of a `mastersrc` argument.

```
from buildbot.plugins import steps

buildinfo = {
    'branch': Property('branch'),
    'got_revision': Property('got_revision')
}
f.addStep(steps.JSONStringDownload(buildinfo, workerdest="buildinfo.json"))
```

JSONStringDownload is similar, except it takes an `o` argument, which must be JSON serializable, and transfers that as a JSON-encoded string to the worker.

```
from buildbot.plugins import steps

f.addStep(steps.JSONPropertiesDownload(workerdest="build-properties.json"))
```

JSONPropertiesDownload transfers a json-encoded string that represents a dictionary where properties maps to a dictionary of build property `name` to property `value`; and `sourcestamp` represents the build's sourcestamp.

Running Commands on the Master

class buildbot.steps.master.**MasterShellCommand**

Occasionally, it is useful to execute some task on the master, for example to create a directory, deploy a build result, or trigger some other centralized processing. This is possible, in a limited fashion, with the *MasterShellCommand* step.

This step operates similarly to a regular *ShellCommand*, but executes on the master, instead of the worker. To be clear, the enclosing `Build` object must still have a worker object, just as for any other step – only, in this step, the worker does not do anything.

In this example, the step renames a tarball based on the day of the week.

```
from buildbot.plugins import steps

f.addStep(steps.FileUpload(workersrc="widgetsoft.tar.gz",
                     masterdest="/var/buildoutputs/widgetsoft-new.tar.gz"))
f.addStep(steps.MasterShellCommand(
    command="mv widgetsoft-new.tar.gz widgetsoft-`date +%a`.tar.gz",
    workdir="/var/buildoutputs"))
```

Note: By default, this step passes a copy of the buildmaster's environment variables to the subprocess. To pass an explicit environment instead, add an `env={..}` argument.

Environment variables constructed using the `env` argument support expansion so that if you just want to prepend `/home/buildbot/bin` to the `PATH` environment variable, you can do it by putting the value `${PATH}` at the end of the value like in the example below. Variables that don't exist on the master will be replaced by `""`.

```
from buildbot.plugins import steps

f.addStep(steps.MasterShellCommand(
               command=["make", "www"],
               env={'PATH': ["/home/buildbot/bin",
                             "${PATH}"]}))
```

Note that environment values must be strings (or lists that are turned into strings). In particular, numeric properties such as `buildnumber` must be substituted using *Interpolate*.

`workdir` (optional) The directory from which the command will be ran.

`interruptSignal` (optional) Signal to use to end the process, if the step is interrupted.

LogRenderable

class `buildbot.steps.master.`**`LogRenderable`**

This build step takes content which can be renderable and logs it in a pretty-printed format. It can be useful for debugging properties during a build.

Setting Properties

These steps set properties on the master based on information from the worker.

SetProperty

class `buildbot.steps.master.`**`SetProperty`**

SetProperty takes two arguments of `property` and `value` where the `value` is to be assigned to the `property` key. It is usually called with the `value` argument being specified as a *Interpolate* object which allows the value to be built from other property values:

```
from buildbot.plugins import steps, util

f.addStep(
    steps.SetProperty(
        property="SomeProperty",
        value=util.Interpolate("sch=%(prop:scheduler)s, worker=%(prop:workername)s")
    )
)
```

SetProperties step

class buildbot.steps.master.**SetProperty**

SetProperties takes a dictionary to be turned into build properties.

It is similar to *SetProperty*, and meant to be used with a *Renderer* function or a dictionary of *Interpolate* objects which allows the value to be built from other property values:

```
"""Example borrowed from Julia's master.cfg
   https://github.com/staticfloat/julia-buildbot (MIT)"""
from buildbot.plugins import *

@util.renderer
def compute_artifact_filename(props):
    # Get the output of the `make print-BINARYDIST_FILENAME` step
    reported_filename = props.getProperty('artifact_filename')

    # First, see if we got a BINARYDIST_FILENAME output
    if reported_filename[:26] == "BINARYDIST_FILENAME=":
        local_filename = util.Interpolate(reported_filename[26:].strip()+"%(prop:os_
↪pkg_ext)s")
    else:
        # If not, use non-sf/consistent_distnames naming
        if is_mac(props):
            local_filename = util.Interpolate("contrib/mac/app/Julia-%(prop:version)s-
↪%(prop:shortcommit)s.%(prop:os_pkg_ext)s")
        elif is_winnt(props):
            local_filename = util.Interpolate("julia-%(prop:version)s-%(prop:tar_
↪arch)s.%(prop:os_pkg_ext)s")
        else:
            local_filename = util.Interpolate("julia-%(prop:shortcommit)s-Linux-
↪%(prop:tar_arch)s.%(prop:os_pkg_ext)s")

    # upload_filename always follows sf/consistent_distname rules
    upload_filename = util.Interpolate("julia-%(prop:shortcommit)s-%(prop:os_name)s
↪%(prop:bits)s.%(prop:os_pkg_ext)s")
    return {
        "local_filename": local_filename
        "upload_filename": upload_filename
    }

f1.addStep(steps.SetProperties(properties=compute_artifact_filename))
```

SetPropertyFromCommand

class buildbot.steps.shell.**SetPropertyFromCommand**

This buildstep is similar to *ShellCommand*, except that it captures the output of the command into a property. It is usually used like this:

```
from buildbot.plugins import steps

f.addStep(steps.SetPropertyFromCommand(command="uname -a", property="uname"))
```

This runs `uname -a` and captures its stdout, stripped of leading and trailing whitespace, in the property `uname`. To avoid stripping, add `strip=False`.

The `property` argument can be specified as a *Interpolate* object, allowing the property name to be built from other property values.

Passing `includeStdout=False` (default `True`) stops capture from stdout.

Passing `includeStderr=True` (default `False`) allows capture from stderr.

The more advanced usage allows you to specify a function to extract properties from the command output. Here you can use regular expressions, string interpolation, or whatever you would like. In this form, `extract_fn` should be passed, and not `Property`. The `extract_fn` function is called with three arguments: the exit status of the command, its standard output as a string, and its standard error as a string. It should return a dictionary containing all new properties.

Note that passing in `extract_fn` will set `includeStderr` to `True`.

```
def glob2list(rc, stdout, stderr):
    jpgs = [l.strip() for l in stdout.split('\n')]
    return {'jpgs': jpgs}

f.addStep(SetPropertyFromCommand(command="ls -1 *.jpg", extract_fn=glob2list))
```

Note that any ordering relationship of the contents of stdout and stderr is lost. For example, given:

```
f.addStep(SetPropertyFromCommand(
    command="echo output1; echo error >&2; echo output2",
    extract_fn=my_extract))
```

Then `my_extract` will see `stdout="output1\noutput2\n"` and `stderr="error\n"`.

Avoid using the `extract_fn` form of this step with commands that produce a great deal of output, as the output is buffered in memory until complete.

class buildbot.steps.worker.**SetPropertiesFromEnv**

SetPropertiesFromEnv

Buildbot workers (later than version 0.8.3) provide their environment variables to the master on connect. These can be copied into Buildbot properties with the *SetPropertiesFromEnv* step. Pass a variable or list of variables in the `variables` parameter, then simply use the values as properties in a later step.

Note that on Windows, environment variables are case-insensitive, but Buildbot property names are case sensitive. The property will have exactly the variable name you specify, even if the underlying environment variable is capitalized differently. If, for example, you use `variables=['Tmp']`, the result will be a property named `Tmp`, even though the environment variable is displayed as `TMP` in the Windows GUI.

```
from buildbot.plugins import steps, util

f.addStep(steps.SetPropertiesFromEnv(variables=["SOME_JAVA_LIB_HOME", "JAVAC"]))
f.addStep(steps.Compile(commands=[util.Interpolate("%(prop:JAVAC)s"),
                                  "-cp",
                                  util.Interpolate("%(prop:SOME_JAVA_LIB_HOME)s")]))
```

Note that this step requires that the worker be at least version 0.8.3. For previous versions, no environment variables are available (the worker environment will appear to be empty).

Triggering Schedulers

class buildbot.steps.trigger.**Trigger**

The counterpart to the *Triggerable* scheduler is the *Trigger* build step:

```
from buildbot.plugins import steps

f.addStep(steps.Trigger(schedulerNames=['build-prep'],
                        waitForFinish=True,
                        updateSourceStamp=True,
                        set_properties={ 'quick' : False }))
```

The SourceStamps to use for the triggered build are controlled by the arguments `updateSourceStamp`, `alwaysUseLatest`, and `sourceStamps`.

Hyperlinks are added to the build detail web pages for each triggered build.

schedulerNames lists the *Triggerable* schedulers that should be triggered when this step is executed.

> **Note:** It is possible, but not advisable, to create a cycle where a build continually triggers itself, because the schedulers are specified by name.

unimportantSchedulerNames When `waitForFinish` is `True`, all schedulers in this list will not cause the trigger step to fail. unimportantSchedulerNames must be a subset of schedulerNames If `waitForFinish` is `False`, unimportantSchedulerNames will simply be ignored.

waitForFinish If `True`, the step will not finish until all of the builds from the triggered schedulers have finished.

If `False` (the default) or not given, then the buildstep succeeds immediately after triggering the schedulers.

updateSourceStamp If `True` (the default), then step updates the source stamps given to the *Triggerable* schedulers to include `got_revision` (the revision actually used in this build) as `revision` (the revision to use in the triggered builds). This is useful to ensure that all of the builds use exactly the same source stamps, even if other `Changes` have occurred while the build was running.

If `False` (and neither of the other arguments are specified), then the exact same SourceStamps are used.

alwaysUseLatest If `True`, then no SourceStamps are given, corresponding to using the latest revisions of the repositories specified in the Source steps. This is useful if the triggered builds use to a different source repository.

sourceStamps Accepts a list of dictionaries containing the keys `branch`, `revision`, `repository`, `project`, and optionally `patch_level`, `patch_body`, `patch_subdir`, `patch_author` and `patch_comment` and creates the corresponding SourceStamps. If only one sourceStamp has to be specified then the argument `sourceStamp` can be used for a dictionary containing the keys mentioned above. The arguments `updateSourceStamp`, `alwaysUseLatest`, and `sourceStamp` can be specified using properties.

set_properties allows control of the properties that are passed to the triggered scheduler. The parameter takes a dictionary mapping property names to values. You may use *Interpolate* here to dynamically construct new property values. For the simple case of copying a property, this might look like:

```
set_properties={"my_prop1" : Property("my_prop1")}
```

Note: The `copy_properties` parameter, given a list of properties to copy into the new build request, has been deprecated in favor of explicit use of `set_properties`.

Dynamic Trigger

Sometimes it is desirable to select which scheduler to trigger, and which properties to set dynamically, at the time of the build. For this purpose, Trigger step supports a method that you can customize in order to override statically defined `schedulernames`, `set_properties` and optionally `unimportant`.

`buildbot.steps.source.`**`getSchedulersAndProperties`**`()`

Returns list of dictionaries containing the keys 'sched_name', 'props_to_set' and 'unimportant' optionally via deferred

This method returns a list of dictionaries describing what scheduler to trigger, with which properties and if the scheduler is unimportant. Old style list of tuples is still supported, in which case unimportant is considered `False`. The properties should already be rendered (ie, concrete value, not objects wrapped by `Interpolate` or `Property`). Since this function happens at build-time, the property values are available from the step and can be used to decide what schedulers or properties to use.

With this method, you can also trigger the same scheduler multiple times with different set of properties. The sourcestamp configuration is however the same for each triggered build request.

RPM-Related Steps

These steps work with RPMs and spec files.

RpmBuild

The *`RpmBuild`* step builds RPMs based on a spec file:

```
from buildbot.plugins import steps

f.addStep(steps.RpmBuild(specfile="proj.spec", dist='.el5'))
```

The step takes the following parameters

specfile The `.spec` file to build from

topdir Definition for `_topdir`, defaulting to the workdir.

builddir Definition for `_builddir`, defaulting to the workdir.

rpmdir Definition for `_rpmdir`, defaulting to the workdir.

sourcedir Definition for `_sourcedir`, defaulting to the workdir.

srcrpmdir Definition for `_srcrpmdir`, defaulting to the workdir.

dist Distribution to build, used as the definition for `_dist`.

autoRelease If true, use the auto-release mechanics.

vcsRevision If true, use the version-control revision mechanics. This uses the `got_revision` property to determine the revision and define `_revision`. Note that this will not work with multi-codebase builds.

RpmLint

The *RpmLint* step checks for common problems in RPM packages or spec files:

```
from buildbot.plugins import steps

f.addStep(steps.RpmLint())
```

The step takes the following parameters

fileloc The file or directory to check. In case of a directory, it is recursively searched for RPMs and spec files to check.

config Path to a rpmlint config file. This is passed as the user configuration file if present.

Mock Steps

Mock (http://fedoraproject.org/wiki/Projects/Mock) creates chroots and builds packages in them. It populates the changeroot with a basic system and the packages listed as build requirement. The type of chroot to build is specified with the `root` parameter. To use mock your Buildbot user must be added to the `mock` group.

MockBuildSRPM Step

The *MockBuildSRPM* step builds a SourceRPM based on a spec file and optionally a source directory:

```
from buildbot.plugins import steps

f.addStep(steps.MockBuildSRPM(root='default', spec='mypkg.spec'))
```

The step takes the following parameters

root Use chroot configuration defined in `/etc/mock/<root>.cfg`.

resultdir The directory where the logfiles and the SourceRPM are written to.

spec Build the SourceRPM from this spec file.

sources Path to the directory containing the sources, defaulting to `..`

MockRebuild Step

The *MockRebuild* step rebuilds a SourceRPM package:

```
from buildbot.plugins import steps

f.addStep(steps.MockRebuild(root='default', spec='mypkg-1.0-1.src.rpm'))
```

The step takes the following parameters

root Uses chroot configuration defined in `/etc/mock/<root>.cfg`.

resultdir The directory where the logfiles and the SourceRPM are written to.

srpm The path to the SourceRPM to rebuild.

Debian Build Steps

DebPbuilder

The *DebPbuilder* step builds Debian packages within a chroot built by **pbuilder**. It populates the chroot with a basic system and the packages listed as build requirement. The type of the chroot to build is specified with the `distribution`, `distribution` and `mirror` parameter. To use pbuilder your Buildbot user must have the right to run **pbuilder** as root using **sudo**.

```
from buildbot.plugins import steps

f.addStep(steps.DebPbuilder())
```

The step takes the following parameters

architecture Architecture to build chroot for.

distribution Name, or nickname, of the distribution. Defaults to 'stable'.

basetgz Path of the basetgz to use for building.

mirror URL of the mirror used to download the packages from.

extrapackages List if packages to install in addition to the base system.

keyring Path to a gpg keyring to verify the downloaded packages. This is necessary if you build for a foreign distribution.

components Repos to activate for chroot building.

DebCowbuilder

The *DebCowbuilder* step is a subclass of *DebPbuilder*, which use cowbuilder instead of pbuilder.

DebLintian

The *DebLintian* step checks a build .deb for bugs and policy violations. The packages or changes file to test is specified in `fileloc`

```
from buildbot.plugins import steps, util

f.addStep(steps.DebLintian(fileloc=util.Interpolate("%(prop:deb-changes)s")))
```

Miscellaneous BuildSteps

A number of steps do not fall into any particular category.

HLint

The *HLint* step runs Twisted Lore, a lint-like checker over a set of `.xhtml` files. Any deviations from recommended style is flagged and put in the output log.

The step looks at the list of changes in the build to determine which files to check - it does not check all files. It specifically excludes any `.xhtml` files in the top-level `sandbox/` directory.

The step takes a single, optional, parameter: `python`. This specifies the Python executable to use to run Lore.

```
from buildbot.plugins import steps

f.addStep(steps.HLint())
```

MaxQ

MaxQ (http://maxq.tigris.org/) is a web testing tool that allows you to record HTTP sessions and play them back. The *MaxQ* step runs this framework.

```
from buildbot.plugins import steps

f.addStep(steps.MaxQ(testdir='tests/'))
```

The single argument, `testdir`, specifies where the tests should be run. This directory will be passed to the `run_maxq.py` command, and the results analyzed.

HTTP Requests

Using the *HTTPStep* step, it is possible to perform HTTP requests in order to trigger another REST service about the progress of the build.

Note: This step requires the txrequests (https://pypi.python.org/pypi/txrequests) and requests (http://python-requests.org) Python libraries.

The parameters are the following:

`url` (mandatory) The URL where to send the request

`method` The HTTP method to use (out of `POST`, `GET`, `PUT`, `DELETE`, `HEAD` or `OPTIONS`), default to `POST`.

`params` Dictionary of URL parameters to append to the URL.

`data` The body to attach the request. If a dictionary is provided, form-encoding will take place.

`headers` Dictionary of headers to send.

`other params` Any other keywords supported by the `requests` api (http://docs.python-requests.org/en/master/api/#main-interface) can be passed to this step.

Note: The entire Buildbot master process shares a single Requests `Session` object. This has the advantage of supporting connection re-use and other HTTP/1.1 features. However, it also means that any cookies or other state changed by one step will be visible to other steps, causing unexpected results. This behavior may change in future versions.

When the method is known in advance, class with the name of the method can also be used. In this case, it is not necessary to specify the method.

Example:

```
from buildbot.plugins import steps, util

f.addStep(steps.POST('http://myRESTService.example.com/builds',
                     data = {
                        'builder': util.Property('buildername'),
                        'buildnumber': util.Property('buildnumber'),
                        'workername': util.Property('workername'),
                        'revision': util.Property('got_revision')
                     }))
```

Interlocks

Until now, we assumed that a master can run builds at any worker whenever needed or desired. Some times, you want to enforce additional constraints on builds. For reasons like limited network bandwidth, old worker machines, or a self-willed data base server, you may want to limit the number of builds (or build steps) that can access a resource.

Access Modes

The mechanism used by Buildbot is known as the read/write lock[1]. It allows either many readers or a single writer but not a combination of readers and writers. The general lock has been modified and extended for use in Buildbot. Firstly, the general lock allows an infinite number of readers. In Buildbot, we often want to put an upper limit on the number of readers, for example allowing two out of five possible builds at the same time. To do this, the lock counts the number of active readers. Secondly, the terms *read mode* and *write mode* are confusing in Buildbot context. They have been replaced by *counting mode* (since the lock counts them) and *exclusive mode*. As a result of these changes, locks in Buildbot allow a number of builds (up to some fixed number) in counting mode, or they allow one build in exclusive mode.

Note: Access modes are specified when a lock is used. That is, it is possible to have a single lock that is used by several workers in counting mode, and several workers in exclusive mode. In fact, this is the strength of the modes: accessing a lock in exclusive mode will prevent all counting-mode accesses.

Count

Often, not all workers are equal. To allow for this situation, Buildbot allows to have a separate upper limit on the count for each worker. In this way, you can have at most 3 concurrent builds at a fast worker, 2 at a slightly older worker, and 1 at all other workers.

[1] See http://en.wikipedia.org/wiki/Read/write_lock_pattern for more information.

Scope

The final thing you can specify when you introduce a new lock is its scope. Some constraints are global – they must be enforced over all workers. Other constraints are local to each worker. A *master lock* is used for the global constraints. You can ensure for example that at most one build (of all builds running at all workers) accesses the data base server. With a *worker lock* you can add a limit local to each worker. With such a lock, you can for example enforce an upper limit to the number of active builds at a worker, like above.

Examples

Time for a few examples. Below a master lock is defined to protect a data base, and a worker lock is created to limit the number of builds at each worker.

```
from buildbot.plugins import util

db_lock = util.MasterLock("database")
build_lock = util.WorkerLock("worker_builds",
                             maxCount=1,
                             maxCountForWorker={'fast': 3, 'new': 2})
```

`db_lock` is defined to be a master lock. The `database` string is used for uniquely identifying the lock. At the next line, a worker lock called `build_lock` is created. It is identified by the `worker_builds` string. Since the requirements of the lock are a bit more complicated, two optional arguments are also specified. The `maxCount` parameter sets the default limit for builds in counting mode to `1`. For the worker called `'fast'` however, we want to have at most three builds, and for the worker called `'new'` the upper limit is two builds running at the same time.

The next step is accessing the locks in builds. Buildbot allows a lock to be used during an entire build (from beginning to end), or only during a single build step. In the latter case, the lock is claimed for use just before the step starts, and released again when the step ends. To prevent deadlocks,[2] it is not possible to claim or release locks at other times.

To use locks, you add them with a `locks` argument to a build or a step. Each use of a lock is either in counting mode (that is, possibly shared with other builds) or in exclusive mode, and this is indicated with the syntax `lock.access(mode)`, where `mode` is one of `"counting"` or `"exclusive"`.

A build or build step proceeds only when it has acquired all locks. If a build or step needs a lot of locks, it may be starved[3] by other builds that need fewer locks.

To illustrate use of locks, a few examples.

```
from buildbot.plugins import util, steps

db_lock = util.MasterLock("database")
build_lock = util.WorkerLock("worker_builds",
                             maxCount=1,
                             maxCountForWorker={'fast': 3, 'new': 2})

f = util.BuildFactory()
f.addStep(steps.SVN(repourl="http://example.org/svn/Trunk"))
f.addStep(steps.ShellCommand(command="make all"))
f.addStep(steps.ShellCommand(command="make test",
                             locks=[db_lock.access('exclusive')]))

b1 = {'name': 'full1', 'workername': 'fast',  'builddir': 'f1', 'factory': f,
```

[2] Deadlock is the situation where two or more workers each hold a lock in exclusive mode, and in addition want to claim the lock held by the other worker exclusively as well. Since locks allow at most one exclusive user, both workers will wait forever.

[3] Starving is the situation that only a few locks are available, and they are immediately grabbed by another build. As a result, it may take a long time before all locks needed by the starved build are free at the same time.

```
        'locks': [build_lock.access('counting')] }

b2 = {'name': 'full2', 'workername': 'new',   'builddir': 'f2', 'factory': f,
        'locks': [build_lock.access('counting')] }

b3 = {'name': 'full3', 'workername': 'old',   'builddir': 'f3', 'factory': f,
        'locks': [build_lock.access('counting')] }

b4 = {'name': 'full4', 'workername': 'other', 'builddir': 'f4', 'factory': f,
        'locks': [build_lock.access('counting')] }

c['builders'] = [b1, b2, b3, b4]
```

Here we have four workers `fast`, `new`, `old`, and `other`. Each worker performs the same checkout, make, and test build step sequence. We want to enforce that at most one test step is executed between all workers due to restrictions with the data base server. This is done by adding the `locks=` parameter with the third step. It takes a list of locks with their access mode. Alternatively, this can take a renderable that returns an list of locks with their access mode.

In this case only the `db_lock` is needed. The exclusive access mode is used to ensure there is at most one worker that executes the test step.

In addition to exclusive accessing the data base, we also want workers to stay responsive even under the load of a large number of builds being triggered. For this purpose, the worker lock called `build_lock` is defined. Since the restraint holds for entire builds, the lock is specified in the builder with `'locks': [build_lock.access('counting')]`.

Note that you will occasionally see `lock.access(mode)` written as `LockAccess(lock, mode)`. The two are equivalent, but the former is preferred.

Reporters

- *GerritVerifyStatusPush*

The Buildmaster has a variety of ways to present build status to various users. Each such delivery method is a *Reporter Target* object in the configuration's `services` list. To add reporter targets, you just append more objects to this list:

```
c['services'] = []

m = reporters.MailNotifier(fromaddr="buildbot@localhost",
                           extraRecipients=["builds@lists.example.com"],
                           sendToInterestedUsers=False)
c['services'].append(m)

c['services'].append(reporters.IRC(host="irc.example.com", nick="bb",
                                   channels=[{"channel": "#example1"},
                                             {"channel": "#example2",
                                              "password": "somesecretpassword"}]))
```

Most reporter objects take a `tags=` argument, which can contain a list of tag names: in this case, it will only show status for Builders that contains the named tags.

Note: Implementation Note

Each of these objects should be a `service.BuildbotService` which will be attached to the BuildMaster object when the configuration is processed.

The remainder of this section describes each built-in reporters. A full list of reporters is available in the `reporter`.

MailNotifier

class `buildbot.reporters.mail.`**`MailNotifier`**

The Buildbot can send email when builds finish. The most common use of this is to tell developers when their change has caused the build to fail. It is also quite common to send a message to a mailing list (usually named *builds* or similar) about every build.

The `MailNotifier` reporter is used to accomplish this. You configure it by specifying who mail should be sent to, under what circumstances mail should be sent, and how to deliver the mail. It can be configured to only send out mail for certain builders, and only send messages when the build fails, or when the builder transitions from success to failure. It can also be configured to include various build logs in each message.

If a proper lookup function is configured, the message will be sent to the "interested users" list (*Doing Things With Users*), which includes all developers who made changes in the build. By default, however, Buildbot does not know how to construct an email addressed based on the information from the version control system. See the `lookup` argument, below, for more information.

You can add additional, statically-configured, recipients with the `extraRecipients` argument. You can also add interested users by setting the `owners` build property to a list of users in the scheduler constructor (*Configuring Schedulers*).

Each `MailNotifier` sends mail to a single set of recipients. To send different kinds of mail to different recipients, use multiple `MailNotifiers`. TODO: or subclass MailNotifier and override getRecipients()

The following simple example will send an email upon the completion of each build, to just those developers whose `Changes` were included in the build. The email contains a description of the `Build`, its results, and URLs where more information can be obtained.

```
from buildbot.plugins import reporters
mn = reporters.MailNotifier(fromaddr="buildbot@example.org",
                            lookup="example.org")
c['services'].append(mn)
```

To get a simple one-message-per-build (say, for a mailing list), use the following form instead. This form does not send mail to individual developers (and thus does not need the `lookup=` argument, explained below), instead it only ever sends mail to the *extra recipients* named in the arguments:

```
mn = reporters.MailNotifier(fromaddr="buildbot@example.org",
                            sendToInterestedUsers=False,
                            extraRecipients=['listaddr@example.org'])
```

If your SMTP host requires authentication before it allows you to send emails, this can also be done by specifying `smtpUser` and `smtpPassword`:

```
mn = reporters.MailNotifier(fromaddr="myuser@example.com",
                            sendToInterestedUsers=False,
                            extraRecipients=["listaddr@example.org"],
                            relayhost="smtp.example.com", smtpPort=587,
                            smtpUser="myuser@example.com",
                            smtpPassword="mypassword")
```

Note: If for some reasons you are not able to send a notification with TLS enabled and specified user name and password, you might want to use master/contrib/check_smtp.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/check_smtp.py) to see if it works at all.

If you want to require Transport Layer Security (TLS), then you can also set `useTls`:

```
mn = reporters.MailNotifier(fromaddr="myuser@example.com",
                            sendToInterestedUsers=False,
                            extraRecipients=["listaddr@example.org"],
                            useTls=True, relayhost="smtp.example.com",
                            smtpPort=587, smtpUser="myuser@example.com",
                            smtpPassword="mypassword")
```

Note: If you see `twisted.mail.smtp.TLSRequiredError` exceptions in the log while using TLS, this can be due *either* to the server not supporting TLS or to a missing PyOpenSSL (http://pyopenssl.sourceforge.net/) package on the BuildMaster system.

In some cases it is desirable to have different information then what is provided in a standard MailNotifier message. For this purpose MailNotifier provides the argument `messageFormatter` (an instance of `MessageFormatter`) which allows for the creation of messages with unique content.

For example, if only short emails are desired (e.g., for delivery to phones):

```
from buildbot.plugins import reporters
mn = reporters.MailNotifier(fromaddr="buildbot@example.org",
                            sendToInterestedUsers=False,
                            mode=('problem',),
                            extraRecipients=['listaddr@example.org'],
                            messageFormatter=reporters.MessageFormatter(template=
↪"STATUS: {{ summary }}"))
```

Another example of a function delivering a customized html email is given below:

```
from buildbot.plugins import reporters

template=u'''\
<h4>Build status: {{ summary }}</h4>
<p> Worker used: {{ workername }}</p>
{% for step in build['steps'] %}
<p> {{ step['name'] }}: {{ step['result'] }}</p>
{% endfor %}
<p><b> -- The Buildbot</b></p>
'''

mn = reporters.MailNotifier(fromaddr="buildbot@example.org",
                            sendToInterestedUsers=False,
                            mode=('failing',),
                            extraRecipients=['listaddr@example.org'],
                            messageFormatter=reporters.MessageFormatter(
                                template=template, template_type='html',
                                wantProperties=True, wantSteps=True))
```

MailNotifier arguments

fromaddr The email address to be used in the 'From' header.

sendToInterestedUsers (boolean). If `True` (the default), send mail to all of the Interested Users. If `False`, only send mail to the `extraRecipients` list.

extraRecipients (list of strings). A list of email addresses to which messages should be sent (in addition to the InterestedUsers list, which includes any developers who made `Changes` that went into this build). It is a good idea to create a small mailing list and deliver to that, then let subscribers come and go as they please.

subject (string). A string to be used as the subject line of the message. `%(builder)s` will be replaced with the name of the builder which provoked the message.

mode Mode is a list of strings; however there are two strings which can be used as shortcuts instead of the full lists. The possible shortcuts are:

all Always send mail about builds. Equivalent to (`change`, `failing`, `passing`, `problem`, `warnings`, `exception`).

warnings Equivalent to (`warnings`, `failing`).

(list of strings). A combination of:

cancelled Send mail about builds which were cancelled.

change Send mail about builds which change status.

failing Send mail about builds which fail.

passing Send mail about builds which succeed.

problem Send mail about a build which failed when the previous build has passed.

warnings Send mail about builds which generate warnings.

exception Send mail about builds which generate exceptions.

Defaults to (`failing`, `passing`, `warnings`).

builders (list of strings). A list of builder names for which mail should be sent. Defaults to `None` (send mail for all builds). Use either builders or tags, but not both.

tags (list of strings). A list of tag names to serve status information for. Defaults to `None` (all tags). Use either builders or tags, but not both.

schedulers (list of strings). A list of scheduler names to serve status information for. Defaults to `None` (all schedulers).

branches (list of strings). A list of branch names to serve status information for. Defaults to `None` (all branches).

addLogs (boolean). If `True`, include all build logs as attachments to the messages. These can be quite large. This can also be set to a list of log names, to send a subset of the logs. Defaults to `False`.

addPatch (boolean). If `True`, include the patch content if a patch was present. Patches are usually used on a `Try` server. Defaults to `True`.

buildSetSummary (boolean). If `True`, send a single summary email consisting of the concatenation of all build completion messages rather than a completion message for each build. Defaults to `False`.

relayhost (string). The host to which the outbound SMTP connection should be made. Defaults to 'localhost'

smtpPort (int). The port that will be used on outbound SMTP connections. Defaults to 25.

useTls (boolean). When this argument is `True` (default is `False`) `MailNotifier` requires that STARTTLS encryption is used for the connection with the `relayhost`. Authentication is required for STARTTLS so the arguments `smtpUser` and `smtpPassword` must also be specified.

useSmtps (boolean). When this argument is `True` (default is `False`) `MailNotifier` connects to `relayhost` over an encrypted SSL/TLS connection. This configuration is typically used over port 465.

smtpUser (string). The user name to use when authenticating with the `relayhost`.

smtpPassword (string). The password that will be used when authenticating with the `relayhost`.

lookup (implementer of `IEmailLookup`). Object which provides `IEmailLookup`, which is responsible for mapping User names (which come from the VC system) into valid email addresses.

If the argument is not provided, the `MailNotifier` will attempt to build the `sendToInterestedUsers` from the authors of the Changes that led to the Build via *User Objects*. If the author of one of the Build's Changes has an email address stored, it will added to the recipients list. With this method, `owners` are still added to the recipients. Note that, in the current implementation of user objects, email addresses are not stored; as a result, unless you have specifically added email addresses to the user database, this functionality is unlikely to actually send any emails.

Most of the time you can use a simple Domain instance. As a shortcut, you can pass as string: this will be treated as if you had provided `Domain(str)`. For example, `lookup='example.com'` will allow mail to be sent to all developers whose SVN usernames match their `example.com` account names. See master/buildbot/reporters/mail.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/reporters/mail.py) for more details.

Regardless of the setting of `lookup`, `MailNotifier` will also send mail to addresses in the `extraRecipients` list.

messageFormatter This is an optional instance of the `reporters.MessageFormatter` class that can be used to generate a custom mail message. This class uses the Jinja2 (http://jinja.pocoo.org/docs/dev/templates/) templating language to generate the body and optionally the subject of the mails. Templates can either be given inline (as string), or read from the filesystem.

extraHeaders (dictionary). A dictionary containing key/value pairs of extra headers to add to sent e-mails. Both the keys and the values may be a *Interpolate* instance.

messageFormatterMissingWorker This is an optional instance of the `reporters.messageFormatterMissingWorker` class that can be used to generate a custom mail message for missing workers. This class uses the Jinja2 (http://jinja.pocoo.org/docs/dev/templates/) templating language to generate the body and optionally the subject of the mails. Templates can either be given inline (as string), or read from the filesystem.

MessageFormatter arguments

The easiest way to use the `messageFormatter` parameter is to create a new instance of the `reporters.MessageFormatter` class. The constructor to that class takes the following arguments:

template_dir This is the directory that is used to look for the various templates.

template_filename This is the name of the file in the `template_dir` directory that will be used to generate the body of the mail. It defaults to `default_mail.txt`.

template If this parameter is set, this parameter indicates the content of the template used to generate the body of the mail as string.

template_type This indicates the type of the generated template. Use either 'plain' (the default) or 'html'.

subject_filename This is the name of the file in the `template_dir` directory that contains the content of the subject of the mail.

subject Alternatively, this is the content of the subject of the mail as string.

ctx This is an extension of the standard context that will be given to the templates. Use this to add content to the templates that is otherwise not available.

Alternatively, you can subclass MessageFormatter and override the *buildAdditionalContext* in order to grab more context from the data API.

buildAdditionalContext (*master*, *ctx*)

Parameters

- **master** – the master object
- **ctx** – the context dictionary to enhance

Returns optionally deferred

default implementation will add `self.ctx` into the current template context

wantProperties This parameter (defaults to True) will extend the content of the given `build` object with the Properties from the build.

wantSteps This parameter (defaults to False) will extend the content of the given `build` object with information about the steps of the build. Use it only when necessary as this increases the overhead in term of CPU and memory on the master.

wantLogs This parameter (defaults to False) will extend the content of the steps of the given `build` object with the full Logs of each steps from the build. This requires `wantSteps` to be True. Use it only when mandatory as this increases the overhead in term of CPU and memory on the master greatly.

As a help to those writing Jinja2 templates the following table describes how to get some useful pieces of information from the various data objects:

Name of the builder that generated this event `{{ buildername }}`

Title of the BuildMaster `{{ projects }}`

MailNotifier mode `{{ mode }}` (a combination of `change`, `failing`, `passing`, `problem`, `warnings`, `exception`, `all`)

URL to build page `{{ build_url }}`

URL to buildbot main page `{{ buildbot_url }}`

Status of the build as string. This require extending the context of the Formatter via the `ctx` parameter with: `ctx=dict(statuses=util.Results)`.

```
{{ statuses[results] }}
```

Build text `{{ build['state_string'] }}`

Mapping of property names to (values, source) `{{ build['properties'] }}`

For instance the build reason (from a forced build) `{{ build['properties']['reason'][0] }}`

Worker name `{{ workername }}`

List of responsible users `{{ blamelist | join(', ') }}`

MessageFormatterMissingWorkers arguments

The easiest way to use the `messageFormatterMissingWorkers` parameter is to create a new instance of the `reporters.MessageFormatterMissingWorkers` class.

The constructor to that class takes the same arguments as MessageFormatter, minus `wantLogs`, `wantProperties`, `wantSteps`.

The default `ctx` for the missing worker email is made of:

`buildbot_title` The buildbot title as per `c['title']` from the `master.cfg`

`buildbot_url` The buildbot title as per `c['title']` from the `master.cfg`

`worker` The worker object as defined in the REST api plus two attributes:

`notify` List of emails to be notified for this worker.

`last_connection` String describing the approximate the time of last connection for this worker.

IRC Bot

The *IRC* reporter creates an IRC bot which will attach to certain channels and be available for status queries. It can also be asked to announce builds as they occur, or be told to shut up.

The IRC Bot in buildbot nine, is mostly a rewrite, and not all functionality has been ported yet. Patches are very welcome for restoring the full functionality.

Note: Security Note

Please note that any user having access to your irc channel or can PM the bot will be able to create or stop builds bug #3377 (http://trac.buildbot.net/ticket/3377).

```
from buildbot.plugins import reporters
irc = reporters.IRC("irc.example.org", "botnickname",
                    useColors=False,
                    channels=[{"channel": "#example1"},
                              {"channel": "#example2",
```

```
                              "password": "somesecretpassword"}],
                    password="mysecretnickservpassword",
                    notify_events={
                      'exception': 1,
                      'successToFailure': 1,
                      'failureToSuccess': 1,
                    })
c['services'].append(irc)
```

The following parameters are accepted by this class:

host (mandatory) The IRC server address to connect to.

nick (mandatory) The name this bot will use on the IRC server.

channels (mandatory) This is a list of channels to join on the IRC server. Each channel can be a string (e.g. `#buildbot`), or a dictionary `{'channel': '#buildbot', 'password': 'secret'}` if each channel requires a different password. A global password can be set with the `password` parameter.

pm_to_nicks (optional) This is a list of person to contact on the IRC server.

port (optional, default to 6667) The port to connect to on the IRC server.

allowForce (optional, disabled by default) This allow user to force builds via this bot.

tags (optional) When set, this bot will only communicate about builders containing those tags. (tags functionality is not yet ported)

password (optional) The global password used to register the bot to the IRC server. If provided, it will be sent to Nickserv to claim the nickname: some IRC servers will not allow clients to send private messages until they have logged in with a password.

notify_events (optional) A dictionary of events to be notified on the IRC channels. At the moment, irc bot can listen to build 'start' and 'finish' events. This parameter can be changed during run-time by sending the `notify` command to the bot.

showBlameList (optional, disabled by default) Whether or not to display the blame list for failed builds. (blame list functionality is not ported yet)

useRevisions (optional, disabled by default) Whether or not to display the revision leading to the build the messages are about. (useRevisions functionality is not ported yet)

useSSL (optional, disabled by default) Whether or not to use SSL when connecting to the IRC server. Note that this option requires PyOpenSSL (http://pyopenssl.sourceforge.net/).

lostDelay (optional) Delay to wait before reconnecting to the server when the connection has been lost.

failedDelay (optional) Delay to wait before reconnecting to the IRC server when the connection failed.

useColors (optional, enabled by default) The bot can add color to some of its messages. You might turn it off by setting this parameter to `False`.

allowShutdown (optional, disabled by default) This allow users to shutdown the master.

To use the service, you address messages at the Buildbot, either normally (`botnickname: status`) or with private messages (`/msg botnickname status`). The Buildbot will respond in kind.

If you issue a command that is currently not available, the Buildbot will respond with an error message. If the `noticeOnChannel=True` option was used, error messages will be sent as channel notices instead of messaging.

Some of the commands currently available:

list builders Emit a list of all configured builders

status *BUILDER* Announce the status of a specific Builder: what it is doing right now.

status all Announce the status of all Builders

watch *BUILDER* If the given `Builder` is currently running, wait until the `Build` is finished and then announce the results.

last *BUILDER* Return the results of the last build to run on the given `Builder`.

join *CHANNEL* Join the given IRC channel

leave *CHANNEL* Leave the given IRC channel

notify on|off|list *EVENT* Report events relating to builds. If the command is issued as a private message, then the report will be sent back as a private message to the user who issued the command. Otherwise, the report will be sent to the channel. Available events to be notified are:

started A build has started

finished A build has finished

success A build finished successfully

failure A build failed

exception A build generated and exception

xToY The previous build was x, but this one is Y, where x and Y are each one of success, warnings, failure, exception (except Y is capitalized). For example: `successToFailure` will notify if the previous build was successful, but this one failed

help *COMMAND* Describe a command. Use **help commands** to get a list of known commands.

shutdown *ARG* Control the shutdown process of the Buildbot master. Available arguments are:

check Check if the Buildbot master is running or shutting down

start Start clean shutdown

stop Stop clean shutdown

now Shutdown immediately without waiting for the builders to finish

source Announce the URL of the Buildbot's home page.

version Announce the version of this Buildbot.

Additionally, the config file may specify default notification options as shown in the example earlier.

If the `allowForce=True` option was used, some additional commands will be available:

force build [--codebase=*CODEBASE*] [--branch=*BRANCH*] [--revision=*REVISION*] [--props=PROP1=V
Tell the given `Builder` to start a build of the latest code. The user requesting the build and *REASON* are recorded in the `Build` status. The Buildbot will announce the build's status when it finishes.The user can specify a branch and/or revision with the optional parameters `--branch=`*`BRANCH`* and `--revision=`*`REVISION`*. The user can also give a list of properties with `--props=`*`PROP1=VAL1, PROP2=VAL2...`*.

stop build *BUILDER REASON* Terminate any running build in the given `Builder`. *REASON* will be added to the build status to explain why it was stopped. You might use this if you committed a bug, corrected it right away, and don't want to wait for the first build (which is destined to fail) to complete before starting the second (hopefully fixed) build.

If the *tags* is set (see the tags option in *Builder Configuration*) changes related to only builders belonging to those tags of builders will be sent to the channel.

If the *useRevisions* option is set to *True*, the IRC bot will send status messages that replace the build number with a list of revisions that are contained in that build. So instead of seeing *build #253 of ...*, you would see something like *build containing revisions [a87b2c4]*. Revisions that are stored as hashes are shortened to 7 characters in length, as multiple revisions can be contained in one build and may exceed the IRC message length limit.

Two additional arguments can be set to control how fast the IRC bot tries to reconnect when it encounters connection issues. `lostDelay` is the number of of seconds the bot will wait to reconnect when the connection is lost, where as `failedDelay` is the number of seconds until the bot tries to reconnect when the connection failed. `lostDelay` defaults to a random number between 1 and 5, while `failedDelay` defaults to a random one between 45 and 60. Setting random defaults like this means multiple IRC bots are less likely to deny each other by flooding the server.

GerritStatusPush

class `buildbot.status.status_gerrit.`**`GerritStatusPush`**

`GerritStatusPush` sends review of the `Change` back to the Gerrit server, optionally also sending a message when a build is started. GerritStatusPush can send a separate review for each build that completes, or a single review summarizing the results for all of the builds.

class **`GerritStatusPush`**(*server*, *username*, *reviewCB*, *startCB*, *port*, *reviewArg*, *startArg*, *summaryCB*, *summaryArg*, *identity_file*, *builders*, *notify...*)

Parameters

- **`server`** (*`string`*) – Gerrit SSH server's address to use for push event notifications.
- **`username`** (*`string`*) – Gerrit SSH server's username.
- **`identity_file`** – (optional) Gerrit SSH identity file.
- **`port`** (*`int`*) – (optional) Gerrit SSH server's port (default: 29418)
- **`reviewCB`** – (optional) Called each time a build finishes. Build properties are available. Can be a deferred.
- **`reviewArg`** – (optional) argument passed to the review callback.

  If `reviewCB` callback is specified, it must return a message and optionally labels. If no message is specified, nothing will be sent to Gerrit. It should return a dictionary:

```
{'message': message,
 'labels': {label-name: label-score,
            ...}
}
```

  For example:

```
def gerritReviewCB(builderName, build, result, master, arg):
    if result == util.RETRY:
        return dict()

    message =  "Buildbot finished compiling your patchset\n"
    message += "on configuration: %s\n" % builderName
    message += "The result is: %s\n" % util.Results[result].upper()

    if arg:
        message += "\nFor more details visit:\n"
        message += build['url'] + "\n"

    if result == util.SUCCESS:
        verified = 1
```

```
    else:
        verified = -1

    return dict(message=message, labels={'Verified': verified})
```

Which require an extra import in the config:

```
from buildbot.plugins import util
```

- **startCB** – (optional) Called each time a build is started. Build properties are available. Can be a deferred.
- **startArg** – (optional) argument passed to the start callback.

If `startCB` is specified, it must return a message and optionally labels. If no message is specified, nothing will be sent to Gerrit. It should return a dictionary:

```
{'message': message,
 'labels': {label-name: label-score,
            ...}
}
```

For example:

```
def gerritStartCB(builderName, build, arg):
    message = "Buildbot started compiling your patchset\n"
    message += "on configuration: %s\n" % builderName
    message += "See your build here: %s" % build['url']

    return dict(message=message)
```

- **summaryCB** – (optional) Called each time a buildset finishes. Each build in the buildset has properties available. Can be a deferred.
- **summaryArg** – (optional) argument passed to the summary callback.

If `summaryCB` callback is specified, it must return a message and optionally labels. If no message is specified, nothing will be sent to Gerrit. The message and labels should be a summary of all the builds within the buildset. It should return a dictionary:

```
{'message': message,
 'labels': {label-name: label-score,
            ...}
}
```

For example:

```
def gerritSummaryCB(buildInfoList, results, status, arg):
    success = False
    failure = False

    msgs = []

    for buildInfo in buildInfoList:
        msg = "Builder %(name)s %(resultText)s (%(text)s)" %␣
↪buildInfo
        link = buildInfo.get('url', None)
        if link:
            msg += " - " + link
```

```
        else:
            msg += "."

        msgs.append(msg)

        if buildInfo['result'] == util.SUCCESS:
            success = True
        else:
            failure = True

    if success and not failure:
        verified = 1
    else:
        verified = -1

    return dict(message='\n\n'.join(msgs),
                labels={
                    'Verified': verified
                })
```

- **builders** – (optional) list of builders to send results for. This method allows to filter results for a specific set of builder. By default, or if builders is None, then no filtering is performed.
- **notify** – (optional) control who gets notified by Gerrit once the status is posted. The possible values for *notify* can be found in your version of the Gerrit documentation for the *gerrit review* command.

Note: By default, a single summary review is sent; that is, a default `summaryCB` is provided, but no `reviewCB` or `startCB`.

Note: If `reviewCB` or `summaryCB` do not return any labels, only a message will be pushed to the Gerrit server.

See also:

master/docs/examples/git_gerrit.cfg (https://github.com/buildbot/buildbot/blob/master/master/docs/examples/git_gerrit.cfg) and master/docs/examples/repo_gerrit.cfg (https://github.com/buildbot/buildbot/blob/master/master/docs/examples/repo_gerrit.cfg) in the Buildbot distribution provide a full example setup of Git+Gerrit or Repo+Gerrit of *GerritStatusPush*.

HttpStatusPush

```
from buildbot.plugins import reporters
sp = reporters.HttpStatusPush(serverUrl="http://example.com/submit")
c['services'].append(sp)
```

HttpStatusPush builds on `StatusPush` and sends HTTP requests to `serverUrl`, with all the items json-encoded. It is useful to create a status front end outside of Buildbot for better scalability.

It requires either txrequests (https://pypi.python.org/pypi/txrequests) or treq (https://pypi.python.org/pypi/treq) to be installed to allow interaction with http server.

Note: The json data object sent is completely different from the one that was generated by 0.8.x buildbot. It is indeed

generated using data api.

class **HttpStatusPush**(*serverUrl*, *user=None*, *password=None*, *auth=None*, *format_fn=None*, *builders=None*, *wantProperties=False*, *wantSteps=False*, *wantPreviousBuild=False*, *wantLogs=False*)

Parameters

- **serverUrl** (*string*) – the url where to do the http post
- **user** (*string*) – the BasicAuth user to post as
- **password** (*string*) – the BasicAuth user's password
- **auth** – the authentication method to use. Refer to the documentation of the requests library for more information.
- **format_fn** (*function*) – a function that takes the build as parameter and returns a dictionary to be pushed to the server (as json).
- **builders** (*list*) – only send update for specified builders
- **wantProperties** (*boolean*) – include 'properties' in the build dictionary
- **wantSteps** (*boolean*) – include 'steps' in the build dictionary
- **wantLogs** (*boolean*) – include 'logs' in the steps dictionaries. This needs wantSteps=True. This dumps the *full* content of logs and may consume lots of memory and CPU depending on the log size.
- **wantPreviousBuild** (*boolean*) – include 'prev_build' in the build dictionary

Json object spec

The default json object sent is a build object augmented with some more data as follow.

```
{
    "url": "http://yourbot/path/to/build",
    "<build data api values>": "[...]",
    "buildset": "<buildset data api values>",
    "builder": "<builder data api values>",
    "buildrequest": "<buildrequest data api values>"
}
```

If you want another format, don't hesitate to use the `format_fn` parameter to customize the payload. The `build` parameter given to that function is of type *build*, optionally enhanced with properties, steps, and logs information.

GitHubStatusPush

class `buildbot.reporters.github.`**GitHubStatusPush**

```
from buildbot.plugins import reporters, util

context = Interpolate("buildbot/%(prop:buildername)s")
gs = reporters.GitHubStatusPush(token='githubAPIToken',
                                context=context,
                                startDescription='Build started.',
                                endDescription='Build done.')
factory = util.BuildFactory()
```

```
buildbot_bbtools = util.BuilderConfig(
    name='builder-name',
    workernames=['worker1'],
    factory=factory)
c['builders'].append(buildbot_bbtools)
c['services'].append(gs)
```

GitHubStatusPush publishes a build status using GitHub Status API (http://developer.github.com/v3/repos/statuses).

It requires txrequests (https://pypi.python.org/pypi/txrequests) package to allow interaction with GitHub REST API.

It is configured with at least a GitHub API token.

You can create a token from you own GitHub - Profile - Applications - Register new application (https://github.com/settings/applications) or use an external tool to generate one.

class **GitHubStatusPush**(*token*, *startDescription=None*, *endDescription=None*, *context=None*, *baseURL=None*, *verbose=False*, *builders=None*)

Parameters

- **token** (*string*) – token used for authentication.
- **string startDescription** (*rendereable*) – Custom start message (default: 'Build started.')
- **string endDescription** (*rendereable*) – Custom end message (default: 'Build done.')
- **string context** (*rendereable*) – Passed to GitHub to differentiate between statuses. A static string can be passed or `Interpolate` for dynamic substitution. The default context is *buildbot/%(prop:buildername)s*.
- **baseURL** (*string*) – specify the github api endpoint if you work with GitHub Enterprise
- **verbose** (*boolean*) – if True, logs a message for each successful status push
- **builders** (*list*) – only send update for specified builders

GitHubCommentPush

class buildbot.reporters.github.**GitHubCommentPush**

```
from buildbot.plugins import reporters, util

gc = reporters.GitHubCommentPush(token='githubAPIToken',
                                 startDescription='Build started.',
                                 endDescription='Build done.')
factory = util.BuildFactory()
buildbot_bbtools = util.BuilderConfig(
    name='builder-name',
    workernames=['worker1'],
    factory=factory)
c['builders'].append(buildbot_bbtools)
c['services'].append(gc)
```

GitHubCommentPush publishes a comment on a PR using GitHub Review Comments API (https://developer.github.com/v3/pulls/comments/).

It requires txrequests (https://pypi.python.org/pypi/txrequests) package to allow interaction with GitHub REST API.

It is configured with at least a GitHub API token. By default, it will only comment at the end of a build unless a `startDescription` is provided.

You can create a token from you own GitHub - Profile - Applications - Register new application (https://github.com/settings/applications) or use an external tool to generate one.

class **GitHubCommentPush** (*token*, *startDescription=None*, *endDescription=None*, *baseURL=None*, *verbose=False*, *builders=None*)

Parameters

- **token** (*string*) – token used for authentication.
- **string startDescription** (*rendereable*) – Custom start message (default: None)
- **string endDescription** (*rendereable*) – Custom end message (default: 'Build done.')
- **baseURL** (*string*) – specify the github api endpoint if you work with GitHub Enterprise
- **verbose** (*boolean*) – if True, logs a message for each successful status push
- **builders** (*list*) – only send update for specified builders

StashStatusPush

class buildbot.reporters.stash.**StashStatusPush**

```
from buildbot.plugins import reporters

ss = reporters.StashStatusPush('https://stash.example.com:8080/',
                               'stash_username',
                               'secret_password')
c['services'].append(ss)
```

StashStatusPush publishes build status using Stash Build Integration REST API (https://developer.atlassian.com/static/rest/stash/3.6.0/stash-build-integration-rest.html). The build status is published to a specific commit SHA in Stash. It tracks the last build for each builderName for each commit built.

Specifically, it follows the Updating build status for commits (https://developer.atlassian.com/stash/docs/latest/how-tos/updating-build-status-for-commits.html) document.

It requires txrequests (https://pypi.python.org/pypi/txrequests) package to allow interaction with Stash REST API.

It uses HTTP Basic AUTH. As a result, we recommend you use https in your base_url rather than http.

class **StashStatusPush** (*base_url*, *user*, *password*, *key=None*, *statusName=None*, *startDescription=None*, *endDescription=None*, *verbose=False*, *builders=None*)

Parameters

- **base_url** (*string*) – The base url of the Stash host, up to and optionally including the first / of the path.
- **user** (*string*) – The Stash user to post as.
- **password** (*string*) – The Stash user's password.
- **string key** (*renderable*) – Passed to Stash to differentiate between statuses. A static string can be passed or `Interpolate` for dynamic substitution. The default key is *%(prop:buildername)s*.

- **string statusName** (*renderable*) – The name that is displayed for this status. The default name is nothing, so Stash will use the `key` parameter.
- **string startDescription** (*renderable*) – Custom start message (default: 'Build started.')
- **string endDescription** (*renderable*) – Custom end message (default: 'Build done.')
- **verbose** (*boolean*) – If True, logs a message for each successful status push.
- **builders** (*list*) – Only send update for specified builders.

BitbucketStatusPush

class buildbot.reporters.bitbucket.**BitbucketStatusPush**

```
from buildbot.plugins import reporters
bs = reporters.BitbucketStatusPush('oauth_key', 'oauth_secret')
c['services'].append(bs)
```

BitbucketStatusPush publishes build status using Bitbucket Build Status API (https://confluence.atlassian.com/bitbucket/buildstatus-resource-779295267.html). The build status is published to a specific commit SHA in Bitbucket. It tracks the last build for each builderName for each commit built.

It requires txrequests (https://pypi.python.org/pypi/txrequests) package to allow interaction with the Bitbucket REST and OAuth APIs.

It uses OAuth 2.x to authenticate with Bitbucket. To enable this, you need to go to your Bitbucket Settings -> OAuth page. Click "Add consumer". Give the new consumer a name, eg 'buildbot', and put in any URL as the callback (this is needed for Oauth 2.x but is not used by this reporter, eg 'http://localhost:8010/callback'). Give the consumer Repositories:Write access. After creating the consumer, you will then be able to see the OAuth key and secret.

class BitbucketStatusPush(*oauth_key*, *oauth_secret*, *base_url='https://api.bitbucket.org/2.0/repositories'*, *oauth_url='https://bitbucket.org/site/oauth2/access_token'*, *builders=None*)

Parameters

- **oauth_key** (*string*) – The OAuth consumer key
- **oauth_secret** (*string*) – The OAuth consumer secret
- **base_url** (*string*) – Bitbucket's Build Status API URL
- **oauth_url** (*string*) – Bitbucket's OAuth API URL
- **builders** (*list*) – only send update for specified builders

GitLabStatusPush

class buildbot.reporters.gitlab.**GitLabStatusPush**

```
from buildbot.plugins import reporters

gl = reporters.GitLabStatusPush('private-token', context='continuous-integration/
↪buildbot', baseUrl='https://git.yourcompany.com')
c['services'].append(gl)
```

GitLabStatusPush publishes build status using GitLab Commit Status API (http://doc.gitlab.com/ce/api/commits.html#commit-status). The build status is published to a specific commit SHA in GitLab.

It requires txrequests (https://pypi.python.org/pypi/txrequests) package to allow interaction with GitLab Commit Status API.

It uses private token auth, and the token owner is required to have at least developer access to each repository. As a result, we recommend you use https in your base_url rather than http.

class GitLabStatusPush(*token*, *startDescription=None*, *endDescription=None*, *context=None*, *baseURL=None*, *verbose=False*)

Parameters

- **token** (*string*) – Private token of user permitted to update status for commits
- **startDescription** (*string*) – Description used when build starts
- **endDescription** (*string*) – Description used when build ends
- **context** (*string*) – Name of your build system, eg. continuous-integration/buildbot
- **baseURL** (*string*) – the base url of the GitLab host, up to and optionally including the first / of the path. Do not include /api/
- **verbose** (*string*) – Be more verbose

HipchatStatusPush

class buildbot.reporters.hipchat.**HipchatStatusPush**

```
from buildbot.plugins import reporters

hs = reporters.HipchatStatusPush('private-token', endpoint='https://chat.yourcompany.
↪com')
c['services'].append(hs)
```

HipchatStatusPush publishes a custom message using Hipchat API v2 (https://www.hipchat.com/docs/apiv2). The message is published to a user and/or room in Hipchat,

It requires txrequests (https://pypi.python.org/pypi/txrequests) package to allow interaction with Hipchat API.

It uses API token auth, and the token owner is required to have at least message/notification access to each destination.

HipchatStatusPush(auth_token, endpoint="https://api.hipchat.com", builder_room_map=None, builder_user_map=None, wantProperties=False, wantSteps=False, wantPreviousBuild=False, wantLogs=False)

Parameters

- **auth_token** (*string*) – Private API token with access to the "Send Message" and "Send Notification" scopes.
- **endpoint** (*string*) – (optional) URL of your Hipchat server. Defaults to https://api.hipchat.com
- **builder_room_map** (*dictionary*) – (optional) If specified, will forward events about a builder (based on name) to the corresponding room ID.
- **builder_user_map** (*dictionary*) – (optional) If specified, will forward events about a builder (based on name) to the corresponding user ID.
- **wantProperties** (*boolean*) – (optional) include 'properties' in the build dictionary

- **wantSteps** (*boolean*) – (optional) include 'steps' in the build dictionary
- **wantLogs** (*boolean*) – (optional) include 'logs' in the steps dictionaries. This needs wantSteps=True. This dumps the *full* content of logs.
- **wantPreviousBuild** (*boolean*) – (optional) include 'prev_build' in the build dictionary

Note: No message will be sent if the message is empty or there is no destination found.

Note: If a builder name appears in both the room and user map, the same message will be sent to both destinations.

Json object spec

The default json object contains the minimal required parameters to send a message to Hipchat.

```
{
    "message": "Buildbot started/finished build MyBuilderName (with result success)␣
↪here: http://mybuildbot.com/#/builders/23",
    "id_or_email": "12"
}
```

If you require different parameters, the Hipchat reporter utilizes the template design pattern and will call *getRecipientList getMessage getExtraParams* before sending a message. This allows you to easily override the default implementation for those methods. All of those methods can be deferred.

Method signatures:

getRecipientList(*self*, *build*, *event_name*)

Parameters

- **build** – A `Build` object
- **event_name** (*string*) – the name of the event trigger for this invocation. either 'new' or 'finished'

Returns Deferred

The deferred should return a dictionary containing the key(s) 'id_or_email' for a private user message and/or 'room_id_or_name' for room notifications.

getMessage(*self*, *build*, *event_name*)

Parameters

- **build** – A `Build` object
- **event_name** (*string*) – the name of the event trigger for this invocation. either 'new' or 'finished'

Returns Deferred

The deferred should return a string to send to Hipchat.

getExtraParams(*self*, *build*, *event_name*)

Parameters

- **build** – A `Build` object

- **event_name** (*string*) – the name of the event trigger for this invocation. either 'new' or 'finished'

Returns Deferred

The deferred should return a dictionary containing any extra parameters you wish to include in your JSON POST request that the Hipchat API can consume.

Here's a complete example:

```
class MyHipchatStatusPush(HipChatStatusPush):
    name = "MyHipchatStatusPush"

    # send all messages to the same room
    def getRecipientList(self, build, event_name):
        return {
            'room_id_or_name': 'AllBuildNotifications'
        }

    # only send notifications on finished events
    def getMessage(self, build, event_name):
        event_messages = {
            'finished': 'Build finished.'
        }
        return event_messages.get(event_name, '')

    # color notifications based on the build result
    # and alert room on build failure
    def getExtraParams(self, build, event_name):
        result = {}
        if event_name == 'finished':
            result['color'] = 'green' if build['results'] == 0 else 'red'
            result['notify'] = (build['results'] != 0)
        return result
```

GerritVerifyStatusPush

class buildbot.status.status_gerrit_verify_status.**GerritVerifyStatusPush**

GerritVerifyStatusPush sends a verify status to Gerrit using the verify-status (https://gerrit.googlesource.com/plugins/verify-status) Gerrit plugin.

It is an alternate method to *GerritStatusPush*, which uses the SSH API to send reviews.

The verify-status (https://gerrit.googlesource.com/plugins/verify-status) plugin allows several CI statuses to be sent for the same change, and display them separately in the Gerrit UI.

Most parameters are renderables

```
GerritVerifyStatusPush(
baseURL, auth,
startDescription="Build started.", endDescription="Build done.",
verification_name=Interpolate("%(prop:buildername)s"), abstain=False, category=None, report
verbose=False, **kwargs)
```

Parameters

- **baseURL** (*string*) – Gerrit HTTP base URL

- **auth** (*string*) – a requests authentication configuration. if Gerrit is configured with `BasicAuth`, then it shall be `('login', 'password')` if Gerrit is configured with `DigestAuth`, then it shall be `requests.auth.HTTPDigestAuth('login', 'password')` from the requests module.
- **string startDescription** (*renderable*) – the comment sent when the build is starting.
- **string endDescription** (*renderable*) – the comment sent when the build is finishing.
- **string verification_name** (*renderable*) – the name of the job displayed in the Gerrit UI.
- **boolean abstain** (*renderable*) – whether this results should be counted as voting.
- **boolean category** (*renderable*) – Category of the build.
- **boolean reporter** (*renderable*) – The user that verified this build
- **verbose** (*boolean*) – Whether to log every requests.
- **builders** (*list*) – only send update for specified builders

This reporter is integrated with `GerritChangeSource`, and will update changes detected by this change source.

This reporter can also send reports for changes triggered manually provided that there is a property in the build named `gerrit_changes`, containing the list of changes that were tested. This property must be a list of dictionaries, containing `change_id` and `revision_id` keys, as defined in the revision endpoints of the Gerrit documentation (https://gerrit-review.googlesource.com/Documentation/rest-api-changes.html#revision-endpoints)

Web Server

Note: As of Buildbot 0.9.0, the built-in web server replaces the old `WebStatus` plugin.

Buildbot contains a built-in web server. This server is configured with the `www` configuration key, which specifies a dictionary with the following keys:

port The TCP port on which to serve requests. Note that SSL is not supported. To host Buildbot with SSL, use an HTTP proxy such as lighttpd, nginx, or Apache. If this is `None`, the default, then the master will not implement a web server.

json_cache_seconds The number of seconds into the future at which an HTTP API response should expire.

rest_minimum_version The minimum supported REST API version. Any versions less than this value will not be available. This can be used to ensure that no clients are depending on API versions that will soon be removed from Buildbot.

plugins This key gives a dictionary of additional UI plugins to load, along with configuration for those plugins. These plugins must be separately installed in the Python environment, e.g., `pip install buildbot-waterfall-view`. For example:

```
c['www'] = {
    'plugins': {'waterfall_view': {'num_builds': 50}}
}
```

debug If true, then debugging information will be output to the browser. This is best set to false (the default) on production systems, to avoid the possibility of information leakage.

allowed_origins This gives a list of origins which are allowed to access the Buildbot API (including control via JSONRPC 2.0). It implements cross-origin request sharing (CORS), allowing pages at origins other than the Buildbot UI to use the API. Each origin is interpreted as filename match expression, with `?` matching one character and `*` matching anything. Thus `['*']` will match all origins, and `['https://*.buildbot.net']` will match secure sites under `buildbot.net`. The Buildbot UI will operate correctly without this parameter; it is only useful for allowing access from other web applications.

auth Authentication module to use for the web server. See *Authentication plugins*.

avatar_methods List of methods that can be used to get avatar pictures to use for the web server. By default, buildbot uses Gravatar to get images associated with each users, if you want to disable this you can just specify empty list:

```
c['www'] = {
    'avatar_methods': []
}
```

For use of corporate pictures, you can use LdapUserInfo, which can also acts as an avatar provider. See *Authentication plugins*.

logfileName Filename used for http access logs, relative to the master directory. If set to `None` or the empty string, the content of the logs will land in the main `twisted.log` log file. (Default to `http.log`)

logRotateLength The amount of bytes after which the `http.log` file will be rotated. (Default to the same value as for the `twisted.log` file, set in `buildbot.tac`)

maxRotatedFiles The amount of log files that will be kept when rotating (Default to the same value as for the `twisted.log` file, set in `buildbot.tac`)

versions Custom component versions that you'd like to display on the About page. Buildbot will automatically prepend the versions of Python, twisted and buildbot itself to the list.

`versions` should be a list of tuples. for example:

```
c['www'] = {
    # ...
    'versions': [
        ('master.cfg', '0.1'),
        ('OS', 'Ubuntu 14.04'),
    ]
}
```

The first element of a tuple stands for the name of the component, the second stands for the corresponding version.

custom_templates_dir This directory will be parsed for custom angularJS templates to replace the one of the original website templates. if the directory string is relative, it will be joined to the master's basedir. Either `*.jade` files or `*.html` files can be used, and will be used to override `views/<filename>.html` templates in the angularjs templateCache. Unlike with the regular nodejs based angularjs build system, Python only jade interpreter is used to parse the jade templates. `pip install pyjade` is be required to use jade templates. You can also override plugin's directives, but they have to be in another directory.

```
# replace the template whose source is in:
# www/base/src/app/builders/build/build.tpl.jade
build.jade

# replace the template whose source is in
# www/console_view/src/module/view/builders-header/buildersheader.tpl.jade
console_view/buildersheader.html
```

Known differences between nodejs jade and pyjade:

- quotes in attributes are not quoted. https://github.com/syrusakbary/pyjade/issues/132 This means you should use double quotes for attributes e.g: `tr(ng-repeat="br in buildrequests | orderBy:'-submitted_at'")`

change_hook_dialects See *Change Hooks*.

`cookie_expiration_time`

This allows to define the timeout of the session cookie. Should be a datetime.timedelta (https://docs.python.org/2/library/datetime.html#timedelta-objects). Default is one week.

```
import datetime
c['www'] = {
    # ...
    'cookie_expiration_time': datetime.timedelta(weeks=2)
}
```

Note: The *buildbotURL* configuration value gives the base URL that all masters will use to generate links. The *www* configuration gives the settings for the webserver. In simple cases, the `buildbotURL` contains the hostname and port of the master, e.g., `http://master.example.com:8010/`. In more complex cases, with multiple masters, web proxies, or load balancers, the correspondence may be less obvious.

Authentication plugins

By default, Buildbot does not require people to authenticate in order to see the readonly data. In order to access control features in the web UI, you will need to configure an authentication plugin.

Note: As of Buildbot 0.9.4, user session is managed via a JWT (https://en.wikipedia.org/wiki/JSON_Web_Token) token, using HS256 (https://pyjwt.readthedocs.io/en/latest/algorithms.html) algorithm. The session secret is stored in the database in the `object_state` table with `name` column being `session_secret`. Please make sure appropriate access restriction is made to this database table.

Authentication plugins are implemented as classes, and passed as the `auth` parameter to *www*.

The available classes are described here:

class `buildbot.www.auth.`**NoAuth**
This class is the default authentication plugin, which disables authentication

class `buildbot.www.auth.`**UserPasswordAuth**(*users*)

Parameters **users** – list of `("user","password")` tuples, or a dictionary of `{"user": "password", ..}`

Simple username/password authentication using a list of user/password tuples provided in the configuration file.

```
from buildbot.plugins import util
c['www'] = {
    # ...
    'auth': util.UserPasswordAuth({"homer": "doh!"}),
}
```

class `buildbot.www.auth.`**HTPasswdAuth**(*passwdFile*)

Parameters **passwdFile** – An `.htpasswd` file to read

This class implements simple username/password authentication against a standard `.htpasswd` file.

```
from buildbot.plugins import util
c['www'] = {
    # ...
    'auth': util.HTPasswdAuth("my_htpasswd"),
}
```

class `buildbot.www.oauth2.`**GoogleAuth**(*clientId*, *clientSecret*)

Parameters

- **clientId** – The client ID of your buildbot application
- **clientSecret** – The client secret of your buildbot application

This class implements an authentication with Google (https://developers.google.com/accounts/docs/OAuth2) single sign-on. You can look at the Google (https://developers.google.com/accounts/docs/OAuth2) oauth2 documentation on how to register your Buildbot instance to the Google systems. The developer console will give you the two parameters you have to give to `GoogleAuth`

Register your Buildbot instance with the `BUILDBOT_URL/auth/login` url as the allowed redirect URI.

Example:

```
from buildbot.plugins import util
c['www'] = {
    # ...
    'auth': util.GoogleAuth("clientid", "clientsecret"),
}
```

in order to use this module, you need to install the Python `requests` module

```
pip install requests
```

class `buildbot.www.oauth2.`**GitHubAuth**(*clientId*, *clientSecret*)

Parameters

- **clientId** – The client ID of your buildbot application
- **clientSecret** – The client secret of your buildbot application

This class implements an authentication with GitHub (http://developer.github.com/v3/oauth_authorizations/) single sign-on. It functions almost identically to the *`GoogleAuth`* class.

Register your Buildbot instance with the `BUILDBOT_URL/auth/login` url as the allowed redirect URI.

The user's email-address (for e.g. authorization) is set to the "primary" address set by the user in GitHub. When using group-based authorization, the user's groups are equal to the names of the GitHub organizations the user is a member of.

Example:

```
from buildbot.plugins import util
c['www'] = {
    # ...
    'auth': util.GitHubAuth("clientid", "clientsecret"),
}
```

class `buildbot.www.oauth2.`**GitLabAuth**(*instanceUri*, *clientId*, *clientSecret*)

Parameters

- **instanceUri** – The URI of your GitLab instance
- **clientId** – The client ID of your buildbot application
- **clientSecret** – The client secret of your buildbot application

This class implements an authentication with GitLab (http://doc.gitlab.com/ce/integration/oauth_provider.html) single sign-on. It functions almost identically to the *GoogleAuth* class.

Register your Buildbot instance with the `BUILDBOT_URL/auth/login` url as the allowed redirect URI.

Example:

```
from buildbot.plugins import util
c['www'] = {
    # ...
    'auth': util.GitLabAuth("https://gitlab.com", "clientid", "clientsecret"),
}
```

class `buildbot.www.oauth2.`**BitbucketAuth**(*clientId*, *clientSecret*)

Parameters

- **clientId** – The client ID of your buildbot application
- **clientSecret** – The client secret of your buildbot application

This class implements an authentication with Bitbucket (https://confluence.atlassian.com/bitbucket/oauth-on-bitbucket-cloud-238027431.html) single sign-on. It functions almost identically to the *GoogleAuth* class.

Register your Buildbot instance with the `BUILDBOT_URL/auth/login` url as the allowed redirect URI.

Example:

```
from buildbot.plugins import util
c['www'] = {
    # ...
    'auth': util.BitbucketAuth("clientid", "clientsecret"),
}
```

class `buildbot.www.auth.`**RemoteUserAuth**

Parameters

- **header** – header to use to get the username (defaults to `REMOTE_USER`)
- **headerRegex** – regular expression to get the username from header value (defaults to `"(?P<username>[^ @]+)@(?P<realm>[^ @]+)"`). Note that your at least need to specify a `?P<username>` regular expression named group.
- **userInfoProvider** – user info provider; see *User Information*

If the Buildbot UI is served through a reverse proxy that supports HTTP-based authentication (like apache or lighttpd), it's possible to to tell Buildbot to trust the web server and get the username from th request headers.

Administrator must make sure that it's impossible to get access to Buildbot using other way than through frontend. Usually this means that Buildbot should listen for incoming connections only on localhost (or on some firewall-protected port). The reverse proxy must require HTTP authentication to access Buildbot pages (using any source for credentials, such as htpasswd, PAM, LDAP, Kerberos).

Example:

```
from buildbot.plugins import util
c['www'] = {
    # ...
    'auth': util.RemoteUserAuth(),
}
```

A corresponding Apache configuration example

```
<Location "/">
        AuthType Kerberos
        AuthName "Buildbot login via Kerberos"
        KrbMethodNegotiate On
        KrbMethodK5Passwd On
        KrbAuthRealms <<YOUR CORP REALMS>>
        KrbVerifyKDC off
        KrbServiceName Any
        Krb5KeyTab /etc/krb5/krb5.keytab
        KrbSaveCredentials Off
        require valid-user
        Order allow,deny

        Satisfy Any

        #] SSO
        RewriteEngine On
        RewriteCond %{LA-U:REMOTE_USER} (.+)$
        RewriteRule . - [E=RU:%1,NS]
        RequestHeader set REMOTE_USER %{RU}e

</Location>
```

The advantage of this sort of authentication is that it is uses a proven and fast implementation for authentication. The problem is that the only information that is passed to Buildbot is the username, and there is no way to pass any other information like user email, user groups, etc. That information can be very useful to the mailstatus plugin, or for authorization processes. See *User Information* for a mechanism to supply that information.

User Information

For authentication mechanisms which cannot provide complete information about a user, Buildbot needs another way to get user data. This is useful both for authentication (to fetch more data about the logged-in user) and for avatars (to fetch data about other users).

This extra information is provided by, appropriately enough, user info providers. These can be passed to `RemoteUserAuth` and as an element of `avatar_methods`.

This can also be passed to oauth2 authentication plugins. In this case the username provided by oauth2 will be used, and all other informations will be taken from ldap (Full Name, email, and groups):

Currently only one provider is available:

class `buildbot.ldapuserinfo.`**`LdapUserInfo`**(*uri*, *bindUser*, *bindPw*, *accountBase*, *accountPattern*, *groupBase=None*, *groupMemberPattern=None*, *groupName=None*, *accountFullName*, *accountEmail*, *avatarPattern=None*, *avatarData=None*, *accountExtraFields=None*)

Parameters

- **uri** – uri of the ldap server
- **bindUser** – username of the ldap account that is used to get the infos for other users (usually a "faceless" account)
- **bindPw** – password of the `bindUser`
- **accountBase** – the base dn (distinguished name)of the user database
- **accountPattern** – the pattern for searching in the account database. This must contain the `%(username)s` string, which is replaced by the searched username
- **accountFullName** – the name of the field in account ldap database where the full user name is to be found.
- **accountEmail** – the name of the field in account ldap database where the user email is to be found.
- **groupBase** – the base dn of the groups database.
- **groupMemberPattern** – the pattern for searching in the group database. This must contain the `%(dn)s` string, which is replaced by the searched username's dn
- **groupName** – the name of the field in groups ldap database where the group name is to be found.
- **avatarPattern** – the pattern for searching avatars from emails in the account database. This must contain the `%(email)s` string, which is replaced by the searched email
- **avatarData** – the name of the field in groups ldap database where the avatar picture is to be found. This field is supposed to contain the raw picture, format is automatically detected from jpeg, png or git.
- **accountExtraFields** – extra fields to extracts for use with the authorization policies.

If one of the three optional groups parameters is supplied, then all of them become mandatory. If none is supplied, the retrieved user info has an empty list of groups.

Example:

```
from buildbot.plugins import util

# this configuration works for MS Active Directory ldap implementation
# we use it for user info, and avatars
userInfoProvider = util.LdapUserInfo(
    uri='ldap://ldap.mycompany.com:3268',
    bindUser='ldap_user',
    bindPw='p4$$wd',
    accountBase='dc=corp,dc=mycompany,dc=com',
    groupBase='dc=corp,dc=mycompany,dc=com',
    accountPattern='(&(objectClass=person)(sAMAccountName=%(username)s))',
    accountFullName='displayName',
    accountEmail='mail',
    groupMemberPattern='(&(objectClass=group)(member=%(dn)s))',
    groupName='cn',
    avatarPattern='(&(objectClass=person)(mail=%(email)s))',
    avatarData='thumbnailPhoto',
)
c['www'] = dict(port=PORT, allowed_origins=["*"],
                url=c['buildbotURL'],
                auth=util.RemoteUserAuth(userInfoProvider=userInfoProvider),
                avatar_methods=[userInfoProvider,
                                util.AvatarGravatar()])
```

Note: In order to use this module, you need to install the `python3-ldap` module:

```
pip install python3-ldap
```

In the case of oauth2 authentications, you have to pass the userInfoProvider as keyword argument:

```
from buildbot.plugins import util
userInfoProvider = util.LdapUserInfo(...)
c['www'] = {
    # ...
    'auth': util.GoogleAuth("clientid", "clientsecret",␣
↪userInfoProvider=userInfoProvider),
}
```

Reverse Proxy Configuration

It is usually better to put buildbot behind a reverse proxy in production.

- Provides automatic gzip compression
- Provides SSL support with a widely used implementation
- Provides support for http/2 or spdy for fast parallel REST api access from the browser

Reverse proxy however might be problematic for websocket, you have to configure it specifically to pass web socket requests. Here is an nginx configuration that is known to work (nginx 1.6.2):

```
server {
        # Enable SSL and http2
        listen 443 ssl http2 default_server;

        server_name yourdomain.com;

        root html;
        index index.html index.htm;

        ssl on;
        ssl_certificate /etc/nginx/ssl/server.cer;
        ssl_certificate_key /etc/nginx/ssl/server.key;

        # put a one day session timeout for websockets to stay longer
        ssl_session_cache      shared:SSL:1440m;
        ssl_session_timeout  1440m;

        # please consult latest nginx documentation for current secure encryption␣
↪settings
        ssl_protocols ..
        ssl_ciphers ..
        ssl_prefer_server_ciphers   on;
        #

        # force https
        add_header Strict-Transport-Security "max-age=31536000; includeSubdomains;";
        spdy_headers_comp 5;

        # you could use / if you use domain based proxy instead of path based proxy
```

```
        location /buildbot/ {
            proxy_pass http://127.0.0.1:5000/;
        }
        location /buildbot/sse/ {
            # proxy buffering will prevent sse to work
            proxy_buffering off;
            proxy_pass http://127.0.0.1:5000/sse/;
        }
        # required for websocket
        location /buildbot/ws {
              proxy_http_version 1.1;
              proxy_set_header Upgrade $http_upgrade;
              proxy_set_header Connection "upgrade";
              proxy_pass http://127.0.0.1:5000/ws;
              # raise the proxy timeout for the websocket
              proxy_read_timeout 6000s;
        }
}
```

To run with Apache2, you'll need mod_proxy_wstunnel (https://httpd.apache.org/docs/2.4/mod/mod_proxy_wstunnel.html) in addition to mod_proxy_http (https://httpd.apache.org/docs/2.4/mod/mod_proxy_http.html). Serving HTTPS (mod_ssl (https://httpd.apache.org/docs/2.4/mod/mod_ssl.html)) is advised to prevent issues with enterprise proxies (see *Server Sent Events*), even if you don't need the encryption itself.

Here is a configuration that is known to work (Apache 2.4.10 / Debian 8), directly at the top of the domain.

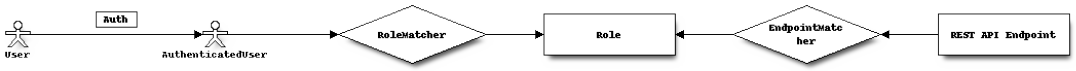
```
<VirtualHost *:443>
    ServerName buildbot.example
    ServerAdmin webmaster@buildbot.example

    <Location /ws>
      ProxyPass ws://127.0.0.1:8020/ws
      ProxyPassReverse ws://127.0.0.1:8020/ws
    </Location>

    ProxyPass /ws !
    ProxyPass / http://127.0.0.1:8020/
    ProxyPassReverse / http://127.0.0.1:8020/

    SetEnvIf X-Url-Scheme https HTTPS=1
    ProxyPreserveHost On

    SSLEngine on
    SSLCertificateFile /path/to/cert.pem
    SSLCertificateKeyFile /path/to/cert.key

    # check Apache2 documentation for current safe SSL settings
    # This is actually the Debian 8 default at the time of this writing:
    SSLProtocol all -SSLv3

</VirtualHost>
```

Authorization rules

The authorization framework in Buildbot is very generic and flexible. Drawback is that it is not very obvious for newcomers. The 'simple' example will however allow you to easily start by implementing an admins-have-all-rights setup.

Please carefully read the following documentation to understand how to setup authorization in Buildbot.

Authorization framework is tightly coupled to the REST API. Authorization framework only works for HTTP, not for other means of interaction like IRC or try scheduler. It allows or denies access to the REST APIs according to rules.



- Roles is a label that you give to a user.

  It is similar but different to the usual notion of group:

  - A user can have several roles, and a role can be given to several users.
  - Role is an application specific notion, while group is more organization specific notion.
  - Groups are given by the auth plugin, e.g `ldap`, `github`, and are not always in the precise control of the buildbot admins.
  - Roles can be dynamically assigned, according to the context. For example, there is the `owner` role, which can be given to a user for a build that he is at the origin, so that he can stop or rebuild only builds of his own.
- Endpoint matchers associate role requirements to REST API endpoints. The default policy is allow in case no matcher matches (see below why)
- Role matchers associate authenticated users to roles.

Authz Configuration

class `buildbot.www.authz.`**Authz**(*allowRules=[]*, *roleMatcher=[]*, *stringsMatcher=util.fnmatchStrMatcher*)

Parameters

- **allowRules** – List of `EndpointMatcherBase` processed in order for each endpoint grant request.
- **roleMatcher** – List of RoleMatchers
- **stringsMatcher** – Selects algorithm used to make strings comparison (used to compare roles and builder names). can be `util.fnmatchStrMatcher` or `util.reStrMatcher` from `from buildbot.plugins import util`

`Authz` needs to be configured in `c['www']['authz']`

Endpoint matchers

Endpoint matchers are responsible for creating rules to match REST endpoints, and requiring roles for them. Endpoint matchers are processed in the order they are configured. The first rule matching an endpoint will prevent further rules from being checked. To continue checking other rules when the result is *deny*, set *defaultDeny=False*. If no endpoint matcher matches, then the access is granted.

One can implement the default deny policy by putting an `AnyEndpointMatcher` with nonexistent role in the end of the list. Please note that this will deny all REST apis, and most of the UI do not implement proper access denied message in case of such error.

The following sequence is implemented by each EndpointMatcher class.

- Check whether the requested endpoint is supported by this matcher
- Get necessary info from data api, and decides whether it matches.
- Look if the users has the required role.

Several endpoints matchers are currently implemented. If you need a very complex setup, you may need to implement your own endpoint matchers. In this case, you can look at the source code for detailed examples on how to write endpoint matchers.

class buildbot.www.authz.endpointmatchers.**EndpointMatcherBase**(*role*, *defaultDeny=True*)

Parameters

- **role** – The role which grants access to this endpoint. List of roles is not supported, but a `fnmatch` expression can be provided to match several roles.
- **defaultDeny** – The role matcher algorithm will stop if this value is true, and if the endpoint matched.

This is the base endpoint matcher. Its arguments are inherited by all the other endpoint matchers.

class buildbot.www.authz.endpointmatchers.**AnyEndpointMatcher**(*role*)

Parameters **role** – The role which grants access to any endpoint.

AnyEndpointMatcher grants all rights to people with given role (usually "admins")

class buildbot.www.authz.endpointmatchers.**AnyControlEndpointMatcher**(*role*)

Parameters **role** – The role which grants access to any control endpoint.

AnyControlEndpointMatcher grants control rights to people with given role (usually "admins") This endpoint matcher is matches current and future control endpoints. You need to add this in the end of your configuration to make sure it is future proof.

class buildbot.www.authz.endpointmatchers.**ForceBuildEndpointMatcher**(*builder*, *role*)

Parameters

- **builder** – name of the builder.
- **role** – The role needed to get access to such endpoints.

ForceBuildEndpointMatcher grants right to force builds.

class buildbot.www.authz.endpointmatchers.**StopBuildEndpointMatcher**(*builder*, *role*)

Parameters

- **builder** – name of the builder.
- **role** – The role needed to get access to such endpoints.

StopBuildEndpointMatcher grants rights to stop builds.

class buildbot.www.authz.endpointmatchers.**RebuildBuildEndpointMatcher**(*builder*, *role*)

Parameters

- **builder** – name of the builder.
- **role** – The role needed to get access to such endpoints.

RebuildBuildEndpointMatcher grants rights to rebuild builds.

class buildbot.www.authz.endpointmatchers.**EnableSchedulerEndpointMatcher**(*builder*, *role*)

Parameters

- **builder** – name of the builder.
- **role** – The role needed to get access to such endpoints.

EnableSchedulerEndpointMatcher grants rights to enable and disable schedulers via the UI.

Role matchers

Endpoint matchers are responsible for creating rules to match people and grant them roles. You can grant roles from groups information provided by the Auth plugins, or if you prefer directly to people's email.

class buildbot.www.authz.roles.**RolesFromGroups**(*groupPrefix*)

Parameters **groupPrefix** – prefix to remove from each group

RolesFromGroups grants roles from the groups of the user. If a user has group `buildbot-admin`, and groupPrefix is `buildbot-`, then user will be granted the role 'admin'

ex:

```
roleMatchers=[
  util.RolesFromGroups(groupPrefix="buildbot-")
]
```

class buildbot.www.authz.roles.**RolesFromEmails**(*roledict*)

Parameters **roledict** – dictionary with key=role, and value=list of email strings

RolesFromEmails grants roles to users according to the hardcoded emails.

ex:

```
roleMatchers=[
  util.RolesFromEmails(admins=["my@email.com"])
]
```

class buildbot.www.authz.roles.**RolesFromOwner**(*roledict*)

Parameters **roledict** – dictionary with key=role, and value=list of email strings

RolesFromOwner grants a given role when property owner matches the email of the user

ex:

```
roleMatchers=[
    RolesFromOwner(role="owner")
]
```

class buildbot.www.authz.roles.**RolesFromUsername**(*roles*, *usernames*)

Parameters

- **roles** – roles to assign when the username matches.
- **usernames** – list of usernames that have the roles.

RolesFromUsername grants the given roles when the `username` property is within the list of usernames.

ex:

```
roleMatchers=[
    RolesFromUsername(roles=["admins"], usernames=["root"]),
    RolesFromUsername(roles=["developers", "integrators"], usernames=["Alice",
↪"Bob"])
]
```

Example Configs

Simple config which allows admin people to control everything, but allow anonymous to look at build results:

```
from buildbot.plugins import *
authz = util.Authz(
  allowRules=[
    util.AnyControlEndpointMatcher(role="admins"),
  ],
  roleMatchers=[
    util.RolesFromEmails(admins=["my@email.com"])
  ]
)
auth=util.UserPasswordAuth({'my@email.com': 'mypass'})
c['www']['auth'] = auth
c['www']['authz'] = authz
```

More complex config with separation per branch:

```
from buildbot.plugins import *

authz = util.Authz(
    stringsMatcher=util.fnmatchStrMatcher,  # simple matcher with '*' glob character
    # stringsMatcher = util.reStrMatcher,   # if you prefer regular expressions
    allowRules=[
        # admins can do anything,
        # defaultDeny=False: if user does not have the admin role, we continue␣
↪parsing rules
        util.AnyEndpointMatcher(role="admins", defaultDeny=False),

        StopBuildEndpointMatcher(role="owner"),

        # *-try groups can start "try" builds
        util.ForceBuildEndpointMatcher(builder="try", role="*-try"),
        # *-mergers groups can start "merge" builds
        util.ForceBuildEndpointMatcher(builder="merge", role="*-mergers"),
        # *-releasers groups can start "release" builds
        util.ForceBuildEndpointMatcher(builder="release", role="*-releasers"),
        # if future Buildbot implement new control, we are safe with this last rule
        util.AnyControlEndpointMatcher(role="admins")
    ],
    roleMatchers=[
        RolesFromGroups(groupPrefix="buildbot-"),
        RolesFromEmails(admins=["homer@springfieldplant.com"],
                        reaper-try=["007@mi6.uk"]),
        # role owner is granted when property owner matches the email of the user
        RolesFromOwner(role="owner")
    ]
)
c['www']['authz'] = authz
```

Using GitHub authentication and allowing access to control endpoints for users in the "BuildBot" organization:

```
from buildbot.plugins import *
authz = util.Authz(
  allowRules=[
    util.AnyControlEndpointMatcher(role="BuildBot")
  ],
  roleMatchers=[
    util.RolesFromGroups()
  ]
)
auth=util.GitHubAuth('CLIENT_ID', 'CLIENT_SECRET')
c['www']['auth'] = auth
c['www']['authz'] = authz
```

Change Hooks

The `/change_hook` url is a magic URL which will accept HTTP requests and translate them into changes for buildbot. Implementations (such as a trivial json-based endpoint and a GitHub implementation) can be found in master/buildbot/www/hooks (https://github.com/buildbot/buildbot/blob/master/master/buildbot/www/hooks). The format of the url is `/change_hook/DIALECT` where DIALECT is a package within the hooks directory. Change_hook is disabled by default and each DIALECT has to be enabled separately, for security reasons

An example www configuration line which enables change_hook and two DIALECTS:

```
c['www'] = dict(
    change_hook_dialects={
                            'base': True,
                            'somehook': {'option1':True,
                                         'option2':False}}))
```

Within the www config dictionary arguments, the `change_hook` key enables/disables the module and `change_hook_dialects` whitelists DIALECTs where the keys are the module names and the values are optional arguments which will be passed to the hooks.

The master/contrib/post_build_request.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/post_build_request.p script allows for the submission of an arbitrary change request. Run **`post_build_request.py --help`** for more information. The `base` dialect must be enabled for this to work.

Change Hooks Auth

By default change hook URL is not protected. Some hooks implement their own authentication method. Other requires the generic method to be secured.

To protect URL against unauthorized access you you may use `change_hook_auth` option.

Note: This method uses `HTTP BasicAuth`, it implies the use of SSL via *Reverse Proxy Configuration* in order to be fully secured.

```
from twisted.cred import strcred
c['www'] = dict(...,
      change_hook_auth=[strcred.makeChecker("file:changehook.passwd")]))
```

create a file `changehook.passwd`:

```
user:password
```

- `change_hook_auth` should be a list of `ICredentialsChecker`

See the details of available options in Twisted documentation (https://twistedmatrix.com/documents/current/core/howto/cred.html)

Mercurial hook

The Mercurial hook uses the base dialect:

```
c['www'] = dict(
    ...,
    change_hook_dialects={'base': True},
)
```

Once this is configured on your buildmaster add the following hook on your server-side Mercurial repository's `hgrc`:

```
[hooks]
changegroup.buildbot = python:/path/to/hgbuildbot.py:hook
```

You'll find `hgbuildbot.py`, and its inline documentation, in the `contrib` directory of Buildbot's repository.

GitHub hook

Note: There is a standalone HTTP server available for receiving GitHub notifications as well: master/contrib/github_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/github_buildbot.py). This script may be useful in cases where you cannot expose the WebStatus for public consumption.

The GitHub hook has the following parameters:

`secret` (default *None*) Secret token to use to validate payloads

`strict` (default *False*) If the hook must be strict regarding valid payloads. If the value is *False* (default), the signature will only be checked if a secret is specified and a signature was supplied with the payload. If the value is *True*, a secret must be provided, and payloads without signature will be ignored.

`codebase` (default *None*) The codebase value to include with created changes. If the value is a function (or any other callable), it will be called with the GitHub event payload as argument and the function must return the codebase value to use for the event.

`class` (default *None*) A class to be used for processing incoming payloads. If the value is *None* (default), the default class – `buildbot.status.web.hooks.github.GitHubEventHandler` – will be used. The default class handles *ping*, *push* and *pull_request* events only. If you'd like to handle other events (see Event Types & Payloads (https://developer.github.com/v3/activity/events/types/) for more information), you'd need to subclass *GitHubEventHandler* and add handler methods for the corresponding events. For example, if you'd like to handle *blah* events, your code should look something like this:

```
from buildbot.status.web.hooks.github import GitHubEventHandler

class MyBlahHandler(GitHubEventHandler):

    def handle_blah(self, payload):
```

```
        # Do some magic here
        return [], 'git'
```

The simples way to use GitHub hook is as follows:

```
c['www'] = dict(...,
    change_hook_dialects={'github': { }})
```

Having added this line, you should add a webhook for your GitHub project (see Creating Webhooks page at GitHub (https://developer.github.com/webhooks/creating/)). The parameters are:

Payload URL This URL should point to `/change_hook/github` relative to the root of the web status. For example, if the base URL is `http://builds.example.com/buildbot`, then point GitHub to `http://builds.example.com/buildbot/change_hook/github`. To specify a project associated to the repository, append `?project=name` to the URL.

Content Type Specify `application/x-www-form-urlencoded` or `application/json`.

Secret Any value. If you provide a non-empty value (recommended), make sure that your hook is configured to use it:

```
c['www'] = dict(
    ...,
    change_hook_dialects={
        'github': {
            'secret': 'MY-SECRET',
            'strict': True
        }
    },
    ...))
```

Which events would you like to trigger this webhook? Leave the default – `Just the push event` – other kind of events are not currently supported.

And then press the `Add Webhook` button.

Warning: The incoming HTTP requests for this hook are not authenticated by default. Anyone who can access the web server can "fake" a request from GitHub, potentially causing the buildmaster to run arbitrary code.

To protect URL against unauthorized access you should use *Change Hooks Auth* option. Then change the the `Payload URL` of your GitHub webhook to `https://user:password@builds.example.com/bbot/change_hook/github`.

Patches are welcome to implement: https://developer.github.com/webhooks/securing/

Note: When using a *ChangeFilter* with a GitHub webhook ensure that your filter matches all desired requests as fields such as `repository` and `project` may differ in different events.

BitBucket hook

The BitBucket hook is as simple as GitHub one and it also takes no options.

```
c['www'] = dict(...,
    change_hook_dialects={ 'bitbucket' : True }))
```

When this is setup you should add a *POST* service pointing to `/change_hook/bitbucket` relative to the root of the web status. For example, it the grid URL is `http://builds.example.com/bbot/grid`, then point BitBucket to `http://builds.example.com/change_hook/bitbucket`. To specify a project associated to the repository, append `?project=name` to the URL.

Note that there is a standalone HTTP server available for receiving BitBucket notifications, as well: master/contrib/bitbucket_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/bitbucket_buildbot.py). This script may be useful in cases where you cannot expose the WebStatus for public consumption.

Warning: As in the previous case, the incoming HTTP requests for this hook are not authenticated by default. Anyone who can access the web status can "fake" a request from BitBucket, potentially causing the buildmaster to run arbitrary code.

To protect URL against unauthorized access you should use *Change Hooks Auth* option. Then, create a BitBucket service hook (see https://confluence.atlassian.com/display/BITBUCKET/POST+Service+Management) with a WebHook URL like `https://user:password@builds.example.com/bbot/change_hook/bitbucket`.

Note that as before, not using `change_hook_auth` can expose you to security risks.

Poller hook

The poller hook allows you to use GET or POST requests to trigger polling. One advantage of this is your buildbot instance can poll at launch (using the pollAtLaunch flag) to get changes that happened while it was down, but then you can still use a commit hook to get fast notification of new changes.

Suppose you have a poller configured like this:

```
c['change_source'] = SVNPoller(
    repourl="https://amanda.svn.sourceforge.net/svnroot/amanda/amanda",
    split_file=split_file_branches,
    pollInterval=24*60*60,
    pollAtLaunch=True)
```

And you configure your WebStatus to enable this hook:

```
c['www'] = dict(...,
    change_hook_dialects={'poller': True}
)
```

Then you will be able to trigger a poll of the SVN repository by poking the `/change_hook/poller` URL from a commit hook like this:

```
curl -s -F poller=https://amanda.svn.sourceforge.net/svnroot/amanda/amanda \
    http://yourbuildbot/change_hook/poller
```

If no `poller` argument is provided then the hook will trigger polling of all polling change sources.

You can restrict which pollers the webhook has access to using the `allowed` option:

```
c['www'] = dict(...,
    change_hook_dialects={'poller': {'allowed': ['https://amanda.svn.sourceforge.net/
↪svnroot/amanda/amanda']}}
)
```

GitLab hook

The GitLab hook is as simple as GitHub one and it also takes no options.

```
c['www'] = dict(...,
    change_hook_dialects={ 'gitlab' : True }
)
```

When this is setup you should add a *POST* service pointing to `/change_hook/gitlab` relative to the root of the web status. For example, it the grid URL is `http://builds.example.com/bbot/grid`, then point GitLab to `http://builds.example.com/change_hook/gitlab`. The project and/or codebase can also be passed in the URL by appending `?project=name` or `?codebase=foo` to the URL. These parameters will be passed along to the scheduler.

Note: Your Git step must be configured with a git@ repourl, not a https: one, else the change from the webhook will not trigger a build.

Warning: As in the previous case, the incoming HTTP requests for this hook are not authenticated by default. Anyone who can access the web status can "fake" a request from your GitLab server, potentially causing the buildmaster to run arbitrary code.

To protect URL against unauthorized access you should use *Change Hooks Auth* option. Then, create a GitLab service hook (see `https://your.gitlab.server/help/web_hooks`) with a WebHook URL like `https://user:password@builds.example.com/bbot/change_hook/gitlab`.

Note that as before, not using `change_hook_auth` can expose you to security risks.

Gitorious Hook

The Gitorious hook is as simple as GitHub one and it also takes no options.

```
c['www'] = dict(...,
    change_hook_dialects={'gitorious': True}
)
```

When this is setup you should add a *POST* service pointing to `/change_hook/gitorious` relative to the root of the web status. For example, it the grid URL is `http://builds.example.com/bbot/grid`, then point Gitorious to `http://builds.example.com/change_hook/gitorious`.

Warning: As in the previous case, the incoming HTTP requests for this hook are not authenticated by default. Anyone who can access the web status can "fake" a request from your Gitorious server, potentially causing the buildmaster to run arbitrary code.

To protect URL against unauthorized access you should use *Change Hooks Auth* option. Then, create a Gitorious web hook with a WebHook URL like `https://user:password@builds.example.com/bbot/change_hook/gitorious`.

Note that as before, not using `change_hook_auth` can expose you to security risks.

Note: Web hooks are only available for local Gitorious installations, since this feature is not offered as part of Gitorious.org yet.

Custom Services

For advanced users or plugins writers, the 'services' key is available, and holds a list of *buildbot.util.service.BuildbotService*. As this feature for advanced users, it is described in the developer section of the manual.

This section will grow as soon as ready-to-use services are created.

DbConfig

DbConfig is an utility for master.cfg to get easy to use key/value storage in the buildbot database

DbConfig can get and store any jsonable object to the db for use by other masters or separate ui plugins to edit them.

The design is voluntary simplistic, the focus is on the easy use rather than efficiency. A separate db connection is created each time get() or set() is called.

Example:

```
from buildbot.plugins import util, worker
c = BuildmasterConfig = {}
c['db_url'] = 'mysql://user@pass:mysqlserver/buildbot'
dbConfig = util.DbConfig(BuildmasterConfig, basedir)
workers = dbConfig.get("workers")
c['workers'] = [
    worker.Worker(worker['name'], worker['passwd'],
                  properties=worker.get('properties')),
    for worker in workers
]
```

class DbConfig

__init__(*BuildmasterConfig*, *basedir*)

Parameters

- **BuildmasterConfig** – the BuildmasterConfig, where db_url is already configured
- **basedir** – basedir global variable of the master.cfg run environment. Sqlite urls are relative to this dir.

get(*name*, *default=MarkerClass*)

Parameters

- **name** – the name of the config variable to retrieve
- **default** – In case the config variable has not been set yet, default is returned if defined, else KeyError is raised.

set(*name*, *value*)

Parameters

- **name** – the name of the config variable to be set

- **value** – the value of the config variable to be set

Transition to "worker" terminology

Since version 0.9.0 of Buildbot "slave"-based terminology is deprecated in favor of "worker"-based terminology.

API change is done in backward compatible way, so old "slave"-containing classes, functions and attributes are still available and can be used. Old API support will be removed in the future versions of Buildbot.

Rename of API introduced in beta versions of Buildbot 0.9.0 done without providing fallback. See release notes for the list of breaking changes of private interfaces.

Old names fallback settings

Use of obsolete names will raise Python warnings with category `buildbot.worker_transition.DeprecatedWorkerAPIWarning`. By default these warnings are printed in the application log. This behaviour can be changed by setting appropriate Python warnings settings via Python's `warnings` module:

```
import warnings
from buildbot.worker_transition import DeprecatedWorkerAPIWarning
# Treat old-name usage as errors:
warnings.simplefilter("error", DeprecatedWorkerAPIWarning)
```

See Python's `warnings` module documentation for complete list of available actions, in particular warnings can be disabled using `"ignore"` action.

It's recommended to configure warnings inside `buildbot.tac`, before using any other Buildbot classes.

Changed API

In general "Slave" and "Buildslave" parts in identifiers and messages were replaced with "Worker"; "SlaveBuilder" with "WorkerForBuilder".

Below is the list of changed API (use of old names from this list will work). Note that some of these symbols are not included in Buildbot's public API. Compatibility is provided as a convenience to those using the private symbols anyway.

Old name	New nam
`buildbot.interfaces.IBuildSlave`	`IWorke`
`buildbot.interfaces.NoSlaveError` (private)	left as is,
`buildbot.interfaces.BuildSlaveTooOldError`	`Worker`
`buildbot.interfaces.LatentBuildSlaveFailedToSubstantiate` (private)	`LatentW`
`buildbot.interfaces.ILatentBuildSlave`	`ILatent`
`buildbot.interfaces.ISlaveStatus` (will be removed in 0.9.x)	`IWorke`
`buildbot.buildslave` module with all contents	*buildbo*
`buildbot.buildslave.AbstractBuildSlave`	buildbo
`buildbot.buildslave.AbstractBuildSlave.slavename` (private)	buildbo
`buildbot.buildslave.AbstractLatentBuildSlave`	buildbo
`buildbot.buildslave.BuildSlave`	*buildbo*
`buildbot.buildslave.ec2`	buildbo
`buildbot.buildslave.ec2.EC2LatentBuildSlave`	*buildbo*

Table 2.2 – continued from

Old name	New nam
buildbot.buildslave.libvirt	buildbo
buildbot.buildslave.libvirt.LibVirtSlave	*buildbo*
buildbot.buildslave.openstack	buildbo
buildbot.buildslave.openstack.OpenStackLatentBuildSlave	*buildbo*
buildbot.config.MasterConfig.slaves	*worker*
buildbot.config.BuilderConfig constructor keyword argument slavename was renamed to	workern
buildbot.config.BuilderConfig constructor keyword argument slavenames was renamed to	workern
buildbot.config.BuilderConfig constructor keyword argument slavebuilddir was renamed to	workerb
buildbot.config.BuilderConfig constructor keyword argument nextSlave was renamed to	nextWo
buildbot.config.BuilderConfig.slavenames	*workern*
buildbot.config.BuilderConfig.slavebuilddir	*workerb*
buildbot.config.BuilderConfig.nextSlave	*nextWo*
buildbot.process.slavebuilder	buildbo
buildbot.process.slavebuilder.AbstractSlaveBuilder	buildbo
buildbot.process.slavebuilder.AbstractSlaveBuilder.slave	buildbo
buildbot.process.slavebuilder.SlaveBuilder	buildbo
buildbot.process.slavebuilder.LatentSlaveBuilder	buildbo
buildbot.process.build.Build.getSlaveName	getWorl
buildbot.process.build.Build.slavename	workern
buildbot.process.builder.enforceChosenSlave	enforce
buildbot.process.builder.Builder.canStartWithSlavebuilder	canSta
buildbot.process.builder.Builder.attaching_slaves	attachi
buildbot.process.builder.Builder.slaves	workers
buildbot.process.builder.Builder.addLatentSlave	addLate
buildbot.process.builder.Builder.getAvailableSlaves	getAva
buildbot.schedulers.forcesched.BuildslaveChoiceParameter	WorkerC
buildbot.process.buildstep.BuildStep.buildslave	*buildbo*
buildbot.process.buildstep.BuildStep.setBuildSlave	*buildbo*
buildbot.process.buildstep.BuildStep.slaveVersion	*buildbo*
buildbot.process.buildstep.BuildStep.slaveVersionIsOlderThan	*buildbo*
buildbot.process.buildstep.BuildStep.checkSlaveHasCommand	*buildbo*
buildbot.process.buildstep.BuildStep.getSlaveName	*buildbo*
buildbot.locks.SlaveLock	buildbo
buildbot.locks.SlaveLock.maxCountForSlave	buildbo
buildbot.locks.SlaveLock constructor argument maxCountForSlave was renamed	maxCou
buildbot.steps.slave	buildbo
buildbot.steps.slave.SlaveBuildStep	buildbo
buildbot.steps.slave.CompositeStepMixin.getFileContentFromSlave	buildbo
buildbot.steps.transfer.FileUpload.slavesrc	workers
buildbot.steps.transfer.FileUpload constructor argument slavesrc was renamed to	workers
buildbot.steps.transfer.DirectoryUpload.slavesrc	workers
buildbot.steps.transfer.DirectoryUpload constructor argument slavesrc was renamed to	workers
buildbot.steps.transfer.MultipleFileUpload.slavesrcs	workers
buildbot.steps.transfer.MultipleFileUpload constructor argument slavesrcs was renamed to	workers
buildbot.steps.transfer.FileDownload.slavedest	workerd
buildbot.steps.transfer.FileDownload constructor argument slavedest was renamed to	workerd
buildbot.steps.transfer.StringDownload.slavedest	workerd
buildbot.steps.transfer.StringDownload constructor argument slavedest was renamed to	workerd
buildbot.steps.transfer.JSONStringDownload.slavedest	workerd

Table 2.2 – continued from

Old name	New nam
`buildbot.steps.transfer.JSONStringDownload` constructor argument `slavedest` was renamed to	`worker`
`buildbot.steps.transfer.JSONPropertiesDownload.slavedest`	`worker`
`buildbot.steps.transfer.JSONPropertiesDownload` constructor argument `slavedest` was renamed to	`worker`
`buildbot.process.remotecommand.RemoteCommand.buildslave`	`worker`

Plugins

`buildbot.buildslave` entry point was renamed to `buildbot.worker`, new plugins should be updated accordingly.

Plugins that use old `buildbot.buildslave` entry point are still available in the configuration file in the same way, as they were in versions prior 0.9.0:

```
from buildbot.plugins import buildslave  # deprecated, use "worker" instead
w = buildslave.ThirdPartyWorker()
```

But also they available using new namespace inside configuration file, so its recommended to use `buildbot.plugins.worker` name even if plugin uses old entry points:

```
from buildbot.plugins import worker
# ThirdPartyWorker can be defined in using `buildbot.buildslave` entry
# point, this still will work.
w = worker.ThirdPartyWorker()
```

Other changes:

- `buildbot.plugins.util.BuildslaveChoiceParameter` is deprecated in favor of `WorkerChoiceParameter`.
- `buildbot.plugins.util.enforceChosenSlave` is deprecated in favor of `enforceChosenWorker`.
- `buildbot.plugins.util.SlaveLock` is deprecated in favor of `WorkerLock`.

`BuildmasterConfig` changes

- `c['slaves']` was replaced with `c['workers']`. Use of `c['slaves']` will work, but is considered deprecated, and will be removed in the future versions of Buildbot.
- Configuration key `c['slavePortnum']` is deprecated in favor of `c['protocols']['pb']['port']`.

Docker latent worker changes

In addition to class being renamed, environment variables that are set inside container `SLAVENAME` and `SLAVEPASS` were renamed to `WORKERNAME` and `WORKERPASS` accordingly. Old environment variable are still available, but are deprecated and will be removed in the future.

EC2 latent worker changes

Use of default values of `keypair_name` and `security_name` constructor arguments of *buildbot.worker.ec2.EC2LatentWorker* is deprecated. Please specify them explicitly.

steps.slave.SetPropertiesFromEnv changes

In addition to buildbot.steps.slave module being renamed to buildbot.steps.worker, default source value for SetPropertiesFromEnv was changed from "SlaveEnvironment" to "WorkerEnvironment".

Local worker changes

Working directory for local workers were changed from master-basedir/slaves/name to master-basedir/workers/name.

Worker Manager changes

slave_config function argument was renamed to worker_config.

Properties

- slavename property is deprecated in favor of workername property. Render of deprecated property will produce warning.

  buildbot.worker.AbstractWorker (previously buildbot.buildslave.AbstractBuildSlave) slavename property source were changed from BuildSlave to Worker (deprecated)

  AbstractWorker now sets workername property with source Worker which should be used.

Metrics

- buildbot.process.metrics.AttachedSlavesWatcher was renamed to buildbot.process.metrics.AttachedWorkersWatcher.
- buildbot.worker.manager.WorkerManager.name (previously buildbot.buildslave.manager.BuildslaveManager.name) metric measurement class name changed from BuildslaveManager to WorkerManager
- buildbot.worker.manager.WorkerManager.managed_services_name (previously buildbot.buildslave.manager.BuildslaveManager.managed_services_name`) metric measurement managed service name changed from ``buildslaves to workers

Renamed events:

Old name	New name
AbstractBuildSlave.attached_slaves	AbstractWorker.attached_workers
BotMaster.attached_slaves	BotMaster.attached_workers
BotMaster.slaveLost()	BotMaster.workerLost()
BotMaster.getBuildersForSlave()	BotMaster.getBuildersForWorker()
AttachedSlavesWatcher	AttachedWorkersWatcher
attached_slaves	attached_workers

Database

Schema changes:

Old name	New name
`buildslaves` table	`workers`
`builds.buildslaveid` (not ForeignKey) column	`workerid` (now ForeignKey)
`configured_buildslaves` table	`configured_workers`
`configured_buildslaves.buildslaveid` (ForeignKey) column	`workerid`
`connected_buildslaves` table	`connected_workers`
`connected_buildslaves.buildslaveid` (ForeignKey) column	`workerid`
`buildslaves_name` index	`workers_name`
`configured_slaves_buildmasterid` index	`configured_workers_buildmasterid`
`configured_slaves_slaves` index	`configured_workers_workers`
`configured_slaves_identity` index	`configured_workers_identity`
`connected_slaves_masterid` index	`connected_workers_masterid`
`connected_slaves_slaves` index	`connected_workers_workers`
`connected_slaves_identity` index	`connected_workers_identity`
`builds_buildslaveid` index	`builds_workerid`

List of database-related changes in API (fallback for old API is provided):

Old name	New name
`buildbot.db.buildslaves`	*`workers`*
`buildbot.db.buildslaves.BuildslavesConnectorComponent`	*`buildbot.db.workers.WorkersConnectorComponent`*
`buildbot.db.buildslaves.BuildslavesConnectorComponent.getBuildslaves` (rewritten in nine)	*`buildbot.db.workers.WorkersConnectorComponent.getWorkers`*
`buildbot.db.connector.DBConnector.buildslaves`	`buildbot.db.connector.DBConnector.workers`

`usePTY` changes

`usePTY` default value has been changed from `slave-config` to `None` (use of `slave-config` will still work, but discouraged).

`buildbot-worker`

`buildbot-slave` package has been renamed to `buildbot-worker`.

`buildbot-worker` has backward incompatible changes and requires buildmaster >= 0.9.0b8. `buildbot-slave` from 0.8.x will work with both 0.8.x and 0.9.x versions of buildmaster, so there is no need to upgrade currently deployed buildbot-slaves during switch from 0.8.x to 0.9.x.

Table 2.3: Master/worker compatibility table

	master 0.8.x	master 0.9.x
buildbot-slave	yes	yes
buildbot-worker	no	yes

`buildbot-worker` doesn't support worker-side specification of `usePTY` (with `--usepty` command line switch of `buildbot-worker create-worker`), you need to specify this option on master side.

`getSlaveInfo` remote command was renamed to `getWorkerInfo` in `buildbot-worker`.

Customization

For advanced users, Buildbot acts as a framework supporting a customized build application. For the most part, such configurations consist of subclasses set up for use in a regular Buildbot configuration file.

This chapter describes some of the more common idioms in advanced Buildbot configurations.

At the moment, this chapter is an unordered set of suggestions:

If you'd like to clean it up, fork the project on GitHub and get started!

Programmatic Configuration Generation

Bearing in mind that `master.cfg` is a Python file, large configurations can be shortened considerably by judicious use of Python loops. For example, the following will generate a builder for each of a range of supported versions of Python:

```
pythons = ['python2.4', 'python2.5', 'python2.6', 'python2.7',
           'python3.2', 'python3.3']
pytest_workers = ["worker%s" % n for n in range(10)]
for python in pythons:
    f = util.BuildFactory()
    f.addStep(steps.SVN(...))
    f.addStep(steps.ShellCommand(command=[python, 'test.py']))
    c['builders'].append(util.BuilderConfig(
            name="test-%s" % python,
            factory=f,
            workernames=pytest_workers))
```

Next step would be the loading of `pythons` list from a .yaml/.ini file.

Collapse Request Functions

The logic Buildbot uses to decide which build request can be merged can be customized by providing a Python function (a callable) instead of `True` or `False` described in *Collapsing Build Requests*.

Arguments for the callable are:

master pointer to the master object, which can be used to make additional data api calls via *master.data.get*

builder dictionary of type *builder*

req1 dictionary of type *buildrequest*

req2 dictionary of type *buildrequest*

Warning: The number of invocations of the callable is proportional to the square of the request queue length, so a long-running callable may cause undesirable delays when the queue length grows.

It should return true if the requests can be merged, and False otherwise. For example:

```
@defer.inlineCallbacks
def collapseRequests(master, builder, req1, req2):
    "any requests with the same branch can be merged"

    # get the buildsets for each buildrequest
    selfBuildset , otherBuildset = yield defer.gatherResults([
        master.data.get(('buildsets', req1['buildsetid'])),
        master.data.get(('buildsets', req2['buildsetid']))
        ])
    selfSourcestamps = selfBuildset['sourcestamps']
    otherSourcestamps = otherBuildset['sourcestamps']

    if len(selfSourcestamps) != len(otherSourcestamps):
        defer.returnValue(False)

    for selfSourcestamp, otherSourcestamp in zip(selfSourcestamps, otherSourcestamps):
        if selfSourcestamp['branch'] != otherSourcestamp['branch']:
            defer.returnValue(False)

    defer.returnValue(True)

c['collapseRequests'] = collapseRequests
```

In many cases, the details of the *sourcestamp* and *buildrequest* are important.

In the following example, only *buildrequest* with the same "reason" are merged; thus developers forcing builds for different reasons will see distinct builds.

Note the use of the `buildrequest.BuildRequest.canBeCollapsed` method to access the source stamp compatibility algorithm.

```
@defer.inlineCallbacks
def collapseRequests(master, builder, req1, req2):
    canBeCollapsed = yield buildrequest.BuildRequest.canBeCollapsed(master, req1,␣
↪req2)
    if canBeCollapsed and req1.reason == req2.reason:
       defer.returnValue(True)
    else:
       defer.returnValue(False)
c['collapseRequests'] = collapseRequests
```

Another common example is to prevent collapsing of requests coming from a *Trigger* step. *Trigger* step can indeed be used in order to implement parallel testing of the same source.

Buildrequests will all have the same sourcestamp, but probably different properties, and shall not be collapsed.

Note: In most of the cases, just setting collapseRequests=False for triggered builders will do the trick.

In other cases, `parent_buildid` from buildset can be used:

```
@defer.inlineCallbacks
def collapseRequests(master, builder, req1, req2):
    canBeCollapsed = yield buildrequest.BuildRequest.canBeCollapsed(master, req1,␣
↪req2)
    selfBuildset , otherBuildset = yield defer.gatherResults([
        master.data.get(('buildsets', req1['buildsetid'])),
```

```
        master.data.get(('buildsets', req2['buildsetid']))
    ])
    if canBeCollapsed and selfBuildset['parent_buildid'] != None and otherBuildset[
↪'parent_buildid'] != None:
        defer.returnValue(True)
    else:
        defer.returnValue(False)
c['collapseRequests'] = collapseRequests
```

If it's necessary to perform some extended operation to determine whether two requests can be merged, then the `collapseRequests` callable may return its result via Deferred.

Warning: Again, the number of invocations of the callable is proportional to the square of the request queue length, so a long-running callable may cause undesirable delays when the queue length grows.

For example:

```
@defer.inlineCallbacks
def collapseRequests(master, builder, req1, req2):
    info1, info2 = yield defer.gatherResults([
        getMergeInfo(req1),
        getMergeInfo(req2),
    ])
    defer.returnValue(info1 == info2)

c['collapseRequests'] = collapseRequests
```

Builder Priority Functions

The *prioritizeBuilders* configuration key specifies a function which is called with two arguments: a `BuildMaster` and a list of `Builder` objects. It should return a list of the same `Builder` objects, in the desired order. It may also remove items from the list if builds should not be started on those builders. If necessary, this function can return its results via a Deferred (it is called with `maybeDeferred`).

A simple `prioritizeBuilders` implementation might look like this:

```
def prioritizeBuilders(buildmaster, builders):
    """Prioritize builders.  'finalRelease' builds have the highest
    priority, so they should be built before running tests, or
    creating builds."""
    builderPriorities = {
        "finalRelease": 0,
        "test": 1,
        "build": 2,
    }
    builders.sort(key=lambda b: builderPriorities.get(b.name, 0))
    return builders

c['prioritizeBuilders'] = prioritizeBuilders
```

Build Priority Functions

When a builder has multiple pending build requests, it uses a `nextBuild` function to decide which build it should start first. This function is given two parameters: the `Builder`, and a list of `BuildRequest` objects representing pending build requests.

A simple function to prioritize release builds over other builds might look like this:

```
def nextBuild(bldr, requests):
    for r in requests:
        if r.source.branch == 'release':
            return r
    return requests[0]
```

If some non-immediate result must be calculated, the `nextBuild` function can also return a Deferred:

```
def nextBuild(bldr, requests):
    d = get_request_priorities(requests)
    def pick(priorities):
        if requests:
            return sorted(zip(priorities, requests))[0][1]
    d.addCallback(pick)
    return d
```

The `nextBuild` function is passed as parameter to `BuilderConfig`:

```
... BuilderConfig(..., nextBuild=nextBuild, ...) ...
```

Customizing SVNPoller

Each source file that is tracked by a Subversion repository has a fully-qualified SVN URL in the following form: (*REPOURL*) (*PROJECT-plus-BRANCH*) (*FILEPATH*). When you create the *SVNPoller*, you give it a `repourl` value that includes all of the *REPOURL* and possibly some portion of the *PROJECT-plus-BRANCH* string. The *SVNPoller* is responsible for producing Changes that contain a branch name and a *FILEPATH* (which is relative to the top of a checked-out tree). The details of how these strings are split up depend upon how your repository names its branches.

PROJECT/BRANCHNAME/FILEPATH repositories

One common layout is to have all the various projects that share a repository get a single top-level directory each, with `branches`, `tags`, and `trunk` subdirectories:

```
amanda/trunk
      /branches/3_2
               /3_3
      /tags/3_2_1
           /3_2_2
           /3_3_0
```

To set up a *SVNPoller* that watches the Amanda trunk (and nothing else), we would use the following, using the default `split_file`:

```
from buildbot.plugins import changes
c['change_source'] = changes.SVNPoller(
   repourl="https://svn.amanda.sourceforge.net/svnroot/amanda/amanda/trunk")
```

In this case, every Change that our *SVNPoller* produces will have its branch attribute set to None, to indicate that the Change is on the trunk. No other sub-projects or branches will be tracked.

If we want our ChangeSource to follow multiple branches, we have to do two things. First we have to change our repourl= argument to watch more than just amanda/trunk. We will set it to amanda so that we'll see both the trunk and all the branches. Second, we have to tell *SVNPoller* how to split the (*PROJECT-plus-BRANCH*)(*FILEPATH*) strings it gets from the repository out into (*BRANCH*) and (*FILEPATH*).

We do the latter by providing a split_file function. This function is responsible for splitting something like branches/3_3/common-src/amanda.h into branch='branches/3_3' and filepath='common-src/amanda.h'. The function is always given a string that names a file relative to the subdirectory pointed to by the *SVNPoller*'s repourl= argument. It is expected to return a dictionary with at least the path key. The splitter may optionally set branch, project and repository. For backwards compatibility it may return a tuple of (branchname, path). It may also return None to indicate that the file is of no interest.

Note: The function should return branches/3_3 rather than just 3_3 because the SVN checkout step, will append the branch name to the baseURL, which requires that we keep the branches component in there. Other VC schemes use a different approach towards branches and may not require this artifact.

If your repository uses this same {PROJECT}/{BRANCH}/{FILEPATH} naming scheme, the following function will work:

```
def split_file_branches(path):
    pieces = path.split('/')
    if len(pieces) > 1 and pieces[0] == 'trunk':
        return (None, '/'.join(pieces[1:]))
    elif len(pieces) > 2 and pieces[0] == 'branches':
        return ('/'.join(pieces[0:2]),
                '/'.join(pieces[2:]))
    else:
        return None
```

In fact, this is the definition of the provided split_file_branches function. So to have our Twisted-watching *SVNPoller* follow multiple branches, we would use this:

```
from buildbot.plugins import changes, util
c['change_source'] = changes.SVNPoller("svn://svn.twistedmatrix.com/svn/Twisted",
                                       split_file=util.svn.split_file_branches)
```

Changes for all sorts of branches (with names like "branches/1.5.x", and None to indicate the trunk) will be delivered to the Schedulers. Each Scheduler is then free to use or ignore each branch as it sees fit.

If you have multiple projects in the same repository your split function can attach a project name to the Change to help the Scheduler filter out unwanted changes:

```
from buildbot.plugins import util
def split_file_projects_branches(path):
    if not "/" in path:
        return None
    project, path = path.split("/", 1)
    f = util.svn.split_file_branches(path)
    if f:
        info = dict(project=project, path=f[1])
        if f[0]:
            info['branch'] = f[0]
```

```
        return info
    return f
```

Again, this is provided by default. To use it you would do this:

```
from buildbot.plugins import changes, util
c['change_source'] = changes.SVNPoller(
   repourl="https://svn.amanda.sourceforge.net/svnroot/amanda/",
   split_file=util.svn.split_file_projects_branches)
```

Note here that we are monitoring at the root of the repository, and that within that repository is a `amanda` subdirectory which in turn has `trunk` and `branches`. It is that `amanda` subdirectory whose name becomes the `project` field of the Change.

BRANCHNAME/PROJECT/FILEPATH repositories

Another common way to organize a Subversion repository is to put the branch name at the top, and the projects underneath. This is especially frequent when there are a number of related sub-projects that all get released in a group.

For example, `Divmod.org` hosts a project named *Nevow* as well as one named *Quotient*. In a checked-out Nevow tree there is a directory named *formless* that contains a Python source file named `webform.py`. This repository is accessible via webdav (and thus uses an *http:* scheme) through the divmod.org hostname. There are many branches in this repository, and they use a `({BRANCHNAME})/({PROJECT})` naming policy.

The fully-qualified SVN URL for the trunk version of `webform.py` is `http://divmod.org/svn/Divmod/trunk/Nevow/formless/webform.py`. The 1.5.x branch version of this file would have a URL of `http://divmod.org/svn/Divmod/branches/1.5.x/Nevow/formless/webform.py`. The whole Nevow trunk would be checked out with `http://divmod.org/svn/Divmod/trunk/Nevow`, while the Quotient trunk would be checked out using `http://divmod.org/svn/Divmod/trunk/Quotient`.

Now suppose we want to have an *SVNPoller* that only cares about the Nevow trunk. This case looks just like the *PROJECT/BRANCH* layout described earlier:

```
from buildbot.plugins import changes
c['change_source'] = changes.SVNPoller("http://divmod.org/svn/Divmod/trunk/Nevow")
```

But what happens when we want to track multiple Nevow branches? We have to point our `repourl=` high enough to see all those branches, but we also don't want to include Quotient changes (since we're only building Nevow). To accomplish this, we must rely upon the `split_file` function to help us tell the difference between files that belong to Nevow and those that belong to Quotient, as well as figuring out which branch each one is on.

```
from buildbot.plugins import changes
c['change_source'] = changes.SVNPoller("http://divmod.org/svn/Divmod",
                                       split_file=my_file_splitter)
```

The `my_file_splitter` function will be called with repository-relative pathnames like:

`trunk/Nevow/formless/webform.py` This is a Nevow file, on the trunk. We want the Change that includes this to see a filename of `formless/webform.py`, and a branch of `None`

`branches/1.5.x/Nevow/formless/webform.py` This is a Nevow file, on a branch. We want to get `branch='branches/1.5.x'` and `filename='formless/webform.py'`.

`trunk/Quotient/setup.py` This is a Quotient file, so we want to ignore it by having `my_file_splitter` return `None`.

`branches/1.5.x/Quotient/setup.py` This is also a Quotient file, which should be ignored.

The following definition for `my_file_splitter` will do the job:

```
def my_file_splitter(path):
    pieces = path.split('/')
    if pieces[0] == 'trunk':
        branch = None
        pieces.pop(0) # remove 'trunk'
    elif pieces[0] == 'branches':
        pieces.pop(0) # remove 'branches'
        # grab branch name
        branch = 'branches/' + pieces.pop(0)
    else:
        return None # something weird
    projectname = pieces.pop(0)
    if projectname != 'Nevow':
        return None # wrong project
    return dict(branch=branch, path='/'.join(pieces))
```

If you later decide you want to get changes for Quotient as well you could replace the last 3 lines with simply:

```
return dict(project=projectname, branch=branch, path='/'.join(pieces))
```

Writing Change Sources

For some version-control systems, making Buildbot aware of new changes can be a challenge. If the pre-supplied classes in *Change Sources* are not sufficient, then you will need to write your own.

There are three approaches, one of which is not even a change source. The first option is to write a change source that exposes some service to which the version control system can "push" changes. This can be more complicated, since it requires implementing a new service, but delivers changes to Buildbot immediately on commit.

The second option is often preferable to the first: implement a notification service in an external process (perhaps one that is started directly by the version control system, or by an email server) and delivers changes to Buildbot via *PBChangeSource*. This section does not describe this particular approach, since it requires no customization within the buildmaster process.

The third option is to write a change source which polls for changes - repeatedly connecting to an external service to check for new changes. This works well in many cases, but can produce a high load on the version control system if polling is too frequent, and can take too long to notice changes if the polling is not frequent enough.

Writing a Notification-based Change Source

A custom change source must implement `buildbot.interfaces.IChangeSource`.

The easiest way to do this is to subclass *`buildbot.changes.base.ChangeSource`*, implementing the `describe` method to describe the instance. `ChangeSource` is a Twisted service, so you will need to implement the `startService` and `stopService` methods to control the means by which your change source receives notifications.

When the class does receive a change, it should call `self.master.addChange(..)` to submit it to the buildmaster. This method shares the same parameters as `master.db.changes.addChange`, so consult the API documentation for that function for details on the available arguments.

You will probably also want to set `compare_attrs` to the list of object attributes which Buildbot will use to compare one change source to another when reconfiguring. During reconfiguration, if the new change source is different from the old, then the old will be stopped and the new started.

Writing a Change Poller

Polling is a very common means of seeking changes, so Buildbot supplies a utility parent class to make it easier. A poller should subclass *buildbot.changes.base.PollingChangeSource*, which is a subclass of *ChangeSource*. This subclass implements the `Service` methods, and calls the `poll` method according to the `pollInterval` and `pollAtLaunch` options. The `poll` method should return a Deferred to signal its completion.

Aside from the service methods, the other concerns in the previous section apply here, too.

Writing a New Latent Worker Implementation

Writing a new latent worker should only require subclassing `buildbot.worker.AbstractLatentWorker` and implementing `start_instance` and `stop_instance`.

```
def start_instance(self):
    # responsible for starting instance that will try to connect with this
    # master. Should return deferred. Problems should use an errback. The
    # callback value can be None, or can be an iterable of short strings to
    # include in the "substantiate success" status message, such as
    # identifying the instance that started.
    raise NotImplementedError

def stop_instance(self, fast=False):
    # responsible for shutting down instance. Return a deferred. If `fast`,
    # we're trying to shut the master down, so callback as soon as is safe.
    # Callback value is ignored.
    raise NotImplementedError
```

See *buildbot.worker.ec2.EC2LatentWorker* for an example.

Custom Build Classes

The standard `BuildFactory` object creates `Build` objects by default. These Builds will each execute a collection of `BuildSteps` in a fixed sequence. Each step can affect the results of the build, but in general there is little intelligence to tie the different steps together.

By setting the factory's `buildClass` attribute to a different class, you can instantiate a different build class. This might be useful, for example, to create a build class that dynamically determines which steps to run. The skeleton of such a project would look like:

```
class DynamicBuild(Build):
    # override some methods
    ...

f = factory.BuildFactory()
f.buildClass = DynamicBuild
f.addStep(...)
```

Factory Workdir Functions

Note: While factory workdir function is still supported, it is better to just use the fact that workdir is a renderables attribute of every steps. A Renderable has access to much more contextual information, and also can return a de-

ferred. So you could say `build_factory.workdir = util.Interpolate("%(src:repository)s` to achieve similar goal.

It is sometimes helpful to have a build's workdir determined at runtime based on the parameters of the build. To accomplish this, set the `workdir` attribute of the build factory to a callable. That callable will be invoked with the list of `SourceStamp` for the build, and should return the appropriate workdir. Note that the value must be returned immediately - Deferreds are not supported.

This can be useful, for example, in scenarios with multiple repositories submitting changes to Buildbot. In this case you likely will want to have a dedicated workdir per repository, since otherwise a sourcing step with mode = "update" will fail as a workdir with a working copy of repository A can't be "updated" for changes from a repository B. Here is an example how you can achieve workdir-per-repo:

```
def workdir(source_stamps):
    return hashlib.md5(source_stamps[0].repository).hexdigest()[:8]

build_factory = factory.BuildFactory()
build_factory.workdir = workdir

build_factory.addStep(Git(mode="update"))
# ...
builders.append ({'name': 'mybuilder',
                  'workername': 'myworker',
                  'builddir': 'mybuilder',
                  'factory': build_factory})
```

The end result is a set of workdirs like

```
Repo1 => <worker-base>/mybuilder/a78890ba
Repo2 => <worker-base>/mybuilder/0823ba88
```

You could make the `workdir` function compute other paths, based on parts of the repo URL in the sourcestamp, or lookup in a lookup table based on repo URL. As long as there is a permanent 1:1 mapping between repos and workdir, this will work.

Writing New BuildSteps

Warning: Buildbot has transitioned to a new, simpler style for writing custom steps. See *New-Style Build Steps* for details. This section documents new-style steps. Old-style steps are supported in Buildbot-0.9.0, but not in later releases.

While it is a good idea to keep your build process self-contained in the source code tree, sometimes it is convenient to put more intelligence into your Buildbot configuration. One way to do this is to write a custom *BuildStep*. Once written, this Step can be used in the `master.cfg` file.

The best reason for writing a custom `BuildStep` is to better parse the results of the command being run. For example, a *BuildStep* that knows about JUnit could look at the logfiles to determine which tests had been run, how many passed and how many failed, and then report more detailed information than a simple `rc==0` -based *good/bad* decision.

Buildbot has acquired a large fleet of build steps, and sports a number of knobs and hooks to make steps easier to write. This section may seem a bit overwhelming, but most custom steps will only need to apply one or two of the techniques outlined here.

For complete documentation of the build step interfaces, see *BuildSteps*.

Writing BuildStep Constructors

Build steps act as their own factories, so their constructors are a bit more complex than necessary. The configuration file instantiates a *BuildStep* object, but the step configuration must be re-used for multiple builds, so Buildbot needs some way to create more steps.

Consider the use of a `BuildStep` in `master.cfg`:

```
f.addStep(MyStep(someopt="stuff", anotheropt=1))
```

This creates a single instance of class `MyStep`. However, Buildbot needs a new object each time the step is executed. An instance of *BuildStep* remembers how it was constructed, and can create copies of itself. When writing a new step class, then, keep in mind are that you cannot do anything "interesting" in the constructor – limit yourself to checking and storing arguments.

It is customary to call the parent class's constructor with all otherwise-unspecified keyword arguments. Keep a `**kwargs` argument on the end of your options, and pass that up to the parent class's constructor.

The whole thing looks like this:

```
class Frobnify(LoggingBuildStep):
    def __init__(self,
                 frob_what="frobee",
                 frob_how_many=None,
                 frob_how=None,
                 **kwargs):

        # check
        if frob_how_many is None:
            raise TypeError("Frobnify argument how_many is required")

        # override a parent option
        kwargs['parentOpt'] = 'xyz'

        # call parent
        LoggingBuildStep.__init__(self, **kwargs)

        # set Frobnify attributes
        self.frob_what = frob_what
        self.frob_how_many = how_many
        self.frob_how = frob_how

class FastFrobnify(Frobnify):
    def __init__(self,
                 speed=5,
                 **kwargs):
        Frobnify.__init__(self, **kwargs)
        self.speed = speed
```

Step Execution Process

A step's execution occurs in its *run* method. When this method returns (more accurately, when the Deferred it returns fires), the step is complete. The method's result must be an integer, giving the result of the step. Any other output from the step (logfiles, status strings, URLs, etc.) is the responsibility of the `run` method.

The *ShellCommand* class implements this `run` method, and in most cases steps subclassing `ShellCommand` simply implement some of the subsidiary methods that its `run` method calls.

Running Commands

To spawn a command in the worker, create a *RemoteCommand* instance in your step's `run` method and run it with `runCommand`:

```
cmd = RemoteCommand(args)
d = self.runCommand(cmd)
```

The `CommandMixin` class offers a simple interface to several common worker-side commands.

For the much more common task of running a shell command on the worker, use `ShellMixin`. This class provides a method to handle the myriad constructor arguments related to shell commands, as well as a method to create new *RemoteCommand* instances. This mixin is the recommended method of implementing custom shell-based steps. The older pattern of subclassing `ShellCommand` is no longer recommended.

A simple example of a step using the shell mixin is:

```
class RunCleanup(buildstep.ShellMixin, buildstep.BuildStep):
    def __init__(self, cleanupScript='./cleanup.sh', **kwargs):
        self.cleanupScript = cleanupScript
        kwargs = self.setupShellMixin(kwargs, prohibitArgs=['command'])
        buildstep.BuildStep.__init__(self, **kwargs)

    @defer.inlineCallbacks
    def run(self):
        cmd = yield self.makeRemoteShellCommand(
                command=[self.cleanupScript])
        yield self.runCommand(cmd)
        if cmd.didFail():
            cmd = yield self.makeRemoteShellCommand(
                    command=[self.cleanupScript, '--force'],
                    logEnviron=False)
            yield self.runCommand(cmd)
        defer.returnValue(cmd.results())

@defer.inlineCallbacks
def run(self):
    cmd = RemoteCommand(args)
    log = yield self.addLog('output')
    cmd.useLog(log, closeWhenFinished=True)
    yield self.runCommand(cmd)
```

Updating Status Strings

Each step can summarize its current status in a very short string. For example, a compile step might display the file being compiled. This information can be helpful users eager to see their build finish.

Similarly, a build has a set of short strings collected from its steps summarizing the overall state of the build. Useful information here might include the number of tests run, but probably not the results of a `make clean` step.

As a step runs, Buildbot calls its *getCurrentSummary* method as necessary to get the step's current status. "As necessary" is determined by calls to *buildbot.process.buildstep.BuildStep.updateSummary*. Your step should call this method every time the status summary may have changed. Buildbot will take care of rate-limiting summary updates.

When the step is complete, Buildbot calls its *getResultSummary* method to get a final summary of the step along with a summary for the build.

About Logfiles

Each BuildStep has a collection of log files. Each one has a short name, like *stdio* or *warnings*. Each log file contains an arbitrary amount of text, usually the contents of some output file generated during a build or test step, or a record of everything that was printed to `stdout/stderr` during the execution of some command.

Each can contain multiple *channels*, generally limited to three basic ones: stdout, stderr, and *headers*. For example, when a shell command runs, it writes a few lines to the headers channel to indicate the exact argv strings being run, which directory the command is being executed in, and the contents of the current environment variables. Then, as the command runs, it adds a lot of `stdout` and `stderr` messages. When the command finishes, a final *header* line is added with the exit code of the process.

Status display plugins can format these different channels in different ways. For example, the web page shows log files as text/html, with header lines in blue text, stdout in black, and stderr in red. A different URL is available which provides a text/plain format, in which stdout and stderr are collapsed together, and header lines are stripped completely. This latter option makes it easy to save the results to a file and run **`grep`** or whatever against the output.

Writing Log Files

Most commonly, logfiles come from commands run on the worker. Internally, these are configured by supplying the *`RemoteCommand`* instance with log files via the `useLog` method:

```
@defer.inlineCallbacks
def run(self):
    ...
    log = yield self.addLog('stdio')
    cmd.useLog(log, closeWhenFinished=True, 'stdio')
    yield self.runCommand(cmd)
```

The name passed to `useLog` must match that configured in the command. In this case, `stdio` is the default.

If the log file was already added by another part of the step, it can be retrieved with *`getLog`*:

```
stdioLog = self.getLog('stdio')
```

Less frequently, some master-side processing produces a log file. If this log file is short and easily stored in memory, this is as simple as a call to *`addCompleteLog`*:

```
@defer.inlineCallbacks
def run(self):
    ...
    summary = u'\n'.join('%s: %s' % (k, count)
                         for (k, count) in self.lint_results.iteritems())
    yield self.addCompleteLog('summary', summary)
```

Note that the log contents must be a unicode string.

Longer logfiles can be constructed line-by-line using the `add` methods of the log file:

```
@defer.inlineCallbacks
def run(self):
    ...
    updates = yield self.addLog('updates')
    while True:
        ...
        yield updates.addStdout(some_update)
```

Again, note that the log input must be a unicode string.

Finally, *addHTMLLog* is similar to *addCompleteLog*, but the resulting log will be tagged as containing HTML. The web UI will display the contents of the log using the browser.

The `logfiles=` argument to *ShellCommand* and its subclasses creates new log files and fills them in realtime by asking the worker to watch a actual file on disk. The worker will look for additions in the target file and report them back to the `BuildStep`. These additions will be added to the log file by calling `addStdout`.

All log files can be used as the source of a *LogObserver* just like the normal `stdio` `LogFile`. In fact, it's possible for one *LogObserver* to observe a logfile created by another.

Reading Logfiles

For the most part, Buildbot tries to avoid loading the contents of a log file into memory as a single string. For large log files on a busy master, this behavior can quickly consume a great deal of memory.

Instead, steps should implement a *LogObserver* to examine log files one chunk or line at a time.

For commands which only produce a small quantity of output, *RemoteCommand* will collect the command's stdout into its *stdout* attribute if given the `collectStdout=True` constructor argument.

Adding LogObservers

Most shell commands emit messages to stdout or stderr as they operate, especially if you ask them nicely with a option *–verbose* flag of some sort. They may also write text to a log file while they run. Your `BuildStep` can watch this output as it arrives, to keep track of how much progress the command has made or to process log output for later summarization.

To accomplish this, you will need to attach a *LogObserver* to the log. This observer is given all text as it is emitted from the command, and has the opportunity to parse that output incrementally.

There are a number of pre-built *LogObserver* classes that you can choose from (defined in *buildbot.process.buildstep*, and of course you can subclass them to add further customization. The `LogLineObserver` class handles the grunt work of buffering and scanning for end-of-line delimiters, allowing your parser to operate on complete `stdout`/`stderr` lines.

For example, let's take a look at the `TrialTestCaseCounter`, which is used by the *Trial* step to count test cases as they are run. As Trial executes, it emits lines like the following:

```
buildbot.test.test_config.ConfigTest.testDebugPassword ... [OK]
buildbot.test.test_config.ConfigTest.testEmpty ... [OK]
buildbot.test.test_config.ConfigTest.testIRC ... [FAIL]
buildbot.test.test_config.ConfigTest.testLocks ... [OK]
```

When the tests are finished, trial emits a long line of `======` and then some lines which summarize the tests that failed. We want to avoid parsing these trailing lines, because their format is less well-defined than the *[OK]* lines.

A simple version of the parser for this output looks like this. The full version is in master/buildbot/steps/python_twisted.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/steps/python_twisted.py).

```
from buildbot.plugins import util

class TrialTestCaseCounter(util.LogLineObserver):
    _line_re = re.compile(r'^([\w\.]+) \.\.\. \[([^\]]+)\]$')
    numTests = 0
    finished = False
```

```
    def outLineReceived(self, line):
        if self.finished:
            return
        if line.startswith("=" * 40):
            self.finished = True
            return

        m = self._line_re.search(line.strip())
        if m:
            testname, result = m.groups()
            self.numTests += 1
            self.step.setProgress('tests', self.numTests)
```

This parser only pays attention to stdout, since that's where trial writes the progress lines. It has a mode flag named `finished` to ignore everything after the `====` marker, and a scary-looking regular expression to match each line while hopefully ignoring other messages that might get displayed as the test runs.

Each time it identifies a test has been completed, it increments its counter and delivers the new progress value to the step with `self.step.setProgress`. This helps Buildbot to determine the ETA for the step.

To connect this parser into the *Trial* build step, `Trial.__init__` ends with the following clause:

```
# this counter will feed Progress along the 'test cases' metric
counter = TrialTestCaseCounter()
self.addLogObserver('stdio', counter)
self.progressMetrics += ('tests',)
```

This creates a `TrialTestCaseCounter` and tells the step that the counter wants to watch the `stdio` log. The observer is automatically given a reference to the step in its `step` attribute.

Using Properties

In custom `BuildSteps`, you can get and set the build properties with the *getProperty* and *setProperty* methods. Each takes a string for the name of the property, and returns or accepts an arbitrary JSON-able (lists, dicts, strings, and numbers) object. For example:

```
class MakeTarball(ShellCommand):
    def start(self):
        if self.getProperty("os") == "win":
            self.setCommand([ ... ]) # windows-only command
        else:
            self.setCommand([ ... ]) # equivalent for other systems
        ShellCommand.start(self)
```

Remember that properties set in a step may not be available until the next step begins. In particular, any `Property` or `Interpolate` instances for the current step are interpolated before the step starts, so they cannot use the value of any properties determined in that step.

Using Statistics

Statistics can be generated for each step, and then summarized across all steps in a build. For example, a test step might set its `warnings` statistic to the number of warnings observed. The build could then sum the `warnings` on all steps to get a total number of warnings.

Statistics are set and retrieved with the *setStatistic* and *getStatistic* methods. The *hasStatistic* method determines whether a statistic exists.

The Build method *getSummaryStatistic* can be used to aggregate over all steps in a Build.

BuildStep URLs

Each BuildStep has a collection of *links*. Each has a name and a target URL. The web display displays clickable links for each link, making them a useful way to point to extra information about a step. For example, a step that uploads a build result to an external service might include a link to the uploaded file.

To set one of these links, the `BuildStep` should call the *addURL* method with the name of the link and the target URL. Multiple URLs can be set. For example:

```
@defer.inlineCallbacks
def run(self):
    ... # create and upload report to coverage server
    url = 'http://coverage.example.com/reports/%s' % reportname
    yield self.addURL('coverage', url)
```

Discovering files

When implementing a `BuildStep` it may be necessary to know about files that are created during the build. There are a few worker commands that can be used to find files on the worker and test for the existence (and type) of files and directories.

The worker provides the following file-discovery related commands:

- *stat* calls `os.stat` for a file in the worker's build directory. This can be used to check if a known file exists and whether it is a regular file, directory or symbolic link.
- *listdir* calls `os.listdir` for a directory on the worker. It can be used to obtain a list of files that are present in a directory on the worker.
- *glob* calls `glob.glob` on the worker, with a given shell-style pattern containing wildcards.

For example, we could use stat to check if a given path exists and contains `*.pyc` files. If the path does not exist (or anything fails) we mark the step as failed; if the path exists but is not a directory, we mark the step as having "warnings".

```
from buildbot.plugins import steps, util
from buildbot.interfaces import WorkerTooOldError
import stat

class MyBuildStep(steps.BuildStep):

    def __init__(self, dirname, **kwargs):
        buildstep.BuildStep.__init__(self, **kwargs)
        self.dirname = dirname

    def start(self):
        # make sure the worker knows about stat
        workerver = (self.workerVersion('stat'),
                     self.workerVersion('glob'))
        if not all(workerver):
            raise WorkerTooOldError('need stat and glob')

        cmd = buildstep.RemoteCommand('stat', {'file': self.dirname})

        d = self.runCommand(cmd)
```

```
        d.addCallback(lambda res: self.evaluateStat(cmd))
        d.addErrback(self.failed)
        return d

    def evaluateStat(self, cmd):
        if cmd.didFail():
            self.step_status.setText(["File not found."])
            self.finished(util.FAILURE)
            return
        s = cmd.updates["stat"][-1]
        if not stat.S_ISDIR(s[stat.ST_MODE]):
            self.step_status.setText(["'tis not a directory"])
            self.finished(util.WARNINGS)
            return

        cmd = buildstep.RemoteCommand('glob', {'path': self.dirname + '/*.pyc'})

        d = self.runCommand(cmd)
        d.addCallback(lambda res: self.evaluateGlob(cmd))
        d.addErrback(self.failed)
        return d

    def evaluateGlob(self, cmd):
        if cmd.didFail():
            self.step_status.setText(["Glob failed."])
            self.finished(util.FAILURE)
            return
        files = cmd.updates["files"][-1]
        if len(files):
            self.step_status.setText(["Found pycs"]+files)
        else:
            self.step_status.setText(["No pycs found"])
        self.finished(util.SUCCESS)
```

For more information on the available commands, see *Master-Worker API*.

Todo

Step Progress BuildStepFailed

Writing Dashboards with Flask (http://flask.pocoo.org/) or Bottle (https://bottlepy.org/docs/dev/)

Buildbot Nine UI is written in Javascript. This allows it to be reactive and real time, but comes at a price of a fair complexity. Sometimes, you need a dashboard displaying your build results in your own manner but learning AngularJS for that is just too much.

There is a Buildbot plugin which allows to write a server side generated dashboard, and integrate it in the UI.

```
# This needs buildbot and buildbot_www >= 0.9.5
pip install buildbot_wsgi_dashboards flask
```

- This plugin can use any WSGI compatible web framework, Flask (http://flask.pocoo.org/) is a very common one, Bottle (https://bottlepy.org/docs/dev/) is another popular option.

- The application needs to implement a `/index.html` route, which will render the html code representing the dashboard.
- The application framework runs in a thread outside of Twisted. No need to worry about Twisted and asynchronous code. You can use python-requests (http://docs.python-requests.org/en/master/) or any library from the python ecosystem to access other servers.
- You could use HTTP in order to access Buildbot *REST API*, but you can also use the *Data API*, via the provided synchronous wrapper.

  buildbot_api.dataGet(path, filters=None, fields=None, order=None, limit=None, offse

  Parameters
  - **path** (*tuple*) – A tuple of path elements representing the API path to fetch. Numbers can be passed as strings or integers.
  - **filters** – result spec filters
  - **fields** – result spec fields
  - **order** – result spec order
  - **limit** – result spec limit
  - **offset** – result spec offset

  Raises *InvalidPathError*

  Returns a resource or list, or None

  This is a blocking wrapper to master.data.get as described in *Data API*. The available paths are described in the *REST API*, as well as the nature of return values depending on the kind of data that is fetched. Path can be either the REST path e.g. `"builders/2/builds/4"` or tuple e.g. `("builders", 2, "builds", 4)`. The latter form being more convenient if some path parts are coming from variables. The *Data API* and *REST API* are functionally equivalent except:
  - –*Data API* does not have HTTP connection overhead.
  - –*Data API* does not enforce authorization rules.

  `buildbot_api.dataGet` is accessible via the WSGI application object passed to `wsgi_dashboards` plugin (as per the example).
- That html code output of the server runs inside AngularJS application.
  - It will use the CSS of the AngularJS application (including the Bootstrap (http://getbootstrap.com/css/) CSS base). You can use custom style-sheet with a standard `style` tag within your html. Custom CSS will be shared with the whole Buildbot application once your dashboard is loaded. So you should make sure your custom CSS rules only apply to your dashboard (e.g. by having a specific class for your dashboard's main div)
  - It can use some of the AngularJS directives defined by Buildbot UI (currently only buildsummary is usable).
  - It has full access to the application JS context.

Here is an example of code that you can use in your `master.cfg` to create a simple dashboard:

```
from __future__ import absolute_import
from __future__ import print_function

import os
import time

import requests

from flask import Flask
from flask import render_template

mydashboardapp = Flask('test', root_path=os.path.dirname(__file__))
```

```
# this allows to work on the template without having to restart Buildbot
mydashboardapp.config['TEMPLATES_AUTO_RELOAD'] = True


@mydashboardapp.route("/index.html")
def main():
    # This code fetches build data from the data api, and give it to the template
    builders = mydashboardapp.buildbot_api.dataGet("/builders")

    builds = mydashboardapp.buildbot_api.dataGet("/builds", limit=20)

    # properties are actually not used in the template example, but this is how you␣
↪get more properties
    for build in builds:
        build['properties'] = mydashboardapp.buildbot_api.dataGet(("builds", build[
↪'buildid'], "properties"))

    # Example on how to use requests to get some info from other web servers
    code_frequency_url = "https://api.github.com/repos/buildbot/buildbot/stats/code_
↪frequency"
    results = requests.get(code_frequency_url)
    while results.status_code == 202:
        # while github calculates statistics, it returns code 202.
        # this is no problem, we just sleep in our thread..
        time.sleep(500)
        results = requests.get(code_frequency_url)

    # some post processing of the data from github
    graph_data = []
    for i, data in enumerate(results.json()):
        graph_data.append(
            dict(x=data[0], y=data[1])
        )

    # mydashboard.html is a template inside the template directory
    return render_template('mydashboard.html', builders=builders, builds=builds,
                           graph_data=graph_data)

# Here we assume c['www']['plugins'] has already be created earlier.
# Please see the web server documentation to understand how to configure the other␣
↪parts.
c['www']['plugins']['wsgi_dashboards'] = [  # This is a list of dashboards, you can␣
↪create several
    {
        'name': 'mydashboard',  # as used in URLs
        'caption': 'My Dashboard',  # Title displayed in the UI'
        'app': mydashboardapp,
        'order': 5,  # priority of the dashboard in the left menu (lower is higher in␣
↪the menu)
        'icon': 'area-chart'  # available icon list can be found at http://
↪fontawesome.io/icons/
    }
]
```

Then you need a `templates/mydashboard.html` file near your `master.cfg`.

This template is a standard Jinja (http://jinja.pocoo.org/) template which is the default templating engine of Flask (http://flask.pocoo.org/).

```
<div class="container mydashboard">
    <style>
    /* only modify th from this dashboard! */
    .mydashboard table th {
        font-size 24pt;
    }
    </style>
    <!-- Create a table of builds organised by builders in columns -->
    <table class="table">
        <tr>
            <!-- Generate the table header with name of builders -->
            {% for builder in builders %}
            <th>
                {{builder.name}}
            </th>
            {% endfor %}
        </tr>
        {% for build in builds %}
        <tr>
            {% for builder in builders %}
            <th>
                <!-- If this build is from this builderid, then we render it in this␣
↪cell -->
                {% if build.builderid == builder.builderid %}
                <!-- for representing a build, you can choose one of those three␣
↪forms -->
                    <!-- 1) We use buildbot internal CSS styles display our builds,␣
↪with links to the standard UI  -->
                        <a class="badge-status badge results_{{build.results_text |␣
↪upper}}" href="#/builders/{{build.builderid}}/builds/{{build.number}}">
                            {{build.number}}
                        </a>
                    <!-- 2) The buildsummary directive is very powerful and will␣
↪display steps, sub-builds, logs, urls. -->
                        <buildsummary buildid="{{build.buildid}}" condensed="1"/>
                    <!-- 3) If you need something lighter, there is the build sticker␣
↪directive -->
                        <buildsticker buildid="{{build.buildid}}"/>
                    <!-- Note that those two directives will make additional HTTP␣
↪requests from the browser in order to fetch the necessary data they need to be␣
↪rendered. -->

                {% endif %}
            </th>
            {% endfor %}
        </tr>
        {% endfor %}
    </table>
    <!-- Example of line chart using Chart.js -->
    <canvas id="myChart" width="400" height="400"></canvas>
        <script>
        // We use Chart.js for rendering a chart, we first have to download it from␣
↪internet
        // (will be cached by the browser)
        $.getScript("https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.5.0/Chart.
↪bundle.min.js",
        // See http://www.chartjs.org/docs/ for more details
        function createChart() {
```

```
            var scatterChart = new Chart("myChart", {
            type: 'line',
            data: {
                datasets: [{
                    label: 'Github statistics',
                    // Here the data from the python is passed to the javascript via␣
↪tojson and safe jinja filters
                    // http://flask.pocoo.org/docs/0.12/templating/#standard-filters
                    data: {{graph_data | tojson |safe }}
                }]
            },
            options: {
                scales: {
                    xAxes: [{
                        type: 'linear',
                        position: 'bottom'
                    }]
                }
            }
        });
    });
    </script>

</div>
```

A Somewhat Whimsical Example (or "It's now customized, how do I deploy it?")

Let's say that we've got some snazzy new unit-test framework called Framboozle. It's the hottest thing since sliced bread. It slices, it dices, it runs unit tests like there's no tomorrow. Plus if your unit tests fail, you can use its name for a Web 2.1 startup company, make millions of dollars, and hire engineers to fix the bugs for you, while you spend your afternoons lazily hang-gliding along a scenic pacific beach, blissfully unconcerned about the state of your tests.[1]

To run a Framboozle-enabled test suite, you just run the 'framboozler' command from the top of your source code tree. The 'framboozler' command emits a bunch of stuff to stdout, but the most interesting bit is that it emits the line "FNURRRGH!" every time it finishes running a test case You'd like to have a test-case counting LogObserver that watches for these lines and counts them, because counting them will help the buildbot more accurately calculate how long the build will take, and this will let you know exactly how long you can sneak out of the office for your hang-gliding lessons without anyone noticing that you're gone.

This will involve writing a new `BuildStep` (probably named "Framboozle") which inherits from *`ShellCommand`*. The `BuildStep` class definition itself will look something like this:

```
from buildbot.plugins import steps, util

class FNURRRGHCounter(util.LogLineObserver):
    numTests = 0
    def outLineReceived(self, line):
        if "FNURRRGH!" in line:
            self.numTests += 1
            self.step.setProgress('tests', self.numTests)

class Framboozle(steps.ShellCommand):
    command = ["framboozler"]

    def __init__(self, **kwargs):
```

[1] framboozle.com is still available. Remember, I get 10% :).

```
        steps.ShellCommand.__init__(self, **kwargs)    # always upcall!
        counter = FNURRRGHCounter()
        self.addLogObserver('stdio', counter)
        self.progressMetrics += ('tests',)
```

So that's the code that we want to wind up using. How do we actually deploy it?

You have a number of different options:

Inclusion in the `master.cfg` file

The simplest technique is to simply put the step class definitions in your `master.cfg` file, somewhere before the `BuildFactory` definition where you actually use it in a clause like:

```
f = BuildFactory()
f.addStep(SVN(repourl="stuff"))
f.addStep(Framboozle())
```

Remember that `master.cfg` is secretly just a Python program with one job: populating the `BuildmasterConfig` dictionary. And Python programs are allowed to define as many classes as they like. So you can define classes and use them in the same file, just as long as the class is defined before some other code tries to use it.

This is easy, and it keeps the point of definition very close to the point of use, and whoever replaces you after that unfortunate hang-gliding accident will appreciate being able to easily figure out what the heck this stupid "Framboozle" step is doing anyways. The downside is that every time you reload the config file, the Framboozle class will get redefined, which means that the buildmaster will think that you've reconfigured all the Builders that use it, even though nothing changed. Bleh.

Python file somewhere on the system

Instead, we can put this code in a separate file, and import it into the master.cfg file just like we would the normal buildsteps like *`ShellCommand`* and *`SVN`*.

Create a directory named `~/lib/python`, put the step class definitions in `~/lib/python/framboozle.py`, and run your buildmaster using:

```
PYTHONPATH=~/lib/python buildbot start MASTERDIR
```

or use the `Makefile.buildbot` to control the way `buildbot start` works. Or add something like this to something like your `~/.bashrc` or `~/.bash_profile` or `~/.cshrc`:

```
export PYTHONPATH=~/lib/python
```

Once we've done this, our `master.cfg` can look like:

```
from framboozle import Framboozle
f = BuildFactory()
f.addStep(SVN(repourl="stuff"))
f.addStep(Framboozle())
```

or:

```
import framboozle
f = BuildFactory()
f.addStep(SVN(repourl="stuff"))
f.addStep(framboozle.Framboozle())
```

(check out the Python docs for details about how `import` and `from A import B` work).

What we've done here is to tell Python that every time it handles an "import" statement for some named module, it should look in our `~/lib/python/` for that module before it looks anywhere else. After our directories, it will try in a bunch of standard directories too (including the one where buildbot is installed). By setting the `PYTHONPATH` environment variable, you can add directories to the front of this search list.

Python knows that once it "import"s a file, it doesn't need to re-import it again. This means that reconfiguring the buildmaster (with `buildbot reconfig`, for example) won't make it think the Framboozle class has changed every time, so the Builders that use it will not be spuriously restarted. On the other hand, you either have to start your buildmaster in a slightly weird way, or you have to modify your environment to set the `PYTHONPATH` variable.

Install this code into a standard Python library directory

Find out what your Python's standard include path is by asking it:

```
80:warner@luther% python
Python 2.4.4c0 (#2, Oct  2 2006, 00:57:46)
[GCC 4.1.2 20060928 (prerelease) (Debian 4.1.1-15)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> import pprint
>>> pprint.pprint(sys.path)
['',
 '/usr/lib/python24.zip',
 '/usr/lib/python2.4',
 '/usr/lib/python2.4/plat-linux2',
 '/usr/lib/python2.4/lib-tk',
 '/usr/lib/python2.4/lib-dynload',
 '/usr/local/lib/python2.4/site-packages',
 '/usr/lib/python2.4/site-packages',
 '/usr/lib/python2.4/site-packages/Numeric',
 '/var/lib/python-support/python2.4',
 '/usr/lib/site-python']
```

In this case, putting the code into `/usr/local/lib/python2.4/site-packages/framboozle.py` would work just fine. We can use the same `master.cfg` `import framboozle` statement as in Option 2. By putting it in a standard include directory (instead of the decidedly non-standard `~/lib/python`), we don't even have to set `PYTHONPATH` to anything special. The downside is that you probably have to be root to write to one of those standard include directories.

Distribute a Buildbot Plug-In

First of all, you must prepare a Python package (if you do not know what that is, please check *How to package Buildbot plugins*, where you can find a couple of pointers to tutorials).

When you have a package, you will have a special file called `setup.py`. This file needs to be updated to include a pointer to your new step:

```
setup(
    ...
    entry_points = {
        ...,
        'buildbot.steps': [
            'Framboozle = framboozle:Framboozle'
        ]
    },
    ...
)
```

Where:

- `buildbot.steps` is the kind of plugin you offer (more information about possible kinds you can find in *How to package Buildbot plugins*)
- `framboozle:Framboozle` consists of two parts: `framboozle` is the name of the Python module where to look for `Framboozle` class, which implements the plugin
- `Framboozle` is the name of the plugin.

  This will allow users of your plugin to use it just like any other Buildbot plugins:

  ```
  from buildbot.plugins import steps

  ... steps.Framboozle ...
  ```

Now you can upload it to PyPI (http://pypi.python.org/) where other people can download it from and use in their build systems. Once again, the information about how to prepare and upload a package to PyPI (http://pypi.python.org/) can be found in tutorials listed in *How to package Buildbot plugins*.

Submit the code for inclusion in the Buildbot distribution

Make a fork of buildbot on http://github.com/buildbot/buildbot or post a patch in a bug at http://trac.buildbot.net/. In either case, post a note about your patch to the mailing list, so others can provide feedback and, eventually, commit it.

When it's committed to the master, the usage is the same as in the previous approach:

```
from buildbot.plugins import steps, util

...
f = util.BuildFactory()
f.addStep(steps.SVN(repourl="stuff"))
f.addStep(steps.Framboozle())
...
```

And then you don't even have to install `framboozle.py` anywhere on your system, since it will ship with Buildbot. You don't have to be root, you don't have to set `PYTHONPATH`. But you do have to make a good case for Framboozle being worth going into the main distribution, you'll probably have to provide docs and some unit test cases, you'll need to figure out what kind of beer the author likes (IPA's and Stouts for Dustin), and then you'll have to wait until

the next release. But in some environments, all this is easier than getting root on your buildmaster box, so the tradeoffs may actually be worth it.

Summary

Putting the code in master.cfg (1) makes it available to that buildmaster instance. Putting it in a file in a personal library directory (2) makes it available for any buildmasters you might be running. Putting it in a file in a system-wide shared library directory (3) makes it available for any buildmasters that anyone on that system might be running. Getting it into the buildbot's upstream repository (4) makes it available for any buildmasters that anyone in the world might be running. It's all a matter of how widely you want to deploy that new class.

New-Style Build Steps

In Buildbot-0.9.0, many operations performed by BuildStep subclasses return a Deferred. As a result, custom build steps which call these methods will need to be rewritten.

Buildbot-0.8.9 supports old-style steps natively, while new-style steps are emulated. Buildbot-0.9.0 supports new-style steps natively, while old-style steps are emulated. Later versions of Buildbot will not support old-style steps at all. All custom steps should be rewritten in the new style as soon as possible.

Buildbot distinguishes new-style from old-style steps by the presence of a *run* method. If this method is present, then the step is a new-style step.

Summary of Changes

- New-style steps have a `run` method that is simpler to implement than the old `start` method.
- Many methods are now asynchronous (return Deferreds), as they perform operations on the database.
- Logs are now implemented by a completely different class. This class supports the same log-writing methods (`addStderr` and so on), although they are now asynchronous. However, it does not support log-reading methods such as `getText`. It was never advisable to handle logs as enormous strings. New-style steps should, instead, use a LogObserver or (in Buildbot-0.9.0) fetch log lines bit by bit using the data API.
- *buildbot.process.buildstep.LoggingBuildStep* is deprecated and cannot be used in new-style steps. Mix in `buildbot.process.buildstep.ShellMixin` instead.
- Step strings, derived by parameters like `description`, `descriptionDone`, and `descriptionSuffix`, are no longer treated as lists. For backward compatibility, the parameters may still be given as lists, but will be joined with spaces during execution (using *join_list*).

Backward Compatibility

Some hacks are in place to support old-style steps. These hacks are only activated when an old-style step is detected. Support for old-style steps will be dropped soon after Buildbot-0.9.0 is released.

- The Deferreds from all asynchronous methods invoked during step execution are gathered internally. The step is not considered finished until all such Deferreds have fired, and is marked EXCEPTION if any fail. For logfiles, this is accomplished by means of a synchronous wrapper class.
- Logfile data is available while the step is still in memory. This means that logs returned from `step.getLog` have the expected methods `getText`, `readlines` and so on.

- `ShellCommand` subclasses implicitly gather all stdio output in memory and provide it to the `createSummary` method.

Rewriting `start`

If your custom buildstep implements the `start` method, then rename that method to `run` and set it up to return a Deferred, either explicitly or via `inlineCallbacks`. The value of the Deferred should be the result of the step (one of the codes in *buildbot.process.results*), or a Twisted failure instance to complete the step as EXCEPTION. The new `run` method should *not* call `self.finished` or `self.failed`, instead signalling the same via Deferred.

For example, the following old-style `start` method

```
def start(self):  ## old style
    cmd = remotecommand.RemoteCommand('stat', {'file': self.file })
    d = self.runCommand(cmd)
    d.addCallback(lambda res: self.convertResult(cmd))
    d.addErrback(self.failed)
```

Becomes

```
@defer.inlineCallbacks
def run(self):  ## new style
    cmd = remotecommand.RemoteCommand('stat', {'file': self.file })
    yield self.runCommand(cmd)
    defer.returnValue(self.convertResult(cmd))
```

Newly Asynchronous Methods

The following methods now return a Deferred:

- *buildbot.process.buildstep.BuildStep.addLog*
- `log.addStdout`
- `log.addStderr`
- `log.addHeader`
- `log.finish` (see "Log Objects", below)
- *buildbot.process.remotecommand.RemoteCommand.addStdout*
- *buildbot.process.remotecommand.RemoteCommand.addStderr*
- *buildbot.process.remotecommand.RemoteCommand.addHeader*
- *buildbot.process.remotecommand.RemoteCommand.addToLog*
- *buildbot.process.buildstep.BuildStep.addCompleteLog*
- *buildbot.process.buildstep.BuildStep.addHTMLLog*
- *buildbot.process.buildstep.BuildStep.addURL*

Any custom code in a new-style step that calls these methods must handle the resulting Deferred. In some cases, that means that the calling method's signature will change. For example

```
def summarize(self):  ## old-style
    for m in self.MESSAGES:
        if counts[m]:
            self.addCompleteLog(m, "".join(summaries[m]))
        self.setProperty("count-%s" % m, counts[m], "counter")
```

Is a synchronous function, not returning a Deferred. However, when converted to a new-style test, it must handle Deferreds from the methods it calls, so it must be asynchronous. Syntactically, `inlineCallbacks` makes the change fairly simple:

```
@defer.inlineCallbacks
def summarize(self):  ## new-style
    for m in self.MESSAGES:
        if counts[m]:
            yield self.addCompleteLog(m, "".join(summaries[m]))
        self.setProperty("count-%s" % m, counts[m], "counter")
```

However, this method's callers must now handle the Deferred that it returns. All methods that can be overridden in custom steps can return a Deferred.

Properties

Good news! The API for properties is the same synchronous API as was available in old-style steps. Properties are handled synchronously during the build, and persisted to the database at completion of each step.

Log Objects

Old steps had two ways of interacting with logfiles, both of which have changed.

The first is writing to logs while a step is executing. When using *addCompleteLog* or *addHTMLLog*, this is straightforward, except that in new-style steps these methods return a Deferred.

The second method is via *buildbot.process.buildstep.BuildStep.addLog*. In new-style steps, the returned object (via Deferred) has the following methods to add log content:

- *addStdout*
- *addStderr*
- *addHeader*
- *finish*

All of these methods now return Deferreds. None of the old log-reading methods are available on this object:

- `hasContents`
- `getText`
- `readLines`
- `getTextWithHeaders`
- `getChunks`

If your step uses such methods, consider using a *LogObserver* instead, or using the Data API to get the required data.

The undocumented and unused `subscribeConsumer` method of logfiles has also been removed.

The *subscribe* method now takes a callable, rather than an instance, and does not support catchup. This method was primarily used by *LogObserver*, the implementation of which has been modified accordingly. Any other uses of the subscribe method should be refactored to use a *LogObserver*.

Status Strings

The `self.step_status.setText` and `setText2` methods have been removed. Similarly, the `_describe` and `describe` methods are not used in new-style steps. In fact, steps no longer set their status directly.

Instead, steps call *buildbot.process.buildstep.BuildStep.updateSummary* whenever the status may have changed. This method calls *getCurrentSummary* or *getResultSummary* as appropriate and update displays of the step's status. Steps override the latter two methods to provide appropriate summaries.

Statistics

Support for statistics has been moved to the `BuildStep` and `Build` objects. Calls to `self.step_status.setStatistic` should be rewritten as `self.setStatistic`.

Command-line Tool

This section describes command-line tools available after buildbot installation. Since version 0.8 the one-for-all **buildbot** command-line tool was divided into two parts namely **buildbot** and **buildslave**, starting from version 0.9 **buildslave** command was replaced with **buildbot-worker** command. The last one was separated from main command-line tool to minimize dependencies required for running a worker while leaving all other functions to **buildbot** tool.

Every command-line tool has a list of global options and a set of commands which have their own options. One can run these tools in the following way:

```
buildbot [global options] command [command options]
buildbot-worker [global options] command [command options]
```

The `buildbot` command is used on the master, while `buildbot-worker` is used on the worker. Global options are the same for both tools which perform the following actions:

--help Print general help about available commands and global options and exit. All subsequent arguments are ignored.

--verbose Set verbose output.

--version Print current buildbot version and exit. All subsequent arguments are ignored.

You can get help on any command by specifying `--help` as a command option:

```
buildbot command --help
```

You can also use manual pages for **buildbot** and **buildbot-worker** for quick reference on command-line options.

The remainder of this section describes each buildbot command. See `cmdline` for a full list.

buildbot

The **buildbot** command-line tool can be used to start or stop a buildmaster or buildbot, and to interact with a running buildmaster. Some of its subcommands are intended for buildmaster admins, while some are for developers who are editing the code that the buildbot is monitoring.

Administrator Tools

The following **buildbot** sub-commands are intended for buildmaster administrators:

create-master

```
buildbot create-master -r {BASEDIR}
```

This creates a new directory and populates it with files that allow it to be used as a buildmaster's base directory.

You will usually want to use the option *-r* option to create a relocatable `buildbot.tac`. This allows you to move the master directory without editing this file.

upgrade-master

```
buildbot upgrade-master {BASEDIR}
```

This upgrades a previously created buildmaster's base directory for a new version of buildbot master source code. This will copy the web server static files, and potencially upgrade the db.

start

```
buildbot start [--nodaemon] {BASEDIR}
```

This starts a buildmaster which was already created in the given base directory. The daemon is launched in the background, with events logged to a file named `twistd.log`.

The option *–nodaemon* option instructs Buildbot to skip daemonizing. The process will start in the foreground. It will only return to the command-line when it is stopped.

restart

```
buildbot restart [--nodaemon] {BASEDIR}
```

Restart the buildmaster. This is equivalent to `stop` followed by `start` The option *–nodaemon* option has the same meaning as for `start`.

stop

```
buildbot stop {BASEDIR}
```

This terminates the daemon (either buildmaster or worker) running in the given directory. The `--clean` option shuts down the buildmaster cleanly. With `--no-wait` option `buildbot stop` command will send buildmaster shutdown signal and will immediately exit, not waiting for complete buildmaster shutdown.

sighup

```
buildbot sighup {BASEDIR}
```

This sends a SIGHUP to the buildmaster running in the given directory, which causes it to re-read its `master.cfg` file.

checkconfig

```
buildbot checkconfig {BASEDIR|CONFIG_FILE}
```

This checks if the buildmaster configuration is well-formed and contains no deprecated or invalid elements. If no arguments are used or the base directory is passed as the argument the config file specified in `buildbot.tac` is checked. If the argument is the path to a config file then it will be checked without using the `buildbot.tac` file.

cleanupdb

```
buildbot cleanupdb {BASEDIR|CONFIG_FILE} [-q]
```

This command is frontend for various database maintainance jobs:

- optimiselogs: This optimization groups logs into bigger chunks to apply higher level of compression.

Developer Tools

These tools are provided for use by the developers who are working on the code that the buildbot is monitoring.

try

This lets a developer to ask the question `What would happen if I committed this patch right now?`. It runs the unit test suite (across multiple build platforms) on the developer's current code, allowing them to make sure they will not break the tree when they finally commit their changes.

The `buildbot try` command is meant to be run from within a developer's local tree, and starts by figuring out the base revision of that tree (what revision was current the last time the tree was updated), and a patch that can be applied to that revision of the tree to make it match the developer's copy. This `(revision, patch)` pair is then sent to the buildmaster, which runs a build with that `SourceStamp`. If you want, the tool will emit status messages as the builds run, and will not terminate until the first failure has been detected (or the last success).

There is an alternate form which accepts a pre-made patch file (typically the output of a command like **`svn diff`**). This `--diff` form does not require a local tree to run from. See *try –diff* concerning the `--diff` command option.

For this command to work, several pieces must be in place: the *`Try_Jobdir`* or :*`Try_Userpass`*, as well as some client-side configuration.

Locating the master

The **try** command needs to be told how to connect to the try scheduler, and must know which of the authentication approaches described above is in use by the buildmaster. You specify the approach by using `--connect=ssh` or `--connect=pb` (or `try_connect = 'ssh'` or `try_connect = 'pb'` in `.buildbot/options`).

For the PB approach, the command must be given a option *–master* argument (in the form *HOST*`:`*PORT*) that points to TCP port that you picked in the `Try_Userpass` scheduler. It also takes a option *–username* and option *–passwd* pair of arguments that match one of the entries in the buildmaster's `userpass` list. These arguments can also be provided as `try_master`, `try_username`, and `try_password` entries in the `.buildbot/options` file.

For the SSH approach, the command must be given option *–host* and option *–username*, to get to the buildmaster host. It must also be given option *–jobdir*, which points to the inlet directory configured above. The jobdir can be relative to the user's home directory, but most of the time you will use an explicit path like `~buildbot/project/trydir`. These arguments can be provided in `.buildbot/options` as `try_host`, `try_username`, `try_password`, and `try_jobdir`.

If you need to use something different from the default `ssh` command for connecting to the remote system, you can use *–ssh* command line option or `try_ssh` in the configuration file.

The SSH approach also provides a option *–buildbotbin* argument to allow specification of the buildbot binary to run on the buildmaster. This is useful in the case where buildbot is installed in a *virtualenv* on the buildmaster host, or in other circumstances where the buildbot command is not on the path of the user given by option *–username*. The option *–buildbotbin* argument can be provided in `.buildbot/options` as `try_buildbotbin`

The following command line arguments are deprecated, but retained for backward compatibility:

--tryhost is replaced by option *–host*

--trydir is replaced by option *–jobdir*

--master is replaced by option *–masterstatus*

Likewise, the following `.buildbot/options` file entries are deprecated, but retained for backward compatibility:

- `try_dir` is replaced by `try_jobdir`
- `masterstatus` is replaced by `try_masterstatus`

Waiting for results

If you provide the option *–wait* option (or `try_wait = True` in `.buildbot/options`), the `buildbot try` command will wait until your changes have either been proven good or bad before exiting. Unless you use the option *–quiet* option (or `try_quiet=True`), it will emit a progress message every 60 seconds until the builds have completed.

The SSH connection method does not support waiting for results.

Choosing the Builders

A trial build is performed on multiple Builders at the same time, and the developer gets to choose which Builders are used (limited to a set selected by the buildmaster admin with the `TryScheduler`'s `builderNames=` argument). The set you choose will depend upon what your goals are: if you are concerned about cross-platform compatibility, you should use multiple Builders, one from each platform of interest. You might use just one builder if that platform has libraries or other facilities that allow better test coverage than what you can accomplish on your own machine, or faster test runs.

The set of Builders to use can be specified with multiple option *–builder* arguments on the command line. It can also be specified with a single `try_builders` option in `.buildbot/options` that uses a list of strings to specify all the Builder names:

```
try_builders = ["full-OSX", "full-win32", "full-linux"]
```

If you are using the PB approach, you can get the names of the builders that are configured for the try scheduler using the `get-builder-names` argument:

```
buildbot try --get-builder-names --connect=pb --master=... --username=... --passwd=...
```

Specifying the VC system

The **`try`** command also needs to know how to take the developer's current tree and extract the (revision, patch) source-stamp pair. Each VC system uses a different process, so you start by telling the **`try`** command which VC system you are using, with an argument like option *–vc=cvs* or option *–vc=git*. This can also be provided as `try_vc` in `.buildbot/options`.

The following names are recognized: `bzr` `cvs` `darcs` `hg` `git` `mtn` `p4` `svn`

Finding the top of the tree

Some VC systems (notably CVS and SVN) track each directory more-or-less independently, which means the **`try`** command needs to move up to the top of the project tree before it will be able to construct a proper full-tree patch. To accomplish this, the **`try`** command will crawl up through the parent directories until it finds a marker file. The default name for this marker file is `.buildbot-top`, so when you are using CVS or SVN you should `touch .buildbot-top` from the top of your tree before running **`buildbot try`**. Alternatively, you can use a filename like `ChangeLog` or `README`, since many projects put one of these files in their top-most directory (and nowhere else). To set this filename, use `--topfile=ChangeLog`, or set it in the options file with `try_topfile = 'ChangeLog'`.

You can also manually set the top of the tree with `--topdir=~/trees/mytree`, or `try_topdir = '~/trees/mytree'`. If you use `try_topdir`, in a `.buildbot/options` file, you will need a separate options file for each tree you use, so it may be more convenient to use the `try_topfile` approach instead.

Other VC systems which work on full projects instead of individual directories (Darcs, Mercurial, Git, Monotone) do not require **`try`** to know the top directory, so the option *–try-topfile* and option *–try-topdir* arguments will be ignored.

If the **`try`** command cannot find the top directory, it will abort with an error message.

The following command line arguments are deprecated, but retained for backward compatibility:

- `--try-topdir` is replaced by option *–topdir*
- `--try-topfile` is replaced by option *–topfile*

Determining the branch name

Some VC systems record the branch information in a way that `try` can locate it. For the others, if you are using something other than the default branch, you will have to tell the buildbot which branch your tree is using. You can do this with either the option *–branch* argument, or a `try_branch` entry in the `.buildbot/options` file.

Determining the revision and patch

Each VC system has a separate approach for determining the tree's base revision and computing a patch.

CVS **`try`** pretends that the tree is up to date. It converts the current time into a option *-D* time specification, uses it as the base revision, and computes the diff between the upstream tree as of that point in time versus the current contents. This works, more or less, but requires that the local clock be in reasonably good sync with the repository.

SVN **`try`** does a **`svn status -u`** to find the latest repository revision number (emitted on the last line in the `Status against revision: NN` message). It then performs an `svn diff -rNN` to find out how your tree differs from the repository version, and sends the resulting patch to the buildmaster. If your tree is not up to date, this will result in the `try` tree being created with the latest revision, then *backwards* patches applied to bring it `back` to the version you actually checked out (plus your actual code changes), but this will still result in the correct tree being used for the build.

bzr **`try`** does a `bzr revision-info` to find the base revision, then a `bzr diff -r$base..` to obtain the patch.

Mercurial `hg parents --template '{node}\n'` emits the full revision id (as opposed to the common 12-char truncated) which is a SHA1 hash of the current revision's contents. This is used as the base revision. `hg diff` then provides the patch relative to that revision. For **`try`** to work, your working directory must only have patches that are available from the same remotely-available repository that the build process' `source.Mercurial` will use.

Perforce **`try`** does a `p4 changes -m1 ...` to determine the latest changelist and implicitly assumes that the local tree is synced to this revision. This is followed by a `p4 diff -du` to obtain the patch. A p4 patch differs slightly from a normal diff. It contains full depot paths and must be converted to paths relative to the branch top. To convert the following restriction is imposed. The p4base (see *`P4Source`*) is assumed to be `//depot`

Darcs **`try`** does a `darcs changes --context` to find the list of all patches back to and including the last tag that was made. This text file (plus the location of a repository that contains all these patches) is sufficient to re-create the tree. Therefore the contents of this `context` file *are* the revision stamp for a Darcs-controlled source tree. It then does a `darcs diff -u` to compute the patch relative to that revision.

Git `git branch -v` lists all the branches available in the local repository along with the revision ID it points to and a short summary of the last commit. The line containing the currently checked out branch begins with `*` (star and space) while all the others start with (two spaces). **`try`** scans for this line and extracts the branch name and revision from it. Then it generates a diff against the base revision.

Todo

I'm not sure if this actually works the way it's intended since the extracted base revision might not actually exist in the upstream repository. Perhaps we need to add a –remote option to specify the remote tracking branch to generate a diff against.

Monotone **`mtn automate get_base_revision_id`** emits the full revision id which is a SHA1 hash of the current revision's contents. This is used as the base revision. **`mtn diff`** then provides the patch relative to that revision. For **`try`** to work, your working directory must only have patches that are available from the same remotely-available repository that the build process' `source.Monotone` will use.

patch information

You can provide the option *–who=dev* to designate who is running the try build. This will add the `dev` to the Reason field on the try build's status web page. You can also set `try_who = dev` in the `.buildbot/options` file. Note

that option *–who=dev* will not work on version 0.8.3 or earlier masters.

Similarly, option *–comment=COMMENT* will specify the comment for the patch, which is also displayed in the patch information. The corresponding config-file option is `try_comment`.

Sending properties

You can set properties to send with your change using either the option *–property=key=value* option, which sets a single property, or the option *–properties=key1=value1,key2=value2...* option, which sets multiple comma-separated properties. Either of these can be sepcified multiple times. Note that the option *–properties* option uses commas to split on properties, so if your property value itself contains a comma, you'll need to use the option *–property* option to set it.

try –diff

Sometimes you might have a patch from someone else that you want to submit to the buildbot. For example, a user may have created a patch to fix some specific bug and sent it to you by email. You've inspected the patch and suspect that it might do the job (and have at least confirmed that it doesn't do anything evil). Now you want to test it out.

One approach would be to check out a new local tree, apply the patch, run your local tests, then use `buildbot try` to run the tests on other platforms. An alternate approach is to use the `buildbot try --diff` form to have the buildbot test the patch without using a local tree.

This form takes a option *–diff* argument which points to a file that contains the patch you want to apply. By default this patch will be applied to the TRUNK revision, but if you give the optional option *–baserev* argument, a tree of the given revision will be used as a starting point instead of TRUNK.

You can also use `buildbot try --diff=-` to read the patch from `stdin`.

Each patch has a `patchlevel` associated with it. This indicates the number of slashes (and preceding pathnames) that should be stripped before applying the diff. This exactly corresponds to the option *-p* or option *–strip* argument to the **`patch`** utility. By default `buildbot try --diff` uses a patchlevel of 0, but you can override this with the option *-p* argument.

When you use option *–diff*, you do not need to use any of the other options that relate to a local tree, specifically option *–vc*, option *–try-topfile*, or option *–try-topdir*. These options will be ignored. Of course you must still specify how to get to the buildmaster (with option *–connect*, option *–tryhost*, etc).

Other Tools

These tools are generally used by buildmaster administrators.

sendchange

This command is used to tell the buildmaster about source changes. It is intended to be used from within a commit script, installed on the VC server. It requires that you have a `PBChangeSource` (*`PBChangeSource`*) running in the buildmaster (by being set in `c['change_source']`).

```
buildbot sendchange --master {MASTERHOST}:{PORT} --auth {USER}:{PASS}
        --who {USER} {FILENAMES..}
```

The option *–auth* option specifies the credentials to use to connect to the master, in the form `user:pass`. If the password is omitted, then sendchange will prompt for it. If both are omitted, the old default (username "change" and

password "changepw") will be used. Note that this password is well-known, and should not be used on an internet-accessible port.

The option *–master* and option *–username* arguments can also be given in the options file (see *.buildbot config directory*). There are other (optional) arguments which can influence the `Change` that gets submitted:

--branch (or option `branch`) This provides the (string) branch specifier. If omitted, it defaults to `None`, indicating the `default branch`. All files included in this Change must be on the same branch.

--category (or option `category`) This provides the (string) category specifier. If omitted, it defaults to `None`, indicating `no category`. The category property can be used by schedulers to filter what changes they listen to.

--project (or option `project`) This provides the (string) project to which this change applies, and defaults to ''. The project can be used by schedulers to decide which builders should respond to a particular change.

--repository (or option `repository`) This provides the repository from which this change came, and defaults to `''`.

--revision This provides a revision specifier, appropriate to the VC system in use.

--revision_file This provides a filename which will be opened and the contents used as the revision specifier. This is specifically for Darcs, which uses the output of `darcs changes --context` as a revision specifier. This context file can be a couple of kilobytes long, spanning a couple lines per patch, and would be a hassle to pass as a command-line argument.

--property This parameter is used to set a property on the `Change` generated by `sendchange`. Properties are specified as a *`name`*`:`*`value`* pair, separated by a colon. You may specify many properties by passing this parameter multiple times.

--comments This provides the change comments as a single argument. You may want to use option *–logfile* instead.

--logfile This instructs the tool to read the change comments from the given file. If you use − as the filename, the tool will read the change comments from stdin.

--encoding Specifies the character encoding for all other parameters, defaulting to `'utf8'`.

--vc Specifies which VC system the Change is coming from, one of: `cvs`, `svn`, `darcs`, `hg`, `bzr`, `git`, `mtn`, or `p4`. Defaults to `None`.

user

Note that in order to use this command, you need to configure a *CommandlineUserManager* instance in your *master.cfg* file, which is explained in *Users Options*.

This command allows you to manage users in buildbot's database. No extra requirements are needed to use this command, aside from the Buildmaster running. For details on how Buildbot manages users, see *Users*.

--master The **`user`** command can be run virtually anywhere provided a location of the running buildmaster. The option *–master* argument is of the form *`MASTERHOST`*`:`*`PORT`*.

--username PB connection authentication that should match the arguments to *CommandlineUserManager*.

--passwd PB connection authentication that should match the arguments to *CommandlineUserManager*.

--op There are four supported values for the option *–op* argument: `add`, `update`, `remove`, and `get`. Each are described in full in the following sections.

--bb_username Used with the option *–op=update* option, this sets the user's username for web authentication in the database. It requires option *–bb_password* to be set along with it.

--bb_password Also used with the option *–op=update* option, this sets the password portion of a user's web authentication credentials into the database. The password is first encrypted prior to storage for security reasons.

--ids When working with users, you need to be able to refer to them by unique identifiers to find particular users in the database. The option *–ids* option lets you specify a comma separated list of these identifiers for use with the **`user`** command.

The option *–ids* option is used only when using option *–op=remove* or option *–op=get*.

--info Users are known in buildbot as a collection of attributes tied together by some unique identifier (see *Users*). These attributes are specified in the form `{TYPE}={VALUE}` when using the option *–info* option. These `{TYPE}={VALUE}` pairs are specified in a comma separated list, so for example:

```
--info=svn=jdoe,git='John Doe <joe@example.com>'
```

The option *–info* option can be specified multiple times in the **`user`** command, as each specified option will be interpreted as a new user. Note that option *–info* is only used with option *–op=add* or with option *–op=update*, and whenever you use option *–op=update* you need to specify the identifier of the user you want to update. This is done by prepending the option *–info* arguments with `{ID:}`. If we were to update `'jschmo'` from the previous example, it would look like this:

```
--info=jdoe:git='Joe Doe <joe@example.com>'
```

Note that option *–master*, option *–username*, option *–passwd*, and option *–op* are always required to issue the **`user`** command.

The option *–master*, option *–username*, and option *–passwd* options can be specified in the option file with keywords `user_master`, `user_username`, and `user_passwd`, respectively. If `user_master` is not specified, then option *–master* from the options file will be used instead.

Below are examples of how each command should look. Whenever a **`user`** command is successful, results will be shown to whoever issued the command.

For option *–op=add*:

```
buildbot user --master={MASTERHOST} --op=add \
        --username={USER} --passwd={USERPW} \
        --info={TYPE}={VALUE},...
```

For option *–op=update*:

```
buildbot user --master={MASTERHOST} --op=update \
        --username={USER} --passwd={USERPW} \
        --info={ID}:{TYPE}={VALUE},...
```

For option *–op=remove*:

```
buildbot user --master={MASTERHOST} --op=remove \
        --username={USER} --passwd={USERPW} \
        --ids={ID1},{ID2},...
```

For option *–op=get*:

```
buildbot user --master={MASTERHOST} --op=get \
        --username={USER} --passwd={USERPW} \
        --ids={ID1},{ID2},...
```

A note on option *–op=update*: when updating the option *–bb_username* and option *–bb_password*, the option *–info* doesn't need to have additional `{TYPE}={VALUE}` pairs to update and can just take the `{ID}` portion.

`.buildbot` config directory

Many of the **buildbot** tools must be told how to contact the buildmaster that they interact with. This specification can be provided as a command-line argument, but most of the time it will be easier to set them in an `options` file. The **buildbot** command will look for a special directory named `.buildbot`, starting from the current directory (where the command was run) and crawling upwards, eventually looking in the user's home directory. It will look for a file named `options` in this directory, and will evaluate it as a Python script, looking for certain names to be set. You can just put simple `name = 'value'` pairs in this file to set the options.

For a description of the names used in this file, please see the documentation for the individual **buildbot** sub-commands. The following is a brief sample of what this file's contents could be.

```
# for status-reading tools
masterstatus = 'buildbot.example.org:12345'
# for 'sendchange' or the debug port
master = 'buildbot.example.org:18990'
```

Note carefully that the names in the `options` file usually do not match the command-line option name.

master Equivalent to option *–master* for *sendchange*. It is the location of the `pb.PBChangeSource` for `sendchange.

username Equivalent to option *–username* for the *sendchange* command.

branch Equivalent to option *–branch* for the *sendchange* command.

category Equivalent to option *–category* for the *sendchange* command.

try_connect Equivalent to option *–connect*, this specifies how the *try* command should deliver its request to the buildmaster. The currently accepted values are `ssh` and `pb`.

try_builders Equivalent to option *–builders*, specifies which builders should be used for the *try* build.

try_vc Equivalent to option *–vc* for *try*, this specifies the version control system being used.

try_branch Equivalent to option *–branch*, this indicates that the current tree is on a non-trunk branch.

`try_topdir`

try_topfile Use `try_topdir`, equivalent to option *–try-topdir*, to explicitly indicate the top of your working tree, or `try_topfile`, equivalent to option *–try-topfile* to name a file that will only be found in that top-most directory.

`try_host`

`try_username`

try_dir When `try_connect` is `ssh`, the command will use `try_host` for option *–tryhost*, `try_username` for option *–username*, and `try_dir` for option *–trydir*. Apologies for the confusing presence and absence of 'try'.

`try_username`

`try_password`

try_master Similarly, when `try_connect` is `pb`, the command will pay attention to `try_username` for option *–username*, `try_password` for option *–passwd*, and `try_master` for option *–master*.

`try_wait`

masterstatus `try_wait` and `masterstatus` (equivalent to option *–wait* and `master`, respectively) are used to ask the *`try`* command to wait for the requested build to complete.

worker

buildbot-worker command-line tool is used for worker management only and does not provide any additional functionality. One can create, start, stop and restart the worker.

create-worker

This creates a new directory and populates it with files that let it be used as a worker's base directory. You must provide several arguments, which are used to create the initial `buildbot.tac` file.

The option *-r* option is advisable here, just like for `create-master`.

```
buildbot-worker create-worker -r {BASEDIR} {MASTERHOST}:{PORT} {WORKERNAME} {PASSWORD}
```

The create-worker options are described in *Worker Options*.

start

This starts a worker which was already created in the given base directory. The daemon is launched in the background, with events logged to a file named `twistd.log`.

```
buildbot-worker start [--nodaemon] BASEDIR
```

The option *–nodaemon* option instructs Buildbot to skip daemonizing. The process will start in the foreground. It will only return to the command-line when it is stopped.

restart

```
buildbot-worker restart [--nodaemon] BASEDIR
```

This restarts a worker which is already running. It is equivalent to a `stop` followed by a `start`.

The option *–nodaemon* option has the same meaning as for `start`.

stop

This terminates the daemon worker running in the given directory.

```
buildbot stop BASEDIR
```

Resources

The Buildbot home page is http://buildbot.net/.

For configuration questions and general discussion, please use the `buildbot-devel` mailing list. The subscription instructions and archives are available at http://lists.sourceforge.net/lists/listinfo/buildbot-devel

The `#buildbot` channel on Freenode's IRC servers hosts development discussion, and often folks are available to answer questions there, as well.

Optimization

If you're feeling your Buildbot is running a bit slow, here are some tricks that may help you, but use them at your own risk.

Properties load speedup

For example, if most of your build properties are strings, you can gain an approx. 30% speedup if you put this snippet of code inside your master.cfg file:

```
def speedup_json_loads():
    import json, re

    original_decode = json._default_decoder.decode
    my_regexp = re.compile(r'^\["([^"]*)",\s+"([^"]*)"\]$')
    def decode_with_re(str, *args, **kw):
        m = my_regexp.match(str)
        try:
            return list(m.groups())
        except Exception:
            return original_decode(str, *args, **kw)
    json._default_decoder.decode = decode_with_re

speedup_json_loads()
```

It patches json decoder so that it would first try to extract a value from JSON that is a list of two strings (which is the case for a property being a string), and would fallback to general JSON decoder on any error.

Plugin Infrastructure in Buildbot

New in version 0.8.11.

Plugin infrastructure in Buildbot allows easy use of components that are not part of the core. It also allows unified access to components that are included in the core.

The following snippet

```
from buildbot.plugins import kind

... kind.ComponentClass ...
```

allows to use a component of kind `kind`. Available `kind`s are:

worker workers, described in *Workers*

changes change source, described in *Change Sources*

schedulers schedulers, described in *Schedulers*

steps build steps, described in *Build Steps*

reporters reporters (or reporter targets), described in *Reporters*

util utility classes. For example, *BuilderConfig*, *Build Factories*, *ChangeFilter* and *Locks* are accessible through `util`.

Web interface plugins are not used directly: as described in *web server configuration* section, they are listed in the corresponding section of the web server configuration dictionary.

Note: If you are not very familiar with Python and you need to use different kinds of components, start your `master.cfg` file with:

```
from buildbot.plugins import *
```

As a result, all listed above components will be available for use. This is what sample `master.cfg` file uses.

Finding Plugins

Buildbot maintains a list of plugins at http://trac.buildbot.net/wiki/Plugins.

Developing Plugins

Distribute a Buildbot Plug-In contains all necesary information for you to develop new plugins. Please edit http://trac.buildbot.net/wiki/Plugins to add a link to your plugin!

Plugins of note

Plugins were introduced in Buildbot-0.8.11, so as of this writing, only components that are bundled with Buildbot are available as plugins.

If you have an idea/need about extending Buildbot, head to *How to package Buildbot plugins*, create your own plugins and let the world now how Buildbot can be made even more useful.

Deployment

This page aims at describing the common pitfalls and best practices when deploying buildbot.

Using A Database Server

Buildbot uses the sqlite3 database backend by default. If you plan to host a lot of data, you may consider using a more suitable database server.

If you want to use a database server (e.g., MySQL or Postgres) as the database backend for your Buildbot, use option *buildbot create-master –db* to specify the *connection string* for the database, and make sure that the same URL appears in the `db_url` of the *db* parameter in your configuration file.

Additional Requirements

Depending on the selected database, further Python packages will be required. Consult the SQLAlchemy (http://www.sqlalchemy.org/) dialect list for a full description. The most common choice for MySQL is mysqlclient (https://github.com/PyMySQL/mysqlclient-python). Any reasonably recent version should suffice.

The most common choice for Postgres is Psycopg (http://initd.org/psycopg/) Any reasonably recent version should suffice.

Maintenance

The buildmaster can be configured to send out email notifications when a worker has been offline for a while. Be sure to configure the buildmaster with a contact email address for each worker so these notifications are sent to someone who can bring it back online.

If you find you can no longer provide a worker to the project, please let the project admins know, so they can put out a call for a replacement.

The Buildbot records status and logs output continually, each time a build is performed. The status tends to be small, but the build logs can become quite large. Each build and log are recorded in a separate file, arranged hierarchically under the buildmaster's base directory. To prevent these files from growing without bound, you should periodically delete old build logs. A simple cron job to delete anything older than, say, two weeks should do the job. The only trick is to leave the `buildbot.tac` and other support files alone, for which **find**'s `-mindepth` argument helps skip everything in the top directory. You can use something like the following (assuming builds are stored in `./builds/` directory):

```
@weekly cd BASEDIR && find . -mindepth 2 i-path './builds/*' \
    -prune -o -type f -mtime +14 -exec rm {} \;
@weekly cd BASEDIR && find twistd.log* -mtime +14 -exec rm {} \;
```

Alternatively, you can configure a maximum number of old logs to be kept using the `--log-count` command line option when running `buildbot-worker create-worker` or `buildbot create-master`.

Troubleshooting

Here are a few hints on diagnosing common problems.

Starting the worker

Cron jobs are typically run with a minimal shell (`/bin/sh`, not `/bin/bash`), and tilde expansion is not always performed in such commands. You may want to use explicit paths, because the `PATH` is usually quite short and doesn't include anything set by your shell's startup scripts (`.profile`, `.bashrc`, etc). If you've installed buildbot (or other Python libraries) to an unusual location, you may need to add a `PYTHONPATH` specification (note that Python will do tilde-expansion on `PYTHONPATH` elements by itself). Sometimes it is safer to fully-specify everything:

```
@reboot PYTHONPATH=~/lib/python /usr/local/bin/buildbot \
    start /usr/home/buildbot/basedir
```

Take the time to get the `@reboot` job set up. Otherwise, things will work fine for a while, but the first power outage or system reboot you have will stop the worker with nothing but the cries of sorrowful developers to remind you that it has gone away.

Connecting to the buildmaster

If the worker cannot connect to the buildmaster, the reason should be described in the `twistd.log` logfile. Some common problems are an incorrect master hostname or port number, or a mistyped bot name or password. If the worker loses the connection to the master, it is supposed to attempt to reconnect with an exponentially-increasing backoff. Each attempt (and the time of the next attempt) will be logged. If you get impatient, just manually stop and re-start the worker.

When the buildmaster is restarted, all workers will be disconnected, and will attempt to reconnect as usual. The reconnect time will depend upon how long the buildmaster is offline (i.e. how far up the exponential backoff curve the workers have travelled). Again, `buildbot-worker restart` *`BASEDIR`* will speed up the process.

Contrib Scripts

While some features of Buildbot are included in the distribution, others are only available in master/contrib/ (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/) in the `buildbot-contrib` source directory. The latest versions of such scripts are available at master/contrib (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib).

CHAPTER 3

Buildbot Development

This chapter is the official repository for the collected wisdom of the Buildbot hackers. It is intended both for developers writing patches that will be included in Buildbot itself, and for advanced users who wish to customize Buildbot.

General Documents

This section gives some general information about Buildbot development.

Master Organization

Buildbot makes heavy use of Twisted Python's support for services - software modules that can be started and stopped dynamically. Buildbot adds the ability to reconfigure such services, too - see *Reconfiguration*. Twisted arranges services into trees; the following section describes the service tree on a running master.

BuildMaster Object

The hierarchy begins with the master, a `buildbot.master.BuildMaster` instance. Most other services contain a reference to this object in their `master` attribute, and in general the appropriate way to access other objects or services is to begin with `self.master` and navigate from there.

The master has a number of useful attributes:

`master.metrics` A `buildbot.process.metrics.MetricLogObserver` instance that handles tracking and reporting on master metrics.

`master.caches` A `buildbot.process.caches.CacheManager` instance that provides access to object caches.

`master.pbmanager` A `buildbot.pbmanager.PBManager` instance that handles incoming PB connections, potentially on multiple ports, and dispatching those connections to appropriate components based on the supplied username.

master.workers A *buildbot.worker.manager.WorkerManager* instance that provides wrapper around multiple master-worker protocols(e.g. PB) to unify calls for them from higher level code

master.change_svc A `buildbot.changes.manager.ChangeManager` instance that manages the active change sources, as well as the stream of changes received from those sources. All active change sources are child services of this instance.

master.botmaster A `buildbot.process.botmaster.BotMaster` instance that manages all of the workers and builders as child services.

The botmaster acts as the parent service for a `buildbot.process.botmaster.BuildRequestDistributor` instance (at `master.botmaster.brd`) as well as all active workers (`buildbot.worker.AbstractWorker` instances) and builders (`buildbot.process.builder.Builder` instances).

master.scheduler_manager A `buildbot.schedulers.manager.SchedulerManager` instance that manages the active schedulers. All active schedulers are child services of this instance.

master.user_manager A `buildbot.process.users.manager.UserManagerManager` instance that manages access to users. All active user managers are child services of this instance.

master.db A *buildbot.db.connector.DBConnector* instance that manages access to the buildbot database. See *Database* for more information.

master.debug A `buildbot.process.debug.DebugServices` instance that manages debugging-related access – the manhole, in particular.

master.status A `buildbot.status.master.Status` instance that provides access to all status data. This instance is also the service parent for all status listeners.

master.masterid This is the ID for this master, from the `masters` table. It is used in the database and messages to uniquely identify this master.

Definitions

Buildbot uses some terms and concepts that have specific meanings.

Repository

See *Repository*.

Project

See *Project*.

Version Control Comparison

Buildbot supports a number of version control systems, and they don't all agree on their terms. This table should help to disambiguate them.

Name	Change	Revision	Branches
CVS	patch [1]	timestamp	unnamed
Subversion	revision	integer	directories
Git	commit	sha1 hash	named refs
Mercurial	changeset	sha1 hash	different repos or (permanently) named commits
Darcs	?	none [2]	different repos
Bazaar	?	?	?
Perforce	?	?	?
BitKeeper	changeset	?	different repos

- [1] note that CVS only tracks patches to individual files. Buildbot tries to recognize coordinated changes to multiple files by correlating change times.
- [2] Darcs does not have a concise way of representing a particular revision of the source.

Buildbot Coding Style

Documentation

Buildbot strongly encourages developers to document the methods, behavior, and usage of classes that users might interact with. However, this documentation should be in `.rst` files under `master/docs/developer`, rather than in docstrings within the code. For private methods or where code deserves some kind of explanatory preface, use comments instead of a docstring. While some docstrings remain within the code, these should be migrated to documentation files and removed as the code is modified.

Within the reStructuredText files, write with each English sentence on its own line. While this does not affect the generated output, it makes git diffs between versions of the documentation easier to read, as they are not obscured by changes due to re-wrapping. This convention is not followed everywhere, but we are slowly migrating documentation from the old (wrapped) style as we update it.

Symbol Names

Buildbot follows PEP8 (https://www.python.org/dev/peps/pep-0008/) regarding the formatting of symbol names. Because Buildbot uses Twisted so heavily, and Twisted uses interCaps, this is not very consistently applied throughout the codebase.

The single exception to PEP8 is in naming of functions and methods. That is, you should spell methods and functions with the first character in lower-case, and the first letter of subsequent words capitalized, e.g., `compareToOther` or `getChangesGreaterThan`.

Symbols used as parameters to functions used in configuration files should use underscores.

In summary, then:

Symbol Type	Format
Methods	interCaps
Functions	interCaps
Function Arguments	under_scores
API method Arguments	interCaps
Classes	InitialCaps
Variables	under_scores
Constants	ALL_CAPS

Twisted Idioms

Programming with Twisted Python can be daunting. But sticking to a few well-defined patterns can help avoid surprises.

Prefer to Return Deferreds

If you're writing a method that doesn't currently block, but could conceivably block sometime in the future, return a Deferred and document that it does so. Just about anything might block - even getters and setters!

Helpful Twisted Classes

Twisted has some useful, but little-known classes. Brief descriptions follow, but you should consult the API documentation or source code for the full details.

`twisted.internet.task.LoopingCall` Calls an asynchronous function repeatedly at set intervals. Note that this will stop looping if the function fails. In general, you will want to wrap the function to capture and log errors.

`twisted.application.internet.TimerService` Similar to `t.i.t.LoopingCall`, but implemented as a service that will automatically start and stop the function calls when the service starts and stops. See the warning about failing functions for `t.i.t.LoopingCall`.

Sequences of Operations

Especially in Buildbot, we're often faced with executing a sequence of operations, many of which may block.

In all cases where this occurs, there is a danger of pre-emption, so exercise the same caution you would if writing a threaded application.

For simple cases, you can use nested callback functions. For more complex cases, inlineCallbacks is appropriate. In all cases, please prefer maintainability and readability over performance.

Nested Callbacks

First, an admonition: do not create extra class methods that represent the continuations of the first:

```
def myMethod(self):
    d = ...
    d.addCallback(self._myMethod_2)  # BAD!
def _myMethod_2(self, res):          # BAD!
    ...
```

Invariably, this extra method gets separated from its parent as the code evolves, and the result is completely unreadable. Instead, include all of the code for a particular function or method within the same indented block, using nested functions:

```
def getRevInfo(revname):
    # for example only! See above a better implementation with inlineCallbacks
    results = {}
    d = defer.succeed(None)
    def rev_parse(_): # note use of '_' to quietly indicate an ignored parameter
        return utils.getProcessOutput(git, [ 'rev-parse', revname ])
```

```
    d.addCallback(rev_parse)
    def parse_rev_parse(res):
        results['rev'] = res.strip()
        return utils.getProcessOutput(git, [ 'log', '-1', '--format=%s%n%b', results[
↪'rev'] ])
    d.addCallback(parse_rev_parse)
    def parse_log(res):
        results['comments'] = res.strip()
    d.addCallback(parse_log)
    def set_results(_):
        return results
    d.addCallback(set_results)
    return d
```

it is usually best to make the first operation occur within a callback, as the deferred machinery will then handle any exceptions as a failure in the outer Deferred. As a shortcut, `d.addCallback` works as a decorator:

```
d = defer.succeed(None)
@d.addCallback
def rev_parse(_): # note use of '_' to quietly indicate an ignored parameter
    return utils.getProcessOutput(git, [ 'rev-parse', revname ])
```

Note: `d.addCallback` is not really a decorator as it does not return a modified function. As a result in previous code, `rev_parse` value is actually the Deferred. In general the `inlineCallbacks` method is preferred inside new code as it keeps the code easier to read. As a general rule of thumb, when you need more than 2 callbacks in the same method, it's time to switch it to `inlineCallbacks`. This would be for example the case for previous `getRevInfo`. See this discussion (https://github.com/buildbot/buildbot/pull/2523) with Twisted experts for more information.

Be careful with local variables. For example, if `parse_rev_parse`, above, merely assigned `rev = res.strip()`, then that variable would be local to `parse_rev_parse` and not available in `set_results`. Mutable variables (dicts and lists) at the outer function level are appropriate for this purpose.

Note: do not try to build a loop in this style by chaining multiple Deferreds! Unbounded chaining can result in stack overflows, at least on older versions of Twisted. Use `inlineCallbacks` instead.

In most of the cases if you need more than two callbacks in a method, it is more readable and maintainable to use inlineCallbacks.

inlineCallbacks

`twisted.internet.defer.inlineCallbacks` is a great help to writing code that makes a lot of asynchronous calls, particularly if those calls are made in loop or conditionals. Refer to the Twisted documentation for the details, but the style within Buildbot is as follows:

```
from twisted.internet import defer

@defer.inlineCallbacks
def mymethod(self, x, y):
    xval = yield getSomething(x)

    for z in (yield getZValues()):
```

```
        y += z

    if xval > 10:
        defer.returnValue(xval + y)
        return

    self.someOtherMethod()
```

The key points to notice here:

- Always import `defer` as a module, not the names within it.
- Use the decorator form of `inlineCallbacks`.
- In most cases, the result of a `yield` expression should be assigned to a variable. It can be used in a larger expression, but remember that Python requires that you enclose the expression in its own set of parentheses.
- Python does not permit returning a value from a generator, so statements like `return xval + y` are invalid. Instead, yield the result of `defer.returnValue`. Although this function does cause an immediate function exit, for clarity follow it with a bare `return`, as in the example, unless it is the last statement in a function.

The great advantage of `inlineCallbacks` is that it allows you to use all of the usual Pythonic control structures in their natural form. In particular, it is easy to represent a loop, or even nested loops, in this style without losing any readability.

Note that code using `deferredGenerator` is no longer acceptable in Buildbot.

The previous `getRevInfo` example implementation should rather be written as:

```
@defer.inlineCallbacks
def getRevInfo(revname):
    results = {}
    res = yield utils.getProcessOutput(git, [ 'rev-parse', revname ])
    results['rev'] = res.strip()
    res = yield utils.getProcessOutput(git, [ 'log', '-1', '--format=%s%n%b', results[
↪'rev'] ])
    results['comments'] = res.strip()
    defer.returnValue(results)
```

Locking

Remember that asynchronous programming does not free you from the need to worry about concurrency issues. Particularly if you are executing a sequence of operations, each time you wait for a Deferred, arbitrary other actions can take place.

In general, you should try to perform actions atomically, but for the rare situations that require synchronization, the following might be useful:

- `twisted.internet.defer.DeferredLock`
- *`buildbot.util.misc.deferredLocked`*

Joining Sequences

It's often the case that you'll want to perform multiple operations in parallel, and rejoin the results at the end. For this purpose, you'll want to use a DeferredList (http://twistedmatrix.com/documents/current/api/twisted.internet.defer.DeferredList.html) :

```
def getRevInfo(revname):
    results = {}
    finished = dict(rev_parse=False, log=False)

    rev_parse_d = utils.getProcessOutput(git, [ 'rev-parse', revname ])
    def parse_rev_parse(res):
        return res.strip()
    rev_parse_d.addCallback(parse_rev_parse)

    log_d = utils.getProcessOutput(git, [ 'log', '-1', '--format=%s%n%b', results['rev
↪'] ])
    def parse_log(res):
        return res.strip()
    log_d.addCallback(parse_log)

    d = defer.DeferredList([rev_parse_d, log_d], consumeErrors=1,␣
↪fireOnFirstErrback=1)
    def handle_results(results):
        return dict(rev=results[0][1], log=results[1][1])
    d.addCallback(handle_results)
    return d
```

Here the deferred list will wait for both `rev_parse_d` and `log_d` to fire, or for one of them to fail. You may attach callbacks and errbacks to a `DeferredList` just as for a deferred.

Functions running outside of the main thread

It is very important in Twisted to be able to distinguish functions that runs in the main thread and functions that don't, as reactors and deferreds can only be used in the main thread. To make this distinction clearer, every functions meant to be started in a secondary thread must be prefixed with `thd_`.

CoffeeScript Coding Style

The Buildbot development team is primarily Python experts and not front-end experts. Whlie we did spend lot of time looking for front end best practices, we are happy to accept suggestions to this coding-style and best-practices guide.

Here is a summary of what is the expected coding style for buildbot contributions, as well as some common gotcha's for developers with a Python background.

CoffeeScript looks like Python

Buildbot follows Python pep8 coding style as much as possible, except for naming convention (where twisted's inter-Caps are preferred). The same rules apply for CoffeeScript, whenever they makes sense:

Symbol Type	Format
Methods	interCaps
Functions	interCaps
Function Arguments	interCaps
Classes	InitialCaps
Controllers	interCaps
Services	interCaps
Filters	interCaps
Constants	ALL_CAPS

Coffeelint should be happy

Buildbot ships with a Gruntfile containing a coffeelint configuration which is expected to pass for buildbot CoffeeScript code.

CoffeeScript syntax sugar

CoffeeScript does not have inlineCallbacks, but have some syntax sugar for helping readability of nested callbacks. However, those syntax sugars sometimes leads to surprises. Make sure you check the generated javascript in case of weird behavior.

Follow the following suggestions:

- Use implicit parentheses for multi line function calls or object construction:

```
# GOOD
d.then (res) ->
   $scope.val = res

# BAD
d.then((res) ->
   $scope.val = res
)
```

```
# push a dictionary into a list
# GOOD
l.push
    k1: v1
    k2: v2

# BAD
l.push(
    k1: v1
    k2: v2
)

# BAD
l.push({
    k1: v1
    k2: v2
})
```

- Use explicit parentheses for single line function calls

```
# GOOD
myFunc(service.getA(b))

# BAD
myFunc service.getA b
# (not enough visually-distinct from:)
myFunc service.getA, b
# which means
myFunc(service.getA, b)
```

- always use `return` for multiline functions

  In CoffeeScript, "everything is an expression", and the default return value is the result of the last expression. This is considered too error prone for Python and JS developers who are used to "return None" by default. In

buildbot code, every multiline function must end with an explicit `return` statement.

```
# BAD: implicitly returns the return value of b()
myFunc = ->
    if (a)
        b()

# GOOD
myFunc = ->
    if (a)
        b()
    return null

# GOOD
myFunc = ->
    if (a)
        return b()
    return null
```

- never use return for single line functions

  Single line functions is equivalent to Python `lambda` functions and thus must not use `return`.

```
# GOOD
# if p resolves with a non-null list, will return the list with all element␣
↪incremented
p = p.then( (res) -> _.each(res, (a) -> a + 1))
```

CoffeeScript does not include batteries

There is a very limited standard library in JS, and none in CoffeeScript. However, de-facto general purpose libraries have emerged.

- JQuery considered harmful to access the DOM directly.

  Buildbot ships with JQuery, because it is supposed to be more optimized than AngularJS's own jqlite, and because some 3rd party directives are requiring it. However, it must not be used in Buildbot services or controllers, and should be avoided in directives. The Buildbot UI should follow AngularJS best practices and only modify DOM via templates.

- Lodash is a clone of Underscore.js, and provides good utilities for standard types manipulation (array and objects). Underscore-string is also available for string manipulation function (e.g. startsWith, endsWith)

  Avoid using lodash decoration form. Those are considered tricky to use.

```
# GOOD
_.each(res, (a) -> a + 1))

# BAD
_(res).each((a) -> a + 1))
```

- Require.js is used as technical solution for plugin loading. It should not be used appart from this.
- Moment.js is used for manipulating dates and displaying them to the user in a human readable form (e.g "one month ago"). It can be used anywhere it is useful.

$q "A+ promises" VS twisted's deferred

The AngularJS `$q` module implements A+ promises. At first sight, this looks like Twisted Deferreds.

> **Warning:** d.addCallbacks(successCb, errorCb) is not equivalent to p.then(successCb, errorCb)!

- Once a Twisted deferred has been "called", its result is changed with the return value of each callback in the callback queue.
- Once a $q promise has been "resolved", its result is immutable. p.then() itself returns another promise which can be used to alter result of another promise.

```
d = someFunction()
@d.addCallback
def addOneToResult(res):
    return res + 1
return d # we return the same deferred as the one returned by someFunction()
```

Translate in coffeeScript to:

```
p = someFunction()
p = p.then (res) ->  ## note assignment
    return res + 1
return p  # we return the another promise as the one returned by someFunction()
```

- With `$q`, only the promise creator can resolve it.

```
someFunction = ->
    d = $q.defer()
    $timeout ->
            d.resolve("foo")
        , 100
    return d.promise
p = someFunction()
p.resolve() # cannot work, we can only call the "then" method of a promise
```

Buildbot's Test Suite

Buildbot's master tests are under `buildbot.test`, `buildbot-worker` package tests are under `buildbot_worker.test`. Tests for the workers are similar to the master, although in some cases helpful functionality on the master is not re-implemented on the worker.

Quick-Start

Buildbot uses Twisted trial (http://twistedmatrix.com/trac/wiki/TwistedTrial) to run its test suite. Following is a quick shell session to put you on the right track.

```
# the usual buildbot development bootstrap with git and virtualenv
git clone https://github.com/buildbot/buildbot
cd buildbot

# helper script which creates the virtualenv for development
make virtualenv
. .venv/bin/activate
```

```
# now we run the test suite
trial buildbot

# find all tests that talk about mail
trial -n --reporter=bwverbose buildbot | grep mail

# run only one test module
trial buildbot.test.unit.test_reporters_mail
```

Suites

Tests are divided into a few suites:

- Unit tests (`buildbot.test.unit`) - these follow unit-testing practices and attempt to maximally isolate the system under test. Unit tests are the main mechanism of achieving test coverage, and all new code should be well-covered by corresponding unit tests.
  - Interface tests are a special type of unit tests, and are found in the same directory and often the same file. In many cases, Buildbot has multiple implementations of the same interface – at least one "real" implementation and a fake implementation used in unit testing. The interface tests ensure that these implementations all meet the same standards. This ensures consistency between implementations, and also ensures that the unit tests are testing against realistic fakes.
- Integration tests (`buildbot.test.integration`) - these test combinations of multiple units. Of necessity, integration tests are incomplete - they cannot test every condition; difficult to maintain - they tend to be complex and touch a lot of code; and slow - they usually require considerable setup and execute a lot of code. As such, use of integration tests is limited to a few, broad tests to act as a failsafe for the unit and interface tests.
- Regression tests (`buildbot.test.regressions`) - these test to prevent re-occurrence of historical bugs. In most cases, a regression is better tested by a test in the other suites, or unlike to recur, so this suite tends to be small.
- Fuzz tests (`buildbot.test.fuzz`) - these tests run for a long time and apply randomization to try to reproduce rare or unusual failures. The Buildbot project does not currently have a framework to run fuzz tests regularly.

Unit Tests

Every code module should have corresponding unit tests. This is not currently true of Buildbot, due to a large body of legacy code, but is a goal of the project. All new code must meet this requirement.

Unit test modules are be named after the package or class they test, replacing `.` with `_` and omitting the `buildbot_`. For example, test_schedulers_timed_Periodic.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/unit/test_schedulers_timed_Periodic.py) tests the `Periodic` class in master/buildbot/schedulers/timed.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/schedule
Modules with only one class, or a few trivial classes, can be tested in a single test module. For more complex situations, prefer to use multiple test modules.

Unit tests using renderables require special handling. The following example shows how the same test would be written with the 'param' parameter and with the same parameter as a renderable.:

```
def test_param(self):
    f = self.ConcreteClass(param='val')
    self.assertEqual(f.param, 'val')
```

When the parameter is renderable, you need to instantiate the Class before you can you renderables:

```
def setUp(self):
    self.build = Properties(paramVal='val')

@defer.inlineCallbacks
def test_param_renderable(self):
    f = self.ConcreteClass(param=Interpolate('%(kw:rendered_val)s',
                           rendered_val=Property('paramVal'))
    yield f.start_instance(self.build)
    self.assertEqual(f.param, 'val')
```

Interface Tests

Interface tests exist to verify that multiple implementations of an interface meet the same requirements. Note that the name 'interface' should not be confused with the sparse use of Zope Interfaces in the Buildbot code – in this context, an interface is any boundary between testable units.

Ideally, all interfaces, both public and private, should be tested. Certainly, any *public* interfaces need interface tests.

Interface tests are most often found in files named for the "real" implementation, e.g., test_db_changes.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/unit/test_db_changes.py). When there is ambiguity, test modules should be named after the interface they are testing. Interface tests have the following form:

```
from buildbot.test.util import interfaces
from twistd.trial import unittest

class Tests(interfaces.InterfaceTests):

    # define methods that must be overridden per implementation
    def someSetupMethod(self):
        raise NotImplementedError

    # method signature tests
    def test_signature_someMethod(self):
        @self.assertArgSpecMatches(self.systemUnderTest.someMethod)
        def someMethod(self, arg1, arg2):
            pass

    # tests that all implementations must pass
    def test_something(self):
        pass # ...

class RealTests(Tests):

    # tests that only *real* implementations must pass
    def test_something_else(self):
        pass # ...
```

All of the test methods are defined here, segregated into tests that all implementations must pass, and tests that the fake implementation is not expected to pass. The `test_signature_someMethod` test above illustrates the `buildbot.test.util.interfaces.assertArgSpecMatches` decorator, which can be used to compare the argument specification of a callable with a reference signature conveniently written as a nested function. Wherever possible, prefer to add tests to the `Tests` class, even if this means testing one method (e.g,. `setFoo`) in terms of another (e.g., `getFoo`).

The `assertArgSpecMatches` method can take multiple methods to test; it will check each one in turn.

At the bottom of the test module, a subclass is created for each implementation, implementing the setup methods that were stubbed out in the parent classes:

```
class TestFakeThing(unittest.TestCase, Tests):

    def someSetupMethod(self):
        pass # ...

class TestRealThing(unittest.TestCase, RealTests):

    def someSetupMethod(self):
        pass # ...
```

For implementations which require optional software, such as an AMQP server, this is the appropriate place to signal that tests should be skipped when their prerequisites are not available.

Integration Tests

Integration test modules test several units at once, including their interactions. In general, they serve as a catch-all for failures and bugs that were not detected by the unit and interface tests. As such, they should not aim to be exhaustive, but merely representative.

Integration tests are very difficult to maintain if they reach into the internals of any part of Buildbot. Where possible, try to use the same means as a user would to set up, run, and check the results of an integration test. That may mean writing a `master.cfg` to be parsed, and checking the results by examining the database (or fake DB API) afterward.

Regression Tests

Regression tests are even more rare in Buildbot than integration tests. In many cases, a regression test is not necessary – either the test is better-suited as a unit or interface test, or the failure is so specific that a test will never fail again.

Regression tests tend to be closely tied to the code in which the error occurred. When that code is refactored, the regression test generally becomes obsolete, and is deleted.

Fuzz Tests

Fuzz tests generally run for a fixed amount of time, running randomized tests against a system. They do not run at all during normal runs of the Buildbot tests, unless `BUILDBOT_FUZZ` is defined. This is accomplished with something like the following at the end of each test module:

```
if 'BUILDBOT_FUZZ' not in os.environ:
    del LRUCacheFuzzer
```

Mixins

Buildbot provides a number of purpose-specific mixin classes in master/buildbot/util (https://github.com/buildbot/buildbot/blob/master/master/buildbot/util). These generally define a set of utility functions as well as `setUpXxx` and `tearDownXxx` methods. These methods should be called explicitly from your subclass's `setUp` and `tearDown` methods. Note that some of these methods return Deferreds, which should be handled properly by the caller.

Fakes

Buildbot provides a number of pre-defined fake implementations of internal interfaces, in master/buildbot/test/fake (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/fake). These are designed to be used in unit tests to limit the scope of the test. For example, the fake DB API eliminates the need to create a real database when testing code that uses the DB API, and isolates bugs in the system under test from bugs in the real DB implementation.

The danger of using fakes is that the fake interface and the real interface can differ. The interface tests exist to solve this problem. All fakes should be fully tested in an integration test, so that the fakes pass the same tests as the "real" thing. It is particularly important that the method signatures be compared.

Type Validation

The master/buildbot/test/util/validation.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/util/validation.py) provides a set of classes and definitions for validating Buildbot data types. It supports four types of data:

- DB API dictionaries, as returned from the `getXxx` methods,
- Data API dictionaries, as returned from `get`,
- Data API messages, and
- Simple data types.

These are validated from elsewhere in the codebase with calls to

- `verifyDbDict(testcase, type, value)`,
- `verifyData(testcase, type, options, value)`,
- `verifyMessage(testcase, routingKey, message)`, and
- `verifyType(testcase, name, value, validator)`.

respectively. The `testcase` argument is used to fail the test case if the validation does not succeed. For DB dictionaries and data dictionaries, the `type` identifies the expected data type. For messages, the type is determined from the first element of the routing key.

All messages sent with the fake MQ implementation are automatically validated using `verifyMessage`. The `verifyType` method is used to validate simple types, e.g.,

```
validation.verifyType(self, 'param1', param1, validation.StringValidator())
```

In any case, if `testcase` is None, then the functions will raise an `AssertionError` on failure.

Validator Classes

A validator is an instance of the `Validator` class. Its `validate` method is a generator function that takes a name and an object to validate. It yields error messages describing any deviations of `object` from the designated data type. The `name` argument is used to make such messages more helpful.

A number of validators are supplied for basic types. A few classes deserve special mention:

- `NoneOk` wraps another validator, allowing the object to be None.
- `Any` will match any object without error.
- `IdentifierValidator` will match identifiers; see *identifier*.
- `DictValidator` takes key names as keyword arguments, with the values giving validators for each key. The `optionalNames` argument is a list of keys which may be omitted without error.

- `SourcedPropertiesValidator` matches dictionaries with (value, source) keys, the representation used for properties in the data API.
- `MessageValidator` validates messages. It checks that the routing key is a tuple of strings. The first tuple element gives the message type. The last tuple element is the event, and must be a member of the `events` set. The remaining "middle" tuple elements must match the message values identified by `keyFields`. The `messageValidator` should be a `DictValidator` configured to check the message body. This validator's `validate` method is called with a tuple `(routingKey, message)`.
- `Selector` allows different validators to be selected based on matching functions. Its `add` method takes a matching function, which should return a boolean, and a validator to use if the matching function returns true. If the matching function is None, it is used as a default. This class is used for message and data validation.

Defining Validators

DB validators are defined in the `dbdict` dictionary, e.g.,

```
dbdict['foodict'] = DictValidator(
    id=IntValidator(),
    name=StringValidator(),
    ...
)
```

Data validators are `Selector` validators, where the selector is the `options` passed to `verifyData`.

```
data['foo'] = Selector()
data['foo'].add(lambda opts : opt.get('fanciness') > 10,
    DictValidator(
        fooid=IntValidator(),
        name=StringValidator(),
        ...
))
```

Similarly, message validators are `Selector` validators, where the selector is the routing key. The underlying validator should be a `MessageValidator`.

```
message['foo'] = Selector()
message['foo'].add(lambda rk : rk[-1] == 'new',
    MessageValidator(
        keyFields=['fooid'],
        events=['new', 'complete'],
        messageValidator=DictValidator(
            fooid=IntValidator(),
            name=StringValidator(),
            ...
       )))
```

Good Tests

Bad tests are worse than no tests at all, since they waste developers' time wondering "was that a spurious failure?" or "what the heck is this test trying to do?" Buildbot needs good tests. So what makes a good test?

Independent of Time

Tests that depend on wall time will fail. As a bonus, they run very slowly. Do not use `reactor.callLater` to wait "long enough" for something to happen.

For testing things that themselves depend on time, consider using `twisted.internet.tasks.Clock`. This may mean passing a clock instance to the code under test, and propagating that instance as necessary to ensure that all of the code using `callLater` uses it. Refactoring code for testability is difficult, but worthwhile.

For testing things that do not depend on time, but for which you cannot detect the "end" of an operation: add a way to detect the end of the operation!

Clean Code

Make your tests readable. This is no place to skimp on comments! Others will attempt to learn about the expected behavior of your class by reading the tests. As a side note, if you use a `Deferred` chain in your test, write the callbacks as nested functions, rather than using methods with funny names:

```
def testSomething(self):
    d = doThisFirst()
    def andThisNext(res):
        pass # ...
    d.addCallback(andThisNext)
    return d
```

This isolates the entire test into one indented block. It is OK to add methods for common functionality, but give them real names and explain in detail what they do.

Good Name

Test method names should follow the pattern `test_`*`METHOD`*`_`*`CONDITION`* where *METHOD* is the method being tested, and *CONDITION* is the condition under which it's tested. Since we can't always test a single method, this is not a hard-and-fast rule.

Assert Only One Thing

Where practical, each test should have a single assertion. This may require a little bit of work to get several related pieces of information into a single Python object for comparison. The problem with multiple assertions is that, if the first assertion fails, the remainder are not tested. The test results then do not tell the entire story.

Prefer Fakes to Mocks

Mock objects are too "compliant", and this often masks errors in the system under test. For example, a mis-spelled method name on a mock object will not raise an exception.

Where possible, use one of the pre-written fake objects (see *Fakes*) instead of a mock object. Fakes themselves should be well-tested using interface tests.

Where they are appropriate, Mock objects can be constructed easily using the aptly-named mock (http://www.voidspace.org.uk/python/mock/) module, which is a requirement for Buildbot's tests.

Small Tests

The shorter each test is, the better. Test as little code as possible in each test.

It is fine, and in fact encouraged, to write the code under test in such a way as to facilitate this. As an illustrative example, if you are testing a new Step subclass, but your tests require instantiating a BuildMaster, you're probably doing something wrong!

This also applies to test modules. Several short, easily-digested test modules are preferred over a 1000-line monster.

Isolation

Each test should be maximally independent of other tests. Do not leave files laying around after your test has finished, and do not assume that some other test has run beforehand. It's fine to use caching techniques to avoid repeated, lengthy setup times.

Be Correct

Tests should be as robust as possible, which at a basic level means using the available frameworks correctly. All Deferreds should have callbacks and be chained properly. Error conditions should be checked properly. Race conditions should not exist (see *Independent of Time*, above).

Be Helpful

Note that tests will pass most of the time, but the moment when they are most useful is when they fail.

When the test fails, it should produce output that is helpful to the person chasing it down. This is particularly important when the tests are run remotely, in which case the person chasing down the bug does not have access to the system on which the test fails. A test which fails sporadically with no more information than "AssertionFailed" is a prime candidate for deletion if the error isn't obvious. Making the error obvious also includes adding comments describing the ways a test might fail.

Keeping State

Python does not allow assignment to anything but the innermost local scope or the global scope with the `global` keyword. This presents a problem when creating nested functions:

```
def test_localVariable(self):
    cb_called = False
    def cb():
        cb_called = True
    cb()
    self.assertTrue(cb_called) # will fail!
```

The `cb_called = True` assigns to a *different variable* than `cb_called = False`. In production code, it's usually best to work around such problems, but in tests this is often the clearest way to express the behavior under test.

The solution is to change something in a common mutable object. While a simple list can serve as such a mutable object, this leads to code that is hard to read. Instead, use `State`:

```
from buildbot.test.state import State

def test_localVariable(self):
    state = State(cb_called=False)
    def cb():
        state.cb_called = True
    cb()
    self.assertTrue(state.cb_called) # passes
```

This is almost as readable as the first example, but it actually works.

Configuration

Wherever possible, Buildbot components should access configuration information as needed from the canonical source, `master.config`, which is an instance of *MasterConfig*. For example, components should not keep a copy of the `buildbotURL` locally, as this value may change throughout the lifetime of the master.

Components which need to be notified of changes in the configuration should be implemented as services, subclassing *ReconfigurableServiceMixin*, as described in *Reconfiguration*.

class `buildbot.config.`**MasterConfig**

The master object makes much of the configuration available from an object named `master.config`. Configuration is stored as attributes of this object. Where possible, other Buildbot components should access this configuration directly and not cache the configuration values anywhere else. This avoids the need to ensure that update-from-configuration methods are called on a reconfig.

Aside from validating the configuration, this class handles any backward-compatibility issues - renamed parameters, type changes, and so on - removing those concerns from other parts of Buildbot.

This class may be instantiated directly, creating an entirely default configuration, or via *FileLoader.loadConfig*, which will load the configuration from a config file.

The following attributes are available from this class, representing the current configuration. This includes a number of global parameters:

title
The title of this buildmaster, from *title*.

titleURL
The URL corresponding to the title, from *titleURL*.

buildbotURL
The URL of this buildmaster, for use in constructing WebStatus URLs; from *buildbotURL*.

changeHorizon
The current change horizon, from *changeHorizon*.

logHorizon
The current log horizon, from *logHorizon*.

buildHorizon
The current build horizon, from *buildHorizon*.

logCompressionLimit
The current log compression limit, from *logCompressionLimit*.

logCompressionMethod
The current log compression method, from *logCompressionMethod*.

logMaxSize
The current log maximum size, from *logMaxSize*.

logMaxTailSize
The current log maximum size, from *logMaxTailSize*.

logEncoding
The encoding to expect when logs are provided as bytestrings, from *logEncoding*.

properties
A `Properties` instance containing global properties, from *properties*.

collapseRequests
A callable, or True or False, describing how to collapse requests; from *collapseRequests*.

prioritizeBuilders
A callable, or None, used to prioritize builders; from *prioritizeBuilders*.

codebaseGenerator
A callable, or None, used to determine the codebase from an incoming `Change`, from *codebaseGenerator*

protocols
The per-protocol port specification for worker connections. Based on *protocols*.

multiMaster
If true, then this master is part of a cluster; based on *multiMaster*.

manhole
The manhole instance to use, or None; from *manhole*.

The remaining attributes contain compound configuration structures, usually dictionaries:

validation
Validation regular expressions, a dictionary from *validation*. It is safe to assume that all expected keys are present.

db
Database specification, a dictionary with key *db_url*. It is safe to assume that this key is present.

metrics
The metrics configuration from *metrics*, or an empty dictionary by default.

caches
The cache configuration, from *caches* as well as the deprecated *buildCacheSize* and *changeCacheSize* parameters.

The keys `Builds` and `Caches` are always available; other keys should use `config.caches.get(cachename, 1)`.

schedulers
The dictionary of scheduler instances, by name, from *schedulers*.

builders
The list of *BuilderConfig* instances from *builders*. Builders specified as dictionaries in the configuration file are converted to instances.

workers
The list of `Worker` instances from *workers*.

change_sources
The list of `IChangeSource` providers from *change_source*.

user_managers
The list of user managers providers from *user_managers*.

www
The web server configuration from *www*. The keys `port` and `url` are always available.

services
The list of additional plugin services

classmethod loadFromDict(*config_dict*, *filename*)

Parameters

- **config_dict** (*dict*) – The dictionary containing the configuration to load.
- **filename** (*string*) – The filename to use when reporting errors.

Returns new *MasterConfig* instance

Load the configuration from the given dictionary.

Loading of the configuration file is generally triggered by the master, using the following class:

class buildbot.config.**FileLoader**

__init__(*basedir*, *filename*)

Parameters

- **basedir** (*string*) – directory to which config is relative
- **filename** (*string*) – the configuration file to load

The filename is treated as relative to the basedir, if it is not absolute.

loadConfig(*basedir*, *filename*)

Returns new *MasterConfig* instance

Load the configuration in the given file. Aside from syntax errors, this will also detect a number of semantic errors such as multiple schedulers with the same name.

buildbot.config.**loadConfigDict**(*basedir*, *filename*)

Parameters

- **basedir** (*string*) – directory to which config is relative
- **filename** (*string*) – the configuration file to load

Raises *ConfigErrors* if any errors occur

Returns dict The `BuildmasterConfig` dictionary.

Load the configuration dictionary in the given file.

The filename is treated as relative to the basedir, if it is not absolute.

Builder Configuration

class buildbot.config.**BuilderConfig**([*keyword args*])
This class parameterizes configuration of builders; see *Builder Configuration* for its arguments. The constructor checks for errors and applies defaults, and sets the properties described here. Most are simply copied from the constructor argument of the same name.

Users may subclass this class to add defaults, for example.

name
The builder's name.

factory
The builder's factory.

workernames
The builder's worker names (a list, regardless of whether the names were specified with `workername` or `workernames`).

builddir
The builder's builddir.

workerbuilddir
The builder's worker-side builddir.

category
The builder's category.

nextWorker
The builder's nextWorker callable.

nextBuild
The builder's nextBuild callable.

canStartBuild
The builder's canStartBuild callable.

locks
The builder's locks.

env
The builder's environment variables.

properties
The builder's properties, as a dictionary.

collapseRequests
The builder's collapseRequests callable.

description
The builder's description, displayed in the web status.

Error Handling

If any errors are encountered while loading the configuration *buildbot.config.error* should be called. This can occur both in the configuration-loading code, and in the constructors of any objects that are instantiated in the configuration - change sources, workers, schedulers, build steps, and so on.

buildbot.config.**error**(*error*)

Parameters **error** – error to report

Raises *ConfigErrors* if called at build-time

This function reports a configuration error. If a config file is being loaded, then the function merely records the error, and allows the rest of the configuration to be loaded. At any other time, it raises *ConfigErrors*. This is done so all config errors can be reported, rather than just the first.

exception buildbot.config.**ConfigErrors**([*errors*])

Parameters **errors** (*list*) – errors to report

This exception represents errors in the configuration. It supports reporting multiple errors to the user simultaneously, e.g., when several consistency checks fail.

errors
A list of detected errors, each given as a string.

addError(*msg*)

Parameters **msg** (*string*) – the message to add

Add another error message to the (presumably not-yet-raised) exception.

Configuration in AngularJS

The AngularJS frontend often needs access to the local master configuration. This is accomplished automatically by converting various pieces of the master configuration to a dictionary.

The *IConfigured* interface represents a way to convert any object into a JSON-able dictionary.

class buildbot.interfaces.**IConfigured**

Providers of this interface provide a method to get their configuration as a dictionary:

getConfigDict()

returns object

Return the configuration of this object. Note that despite the name, the return value may not be a dictionary.

Any object can be "cast" to an *IConfigured* provider. The `getConfigDict` method for basic Python objects simply returns the value.

```
IConfigured(someObject).getConfigDict()
```

class buildbot.util.**ConfiguredMixin**
This class is a basic implementation of *IConfigured*. Its *getConfigDict* method simply returns the instance's `name` attribute.

name
Each object configured must have a `name` attribute.

getConfigDict(*self*)

Returns object

Return a config dictionary representing this object.

All of this is used by to serve `/config.js` to the JavaScript frontend.

Reconfiguration

When the buildmaster receives a signal to begin a reconfig, it re-reads the configuration file, generating a new *MasterConfig* instance, and then notifies all of its child services via the reconfig mechanism described below. The master ensures that at most one reconfiguration is taking place at any time.

See *Master Organization* for the structure of the Buildbot service tree.

To simplify initialization, a reconfiguration is performed immediately on master startup. As a result, services only need to implement their configuration handling once, and can use `startService` for initialization.

See below for instructions on implementing configuration of common types of components in Buildbot.

Note: Because Buildbot uses a pure-Python configuration file, it is not possible to support all forms of reconfiguration. In particular, when the configuration includes custom subclasses or modules, reconfiguration can turn up some surprising behaviors due to the dynamic nature of Python. The reconfig support in Buildbot is intended for "intermediate" uses of the software, where there are fewer surprises.

Reconfigurable Services

Instances which need to be notified of a change in configuration should be implemented as Twisted services, and mix in the *ReconfigurableServiceMixin* class, overriding the *reconfigServiceWithBuildbotConfig* method.

class buildbot.config.**ReconfigurableServiceMixin**

reconfigServiceWithBuildbotConfig(*new_config*)

Parameters **new_config** (*MasterConfig*) – new master configuration

Returns Deferred

This method notifies the service that it should make any changes necessary to adapt to the new configuration values given.

This method will be called automatically after a service is started.

It is generally too late at this point to roll back the reconfiguration, so if possible any errors should be detected in the *MasterConfig* implementation. Errors are handled as best as possible and communicated back to the top level invocation, but such errors may leave the master in an inconsistent state. *ConfigErrors* exceptions will be displayed appropriately to the user on startup.

Subclasses should always call the parent class's implementation. For `MultiService` instances, this will call any child services' `reconfigService` methods, as appropriate. This will be done sequentially, such that the Deferred from one service must fire before the next service is reconfigured.

priority

Child services are reconfigured in order of decreasing priority. The default priority is 128, so a service that must be reconfigured before others should be given a higher priority.

Change Sources

When reconfiguring, there is no method by which Buildbot can determine that a new *ChangeSource* represents the same source as an existing *ChangeSource*, but with different configuration parameters. As a result, the change source manager compares the lists of existing and new change sources using equality, stops any existing sources that are not in the new list, and starts any new change sources that do not already exist.

ChangeSource inherits *ComparableMixin*, so change sources are compared based on the attributes described in their `compare_attrs`.

If a change source does not make reference to any global configuration parameters, then there is no need to inherit *ReconfigurableServiceMixin*, as a simple comparison and `startService` and `stopService` will be sufficient.

If the change source does make reference to global values, e.g., as default values for its parameters, then it must inherit *ReconfigurableServiceMixin* to support the case where the global values change.

Schedulers

Schedulers have names, so Buildbot can determine whether a scheduler has been added, removed, or changed during a reconfig. Old schedulers will be stopped, new schedulers will be started, and both new and existing schedulers will see a call to `reconfigService`, if such a method exists. For backward compatibility, schedulers which do not support reconfiguration will be stopped, and the new scheduler started, when their configuration changes.

If, during a reconfiguration, a new and old scheduler's fully qualified class names differ, then the old class will be stopped and the new class started. This supports the case when a user changes, for example, a *`Nightly`* scheduler to a *`Periodic`* scheduler without changing the name.

Because Buildbot uses *`BaseScheduler`* instances directly in the configuration file, a reconfigured scheduler must extract its new configuration information from another instance of itself.

Custom Subclasses

Custom subclasses are most often defined directly in the configuration file, or in a Python module that is reloaded with `reload` every time the configuration is loaded. Because of the dynamic nature of Python, this creates a new object representing the subclass every time the configuration is loaded – even if the class definition has not changed.

Note that if a scheduler's class changes in a reconfig, but the scheduler's name does not, it will still be treated as a reconfiguration of the existing scheduler. This means that implementation changes in custom scheduler subclasses will not be activated with a reconfig. This behavior avoids stopping and starting such schedulers on every reconfig, but can make development difficult.

One workaround for this is to change the name of the scheduler before each reconfig - this will cause the old scheduler to be stopped, and the new scheduler (with the new name and class) to be started.

Workers

Similar to schedulers, workers are specified by name, so new and old configurations are first compared by name, and any workers to be added or removed are noted. Workers for which the fully-qualified class name has changed are also added and removed. All workers have their `reconfigService` method called.

This method takes care of the basic worker attributes, including changing the PB registration if necessary. Any subclasses that add configuration parameters should override `reconfigService` and update those parameters. As with Schedulers, because the `AbstractWorker` instance is given directly in the configuration, on reconfig instances must extract the configuration from a new instance.

User Managers

Since user managers are rarely used, and their purpose is unclear, they are always stopped and re-started on every reconfig. This may change in figure versions.

Status Receivers

At every reconfig, all status listeners are stopped and new versions started.

Writing Schedulers

Buildbot schedulers are the process objects responsible for requesting builds.

Schedulers are free to decide when to request builds, and to define the parameters of the builds. Many schedulers (e.g., *SingleBranchScheduler*) request builds in response to changes from change sources. Others, such as *Nightly*, request builds at specific times. Still others, like *ForceScheduler*, *Try_Jobdir*, or *Triggerable*, respond to external inputs.

Each scheduler has a unique name, and within a Buildbot cluster, can be active on at most one master. If a scheduler is configured on multiple masters, it will be inactive on all but one master. This provides a form of non-revertive failover for schedulers: if an active scheduler's master fails, an inactive instance of that scheduler on another master will become active.

API Stability

Until Buildbot reaches version 1.0.0, API stability is not guaranteed. The instructions in this document may change incompatibly until that time.

Implementing A Scheduler

A scheduler is a subclass of *BaseScheduler*.

The constructor's arguments form the scheduler's configuration. The first two arguments are `name` and `builderNames`, and are positional. The remaining arguments are keyword arguments, and the subclass's constructor should accept `**kwargs` to pass on to the parent class along with the positional arguments.

```
class MyScheduler(base.BaseScheduler):
    def __init__(self, name, builderNames, arg1=None, arg2=None, **kwargs):
        base.BaseScheduler.__init__(self, name, builderNames, **kwargs)
        self.arg1 = arg1
        self.arg2 = arg2
```

Schedulers are Twisted services, so they can implement `startService` and `stopService`. However, it is more common for scheduler subclasses to override `startActivity` and `stopActivity` instead. See below.

Consuming Changes

A scheduler that needs to be notified of new changes should call *startConsumingChanges* when it becomes active. Change consumption will automatically stop when the scheduler becomes inactive.

Once consumption has started, the *gotChange* method is invoked for each new change. The scheduler is free to do whatever it likes in this method.

Adding Buildsets

To add a new buildset, subclasses should call one of the parent-class methods with the prefix `addBuildsetFor`. These methods call `addBuildset` after applying behaviors common to all schedulers

Any of these methods can be called at any time.

Handling Reconfiguration

When the configuration for a scheduler changes, Buildbot deactivates, stops and removes the old scheduler, then adds, starts, and maybe activates the new scheduler. Buildbot determines whether a scheduler has changed by subclassing *ComparableMixin*. See the documentation for class for an explanation of the `compare_attrs` attribute.

Note: In a future version, schedulers will be converted to handle reconfiguration as reconfigurable services, and will no longer require `compare_attrs` to be set.

Becoming Active and Inactive

An inactive scheduler should not do anything that might interfere with an active scheduler of the same name.

Simple schedulers can consult the *`active`* attribute to determine whether the scheduler is active.

Most schedulers, however, will implement the `activate` method to begin any processing expected of an active scheduler. That may involve calling *`startConsumingChanges`*, beginning a `LoopingCall`, or subscribing to messages.

Any processing begun by the `activate` method, or by an active scheduler, should be stopped by the `deactivate` method. The `deactivate` method's Deferred should not fire until such processing has completely stopped. Schedulers must up-call the parent class's `activate` and `deactivate` methods!

Keeping State

The *`BaseScheduler`* class provides *`getState`* and *`setState`* methods to get and set state values for the scheduler. Active scheduler instances should use these functions to store persistent scheduler state, such that if they fail or become inactive, other instances can pick up where they leave off. A scheduler can cache its state locally, only calling `getState` when it first becomes active. However, it is best to keep the state as up-to-date as possible, by calling `setState` any time the state changes. This prevents loss of state from an unexpected master failure.

Note that the state-related methods do not use locks of any sort. It is up to the caller to ensure that no race conditions exist between getting and setting state. Generally, it is sufficient to rely on there being only one running instance of a scheduler, and cache state in memory.

Utilities

Several small utilities are available at the top-level *`buildbot.util`* package.

`buildbot.util.`**`naturalSort`**(*list*)

> **Parameters** **`list`** – list of strings
>
> **Returns** sorted strings

This function sorts strings "naturally", with embedded numbers sorted numerically. This ordering is good for objects which might have a numeric suffix, e.g., `winworker1`, `winworker2`

`buildbot.util.`**`formatInterval`**(*interval*)

> **Parameters** **`interval`** – duration in seconds
>
> **Returns** human-readable (English) equivalent

This function will return a human-readable string describing a length of time, given a number of seconds.

class `buildbot.util.`**`ComparableMixin`**

This mixin class adds comparability to a subclass. Use it like this:

```
class Widget(FactoryProduct, ComparableMixin):
    compare_attrs = ( 'radius', 'thickness' )
    # ...
```

Any attributes not in `compare_attrs` will not be considered when comparing objects. This is used to implement Buildbot's reconfig logic, where a comparison between the new and existing objects is used to determine whether the new object should replace the existing object. If the comparison shows the objects to be equivalent, then the old object is left in place. If they differ, the old object is removed from the buildmaster and the new object added.

For use in configuration objects (schedulers, changesources, etc.), include any attributes which are set in the constructor based on the user's configuration. Be sure to also include the superclass's list, e.g.:

```
class MyScheduler(base.BaseScheduler):
    compare_attrs = base.BaseScheduler.compare_attrs + ('arg1', 'arg2')
```

A point to note is that the compare_attrs list is cumulative; that is, when a subclass also has a compare_attrs and the parent class has a compare_attrs, the subclass' compare_attrs also includes the parent class' compare_attrs.

This class also implements the `buildbot.interfaces.IConfigured` interface. The configuration is automatically generated, being the dict of all `compare_attrs`.

`buildbot.util.`**`safeTranslate`**(*str*)

Parameters **`str`** – input string

Returns safe version of the input

This function will filter out some inappropriate characters for filenames; it is suitable for adapting strings from the configuration for use as filenames. It is not suitable for use with strings from untrusted sources.

`buildbot.util.`**`epoch2datetime`**(*epoch*)

Parameters **`epoch`** – an epoch time (integer)

Returns equivalent datetime object

Convert a UNIX epoch timestamp to a Python datetime object, in the UTC timezone. Note that timestamps specify UTC time (modulo leap seconds and a few other minor details).

`buildbot.util.`**`datetime2epoch`**(*datetime*)

Parameters **`datetime`** – a datetime object

Returns equivalent epoch time (integer)

Convert an arbitrary Python datetime object into a UNIX epoch timestamp.

`buildbot.util.`**`UTC`**

A `datetime.tzinfo` subclass representing UTC time. A similar class has finally been added to Python in version 3.2, but the implementation is simple enough to include here. This is mostly used in tests to create timezone-aware datetime objects in UTC:

```
dt = datetime.datetime(1978, 6, 15, 12, 31, 15, tzinfo=UTC)
```

`buildbot.util.`**`diffSets`**(*old*, *new*)

Parameters

- **`old`** (`set or iterable`) – old set
- **`new`** (`set or iterable`) – new set

Returns a tuple, (removed, added)

This function compares two sets of objects, returning elements that were added and elements that were removed. This is largely a convenience function for reconfiguring services.

`buildbot.util.`**`makeList`**(*input*)

Parameters **input** – a thing

Returns a list of zero or more things

This function is intended to support the many places in Buildbot where the user can specify either a string or a list of strings, but the implementation wishes to always consider lists. It converts any string to a single-element list, `None` to an empty list, and any iterable to a list. Input lists are copied, avoiding aliasing issues.

`buildbot.util.`**`now`**`()`

Returns epoch time (integer)

Return the current time, using either `reactor.seconds` or `time.time()`.

`buildbot.util.`**`flatten`**`(`*list*`[`, *types* `])`

Parameters

- **list** – potentially nested list
- **types** – An optional iterable of the types to flatten. By default, if unspecified, this flattens both lists and tuples

Returns flat list

Flatten nested lists into a list containing no other lists. For example:

```
>>> flatten([ [  1, 2 ], 3, [ [ 4 ], 5 ] ])
[ 1, 2, 3, 4, 5 ]
```

Both lists and tuples are looked at by default.

`buildbot.util.`**`flattened_iterator`**`(`*list*`[`, *types* `])`

Parameters

- **list** – potentially nested list
- **types** – An optional iterable of the types to flatten. By default, if unspecified, this flattens both lists and tuples.

Returns iterator over every element that isn't in types

Returns a generator that doesn't yield any lists/tuples. For example:

```
   >>> for x in flattened_iterator([ [  1, 2 ], 3, [ [ 4 ] ] ]):
   >>>     print x
   1
   2
   3
   4

Use this for extremely large lists to keep memory-usage down and improve␣
↪performance when you only need to iterate once.
```

`buildbot.util.`**`none_or_str`**`(`*obj*`)`

Parameters **obj** – input value

Returns string or `None`

If `obj` is not None, return its string representation.

`ascii2unicode(str):`

Parameters **str** – string

Returns string as unicode, assuming ascii

This function is intended to implement automatic conversions for user convenience. If given a bytestring, it returns the string decoded as ASCII (and will thus fail for any bytes 0x80 or higher). If given a unicode string, it returns it directly.

string2boolean(str):

Parameters **str** – string

Raises **KeyError** –

Returns boolean

This function converts a string to a boolean. It is intended to be liberal in what it accepts: case-insensitive, "true", "on", "yes", "1", etc. It raises `KeyError` if the value is not recognized.

toJson(obj):

Parameters **obj** – object

Returns UNIX epoch timestamp

This function is a helper for json.dump, that allows to convert non-json able objects to json. For now it supports converting datetime.datetime objects to unix timestamp.

buildbot.util.**NotABranch**

This is a sentinel value used to indicate that no branch is specified. It is necessary since schedulers and change sources consider `None` a valid name for a branch. This is generally used as a default value in a method signature, and then tested against with `is`:

```
if branch is NotABranch:
    pass # ...
```

buildbot.util.**in_reactor**(*fn*)

This decorator will cause the wrapped function to be run in the Twisted reactor, with the reactor stopped when the function completes. It returns the result of the wrapped function. If the wrapped function fails, its traceback will be printed, the reactor halted, and `None` returned.

buildbot.util.**asyncSleep**(*secs*)

Yield a deferred that will fire with no result after `secs` seconds. This is the asynchronous equivalent to `time.sleep`, and can be useful in tests.

buildbot.util.**stripUrlPassword**(*url*)

Parameters **url** – a URL

Returns URL with any password component replaced with `xxxx`

Sanitize a URL; use this before logging or displaying a DB URL.

buildbot.util.**join_list**(*maybe_list*)

Parameters **maybe_list** – list, tuple, byte string, or unicode

Returns unicode

If `maybe_list` is a list or tuple, join it with spaces, casting any strings into unicode using `ascii2unicode`. This is useful for configuration parameters that may be strings or lists of strings.

Notifier():

This is a helper for firing mulitple deferreds with the same result.

buildbot.util.**wait**()

Return a deferred that will fire when when the notifier is notified.

buildbot.util.**notify**(*value*)
Fire all the outstanding deferreds with the given value.

buildbot.util.lru

LRUCache(miss_fn, max_size=50):

Parameters

- **miss_fn** – function to call, with key as parameter, for cache misses. The function should return the value associated with the key argument, or None if there is no value associated with the key.
- **max_size** – maximum number of objects in the cache.

This is a simple least-recently-used cache. When the cache grows beyond the maximum size, the least-recently used items will be automatically removed from the cache.

This cache is designed to control memory usage by minimizing duplication of objects, while avoiding unnecessary re-fetching of the same rows from the database.

All values are also stored in a weak valued dictionary, even after they have expired from the cache. This allows values that are used elsewhere in Buildbot to "stick" in the cache in case they are needed by another component. Weak references cannot be used for some types, so these types are not compatible with this class. Note that dictionaries can be weakly referenced if they are an instance of a subclass of `dict`.

If the result of the `miss_fn` is `None`, then the value is not cached; this is intended to avoid caching negative results.

This is based on Raymond Hettinger's implementation (http://code.activestate.com/recipes/498245-lru-and-lfu-cache-decorators/), licensed under the PSF license, which is GPL-compatible.

buildbot.util.lru.**hits**
cache hits so far

buildbot.util.lru.**refhits**
cache misses found in the weak ref dictionary, so far

buildbot.util.lru.**misses**
cache misses leading to re-fetches, so far

buildbot.util.lru.**max_size**
maximum allowed size of the cache

buildbot.util.lru.**get**(*key*, ***miss_fn_kwargs*)

Parameters

- **key** – cache key
- **miss_fn_kwargs** – keyword arguments to the `miss_fn`

Returns value via Deferred

Fetch a value from the cache by key, invoking `miss_fn(key, **miss_fn_kwargs)` if the key is not in the cache.

Any additional keyword arguments are passed to the `miss_fn` as keyword arguments; these can supply additional information relating to the key. It is up to the caller to ensure that this information is functionally identical for each key value: if the key is already in the cache, the `miss_fn` will not be invoked, even if the keyword arguments differ.

buildbot.util.lru.**put**(*key*, *value*)

Parameters

- **key** – key at which to place the value
- **value** – value to place there

Add the given key and value into the cache. The purpose of this method is to insert a new value into the cache *without* invoking the miss_fn (e.g., to avoid unnecessary overhead).

buildbot.util.lru.**inv**()
Check invariants on the cache. This is intended for debugging purposes.

AsyncLRUCache(miss_fn, max_size=50):

Parameters

- **miss_fn** – This is the same as the miss_fn for class LRUCache, with the difference that this function *must* return a Deferred.
- **max_size** – maximum number of objects in the cache.

This class has the same functional interface as LRUCache, but asynchronous locking is used to ensure that in the common case of multiple concurrent requests for the same key, only one fetch is performed.

buildbot.util.bbcollections

This package provides a few useful collection objects.

Note: This module used to be named `collections`, but without absolute imports (**PEP 328** (https://www.python.org/dev/peps/pep-0328)), this precluded using the standard library's `collections` module.

---

class buildbot.util.bbcollections.**defaultdict**
This is a clone of the Python `collections.defaultdict` for use in Python-2.4. In later versions, this is simply a reference to the built-in *defaultdict*, so Buildbot code can simply use `buildbot.util.collections.defaultdict` everywhere.

class buildbot.util.bbcollections.**KeyedSets**
This is a collection of named sets. In principal, it contains an empty set for every name, and you can add things to sets, discard things from sets, and so on.

```
>>> ks = KeyedSets()
>>> ks['tim']                   # get a named set
set([])
>>> ks.add('tim', 'friendly')   # add an element to a set
>>> ks.add('tim', 'dexterous')
>>> ks['tim']
set(['friendly', 'dexterous'])
>>> 'tim' in ks                 # membership testing
True
>>> 'ron' in ks
False
>>> ks.discard('tim', 'friendly')# discard set element
>>> ks.pop('tim')               # return set and reset to empty
set(['dexterous'])
>>> ks['tim']
set([])
```

This class is careful to conserve memory space - empty sets do not occupy any space.

buildbot.util.eventual

This function provides a simple way to say "please do this later". For example

```
from buildbot.util.eventual import eventually
def do_what_I_say(what, where):
    # ...
    return d
eventually(do_what_I_say, "clean up", "your bedroom")
```

The package defines "later" as "next time the reactor has control", so this is a good way to avoid long loops that block other activity in the reactor.

buildbot.util.eventual.**eventually**(*cb*, **args*, ***kwargs*)

Parameters

- **cb** – callable to invoke later
- **args** – args to pass to `cb`
- **kwargs** – kwargs to pass to `cb`

Invoke the callable `cb` in a later reactor turn.

Callables given to *eventually* are guaranteed to be called in the same order as the calls to *eventually* – writing `eventually(a); eventually(b)` guarantees that `a` will be called before `b`.

Any exceptions that occur in the callable will be logged with `log.err()`. If you really want to ignore them, provide a callable that catches those exceptions.

This function returns None. If you care to know when the callable was run, be sure to provide a callable that notifies somebody.

buildbot.util.eventual.**fireEventually**(*value=None*)

Parameters **value** – value with which the Deferred should fire

Returns Deferred

This function returns a Deferred which will fire in a later reactor turn, after the current call stack has been completed, and after all other Deferreds previously scheduled with *eventually*. The returned Deferred will never fail.

buildbot.util.eventual.**flushEventualQueue**()

Returns Deferred

This returns a Deferred which fires when the eventual-send queue is finally empty. This is useful for tests and other circumstances where it is useful to know that "later" has arrived.

buildbot.util.debounce

It's often necessary to perform some action in response to a particular type of event. For example, steps need to update their status after updates arrive from the worker. However, when many events arrive in quick succession, it's more efficient to only perform the action once, after the last event has occurred.

The `debounce.method(wait)` decorator is the tool for the job.

buildbot.util.debounce.**method**(*wait*)

Parameters

- **wait** – time to wait before invoking, in seconds

- **get_reactor** – A callable that takes the underlying instance and returns the reactor to use. Defaults to `instance.master.reactor`.

Returns a decorator that debounces the underlying method. The underlying method must take no arguments (except `self`).

For each call to the decorated method, the underlying method will be invoked at least once within *wait* seconds (plus the time the method takes to execute). Calls are "debounced" during that time, meaning that multiple calls to the decorated method will result in a single invocation.

Note: This functionality is similar to Underscore's `debounce`, except that the Underscore method resets its timer on every call.

The decorated method is an instance of *Debouncer*, allowing it to be started and stopped. This is useful when the method is a part of a Buildbot service: call `method.start()` from `startService` and `method.stop()` from `stopService`, handling its Deferred appropriately.

class `buildbot.util.debounce.`**Debouncer**

stop()

Returns Deferred

Stop the debouncer. While the debouncer is stopped, calls to the decorated method will be ignored. If a call is pending when `stop` is called, that call will occur immediately. When the Deferred that `stop` returns fires, the underlying method is not executing.

start()

Start the debouncer. This reverses the effects of `stop`. This method can be called on a started debouncer without issues.

buildbot.util.poll

Many Buildbot services perform some periodic, asynchronous operation. Change sources, for example, contact the repositories they monitor on a regular basis. The tricky bit is, the periodic operation must complete before the service stops.

The `@poll.method` decorator makes this behavior easy and reliable.

`buildbot.util.poll.`**method**()

This decorator replaces the decorated method with a *Poller* instance configured to call the decorated method periodically. The poller is initially stopped, so peroidic calls will not begin until its `start` method is called. The start polling interval is specified when the poller is started.

If the decorated method fails or raises an exception, the Poller logs the error and re-schedules the call for the next interval.

If a previous invocation of the method has not completed when the interval expires, then the next invocation is skipped and the interval timer starts again.

A common idiom is to call `start` and `stop` from `startService` and `stopService`:

```
class WatchThings(object):

    @poll.method
    def watch(self):
        d = self.beginCheckingSomething()
        return d
```

```
    def startService(self):
        self.watch.start(interval=self.pollingInterval, now=False)

    def stopService(self):
        return self.watch.stop()
```

class buildbot.util.poll.**Poller**

start(*interval=N*, *now=False*)

Parameters

- **interval** – time, in seconds, between invocations
- **now** – if true, call the decorated method immediately on startup.

Start the poller.

stop()

Returns Deferred

Stop the poller. The returned Deferred fires when the decorated method is complete.

__call__()

Force a call to the decorated method now. If the decorated method is currently running, another call will begin as soon as it completes.

buildbot.util.maildir

Several Buildbot components make use of maildirs (http://www.courier-mta.org/maildir.html) to hand off messages between components. On the receiving end, there's a need to watch a maildir for incoming messages and trigger some action when one arrives.

class buildbot.util.maildir.**MaildirService**(*basedir*)

param basedir (optional) base directory of the maildir

A *MaildirService* instance watches a maildir for new messages. It should be a child service of some MultiService instance. When running, this class uses the linux dirwatcher API (if available) or polls for new files in the 'new' maildir subdirectory. When it discovers a new message, it invokes its *messageReceived* method.

To use this class, subclass it and implement a more interesting *messageReceived* function.

setBasedir(*basedir*)

Parameters **basedir** – base directory of the maildir

If no basedir is provided to the constructor, this method must be used to set the basedir before the service starts.

messageReceived(*filename*)

Parameters **filename** – unqualified filename of the new message

This method is called with the short filename of the new message. The full name of the new file can be obtained with os.path.join(maildir, 'new', filename). The method is un-implemented in the *MaildirService* class, and must be implemented in subclasses.

moveToCurDir(*filename*)

Parameters **filename** – unqualified filename of the new message

Returns open file object

Call this from *messageReceived* to start processing the message; this moves the message file to the 'cur' directory and returns an open file handle for it.

buildbot.util.misc

buildbot.util.misc.**deferredLocked**(*lock*)

Parameters **lock** – a twisted.internet.defer.DeferredLock instance or a string naming an instance attribute containing one

This is a decorator to wrap an event-driven method (one returning a Deferred) in an acquire/release pair of a designated DeferredLock. For simple functions with a static lock, this is as easy as:

```
someLock = defer.DeferredLock()

@util.deferredLocked(someLock)
def someLockedFunction():
    # ..
    return d
```

For class methods which must access a lock that is an instance attribute, the lock can be specified by a string, which will be dynamically resolved to the specific instance at runtime:

```
def __init__(self):
    self.someLock = defer.DeferredLock()

@util.deferredLocked('someLock')
def someLockedFunction():
    # ..
    return d
```

buildbot.util.misc.**cancelAfter**(*seconds*, *deferred*)

Parameters

- **seconds** – timeout in seconds
- **deferred** – deferred to cancel after timeout expires

Returns the deferred passed to the function

Cancel the given deferred after the given time has elapsed, if it has not already been fired. Whent his occurs, the deferred's errback will be fired with a twisted.internet.defer.CancelledError failure.

buildbot.util.netstrings

Similar to maildirs, netstrings (http://cr.yp.to/proto/netstrings.txt) are used occasionally in Buildbot to encode data for interchange. While Twisted supports a basic netstring receiver protocol, it does not have a simple way to apply that to a non-network situation.

class buildbot.util.netstrings.**NetstringParser**

This class parses strings piece by piece, either collecting the accumulated strings or invoking a callback for each one.

feed(*data*)

Parameters **data** – a portion of netstring-formatted data

Raises twisted.protocols.basic.NetstringParseError

Add arbitrarily-sized data to the incoming-data buffer. Any complete netstrings will trigger a call to the stringReceived method.

Note that this method (like the Twisted class it is based on) cannot detect a trailing partial netstring at EOF - the data will be silently ignored.

stringReceived(string):

Parameters **string** – the decoded string

This method is called for each decoded string as soon as it is read completely. The default implementation appends the string to the *strings* attribute, but subclasses can do anything.

strings
The strings decoded so far, if stringReceived is not overridden.

buildbot.util.sautils

This module contains a few utilities that are not included with SQLAlchemy.

class buildbot.util.sautils.**InsertFromSelect**(*table*, *select*)

Parameters

- **table** – table into which insert should be performed
- **select** – select query from which data should be drawn

This class is taken directly from SQLAlchemy's compiler.html (http://www.sqlalchemy.org/docs/core/compiler.html#compiling-sub-elements-of-a-custom-expression-construct), and allows a Pythonic representation of INSERT INTO .. SELECT .. queries.

buildbot.util.sautils.**sa_version**()
Return a 3-tuple representing the SQLAlchemy version. Note that older versions that did not have a __version__ attribute are represented by (0,0,0).

buildbot.util.pathmatch

class buildbot.util.pathmatch.**Matcher**
This class implements the path-matching algorithm used by the data API.

Patterns are tuples of strings, with strings beginning with a colon (:) denoting variables. A character can precede the colon to indicate the variable type:

- i specifies an identifier (*identifier*).
- n specifies a number (parseable by int).

A tuple of strings matches a pattern if the lengths are identical, every variable matches and has the correct type, and every non-variable pattern element matches exactly.

A matcher object takes patterns using dictionary-assignment syntax:

```
ep = ChangeEndpoint()
matcher[('change', 'n:changeid')] = ep
```

and performs matching using the dictionary-lookup syntax:

```
changeEndpoint, kwargs = matcher[('change', '13')]
# -> (ep, {'changeid': 13})
```

where the result is a tuple of the original assigned object (the `Change` instance in this case) and the values of any variables in the path.

iterPatterns()
Returns an iterator which yields all patterns in the matcher as tuples of (pattern, endpoint).

buildbot.util.topicmatch

class buildbot.util.topicmatch.**TopicMatcher**(*topics*)

Parameters **topics** (*list*) – topics to match

This class implements the AMQP-defined syntax: routing keys are treated as dot-separated sequences of words and matched against topics. A star (`*`) in the topic will match any single word, while an octothorpe (`#`) will match zero or more words.

matches(*routingKey*)

Parameters **routingKey** (*string*) – routing key to examine

Returns True if the routing key matches a topic

buildbot.util.subscription

The classes in the `buildbot.util.subscription` module are used for master-local subscriptions. In the near future, all uses of this module will be replaced with message-queueing implementations that allow subscriptions and subscribers to span multiple masters.

buildbot.util.croniter

This module is a copy of https://github.com/taichino/croniter, and provides support for converting cron-like time specifications into actual times.

buildbot.util.state

The classes in the `buildbot.util.subscription` module are used for dealing with object state stored in the database.

class buildbot.util.state.**StateMixin**
This class provides helper methods for accessing the object state stored in the database.

name
This must be set to the name to be used to identify this object in the database.

master
This must point to the `BuildMaster` object.

getState(*name*, *default*)

Parameters

- **name** – name of the value to retrieve
- **default** – (optional) value to return if *name* is not present

Returns state value via a Deferred

Raises

- **KeyError** – if *name* is not present and no default is given
- **TypeError** – if JSON parsing fails

Get a named state value from the object's state.

getState(*name*, *value*)

Parameters

- **name** – the name of the value to change
- **value** – the value to set - must be a JSONable object
- **returns** – Deferred

Raises **TypeError** – if JSONification fails

Set a named state value in the object's persistent state. Note that value must be json-able.

buildbot.util.identifiers

This module makes it easy to manipulate identifiers.

buildbot.util.identifiers.**isIdentifier**(*maxLength*, *object*)

Parameters

- **maxLength** – maximum length of the identifier
- **object** – object to test for identifier-ness

Returns boolean

Is object a *identifier*?

buildbot.util.identifiers.**forceIdentifier**(*maxLength*, *str*)

Parameters

- **maxLength** – maximum length of the identifier
- **str** – string to coerce to an identifier

Returns identifer of maximum length `maxLength`

Coerce a string (assuming ASCII for bytestrings) into an identifier. This method will replace any invalid characters with _ and truncate to the given length.

buildbot.util.identifiers.**incrementIdentifier**(*maxLength*, *str*)

Parameters

- **maxLength** – maximum length of the identifier
- **str** – identifier to increment

Returns identifer of maximum length `maxLength`

Raises ValueError if no suitable identifier can be constructed

"Increment" an identifier by adding a numeric suffix, while keeping the total length limited. This is useful when selecting a unique identifier for an object. Maximum-length identifiers like `_999999` cannot be incremented and will raise `ValueError`.

buildbot.util.lineboundaries

class buildbot.util.lineboundaries.**LineBoundaryFinder**

This class accepts a sequence of arbitrary strings and invokes a callback only with complete (newline-terminated) substrings. It buffers any partial lines until a subsequent newline is seen. It considers any of `\r`, `\n`, and `\r\n` to be newlines. Because of the ambiguity of an append operation ending in the character `\r` (it may be a bare `\r` or half of `\r\n`), the last line of such an append operation will be buffered until the next append or flush.

Parameters **callback** – asynchronous function to call with newline-terminated strings

append(*text*)

Parameters **text** – text to append to the boundary finder

Returns Deferred

Add additional text to the boundary finder. If the addition of this text completes at least one line, the callback will be invoked with as many complete lines as possible.

flush()

Returns Deferred

Flush any remaining partial line by adding a newline and invoking the callback.

buildbot.util.service

This module implements some useful subclasses of Twisted services.

The first two classes are more robust implementations of two Twisted classes, and should be used universally in Buildbot code.

class buildbot.util.service.**AsyncMultiService**

This class is similar to `twisted.application.service.MultiService`, except that it handles Deferreds returned from child services `startService` and `stopService` methods.

Twisted's service implementation does not support asynchronous `startService` methods. The reasoning is that all services should start at process startup, with no need to coordinate between them. For Buildbot, this is not sufficient. The framework needs to know when startup has completed, so it can begin scheduling builds. This class implements the desired functionality, with a parent service's `startService` returning a Deferred which will only fire when all child services `startService` methods have completed.

This class also fixes a bug with Twisted's implementation of `stopService` which ignores failures in the `stopService` process. With *AsyncMultiService*, any errors in a child's `stopService` will be propagated to the parent's `stopService` method.

class buildbot.util.service.**AsyncService**

This class is similar to `twisted.application.service.Service`, except that its `setServiceParent` method will return a Deferred. That Deferred will fire after the `startService` method has completed, if the service was started because the new parent was already running.

Some services in buildbot must have only one "active" instance at any given time. In a single-master configuration, this requirement is trivial to maintain. In a multiple-master configuration, some arbitration is required to ensure that the service is always active on exactly one master in the cluster.

For example, a particular daily scheduler could be configured on multiple masters, but only one of them should actually trigger the required builds.

class buildbot.util.service.**ClusteredService**

A base class for a service that must have only one "active" instance in a buildbot configuration.

Each instance of the service is started and stopped via the usual twisted `startService` and `stopService` methods. This utility class hooks into those methods in order to run an arbitration strategy to pick the one instance that should actually be "active".

The arbitration strategy is implemented via a polling loop. When each service instance starts, it immediately offers to take over as the active instance (via `_claimService`).

If successful, the `activate` method is called. Once active, the instance remains active until it is explicitly stopped (eg, via `stopService`) or otherwise fails. When this happens, the `deactivate` method is invoked and the "active" status is given back to the cluster (via `_unclaimService`).

If another instance is already active, this offer fails, and the instance will poll periodically to try again. The polling strategy helps guard against active instances that might silently disappear and leave the service without any active instance running.

Subclasses should use these methods to hook into this activation scheme:

activate()
When a particular instance of the service is chosen to be the one "active" instance, this method is invoked. It is the corollary to twisted's `startService`.

deactivate()
When the one "active" instance must be deactivated, this method is invoked. It is the corollary to twisted's `stopService`.

isActive()
Returns whether this particular instance is the active one.

The arbitration strategy is implemented via the following required methods:

_getServiceId()
The "service id" uniquely represents this service in the cluster. Each instance of this service must have this same id, which will be used in the arbitration to identify candidates for activation. This method may return a Deferred.

_claimService()
An instance is attempting to become the one active instance in the cluster. This method must return *True* or *False* (optionally via a Deferred) to represent whether this instance's offer to be the active one was accepted. If this returns *True*, the `activate` method will be called for this instance.

_unclaimService()
Surrender the "active" status back to the cluster and make it available for another instance. This will only be called on an instance that successfully claimed the service and has been activated and after its `deactivate` has been called. Therefore, in this method it is safe to reassign the "active" status to another instance. This method may return a Deferred.

class buildbot.util.service.**SharedService**
This class implements a generic Service that needs to be instantiated only once according to its parameters. It is a common use case to need this for accessing remote services. Having a shared service allows to limit the number of simultaneous access to the same remote service. Thus, several completely independent Buildbot services can use that *SharedService* to access the remote service, and automatically synchronize themselves to not overwhelm it.

__init__(*self*, **args*, ***kwargs*)
Constructor of the service.

Note that unlike *BuildbotService*, *SharedService* is not reconfigurable and uses the classical constructor method.

Reconfigurability would mean to add some kind of reference counting of the users, which will make the design much more complicated to use. This means that the SharedService will not be destroyed when there is

no more users, it will be destroyed at the master's stopService It is important that those *SharedService* life cycles are properly handled. Twisted will indeed wait for any thread pool to finish at master stop, which will not happen if the thread pools are not properly closed.

The lifecycle of the SharedService is the same as a service, it must implement startService and stopService in order to allocate and free its resources.

getName (*cls*, **args*, ***kwargs*)

Class method. Takes same arguments as the constructor of the service. Get a unique name for that instance of a service. This returned name is the key inside the parent's service dictionary that is used to decide if the instance has already been created before or if there is a need to create a new object. Default implementation will hash args and kwargs and use `<classname>_<hash>` as the name.

getService (*cls*, *parentService*, **args*, ***kwargs*)

Parameters **parentService** – an *AsyncMultiService* where to lookup and register the *SharedService* (usually the root service, the master)

Returns instance of the service via Deferred

Class method. Takes same arguments as the constructor of the service (plus the *parentService* at the beginning of the list). Construct an instance of the service if needed, and place it at the beginning of the *parentService* service list. Placing it at the beginning will guarantee that the *SharedService* will be stopped after the other services.

class buildbot.util.service.**BuildbotService**

This class is the combinations of all *Service* classes implemented in buildbot. It is Async, MultiService, and Reconfigurable, and designed to be eventually the base class for all buildbot services. This class makes it easy to manage (re)configured services.

The design separate the check of the config and the actual configuration/start. A service sibling is a configured object that has the same name of a previously started service. The sibling configuration will be used to configure the running service.

Service lifecycle is as follow:

- Buildbot master start
- Buildbot is evaluating the configuration file. BuildbotServices are created, and checkConfig() are called by the generic constructor.
- If everything is fine, all services are started. BuildbotServices startService() is called, and call reconfigService() for the first time.
- User reconfigures buildbot.
- Buildbot is evaluating the configuration file. BuildbotServices siblings are created, and checkConfig() are called by the generic constructor.
- BuildbotServiceManager is figuring out added services, removed services, unchanged services
- BuildbotServiceManager calls stopService() for services that disappeared from the configuration.
- BuildbotServiceManager calls startService() like in buildbot start phase for services that appeared from the configuration.
- BuildbotServiceManager calls reconfigService() for the second time for services that have their configuration changed.

__init__ (*self*, **args*, ***kwargs*)

Constructor of the service. The constructor initialize the service, and store the config arguments in private attributes.

This should *not* be overridden by subclasses, as they should rather override checkConfig.

checkConfig (*self*, **args*, ***kwargs*)
Please override this method to check the parameters of your config. Please use *buildbot.config.error* for error reporting. You can replace them `*args, **kwargs` by actual constructor like arguments with default args, and it have to match self.reconfigService This method is synchronous, and executed in the context of the master.cfg. Please don't block, or use deferreds in this method. Remember that the object that runs checkConfig is not always the object that is actually started. The checked configuration can be passed to another sibling service. Any actual resource creation shall be handled in reconfigService() or startService()

reconfigService (*self*, **args*, ***kwargs*)
This method is called at buildbot startup, and buildbot reconfig. **args* and ***kwargs* are the configuration arguments passed to the constructor in master.cfg. You can replace `them *args, **kwargs` by actual constructor like arguments with default args, and it have to match self.checkConfig

Returns a deferred that should fire when the service is ready. Builds are not started until all services are configured.

BuildbotServices must be aware that during reconfiguration, their methods can still be called by running builds. So they should atomically switch old configuration and new configuration, so that the service is always available.

If this method raises `NotImplementedError`, it means the service is legacy, and do not support reconfiguration. The `BuildbotServiceManager` parent, will detect this, and swap old service with new service. This behaviour allow smooth transition of old code to new reconfigurable service lifecycle but shall not be used for new code.

reconfigServiceWithSibling (*self*, *sibling*)
Internal method that finds the configuration bits in a sibling, an object with same class that is supposed to replace it from a new configuration. We want to reuse the service started at master startup and just reconfigure it. This method handles necessary steps to detect if the config has changed, and eventually call self.reconfigService()

Advanced users can derive this class to make their own services that run inside buildbot, and follow the application lifecycle of buildbot master.

Such services are singletons accessible in nearly every objects of buildbot (buildsteps, status, changesources, etc) using self.master.namedServices['<nameOfYourService'].

As such, they can be used to factorize access to external services, available e.g using a REST api. Having a single service will help with caching, and rate-limiting access of those APIs.

Here is an example on how you would integrate and configure a simple service in your *master.cfg*:

```
class MyShellCommand(ShellCommand):

    def getResultSummary(self):
        # access the service attribute
        service = self.master.namedServices['myService']
        return dict(step=u"arg value: %d" % (service.arg1,))

class MyService(BuildbotService):
    name = "myService"

    def checkConfig(self, arg1):
        if not isinstance(arg1, int):
            config.error("arg1 must be an integer while it is %r" % (arg1,))
            return
        if arg1 < 0:
            config.error("arg1 must be positive while it is %d" % (arg1,))
```

```
    def reconfigService(self, arg1):
        self.arg1 = arg1
        return defer.succeed(None)

c['schedulers'] = [
    ForceScheduler(
        name="force",
        builderNames=["testy"])]

f = BuildFactory()
f.addStep(MyShellCommand(command='echo hei'))
c['builders'] = [
    BuilderConfig(name="testy",
                  workernames=["local1"],
                  factory=f)]

c['services'] = [
    MyService(arg1=1)
]
```

buildbot.util.httpclientservice

class buildbot.util.service.**HTTPClientService**

This class implements a SharedService for doing http client access. The module automatically chooses from txrequests (https://pypi.python.org/pypi/txrequests) and treq (https://pypi.python.org/pypi/treq) and uses whichever is installed. It provides minimalistic API similar to the one from txrequests (https://pypi.python.org/pypi/txrequests) and treq (https://pypi.python.org/pypi/treq). Having a SharedService for this allows to limits the number of simultaneous connection for the same host. While twisted application can managed thousands of connections at the same time, this is often not the case for the services buildbot controls. Both txrequests (https://pypi.python.org/pypi/txrequests) and treq (https://pypi.python.org/pypi/treq) use keep-alive connection polling. Lots of HTTP REST API will however force a connection close in the end of a transaction.

Note: The API described here is voluntary minimalistic, and reflects what is tested. As most of this module is implemented as a pass-through to the underlying libraries, other options can work but have not been tested to work in both backends. If there is a need for more functionality, please add new tests before using them.

---

static getService(*master*, *base_url*, *auth=None*)

Parameters

- **master** – the instance of the master service (available in self.master for all the *BuildbotService* instances)
- **base_url** – The base http url of the service to access. e.g. `http://github.com/`
- **auth** – Authentication information. If auth is a tuple then `BasicAuth` will be used. e.g `('user', 'passwd')` It can also be a `requests.auth` authentication plugin. In this case txrequests (https://pypi.python.org/pypi/txrequests) will be forced, and treq (https://pypi.python.org/pypi/treq) cannot be used.

Returns instance of :*HTTPClientService*

Get an instance of the SharedService. There is one instance per base_url and auth.

The constructor initialize the service, and store the config arguments in private attributes.

This should *not* be overridden by subclasses, as they should rather override checkConfig.

get(*endpoint*, *params=None*)

Parameters

- **endpoint** – endpoint relative to the base_url (starts with `/`)
- **params** – optional dictionary that will be encoded in the query part of the url (e.g. `?param1=foo`)

Returns implementation of :*IHTTPResponse* via deferred

Performs a HTTP `GET`

delete(*endpoint*, *params=None*)

Parameters

- **endpoint** – endpoint relative to the base_url (starts with `/`)
- **params** – optional dictionary that will be encoded in the query part of the url (e.g. `?param1=foo`)

Returns implementation of :*IHTTPResponse* via deferred

Performs a HTTP `DELETE`

post(*endpoint*, *data=None*, *json=None*, *params=None*)

Parameters

- **endpoint** – endpoint relative to the base_url (starts with `/`)
- **data** – optional dictionary that will be encoded in the body of the http requests as `application/x-www-form-urlencoded`
- **json** – optional dictionary that will be encoded in the body of the http requests as `application/json`
- **params** – optional dictionary that will be encoded in the query part of the url (e.g. `?param1=foo`)

Returns implementation of :*IHTTPResponse* via deferred

Performs a HTTP `POST`

Note: json and data cannot be used at the same time.

---

put(*endpoint*, *data=None*, *json=None*, *params=None*)

Parameters

- **endpoint** – endpoint relative to the base_url (starts with `/`)
- **data** – optional dictionary that will be encoded in the body of the http requests as `application/x-www-form-urlencoded`
- **json** – optional dictionary that will be encoded in the body of the http requests as `application/json`
- **params** – optional dictionary that will be encoded in the query part of the url (e.g. `?param1=foo`)

Returns implementation of :*IHTTPResponse* via deferred

Performs a HTTP `PUT`

Note: json and data cannot be used at the same time.

class `buildbot.util.service.`**IHTTPResponse**

Note: *IHTTPResponse* is a subset of treq (https://pypi.python.org/pypi/treq) `Response` API described here (https://treq.readthedocs.io/en/latest/api.html#module-treq.response) The API it is voluntarily minimalistic and reflects what is tested and reliable to use with the three backends (including fake). The api is a subset of the treq (https://pypi.python.org/pypi/treq) API, which is itself a superset of twisted IResponse API (https://twistedmatrix.com/documents/current/api/twisted.web.iweb.IResponse.html). treq (https://pypi.python.org/pypi/treq) is thus implemented as passthrough.

Notably:

- There is no api to automatically decode content, as this is not implemented the same in both backends.
- There is no api to stream content as the two libraries have very different way for doing it, and we do not see use-case where buildbot would need to transfer large content to the master.

content()

Returns raw (`bytes`) content of the response via deferred

json()

Returns json decoded content of the response via deferred

code

Returns http status code of the request's response (e.g 200)

buildbot.test.fake.httpclientservice

class `buildbot.util.service.`**HTTPClientService**

This class implements a fake version of the `buildbot.util.httpclientservice.HTTPClientService` that needs to be used for testing services which needs http client access. It implements the same APIs as `buildbot.util.httpclientservice.HTTPClientService`, plus one that should be used to register the expectations. It should be registered by the test case before the tested service actually requests an HTTPClientService instance, with the same parameters. It will then replace the original implementation automatically (no need to patch anything). The testing methodology is based on AngularJS ngMock (https://docs.angularjs.org/api/ngMock/service/$httpBackend).

getFakeService(cls, master, case, *args, **kwargs):

Parameters

- **master** – the instance of a fake master service
- **case** – a `twisted.python.unittest.TestCase` instance

`getFakeService` returns a fake *HTTPClientService*, and should be used in place of *getService*.

on top of *getService* it will make sure the original *HTTPClientService* is not called, and assert that all expected http requests have been described in the test case.

expect(self, method, ep, params=None, data=None, json=None, code=200,

content=None, content_json=None):

Parameters

- **method** – expected HTTP method
- **ep** – expected endpoint
- **params** – optional expected query parameters
- **data** – optional expected non-json data (bytes)
- **json** – optional expected json data (dictionary or list or string)
- **code** – optional http code that will be received
- **content** – optional content that will be received
- **content_json** – optional content encoded in json that will be received

Records an expectation of HTTP requests that will happen during the test. The order of the requests is important. All the request expectation must be defined in the test.

example:

```
from twisted.internet import defer
from twisted.trial import unittest

from buildbot.test.fake import httpclientservice as fakehttpclientservice
from buildbot.util import httpclientservice
from buildbot.util import service


class myTestedService(service.BuildbotService):
    name = 'myTestedService'

    @defer.inlineCallbacks
    def reconfigService(self, baseurl):
        self._http = yield httpclientservice.HTTPClientService.
↪getService(self.master, baseurl)

    @defer.inlineCallbacks
    def doGetRoot(self):
        res = yield self._http.get("/")
        # note that at this point, only the http response headers are received
        if res.code != 200:
            raise Exception("%d: server did not succeed" % (res.code))
        res_json = yield res.json()
        # res.json() returns a deferred to account for the time needed to␣
↪fetch the entire body
        defer.returnValue(res_json)


class Test(unittest.SynchronousTestCase):

    def setUp(self):
        baseurl = 'http://127.0.0.1:8080'
        self.parent = service.MasterService()
        self._http = self.successResultOf(fakehttpclientservice.
↪HTTPClientService.getFakeService(
            self.parent, self, baseurl))
        self.tested = myTestedService(baseurl)
```

```
        self.successResultOf(self.tested.setServiceParent(self.parent))
        self.successResultOf(self.parent.startService())

    def test_root(self):
        self._http.expect("get", "/", content_json={'foo': 'bar'})

        response = self.successResultOf(self.tested.doGetRoot())
        self.assertEqual(response, {'foo': 'bar'})

    def test_root_error(self):
        self._http.expect("get", "/", content_json={'foo': 'bar'}, code=404)

        response = self.failureResultOf(self.tested.doGetRoot())
        self.assertEqual(response.getErrorMessage(), '404: server did not␣
↪succeed')
```

buildbot.util.ssl

This module is a copy of `twisted.internet.ssl` except it won't crash with `ImportError` if `pyopenssl` is not installed. If you need to use `twisted.internet.ssl`, please instead use `buildbot.util.ssl`, and call `ssl.ensureHasSSL` in `checkConfig` to provide helpful message to the user, only if he enabled SSL for your plugin.

`buildbot.util.service.`**`ensureHasSSL`**(*plugin_name*)

Parameters **`plugin_name`** – name of the plugin. Usually `self.__class__.__name__`

Call this function to provide helpful config error to the user in case of `OpenSSL` not installed.

`buildbot.util.service.`**`skipUnless`**(*f*)

Parameters **`f`** – decorated test

Test decorator which will skip the test if `OpenSSL` is not installed.

Build Result Codes

Buildbot represents the status of a step, build, or buildset using a set of numeric constants. From Python, these constants are available in the module `buildbot.process.results`, but the values also appear in the database and in external tools, so the values are fixed.

`buildbot.process.results.`**`SUCCESS`**
Value: 0; color: green; a successful run.

`buildbot.process.results.`**`WARNINGS`**
Value: 1; color: orange; a successful run, with some warnings.

`buildbot.process.results.`**`FAILURE`**
Value: 2; color: red; a failed run, due to problems in the build itself, as opposed to a Buildbot misconfiguration or bug.

`buildbot.process.results.`**`SKIPPED`**
Value: 3; color: white; a run that was skipped – usually a step skipped by `doStepIf` (see *Common Parameters*)

`buildbot.process.results.`**`EXCEPTION`**
Value: 4; color: purple; a run that failed due to a problem in Buildbot itself.

`buildbot.process.results.`**`RETRY`**
Value: 4; color: purple; a run that should be retried, usually due to a worker disconnection.

buildbot.process.results.**CANCELLED**
Value: 5; color: pink; a run that was cancelled by the user.

buildbot.process.results.**Results**
A dictionary mapping result codes to their lowercase names.

buildbot.process.results.**worst_status**(*a*, *b*)
This function takes two status values, and returns the "worst" status of the two. This is used to aggregate step statuses into build statuses, and build statuses into buildset statuses.

computeResultAndTermination(obj, result, previousResult):

Parameters

- **obj** – an object with the attributes of *ResultComputingConfigMixin*
- **result** – the new result
- **previousResult** – the previous aggregated result

Building on *worst_status*, this function determines what the aggregated overall status is, as well as whether the attempt should be terminated, based on the configuration in obj.

class buildbot.process.results.**ResultComputingConfigMixin**
This simple mixin is intended to help implement classes that will use computeResultAndTermination. The class has, as class attributes, the result computing configuration parameters with default values:

haltOnFailure

flunkOnWarnings

flunkOnFailure

warnOnWarnings

warnOnFailure

The names of these attributes are available in the following attribute:

resultConfig

WWW Server

History and Motivation

One of the goals of the 'nine' project is to rework Buildbot's web services to use a more modern, consistent design and implement UI features in client-side JavaScript instead of server-side Python.

The rationale behind this is that a client side UI relieves pressure on the server while being more responsive for the user. The web server only concentrates on serving data via a REST interface wrapping the *Data API*. This removes a lot of sources of latency where, in previous versions, long synchronous calculations were made on the server to generate complex pages.

Another big advantage is live updates of status pages, without having to poll or reload. The new system uses Comet techniques in order to relay Data API events to connected clients.

Finally, making web services an integral part of Buildbot, rather than a status plugin, allows tighter integration with the rest of the application.

Design Overview

The `www` service exposes three pieces via HTTP:

- A REST interface wrapping *Data API*;
- HTTP-based messaging protocols wrapping the *Messaging and Queues* interface; and
- Static resources implementing the client-side UI.

The REST interface is a very thin wrapper: URLs are translated directly into Data API paths, and results are returned directly, in JSON format. It is based on JSON API (http://jsonapi.org/). Control calls are handled with a simplified form of JSONRPC 2.0 (http://www.jsonrpc.org/specification).

The message interface is also a thin wrapper around Buildbot's MQ mechanism. Clients can subscribe to messages, and receive copies of the messages, in JSON, as they are received by the buildmaster.

The client-side UI is an AngularJS application. Buildbot uses the Python setuptools entry-point mechanism to allow multiple packages to be combined into a single client-side experience. This allows frontend developers and users to build custom components for the web UI without hacking Buildbot itself.

Python development and AngularJS development are very different processes, requiring different environment requirements and skillsets. To maximize hackability, Buildbot separates the two cleanly. An experienced AngularJS hacker should be quite comfortable in the www/ (https://github.com/buildbot/buildbot/blob/master/www/) directory, with a few exceptions described below. Similarly, an experienced Python hacker can simply download the pre-built web UI (from pypi!) and never venture near the www/ (https://github.com/buildbot/buildbot/blob/master/www/) directory.

URLs

The Buildbot web interface is rooted at its base URL, as configured by the user. It is entirely possible for this base URL to contain path components, e.g., `http://build.example.org/buildbot/`, if hosted behind an HTTP proxy. To accomplish this, all URLs are generated relative to the base URL.

Overall, the space under the base URL looks like this:

- `/` – The HTML document that loads the UI
- `/api/v{version}` – The root of the REST APIs, each versioned numerically. Users should, in general, use the latest version.
- `/ws` – The WebSocket endpoint to subscribe to messages from the mq system.
- `/sse` – The server sent event (http://en.wikipedia.org/wiki/Server-sent_events) endpoint where clients can subscribe to messages from the mq system.

REST API

Rest API is described in its own section.

Server-Side Session

The web server keeps a session state for each user, keyed on a session cookie. This session is available from `request.getSession()`, and data is stored as attributes. The following attributes may be available:

`user_info` A dictionary maintained by the *authentication subsystem*. It may have the following information about the logged-in user:

- `username`

- email
- full_name
- groups (a list of group names)

As well as additional fields specific to the user info implementation.

The contents of the user_info dictionary are made available to the UI as config.user.

Message API

Currently messages are implemented with two protocols: WebSockets and server sent event (http://en.wikipedia.org/wiki/Server-sent_events). This may be supplemented with other mechanisms before release.

WebSocket

WebSocket is a protocol for arbitrary messaging to and from browser. As an HTTP extension, the protocol is not yet well supported by all HTTP proxy technologies. Although, it has been reported to work well used behind the https protocol. Only one WebSocket connection is needed per browser.

Client can connect using url ws[s]://<BB_BASE_URL>/ws

The protocol used is a simple in-house protocol based on json. Structure of a command from client is as following:

```
{ "cmd": "<command name>", '_id': <id of the command>, "arg1": arg1, "arg2": arg2 }
```

- cmd is use to reference a command name
- _id is used to track the response, can be any unique number or string. Generated by the client. Needs to be unique per websocket session.

Response is sent asynchronously, reusing _id to track which command is responded.

Success answer example would be:

```
{ "msg": "OK", '_id': 1, code=200 }
```

Error answer example would be:

```
{"_id":1,"code":404,"error":"no such command \'poing\'"}
```

Client can send several command without waiting response.

Responses are not guaranteed to be sent in order.

Several command are implemented:

ping

```
{"_id":1,"cmd":"ping"}
```

server will respond with a "pong" message:

```
{"_id":1,"msg":"pong","code":200}
```

startConsuming start consuming events that match path. path are described in the *Messaging and Queues* section. For size optimization reasons, path are encoded joined with "/", and with None wildcard replaced by '*'.

```
{"_id":1,"cmd":"startConsuming", "path": "change/*/*"}
```

Success answer example will be:

```
{ "msg": "OK", '_id': 1, code=200 }
```

stopConsuming stop consuming events that was previously registered with `path`.

```
{"_id":1,"cmd":"stopConsuming", "path": "change/*/*"}
```

Success answer example will be:

```
{ "msg": "OK", '_id': 1, code=200 }
```

Client will receive events as websocket frames encoded in json with following format:

```
{"k":key,"m":message}
```

Server Sent Events

SSE is a simpler protocol than WebSockets and is more REST compliant. It uses the chunk-encoding HTTP feature to stream the events. SSE also does not works well behind enterprise proxy, unless you use the https protocol

Client can connect using following endpoints

- `http[s]://<BB_BASE_URL>/sse/listen/<path>`: Start listening to events on the http connection. Optionally setup a first event filter on `<path>`. The first message send is a handshake, giving a uuid that can be used to add or remove event filters.
- `http[s]://<BB_BASE_URL>/sse/add/<uuid>/<path>`: Configure a sse session to add an event filter
- `http[s]://<BB_BASE_URL>/sse/remove/<uuid>/<path>`: Configure a sse session to remove an event filter

Note that if a load balancer is setup as a front end to buildbot web masters, the load balancer must be configured to always use the same master given a client ip address for /sse endpoint.

Client will receive events as sse events, encoded with following format:

```
event: event
data: {'key': <key>, 'message': <message>}
```

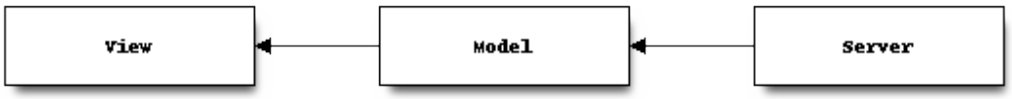
The first event received is a handshake, and is used to inform the client about uuid to use for configuring additional event filters

```
event: handshake
data: <uuid>
```

Javascript Data Module

Data_module is a reusable angularJS module used to access buildbot's data api from the browser. Its main purpose is to handle the 3 way binding.

2 way binding is the angular MVVM concept, which seemlessly synchronise the view and the model. Here we introduce an additional way of synchronisation, which is from the server to the model.



We use the message queue and the websocket interfaces to maintain synchronisation between the server and the client.

The client application just needs to query needed data using a highlevel API, and the data module is using the best approach to make the data always up to date.

Once the binding is setup by the controller, everything is automatically up-to-date.

Base Concepts

Collections

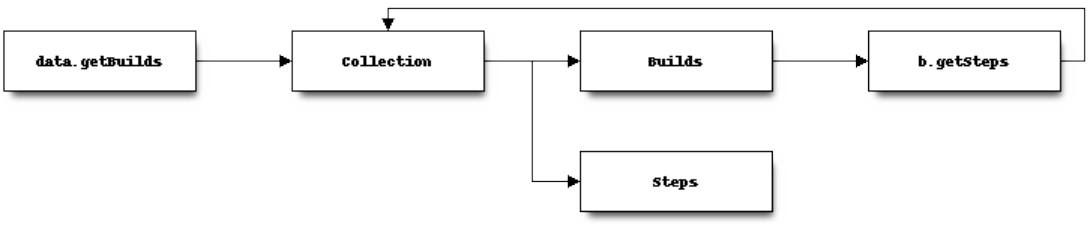
All the data you can get are Collections. Even a query to a single resource returns a collection. A collection is an Array subclass which has extra capabilities:

- It listen to the event stream and is able to maintain itself up-to-date
- It implements client side query in order to garantee up-to-date filtering, ordering and limiting queries.
- It has a fast access to each item it contains via its id.
- It has its own event handlers so that client code can react when the Collection is changing

Wrappers

Each data type contain in a collection is wrapped in a javascript object. This allows to create some custom enhancements to the data model. For example the Change wrapper decodes the author name and email from the "author" field.

Each wrapper class also has specific access method, which allow to access more data from the REST hierachy.



Installation

The Data module is available as a standalone AngularJS module. Installation via bower:

```
bower install buildbot-data --save
```

It is recommended however for coherency to use the installation method via guanlecoja, in the bower section of guanlecoja/config.coffee:

```
'bower':
    'deps':
        'buildbot-data':
            version: '~1.0.14'
            files: 'dist/buildbot-data.js'
```

Inject the `bbData` module to your application:

```
angular.module('myApp', ['bbData'])
```

Service API

DataService

DataService is the service used for accessing the Buildbot data API. It has a modern interface for accessing data in such a way that the updating of the data via web socket is transparent.

Methods:

`.open()`: returns a DataAccessor, which handles 3 way data binding

- open a new accessor every time you need updating data in a controller
- it registers on $destroy event on the scope, and thus automatically unsubscribes from updates when the data is not used anymore.

```
# open a new accessor every time you need updating data in a controller
class DemoController extends Controller
    constructor: ($scope, dataService) ->
        # automatically closes all the bindings when the $scope is destroyed
        data = dataService.open().closeOnDestroy($scope)

        # request new data, it updates automatically
        @builders = data.getBuilders(limit: 10, order: '-started_at')
```

`.getXs([id], [query])`: returns Collection which will eventually contain all the requested data

- it's highly advised to use these instead of the lower level `.get('string')` function
- `Xs` can be the following: `Builds`, `Builders`, `Buildrequests`, `Buildsets`, `Workers`, `Changes`, `Changesources`, `Forceschedulers`, `Masters`, `Schedulers`, `Sourcestamps`
- The collections returns without using an accessor are not automatically updated. So use those methods only when you know the data are not changing

```
# assign builds to $scope.builds and then load the steps when the builds are␣
↪discovered
# onNew is called at initial load
$scope.builds = dataService.getBuilds(builderid: 1)
$scope.builds.onNew = (build) ->
    build.loadSteps()
```

`.get(endpoint, [id], [query])`: returns a <Collection>, when the promise is resolved, the Collection contains all the requested data

```
# assign builds to $scope.builds once the Collection is filled
builderid = 1
$scope.builds = dataService.get("builders/#{builderid}/builds", limit: 1)
$scope.builds.onNew = (build) ->
    build.loadSteps()
```

```
# assign builds to $scope.builds before the Collection is filled using the .
↪getArray() function
$scope.builds = dataService.get('builds', builderid: 1)
```

`.control(url, method, [params])`: returns a promise, sends a JSON RPC2 POST request to the server

```
# open a new accessor every time you need updating data in a controller
dataService.control('forceschedulers/force', 'force').then (response) ->
    $log.debug(response)
, (reason) ->
    $log.error(reason)
```

DataAccessor

DataAccessor object is returned by the dataService.open() method.

Methods:

`.closeOnDestroy($scope)`: registers scope destruction as waterfall destruction for all collection accessed via this accessor.

`.close()`: Destruct all collections previously accessed via this accessor. Destroying a collection means it will unsubscribe to any events necessary to maintain it up-to-date.

`.getXs([id], [query])`: returns Collection which will eventually contain all the requested data Same methods as in the dataService, except here the data will be maintained up-to-date.

Collections

`.get(id)`: returns one element of the collection by id, or undefined, if this id is unknown to the collection. This method does not do any network access, and thus only know about data already fetched.

`.hasOwnProperty(id)`: returns true if this id is known by this collection.

`.close()`: forcefully unsubscribe this connection from auto-update. Normally, this is done automatically on scope destruction, but sometimes, when you got enough data, you want to save bandwidth and disconnect the collection.

`.put(object)`: insert one plain object to the collection. As an external API, this method is only useful for the unit tests to simulate new data coming asynchronously.

`.from(object_list)`: insert several plain objects to the collection. This method is only useful for the unit tests to simulate new data coming asynchronously.

`.onNew = (object) ->`: Callback method which is called when a new object arrives in the collection. This can be called either when initial data is coming via REST api, or when data is coming via the event stream. The affected object is given in parameter. *this* context is the collection.

`.onUpdate = (object) ->`: Callback method which is called when an object is modified. This is called when data is coming via the event stream. The affected object is given in parameter. *this* context is the collection.

`.onChange = (collection) ->`: Callback method which is called when an object is modified. This is called when data is coming via the event stream. *this* context is the collection. The full collection is given in parameter (in case you override `this` via fat arrow).

Wrapper

Wrapper objects are objects stored in the collection. Those objects have specific methods, depending on their types. methods:

`.getXs([id], [query])`: returns a Collection, when the promise is resolved, the Collection contains all the requested data

- same as `dataService.getXs`, but with relative endpoint

```
# assign builds to $scope.builds once the Collection is filled
$scope.builds = dataService.getBuilds(builderid: 1)
$scope.builds.onNew = (b) ->
    b.complete_steps = b.getSteps(complete:true)
    b.running_steps = b.getSteps(complete:false)
```

`.loadXs([id], [query])`: returns a Collection, the Collection contains all the requested data, which is also assigned to `o.Xs`

- `o.loadXs()` is equivalent to `o.xs = o.getXs()`

```
# get builder with id = 1
dataService.getBuilders(1).onNew = (builder) ->
    # load all builds in builder.builds
    builder.loadBuilds().onNew (build) ->
        # load all buildsteps in build.steps
        build.loadSteps()
```

`.control(method, params)`: returns a promise, sends a JSON RPC2 POST request to the server

Base web application

JavaScript Application

The client side of the web UI is written in JavaScript and based on the AngularJS framework and concepts.

This is a Single Page Application" (http://en.wikipedia.org/wiki/Single-page_application) All Buildbot pages are loaded from the same path, at the master's base URL. The actual content of the page is dictated by the fragment in the URL (the portion following the # character). Using the fragment is a common JS techique to avoid reloading the whole page over HTTP when the user changes the URI or clicks a link.

AngularJS

The best place to learn about AngularJS is its own documentation (http://docs.angularjs.org/guide/),

AngularJS strong points are:

- A very powerful MVC system (http://docs.angularjs.org/guide/concepts) allowing automatic update of the UI, when data changes
- A Testing Framework and philosophy (http://docs.angularjs.org/guide/dev_guide.e2e-testing)

- A deferred system (http://docs.angularjs.org/api/ng.$q) similar to the one from Twisted.
- A fast growing community and ecosystem (http://builtwith.angularjs.org/)

On top of Angular we use nodeJS tools to ease development

- gulp buildsystem, seemlessly build the app, can watch files for modification, rebuild and reload browser in dev mode. In production mode, the buildsystem minifies html, css and js, so that the final app is only 3 files to download (+img).
- coffeescript (http://coffeescript.org/), a very expressive langage, preventing some of the major traps of JS.
- jade template langage (http://jade-lang.com/), adds syntax sugar and readbility to angular html templates.
- Bootstrap (http://getbootstrap.com/) is a css library providing know good basis for our styles.
- Font Awesome (http://fortawesome.github.com/Font-Awesome/) is a coherent and large icon library

modules we may or may not want to include:

- momentjs (http://momentjs.com/) is a library implementing human readable relative timings (e.g. "one hour ago")
- ngGrid (https://angular-ui.github.io/ui-grid/) is a grid system for full featured searcheable/sortable/csv exportable grids
- angular-UI (http://angular-ui.github.com/) is a collection of jquery based directives and filters. Probably not very useful for us
- JQuery (http://jquery.com/) the well known JS framework, allows all sort of dom manipulation. Having it inside allows for all kind of hacks we may want to avoid.

Extensibility

Buildbot UI is designed for extensibility. The base application should be pretty minimal, and only include very basic status pages. Base application cannot be disabled so any page not absolutely necessary should be put in plugins. You can also completely replace the default application by another application more suitable to your needs. The `md_base` application is an example rewrite of the app using material design libraries.

Some Web plugins are maintained inside buildbot's git repository, but this is absolutely not necessary. Unofficial plugins are encouraged, please be creative!

Please look at official plugins for working samples.

Typical plugin source code layout is:

```
setup.py                    # standard setup script. Most plugins should use the␣
↪same boilerplate, which helps building guanlecoja app as part of the setup. Minimal␣
↪adaptation is needed
<pluginname>/__init__.py    # python entrypoint. Must contain an "ep" variable of␣
↪type buildbot.www.plugin.Application. Minimal adaptation is needed
guanlecoja/config.coffee    # Configuration for guanlecoja. Few changes are needed␣
↪here. Please see guanlecoja docs for details.
src/..                      # source code for the angularjs application. See␣
↪guanlecoja doc for more info of how it is working.
package.json                # declares npm dependency. normallly, only guanlecoja is␣
↪needed. Typically, no change needed
gulpfile.js                 # entrypoint for gulp, should be a one line call to␣
↪guanlecoja. Typically, no change needed
MANIFEST.in                 # needed by setup.py for sdist generation. You need to␣
↪adapt this file to match the name of your plugin
```

Plugins are packaged as python entry-points for the `buildbot.www` namespace. The python part is defined in the *buildbot.www.plugin* module. The entrypoint must contain a twisted.web Resource, that is populated in the web server in */<pluginname>/*.

The front-end part of the plugin system automatically loads */<pluginname>/scripts.js* and */<pluginname>/styles.css* into the angular.js application. The scripts.js files can register itself as a dependency to the main "app" module, register some new states to $stateProvider, or new menu items via glMenuProvider.

The entrypoint containing a Resource, nothing forbids plugin writers to add more REST apis in */<pluginname>/api*. For that, a reference to the master singleton is provided in `master` attribute of the Application entrypoint. You are even not restricted to twisted, and could even load a wsgi application using flask, django, etc (http://twistedmatrix.com/documents/13.1.0/web/howto/web-in-60/wsgi.html).

Routing

AngularJS uses router to match URL and choose which page to display. The router we use is ui.router. Menu is managed by guanlecoja-ui's glMenuProvider. Please look at ui.router, and guanlecoja-ui documentation for details.

Typically, a route regitration will look like following example.

```
# ng-classify declaration. Declares a config class
class State extends Config
    # Dependancy injection: we inject $stateProvider and glMenuServiceProvider
    constructor: ($stateProvider, glMenuServiceProvider) ->

        # Name of the state
        name = 'console'

        # Menu configuration.
        glMenuServiceProvider.addGroup
            name: name
            caption: 'Console View'     # text of the menu
            icon: 'exclamation-circle'  # icon, from Font-Awesome
            order: 5                    # order in the menu, as menu are declared in␣
↪several places, we need this to control menu order

        # Configuration for the menu-item, here we only have one menu item per menu,␣
↪glMenuProvider won't create submenus
        cfg =
            group: name
            caption: 'Console View'

        # Register new state
        state =
            controller: "#{name}Controller"
            controllerAs: "c"
            templateUrl: "console_view/views/#{name}.html"
            name: name
            url: "/#{name}"
            data: cfg

        $stateProvider.state(state)
```

Directives

We use angular directives as much as possible to implement reusable UI components.

Linking with Buildbot

A running buildmaster needs to be able to find the JavaScript source code it needs to serve the UI. This needs to work in a variety of contexts - Python development, JavaScript development, and end-user installations. To accomplish this, the gulp build process finishes by bundling all of the static data into a Python distribution tarball, along with a little bit of Python glue. The Python glue implements the interface described below, with some care taken to handle multiple contexts.

Hacking Quick-Start

This section describes how to get set up quickly to hack on the JavaScript UI. It does not assume familiarity with Python, although a Python installation is required, as well as `virtualenv`. You will also need `NodeJS`, and `npm` installed.

Prerequisites

Note: Buildbot UI is only tested to build on node 4.x.x.

- Install LTS release of node.js.

  http://nodejs.org/ is a good start for windows and osx

  For Linux, as node.js is evolving very fast, distros versions are often too old, and sometimes distro maintainers make incompatible changes (i.e naming node binary nodejs instead of node) For Ubuntu and other Debian based distros, you want to use following method:

  ```
  curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
  ```

  Please feel free to update this documentation for other distros. Know good source for Linux binary distribution is: https://github.com/nodesource/distributions

- Install gulp globally. Gulp is the build system used for coffeescript development.

  ```
  sudo npm install -g gulp
  ```

Hacking the Buildbot JavaScript

To effectively hack on the Buildbot JavaScript, you'll need a running Buildmaster, configured to operate out of the source directory (unless you like editing minified JS). Start by cloning the project and its git submodules:

```
git clone git://github.com/buildbot/buildbot.git
```

In the root of the source tree, create and activate a virtualenv to install everything in:

```
virtualenv sandbox
source sandbox/bin/activate
```

This creates an isolated Python environment in which you can install packages without affecting other parts of the system. You should see `(sandbox)` in your shell prompt, indicating the sandbox is activated.

Next, install the Buildbot-WWW and Buildbot packages using `--editable`, which means that they should execute from the source directory.

```
pip install --editable pkg
pip install --editable master/
make frontend
```

This will fetch a number of dependencies from pypi, the Python package repository. This will also fetch a bunch a bunch of node.js dependencies used for building the web application, and a bunch of client side js dependencies, with bower

Now you'll need to create a master instance. For a bit more detail, see the Buildbot tutorial (*First Run*).

```
buildbot create-master sandbox/testmaster
mv sandbox/testmaster/master.cfg.sample sandbox/testmaster/master.cfg
buildbot start sandbox/testmaster
```

If all goes well, the master will start up and begin running in the background. As you just installed www in editable mode (aka 'develop' mode), setup.py did build the web site in prod mode, so the everything is minified, making it hard to debug.

When doing web development, you usually run:

```
cd www/base
gulp dev
```

This will compile the base webapp in development mode, and automatically rebuild when files change.

Testing with real data

Front-end only hackers might want to just skip the master and worker setup, and just focus on the UI. It can also be very useful to just try the UI with real data from your production. For those use-cases, `gulp dev proxy` can be used.

This tool is a small nodejs app integrated in the gulp build that can proxy the data and websocket api from a production server to your development environment. Having a proxy is slightly slower, but this can be very useful for testing with real complex data.

You still need to have python virtualenv configured with master package installed, like we described in previous paragraph.

Provided you run it in a buildbot master virtualenv, the following command will start the UI and redirect the api calls to the nine demo server:

```
gulp dev proxy --host nine.buildbot.net
```

You can then just point your browser to localhost:8010, and you will access http://nine.buildbot.net, with your own version of the UI.

Guanlecoja

Buildbot's build environment has been factorized for reuse in other projects and plugins, and is callsed Guanlecoja.

The documentation and meaning of this name is maintained in Guanlecoja's own site. https://github.com/buildbot/guanlecoja/

Testing Setup

buildbot_www uses Karma (http://karma-runner.github.io) to run the coffeescript test suite. This is the official test framework made for angular.js. We don't run the front-end testsuite inside the python 'trial' test suite, because testing python and JS is technically very different.

Karma needs a browser to run the unit test in. It supports all the major browsers. Given our current experience, we did not see any bugs yet that would only happen on a particular browser this is the reason that at the moment, only headless browser "PhantomJS" is used for testing.

We enforce that the tests are run all the time after build. This does not impact the build time by a great factor, and simplify the workflow.

In some case, this might not be desirable, for example if you run the build on headless system, without X. PhantomJS, even if it is headless needs a X server like xvfb. In the case where you are having difficulties to run Phantomjs, you can build without the tests using the command:

```
gulp prod --notests
```

Debug with karma

`console.log` is available via karma. In order to debug the unit tests, you can also use the global variable `dump`, which dumps any object for inspection in the console. This can be handy to be sure that you don't let debug logs in your code to always use `dump`

Material design Base application

`md_base` is a complete replacement for the bootstrap based `base` application. It's UI is based on google's Material Design libraries.

Status

md-base is currently not really usable as the default UI, see the list of issues bug #3397 (http://trac.buildbot.net/ticket/3397).

Authentication

Buildbot's HTTP authentication subsystem supports a rich set of information about users:

- User credentials: Username and proof of ownership of that username.
- User information: Additional information about the user, including
  - email address
  - full name
  - group membership
- Avatar information: a small image to represent the user.

Buildbot's authentication subsystem is designed to support several authentication modes:

- **Simple username/password authentication.** The Buildbot UI prompts for a username and password and the backend verifies them.

- **External authentication by an HTTP Proxy.** An HTTP proxy in front of Buildbot performs the authentication and passes the verified username to Buildbot in an HTTP Header.
- **Authentication by a third-party website.** Buildbot sends the user to another site such as GitHub to authenticate and receives a trustworty assertion of the user's identity from that site.

Implementation

Authentication is implemented by an instance of *AuthBase*. This instance is supplied directly by the user in the configuration file. A reference to the instance is available at `self.master.www.auth`.

Username / Password Authentication

In this mode, the Buildbot UI displays a form allowing the user to specify a username and password. When this form is submitted, the UI makes an AJAX call to `/auth/login` including HTTP Basic Authentication headers. The master verifies the contents of the header and updates the server-side session to indicate a successful login or to contain a failure message. Once the AJAX call is complete, the UI reloads the page, re-fetching `/config.js`, which will include the username or failure message from the session.

Subsequent access is authorized based on the information in the session; the authentication credentials are not sent again.

External Authentication

Buildbot's web service can be run behind an HTTP proxy. Many such proxies can be configured to perform authentication on HTTP connections before forwarding the request to Buildbot. In these cases, the results of the authentication are passed to Buildbot in an HTTP header.

In this mode, authentication proceeds as follows:

- The web browser connects to the proxy, requesting the Buildbot home page
- The proxy negotiates authentication with the browser, as configured
- Once the user is authenticated, the proxy forwards the request goes to the Buildbot web service. The request includes a header, typically `Remote-User`, containing the authenticated username.
- Buildbot reads the header and optionally connects to another service to fetch additional user information about the user.
- Buildbot stores all of the collected information in the server-side session.
- The UI fetches `/config.js`, which includes the user information from the server-side session.

Note that in this mode, the HTTP proxy will send the header with every request, although it is only interpreted during the fetch of `/config.js`.

Kerberos Example

Kerberos is an authentication system which allows passwordless authentication on corporate networks. Users authenticate once on their desktop environment, and the OS, browser, webserver, and corporate directory cooperate in a secure manner to share the authentication to a webserver. This mechanism only takes care about the authentication problem, and no user information is shared other than the username. The kerberos authentication is supported by a Apache front-end in `mod_kerberos`.

Third-Party Authentication

Third-party authentication involves Buildbot redirecting a user's browser to another site to establish the user's identity. Once that is complete, that site redirects the user back to Buildbot, including a cryptographically signed assertion about the user's identity.

The most common implementation of this sort of authentication is oAuth2. Many big internet service companies are providing oAuth2 services to identify their users. Most oAuth2 services provide authentication and user information in the same api.

The following process is used for third-party authentication:

- The web browser connects to buildbot ui
- A session cookie is created, but user is not yet authenticated. The UI adds a widget entitled `Login via GitHub` (or whatever third party is configured)
- When the user clicks on the widget, the UI fetches `/auth/login`, which returns a bare URL on `github.com`. The UI loads that URL in the browser, with an effect similar to a redirect.
- GitHub authenticates the user, if necessary, and requests permission for Buildbot to access the user's information.
- On success, the GitHub web page redirects back to Buildbot's `/auth/login?code=..`, with an authentication code.
- Buildbot uses this code to request more information from GitHub, and stores the results in the server-side session. Finally, Buildbot returns a redirect response, sending the user's browser to the root of the Buildbot UI. The UI code will fetch `/config.js`, which contains the login data from the session.

Logout

A "logout" button is available in the simple and third-party modes. Such a button doesn't make sense for external authentication, since the proxy will immediately re-authenticate the user.

This button fetches `/auth/logout`, which destroys the server-side session. After this point, any stored authentication information is gone and the user is logged out.

Future Additions

- Browserid/Persona: This method is very similar to oauth2, and should be implemented in a similar way (i.e. two stage redirect + token-verify)
- Use the User table in db: This is a very similar to the UserPasswordAuth use cases (form + local db verification). Eventually, this method will require some work on the UI in order to populate the db, add a "register" button, verification email, etc. This has to be done in a ui plugin.

Authorization

Buildbot authorization is designed to address the following requirements

- Most of the configuration is only data: We avoid to require user to write callbacks for most of the use cases. This to allow to load the config from yaml or json and eventually do a UI for authorization config.
- Separation of concerns:
  - Mapping users to roles
  - Mapping roles to REST endpoints.

- Configuration should not need hardcoding endpoint paths.
- Easy to extend

Use cases

- Members of admin group should have access to all resources and actions
- developers can run the "try" builders
- Integrators can run the "merge" builders
- Release team can run the "release" builders
- There are separate teams for different branches or projects, but the roles are identic
- Owners of builds can stop builds or buildrequests
- Secret branch's builds are hidden from people except explicitly authorized

Master-Worker API

This section describes the master-worker interface. It covers communication protocol of "classic" remote Worker, notice that there is other types of workers which behave a bit different, such as *Local Worker* and *Latent Workers*.

Connection

The interface is based on Twisted's Perspective Broker, which operates over TCP connections.

The worker connects to the master, using the parameters supplied to **`buildbot-worker create-worker`**. It uses a reconnecting process with an exponential backoff, and will automatically reconnect on disconnection.

Once connected, the worker authenticates with the Twisted Cred (newcred) mechanism, using the username and password supplied to **`buildbot-worker create-worker`**. The *mind* is the worker bot instance (class `buildbot_worker.pb.BotPb`).

On the master side, the realm is implemented by `buildbot.pbmanager.Dispatcher`, which examines the username of incoming avatar requests. There are special cases for `change`, `debug`, and `statusClient`, which are not discussed here. For all other usernames, the botmaster is consulted, and if a worker with that name is configured, its *`buildbot.worker.Worker`* instance is returned as the perspective.

Workers

At this point, the master-side Worker object has a pointer to the remote, worker-side Bot object in its `self.worker`, and the worker-side Bot object has a reference to the master-side Worker object in its `self.perspective`.

Bot methods

The worker-side Bot object has the following remote methods:

`remote_getCommands` Returns a dictionary for all commands the worker recognizes: the key of the dictionary is command name and command version is the value.

remote_setBuilderList Given a list of builders and their build directories, ensures that those builders, and only those builders, are running. This can be called after the initial connection is established, with a new list, to add or remove builders.

This method returns a dictionary of `WorkerForBuilder` objects - see below.

remote_print Adds a message to the worker logfile.

remote_getWorkerInfo Returns dictionary with the contents of the worker's `info/` directory (i.e. file name is used as key and file contents as the value). This dictionary also contains the keys

environ copy of the workers environment

system OS the worker is running (extracted from Python's `os.name`)

basedir base directory where worker is running

numcpus number of CPUs on the worker, either as configured or as detected (since `buildbot-worker` version 0.9.0)

version worker's version (same as the result of `remote_getVersion` call)

worker_commands worker supported commands (same as the result of `remote_getCommands` call)

remote_getVersion Returns the worker's version.

remote_shutdown Shuts down the worker cleanly.

Worker methods

The master-side object has the following method:

perspective_keepalive Does nothing - used to keep traffic flowing over the TCP connection

Setup

After the initial connection and trading of a mind (`buildbot_worker.pb.BotPb`) for an avatar (Worker), the master calls the Bot's `setBuilderList` method to set up the proper builders on the worker side. This method returns a reference to each of the new worker-side `WorkerForBuilderPb` objects, described below. Each of these is handed to the corresponding master-side `WorkerForBuilder` object.

This immediately calls the remote `setMaster` method, then the `print` method.

Pinging

To ping a remote Worker, the master calls its `print` method.

Building

When a build starts, the master calls the worker's `startBuild` method. Each BuildStep instance will subsequently call the `startCommand` method, passing a reference to itself as the `stepRef` parameter. The `startCommand` method returns immediately, and the end of the command is signalled with a call to a method on the master-side BuildStep object.

Worker For Builders

Each worker has a set of builders which can run on it. These are represented by distinct classes on the master and worker, just like the Worker and Bot objects described above.

On the worker side, builders are represented as instances of the `buildbot_worker.pb.WorkerForBuilderPb` class. On the master side, they are represented by the `buildbot.process.workerforbuilder.WorkerForBuilder` class. The identical names are a source of confusion. The following will refer to these as the worker-side and master-side Worker For Builder classes. Each object keeps a reference to its opposite in `self.remote`.

Worker-Side `WorkerForBuilderPb` Methods

`remote_setMaster` Provides a reference to the master-side Worker For Builder

`remote_print` Adds a message to the worker logfile; used to check round-trip connectivity

`remote_startBuild` Indicates that a build is about to start, and that any subsequent commands are part of that build

`remote_startCommand` Invokes a command on the worker side

`remote_interruptCommand` Interrupts the currently-running command

Master-side `WorkerForBuilder` Methods

The master side does not have any remotely-callable methods.

Commands

Actual work done by the worker is represented on the master side by a *`buildbot.process.remotecommand.RemoteCommand`* instance.

The command instance keeps a reference to the worker-side `buildbot_worker.pb.WorkerForBuilderPb`, and calls methods like `remote_startCommand` to start new commands. Once that method is called, the `WorkerForBuilderPb` instance keeps a reference to the command, and calls the following methods on it:

Master-Side RemoteCommand Methods

`remote_update` Update information about the running command. See below for the format.

`remote_complete` Signal that the command is complete, either successfully or with a Twisted failure.

Updates

Updates from the worker, sent via *`remote_update`*, are a list of individual update elements. Each update element is, in turn, a list of the form `[data, 0]` where the 0 is present for historical reasons. The data is a dictionary, with keys describing the contents. The updates are handled by *`remote_update`*.

Updates with different keys can be combined into a single dictionary or delivered sequentially as list elements, at the worker's option.

To summarize, an `updates` parameter to *`remote_update`* might look like this:

```
[
    [ { 'header' : 'running command..' }, 0 ],
    [ { 'stdout' : 'abcd', 'stderr' : 'local modifications' }, 0 ],
    [ { 'log' : ( 'cmd.log', 'cmd invoked at 12:33 pm\n' ) }, 0 ],
    [ { 'rc' : 0 }, 0 ],
]
```

Defined Commands

The following commands are defined on the workers.

shell

Runs a shell command on the worker. This command takes the following arguments:

`command`

The command to run. If this is a string, will be passed to the system shell as a string. Otherwise, it must be a list, which will be executed directly.

`workdir`

Directory in which to run the command, relative to the builder dir.

`env`

A dictionary of environment variables to augment or replace the existing environment on the worker. In this dictionary, `PYTHONPATH` is treated specially: it should be a list of path components, rather than a string, and will be prepended to the existing Python path.

`initial_stdin`

A string which will be written to the command's standard input before it is closed.

`want_stdout`

If false, then no updates will be sent for stdout.

`want_stderr`

If false, then no updates will be sent for stderr.

`usePTY`

If true, the command should be run with a PTY (POSIX only). This defaults to False.

`not_really`

If true, skip execution and return an update with rc=0.

`timeout`

Maximum time without output before the command is killed.

`maxTime`

Maximum overall time from the start before the command is killed.

`logfiles`

A dictionary specifying logfiles other than stdio. Keys are the logfile names, and values give the workdir-relative filename of the logfile. Alternately, a value can be a dictionary; in this case, the dictionary must have a `filename` key specifying the filename, and can also have the following keys:

`follow`

Only follow the file from its current end-of-file, rather that starting from the beginning.

`logEnviron`

If false, the command's environment will not be logged.

The `shell` command sends the following updates:

`stdout` The data is a bytestring which represents a continuation of the stdout stream. Note that the bytestring boundaries are not necessarily aligned with newlines.

`stderr` Similar to `stdout`, but for the error stream.

`header` Similar to `stdout`, but containing data for a stream of Buildbot-specific metadata.

`rc` The exit status of the command, where – in keeping with UNIX tradition – 0 indicates success and any nonzero value is considered a failure. No further updates should be sent after an `rc`.

`log` This update contains data for a logfile other than stdio. The data associated with the update is a tuple of the log name and the data for that log. Note that non-stdio logs do not distinguish output, error, and header streams.

uploadFile

Upload a file from the worker to the master. The arguments are

`workdir`

The base directory for the filename, relative to the builder's basedir.

`workersrc`

Name of the filename to read from., relative to the workdir.

`writer`

A remote reference to a writer object, described below.

`maxsize`

Maximum size, in bytes, of the file to write. The operation will fail if the file exceeds this size.

`blocksize`

The block size with which to transfer the file.

`keepstamp`

If true, preserve the file modified and accessed times.

The worker calls a few remote methods on the writer object. First, the `write` method is called with a bytestring containing data, until all of the data has been transmitted. Then, the worker calls the writer's `close`, followed (if `keepstamp` is true) by a call to `upload(atime, mtime)`.

This command sends `rc` and `stderr` updates, as defined for the `shell` command.

uploadDirectory

Similar to `uploadFile`, this command will upload an entire directory to the master, in the form of a tarball. It takes the following arguments:

`workdir` `workersrc` `writer` `maxsize` `blocksize`

See `uploadFile`

`compress`

Compression algorithm to use – one of `None`, `'bz2'`, or `'gz'`.

The writer object is treated similarly to the `uploadFile` command, but after the file is closed, the worker calls the master's `unpack` method with no arguments to extract the tarball.

This command sends `rc` and `stderr` updates, as defined for the `shell` command.

downloadFile

This command will download a file from the master to the worker. It takes the following arguments:

`workdir`

Base directory for the destination filename, relative to the builder basedir.

`workerdest`

Filename to write to, relative to the workdir.

`reader`

A remote reference to a reader object, described below.

`maxsize`

Maximum size of the file.

`blocksize`

The block size with which to transfer the file.

`mode`

Access mode for the new file.

The reader object's `read(maxsize)` method will be called with a maximum size, which will return no more than that number of bytes as a bytestring. At EOF, it will return an empty string. Once EOF is received, the worker will call the remote `close` method.

This command sends `rc` and `stderr` updates, as defined for the `shell` command.

mkdir

This command will create a directory on the worker. It will also create any intervening directories required. It takes the following argument:

`dir`

Directory to create.

The `mkdir` command produces the same updates as `shell`.

rmdir

This command will remove a directory or file on the worker. It takes the following arguments:

`dir`

Directory to remove.

`timeout maxTime`

See `shell`, above.

The `rmdir` command produces the same updates as `shell`.

cpdir

This command will copy a directory from place to place on the worker. It takes the following arguments:

`fromdir`

Source directory for the copy operation, relative to the builder's basedir.

`todir`

Destination directory for the copy operation, relative to the builder's basedir.

`timeout maxTime`

See `shell`, above.

The `cpdir` command produces the same updates as `shell`.

stat

This command returns status information about a file or directory. It takes a single parameter, `file`, specifying the filename relative to the builder's basedir.

It produces two status updates:

`stat`

The return value from Python's `os.stat`.

`rc`

0 if the file is found, otherwise 1.

glob

This command finds all pathnames matching a specified pattern that uses shell-style wildcards. It takes a single parameter, `path`, specifying the pattern to pass to Python's `glob.glob` function.

It produces two status updates:

`files`

The list of matching files returned from `glob.glob`

`rc`

0 if the `glob.glob` does not raise exception, otherwise 1.

listdir

This command reads the directory and returns the list with directory contents. It takes a single parameter, `dir`, specifying the directory relative to builder's basedir.

It produces two status updates:

`files`

The list of files in the directory returned from `os.listdir`

`rc`

0 if the `os.listdir` does not raise exception, otherwise 1.

Claiming Build Requests

At Buildbot's core, it is a distributed job (build) scheduling engine. Future builds are represented by build requests, which are created by schedulers.

When a new build request is created, it is added to the `buildrequests` table and an appropriate message is sent.

Distributing

Each master distributes build requests among its builders by examining the list of available build requests, available workers, and accounting for user configuration for build request priority, worker priority, and so on. This distribution process is re-run whenever an event occurs that may allow a new build to start.

Such events can be signalled to master with

- `maybeStartBuildsForBuilder` when a single builder is affected;
- `maybeStartBuildsForWorker` when a single worker is affected; or
- `maybeStartBuildsForAllBuilders` when all builders may be affected.

In particular, when a master receives a new-build-request message, it performs the equivalent of `maybeStartBuildsForBuilder` for the affected builder.

Claiming

If circumstances are right for a master to begin a build, then it attempts to "claim" the build request. In fact, if several build requests were merged, it attempts to claim them as a group, using the `claimBuildRequests` DB method. This method uses transactions and an insert into the `buildrequest_claims` table to ensure that exactly one master succeeds in claiming any particular build request.

If the claim fails, then another master has claimed the affected build requests, and the attempt is abandoned.

If the claim succeeds, then the master sends a message indicating that it has claimed the request. This message can be used by other masters to abandon their attempts to claim this request, although this is not yet implemented.

If the build request is later abandoned (as can happen if, for example, the worker has disappeared), then master will send a message indicating that the request is again unclaimed; like a new-buildrequest message, this message indicates that other masters should try to distribute it once again.

The One That Got Away

The claiming process is complex, and things can go wrong at just about any point. Through master failures or message/database race conditions, it's quite possible for a build request to be "missed", even when resources are available to process it.

To account for this possibility, masters periodically poll the `buildrequests` table for unclaimed requests and try to distribute them. This resiliency avoids "lost" build requests, at the small cost of a polling delay before the requests are scheduled.

String Encodings

Buildbot expects all strings used internally to be valid Unicode strings - not bytestrings.

Note that Buildbot rarely feeds strings back into external tools in such a way that those strings must match. For example, Buildbot does not attempt to access the filenames specified in a Change. So it is more important to store strings in a manner that will be most useful to a human reader (e.g., in logfiles, web status, etc.) than to store them in a lossless format.

Inputs

On input, strings should be decoded, if their encoding is known. Where necessary, the assumed input encoding should be configurable. In some cases, such as filenames, this encoding is not known or not well-defined (e.g., a utf-8 encoded filename in a latin-1 directory). In these cases, the input mechanisms should make a best effort at decoding, and use e.g., the `errors='replace'` option to fail gracefully on un-decodable characters.

Outputs

At most points where Buildbot outputs a string, the target encoding is known. For example, the web status can encode to utf-8. In cases where it is not known, it should be configurable, with a safe fallback (e.g., ascii with `errors='replace'`. For HTML/XML outputs, consider using `errors='xmlcharrefreplace'` instead.

Metrics

New in buildbot 0.8.4 is support for tracking various performance metrics inside the buildbot master process. Currently these are logged periodically according to the `log_interval` configuration setting of the *`metrics`* configuration.

The metrics subsystem is implemented in `buildbot.process.metrics`. It makes use of twisted's logging system to pass metrics data from all over buildbot's code to a central `MetricsLogObserver` object, which is available at `BuildMaster.metrics` or via `Status.getMetrics()`.

Metric Events

`MetricEvent` objects represent individual items to monitor. There are three sub-classes implemented:

`MetricCountEvent` Records incremental increase or decrease of some value, or an absolute measure of some value.

```
from buildbot.process.metrics import MetricCountEvent

# We got a new widget!
MetricCountEvent.log('num_widgets', 1)
```

```
# We have exactly 10 widgets
MetricCountEvent.log('num_widgets', 10, absolute=True)
```

MetricTimeEvent Measures how long things take. By default the average of the last 10 times will be reported.

```
from buildbot.process.metrics import MetricTimeEvent

# function took 0.001s
MetricTimeEvent.log('time_function', 0.001)
```

MetricAlarmEvent Indicates the health of various metrics.

```
from buildbot.process.metrics import MetricAlarmEvent, ALARM_OK

# num_workers looks ok
MetricAlarmEvent.log('num_workers', level=ALARM_OK)
```

Metric Handlers

`MetricsHandler` objects are responsible for collecting `MetricEvents` of a specific type and keeping track of their values for future reporting. There are `MetricsHandler` classes corresponding to each of the `MetricEvent` types.

Metric Watchers

Watcher objects can be added to `MetricsHandlers` to be called when metric events of a certain type are received. Watchers are generally used to record alarm events in response to count or time events.

Metric Helpers

countMethod(name) A function decorator that counts how many times the function is called.

```
from buildbot.process.metrics import countMethod

@countMethod('foo_called')
def foo():
    return "foo!"
```

Timer(name) `Timer` objects can be used to make timing events easier. When `Timer.stop()` is called, a `MetricTimeEvent` is logged with the elapsed time since `timer.start()` was called.

```
from buildbot.process.metrics import Timer

def foo():
    t = Timer('time_foo')
    t.start()
    try:
        for i in range(1000):
            calc(i)
        return "foo!"
    finally:
        t.stop()
```

`Timer` objects also provide a pair of decorators, `startTimer`/`stopTimer` to decorate other functions.

```
from buildbot.process.metrics import Timer

t = Timer('time_thing')

@t.startTimer
def foo():
    return "foo!"

@t.stopTimer
def bar():
    return "bar!"

foo()
bar()
```

timeMethod(name) A function decorator that measures how long a function takes to execute. Note that many functions in buildbot return deferreds, so may return before all the work they set up has completed. Using an explicit `Timer` is better in this case.

```
from buildbot.process.metrics import timeMethod

@timeMethod('time_foo')
def foo():
    for i in range(1000):
        calc(i)
    return "foo!"
```

Secrets

```
class SecretDetails(object):
    """
    ...
    """

    def __init__(self, source, key, value):
```

A `secretDetails` is a python object initialized with a provider name, a key and a value. Each parameter is an object property.

```
secretdetail = SecretDetails("SourceProvider", "myKey", "myValue")
print(secretdetail.source)
"SourceProvider"
print(secretdetail.key)
"myKey"
print(secretdetail.value)
"myValue"
```

A Secret is defined by a key associated to a value, returned from a provider. Secrets returned by providers are stored in a `secretDetails` object.

Secrets manager

The manager is a Buildbot service manager.

```
secretsService = self.master.namedServices['secrets']
secretDetailsList = secretsService.get(self.secrets)
```

The service executes a get method. Depending on the kind of storage chosen and declared in the configuration, the manager gets the selected provider and returns a list of `secretDetails`.

Secrets providers

The secrets providers are implementing the specific getters, related to the storage chosen.

File provider

```
c['secretsProviders'] = [util.SecretInFile(directory="/path/toSecretsFiles"]
```

In the master configuration the provider is instantiated through a Buildbot service secret manager with the file directory path. File secrets provider reads the file named by the key wanted by Buildbot and returns the contained text value. SecretInFile provider allows Buildbot to read secrets in the secret directory.

Vault provider

```
c['secretsProviders'] = [util.SecretInVault(vaultToken=open('VAULT_TOKEN').read(),
                                            vaultServer="http://localhost:8200"
                                            )]
```

In the master configuration, the provider is instantiated through a Buildbot service secret manager with the Vault token and the Vault server address. Vault secrets provider accesses the Vault backend asking the key wanted by Buildbot and returns the contained text value. SecretInVAult provider allows Buildbot to read secrets in the Vault.

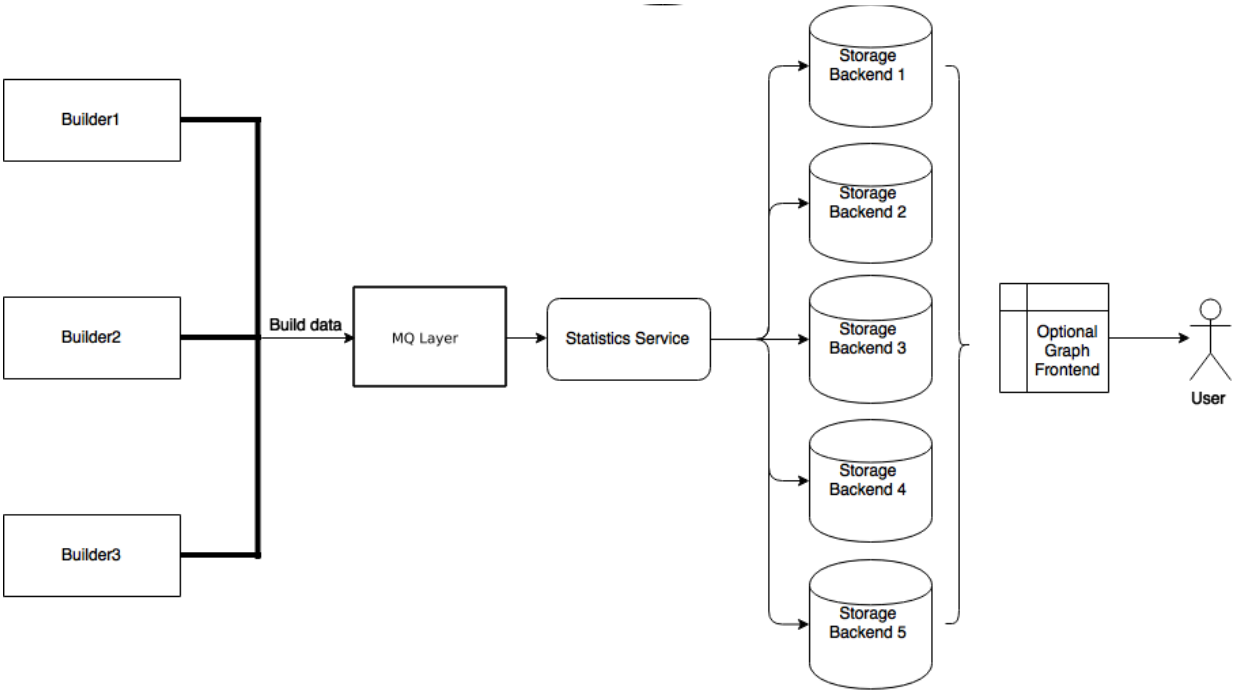
Secret Obfuscation

Secrets are never visible to the normal user via logs and thus are transmitted directly to the workers, using `Obfuscated`. The class Obfuscated replaces secret string value by `####` characters when the secret value is logged.

Statistics Service

The statistic service (or stats service) is implemented in `buildbot.statistics.stats_service`. Please see *`stats-service`* for more information.

Here is a diagram demonstrating the working of the stats service:



Stats Service

class `buildbot.statistics.stats_service.`**`StatsService`**

An instance of this class functions as a `BuildbotService`. The instance of the running service is initialized in the master configuration file (see *stats-service* for more information). The running service is accessible everywhere in Buildbot via the `BuildMaster`. The service is available at `self.master.namedServices['<service-name>']`. It takes the following intialization arguments:

`storage_backends` A list of storage backends. These are instance of subclasses of `StatsStorageBase`.

`name` (str) The name of this service. This name can be used to access the running instance of this service using `self.master.namedServices[name]`.

Please see *stats-service* for examples.

`checkConfig`(*self*, *storage_backends*)

`storage_backends` A list of storage backends.

This method is called automatically to verify that the list of storage backends contains instances of subclasses of `StatsStorageBase`.

`reconfigService`(*self*, *storage_backends*)

`storage_backends` A list of storage backends.

This mehtod is called automatically to reconfigure the running service.

`registerConsumers`(*self*)

Internal method for this class called to register all consumers (methods from Capture classes) to the MQ layer.

`stopService`(*self*)

Internal method for this class to stop the stats service and clean up.

removeConsumers (*self*)
Internal method for this class to stop and remove consumers from the MQ layer.

yieldMetricsValue (*self*, *data_name*, *post_data*, *buildid*)

data_name (str) The name of the data being sent or storage.

post_data A dictionary of key value pair that is sent for storage.

buildid The integer build id of the current build. Obtainable in all `BuildSteps`.

This method should be called to post data that is not generated and stored as build-data in the database. This method generates the `stats-yield-data` event to the mq layer which is then consumed in `postData`.

Storage backends

Storage backends are responsible for storing any statistics/data sent to them. A storage backend will generally be some sort of a database-server running on a machine. .. note:: This machine may be different from the one running `BuildMaster`

Data is captured according to the master config file and then, is sent to each of the storage backends provided by the master configuration (see *`stats-service`*).

Each storage backend has a Python client defined as part of `buildbot.statistics.storage_backends` to aid in posting data by `StatsService`.

Currently, only InfluxDB (https://influxdata.com/time-series-platform/influxdb/) is supported as a storage backend.

class `buildbot.statistis.storage_backends.base.`**StatsStorageBase**
A abstract class for all storage services. It cannot be directly initialized - it would raise a `TypeError` otherwise.

thd_postStatsValue (*self*, *post_data*, *series_name*, *context*)

post_data A dict of key-value pairs that is sent for storage. The keys of this dict can be thought of as columns in a database and the value is the data stored for that column.

series_name (str) The name of the time-series for this statistic.

context (Optional) Any other contextual information about the data. It is a dict of key-value pairs.

An abstract method that needs to be implemented by every child class of this class. Not doing so will result result in a `TypeError` when starting Buildbot.

class `buildbot.statistics.storage_backends.influxdb_client.`**InfluxStorageService**
InfluxDB (https://influxdata.com/time-series-platform/influxdb/) is a distributed, time series database that employs a key-value pair storage system.

This class is a Buildbot client to the InfluxDB storage backend. It is available in the configuration as `statistics.InfluxStorageService`. It takes the following initialization arguments:

url (str) The URL where the service is running.

port (int) The port on which the service is listening.

user (str) Username of a InfluxDB user.

password (str) Password for `user`.

db (str) The name of database to be used.

captures A list of instances of subclasses of `Capture`. This tells which stats are to be stored in this storage backend.

name=None (Optional) (str) The name of this storage backend.

thd_postStatsValue(*self*, *post_data*, *series_name*, *context={}*)

post_data A dict of key-value pairs that is sent for storage. The keys of this dict can be thought of as columns in a database and the value is the data stored for that column.

series_name (str) The name of the time-series for this statistic.

context (Optional) Any other contextual information about the data. It is a dict of key-value pairs.

This method constructs a dictionary of data to be sent to InfluxDB in the proper format and sends the data to the influxDB instance.

Capture Classes

Capture classes are used for declaring which data needs to captured and sent to storage backends for storage.

class `buildbot.statistics.capture.`**Capture**

This is the abstract base class for all capture classes. Not to be used directly. Initlized with the following parameters:

routingKey (tuple) The routing key to be used by `StatsService` to register consumers to the MQ layer for the subclass of this class.

callback The callback registered with the MQ layer for the consumer of a subclass of this class. Each subclass must provide a default callback for this purpose.

_defaultContext(self, msg):
A method for providing default context to the storage backends.

consume(self, routingKey, msg):
This is an abstract method - each subclass of this class should implement its own consume method. If not, then the subclass can't be instantiated. The consume method, when called (from the mq layer), receives the following arguments:

routingKey The routing key which was registered to the MQ layer. Same as the `routingKey` provided to instantiate this class.

msg The message that was sent by the producer.

_store(self, post_data, series_name, context):
This is an abstract method of this class. It must be implemented by all subclasses of this class. It takes the following arguments:

post_data (dict) The key-value pair being sent to the storage backend.

series_name (str) The name of the series to which this data is stored.

context (dict) Any additional information pertaining to data being sent.

class `buildbot.statistics.capture.`**CapturePropertyBase**

This is a base class for both `CaptureProperty` and `CapturePropertyAllBuilders` and abstracts away much of the common functionaltiy between the two classes. Cannot be initialzed directly as it contains an abstract method and raises `TypeError` if tried. It is initialized with the following arguments:

property_name (str) The name of property needed to be recorded as a statistic. This can be a regular expression if `regex=True` (see below).

callback=None The callback function that is used by `CaptureProperty.consumer` to post-process data before formatting it and sending it to the appropriate storage backends. A default callback needs to be prois provided for this.

The default callback:

default_callback(*props*, *property_name*)

It returns property value for `property_name`. It receives the following arguments:

props A dictionary of all build properties.

property_name (str) Name of the build property to return.

regex=False If this is set to `True`, then the property name can be a regular expression. All properties matching this regular expression will be sent for storage.

consume(*self*, *routingKey*, *msg*)
The consumer for all CaptureProperty classes described below. This method filters out the correct properties as per the configuration file and sends those properties for storage. The subclasses of this method do not need to implement this method as it takes care of all the functionaltiy itself. See `Capture` for more information.

_builder_name_matches(self, builder_info):
This is an abstract method and needs to be implemented by all subclasses of this class. This is a helper method to the `consume` method metioned above. It checks whether a builder is allowed to send properties to the storage backend according to the configuration file. It takes one argument:

builder_info (dict) The dictionary returned by the data API containing the builder information.

class `buildbot.statistics.capture.`**CaptureProperty**
The capture class for capturing build properties. It is available in the configuration as `statistics.CaptureProperty`

It takes the following arguments:

builder_name (str) The name of builder in which the property is recorded.

property_name (str) The name of property needed to be recorded as a statistic.

callback=None The callback function that is used by `CaptureProperty.consumer` to post-process data before formatting it and sending it to the appropriate storage backends. A default callback is provided for this (see `CapturePropertyBase` for more information).

regex=False If this is set to `True`, then the property name can be a regular expression. All properties matching this regular expression will be sent for storage.

_builder_name_matches(*self*, *builder_info*)
See `CapturePropertyBase` for more information on this method.

class `buildbot.statistics.capture.`**CapturePropertyAllBuilders**
The capture class to use for capturing build properties on all builders. It is available in the configuration as `statistics.CaptureProperty`

It takes the following arguments:

property_name (str) The name of property needed to be recorded as a statistic.

callback=None The callback function that is used by `CaptureProperty.consumer` to post-process data before formatting it and sending it to the appropriate storage backends. A default callback is provided for this (see `CapturePropertyBase` for more information).

regex=False If this is set to `True`, then the property name can be a regular expression. All properties matching this regular expression will be sent for storage.

_builder_name_matches(*self*, *builder_info*)
See `CapturePropertyBase` for more information on this method.

class buildbot.statistics.capture.**CaptureBuildTimes**
A base class for all Capture classes that deal with build times (start/end/duration). Not to be used directly. Initialized with:

builder_name (str) The name of builder whose times are to be recorded.

callback The callback function that is used by subclass of this class to post-process data before formatting it and sending it to the appropriate storage backends. A default callback is provided for this. Each subclass must provide a deafault callback that is used in initialization of this class should the user not provide a callback.

consume(*self*, *routingKey*, *msg*)
The consumer for all subclasses of this class. See Capture for more information. .. note:: This consumer requires all subclasses to implement:

self._time_type (property) A string used as a key in post_data sent to sotrage services.

self._retValParams(msg) (method) A method that takes in the msg this consumer gets and returns a list of arguments for the capture callback.

_retValParams(*self*, *msg*)
This is an abstract method which needs to be implemented by subclassses. This method needs to return a list of parameters that will be passed to the callback function. See individual build CaptureBuild* classes for more information.

_err_msg(*self*, *build_data*, *builder_name*)
A helper method that returns an error message for the consume method.

_builder_name_matches(*self*, *builder_info*)
This is an abstract method and needs to be implemented by all subclasses of this class. This is a helper method to the consume method metioned above. It checks whether a builder is allowed to send build times to the storage backend according to the configuration file. It takes one argument:

builder_info (dict) The dictionary returned by the data API containing the builder information.

class buildbot.statistics.capture.**CaptureBuildStartTime**
A capture class for capturing build start times. It takes the following arguments:

builder_name (str) The name of builder whose times are to be recorded.

callback=None The callback function for this class. See CaptureBuildTimes for more information.

The default callback:

default_callback(*start_time*)

It returns the start time in ISO format. It takes one argument:

start_time A python datetime object that denotes the build start time.

_retValParams(*self*, *msg*)
Returns a list containing one Python datetime object (start time) from msg dictionary.

_builder_name_matches(*self*, *builder_info*)
See CaptureBuildTimes for more information on this method.

class buildbot.statistics.capture.**CaptureBuildStartTimeAllBuilders**
A capture class for capturing build start times from all builders. It is a subclass of CaptureBuildStartTime. It takes the following arguments:

callback=None The callback function for this class. See CaptureBuildTimes for more information.

The default callback:

See CaptureBuildStartTime.__init__ for the definition.

_builder_name_matches(*self*, *builder_info*)
See `CaptureBuildTimes` for more information on this method.

class `buildbot.statistics.capture.`**CaptureBuildEndTime**
A capture class for capturing build end times. Takes the following arguments:

builder_name (str) The name of builder whose times are to be recorded.

callback=None The callback function for this class. See `CaptureBuildTimes` for more information.

The default callback:

default_callback(*end_time*)

It returns the end time in ISO format. It takes one argument:

end_time A python datetime object that denotes the build end time.

_retValParams(*self*, *msg*)

Returns a list containing two Python datetime object (start time and end time) from `msg` dictionary.

_builder_name_matches(*self*, *builder_info*)
See `CaptureBuildTimes` for more information on this method.

class `buildbot.statistics.capture.`**CaptureBuildEndTimeAllBuilders**
A capture class for capturing build end times from all builders. It is a subclass of `CaptureBuildEndTime`. It takes the following arguments:

callback=None The callback function for this class. See `CaptureBuildTimes` for more information.

The default callback:

See `CaptureBuildEndTime.__init__` for the definition.

_builder_name_matches(*self*, *builder_info*)
See `CaptureBuildTimes` for more information on this method.

class `buildbot.statistics.capture.`**CaptureBuildDuration**
A capture class for capturing build duration. Takes the following arguments:

builder_name (str) The name of builder whose times are to be recorded.

report_in='seconds' Can be one of three: `'seconds'`, `'minutes'`, or `'hours'`. This is the units in which the build time will be reported.

callback=None The callback function for this class. See `CaptureBuildTimes` for more information.

The default callback:

default_callback(*start_time*, *end_time*)

It returns the duration of the build as per the `report_in` argument. It receives the following arguments:

start_time A python datetime object that denotes the build start time.

end_time A python datetime object that denotes the build end time.

_retValParams(*self*, *msg*)

Returns a list containing one Python datetime object (end time) from `msg` dictionary.

_builder_name_matches(*self*, *builder_info*)
See `CaptureBuildTimes` for more information on this method.

class buildbot.statistics.capture.**CaptureBuildDurationAllBuilders**
A capture class for capturing build durations from all builders. It is a subclass of CaptureBuildDuration. It takes the following arguments:

callback=None The callback function for this class. See CaptureBuildTimes for more.

The default callback:

See CaptureBuildDuration.__init__ for the definition.

_builder_name_matches(*self*, *builder_info*)
See CaptureBuildTimes for more information on this method.

class buildbot.statistics.capture.**CaptureDataBase**
This is a base class for both CaptureData and CaptureDataAllBuilders and abstracts away much of the common functionaltiy between the two classes. Cannot be initialzed directly as it contains an abstract method and raises TypeError if tried. It is initialized with the following arguments:

data_name (str) The name of data to be captured. Same as in yieldMetricsValue.

callback=None The callback function for this class.

The default callback:

The default callback takes a value x and return it without changing. As such, x itself acts as the post_data sent to the storage backends.

consume(*self*, *routingKey*, *msg*)
The consumer for this class. See Capture for more.

_builder_name_matches(*self*, *builder_info*)
This is an abstract method and needs to be implemented by all subclasses of this class. This is a helper method to the consume method metioned above. It checks whether a builder is allowed to send properties to the storage backend according to the configuration file. It takes one argument:

builder_info (dict) The dictionary returned by the data API containing the builder information.

class buildbot.statistics.capture.**CaptureData**
A capture class for capturing arbitrary data that is not stored as build-data. See yieldMetricsValue for more. Takes the following arguments for initliazation:

data_name (str) The name of data to be captured. Same as in yieldMetricsValue.

builder_name (str) The name of the builder on which the data is captured.

callback=None The callback function for this class.

The default callback:

See CaptureDataBase of definition.

_builder_name_matches(*self*, *builder_info*)
See CaptureDataBase for more information on this method.

class buildbot.statistics.capture.**CaptureDataAllBuilders**
A capture class to capture arbitrary data on all builders. See yieldMetricsValue for more. It takes the following arguments:

data_name (str) The name of data to be captured. Same as in yieldMetricsValue.

callback=None The callback function for this class.

_builder_name_matches(*self*, *builder_info*)
See CaptureDataBase for more information on this method.

How to package Buildbot plugins

If you customized an existing component (see *Customization*) or created a new component that you believe might be useful for others, you have two options:

- submit the change to the Buildbot main tree, however you need to adhere to certain requirements (see *Buildbot Coding Style*)
- prepare a Python package that contains the functionality you created

Here we cover the second option.

Package the source

To begin with, you must package your changes. If you do not know what a Python package is, these two tutorials will get you going:

- Python Packaging User Guide (https://packaging.python.org/en/latest/)
- The Hitchhiker's Guide to Packaging (https://the-hitchhikers-guide-to-packaging.readthedocs.org/en/latest/)

The former is more recent and, while it addresses everything that you need to know about Python packages, is still work in progress. The latter is a bit dated, though it was the most complete guide for quite some time available for Python developers looking to package their software.

You may also want to check the sample project (https://github.com/pypa/sampleproject), which exemplifies the best Python packaging practices.

Making the plugin package

Buildbot supports several kinds of pluggable components:

- `worker`
- `changes`
- `schedulers`
- `steps`
- `status`
- `util`

(these are described in *Plugin Infrastructure in Buildbot*), and

- `www`

which is described in *web server configuration*.

Once you have your component packaged, it's quite straightforward: you just need to add a few lines to the `entry_points` parameter of your call of `setup` function in `setup.py` file:

```
setup(
    ...
    entry_points = {
        ...,
        'buildbot.kind': [
            'PluginName = PluginModule:PluginClass'
        ]
    },
```

```
    ...
)
```

(You might have seen different ways to specify the value for `entry_points`, however they all do the same thing. Full description of possible ways is available in setuptools documentation (https://setuptools.readthedocs.io/en/latest/setuptools.html#dynamic-discovery-of-services-and-plugins).)

After the setup.py (https://github.com/buildbot/buildbot/blob/master/master/setup.py) file is updated, you can build and install it:

```
$ python setup.py build
$ sudo python setup.py install
```

(depending on your particular setup, you might not need to use **`sudo`**).

After that the plugin should be available for Buildbot and you can use it in your `master.cfg` as:

```
from buildbot.kind import PluginName

... PluginName ...
```

Publish the package

This is the last step before the plugin is available to others.

Once again, there is a number of options available for you:

- just put a link to your version control system
- prepare a source tarball with the plugin (`python setup.py sdist`)
- or publish it on PyPI (https://pypi.python.org)

The last option is probably the best one since it will make your plugin available pretty much to all Python developers.

Once you have published the package, please send a link to buildbot-devel (buildbot-devel@lists.sourceforge.net) mailing list, so we can include a link to your plugin to *Plugin Infrastructure in Buildbot*.

APIs

This section documents Buildbot's APIs. These are the interfaces against which externally-maintained code should be written.

REST API

The REST API is a thin wrapper around the data API's "Getter" and "Control" sections. It is also designed, in keeping with REST principles, to be discoverable. As such, the details of the paths and resources are not documented here. Begin at the root URL, and see the *Data API* documentation for more information.

Versions

The API described here is version 2. The ad-hoc API from Buildbot-0.8.x, version 1, is no longer supported.

The policy for incrementing the version is when there is incompatible change added. Removing a field or endpoint is considered incompatible change. Adding a field or endpoint is not considered incompatible, and thus will only be described as a change in release note. The policy is that we will avoid as much as possible incrementing version.

Getting

To get data, issue a GET request to the appropriate path. For example, with a base URL of `http://build.example.org/buildbot`, the list of masters for builder 9 is available at `http://build.example.org/buildbot/api/v2/builder/9/master`.

resource type: `collection`

Collections

Results are formatted in keeping with the JSON API (http://jsonapi.org/) specification. The top level of every response is an object. Its keys are the plural names of the resource types, and the values are lists of objects, even for a single-resource request. For example:

```
{
  "meta": {
    "total": 2
  },
  "schedulers": [
    {
      "master": null,
      "name": "smoketest",
      "schedulerid": 1
    },
    {
      "master": {
        "active": true,
        "last_active": 1369604067,
        "link": "http://build.example.org/api/v2/master/1",
        "masterid": 1,
        "name": "master3:/BB/master"
      },
      "name": "goaheadtryme",
      "schedulerid": 2
    }
  ]
}
```

A response may optionally contain extra, related resources beyond those requested. The `meta` key contains metadata about the response, including the total count of resources in a collection.

Several query parameters may be used to affect the results of a request. These parameters are applied in the order described (so, it is not possible to sort on a field that is not selected, for example).

Field Selection

If only certain fields of each resource are required, the `field` query parameter can be used to select them. For example, the following will select just the names and id's of all schedulers:

- `http://build.example.org/api/v2/scheduler?field=name&field=schedulerid`

Field selection can be used for either detail (single-entity) or collection (multi-entity) requests. The remaining options only apply to collection requests.

Filtering

Collection responses may be filtered on any simple top-level field.

To select records with a specific value use the query parameter `{field}={value}`. For example, `http://build.example.org/api/v2/scheduler?name=smoketest` selects the scheduler named "smoketest".

Filters can use any of the operators listed below, with query parameters of the form `{field}__{operator}={value}`.

eq equality, or with the same parameter appearing multiple times, equality with one of the given values (so *foo__eq=x&foo__eq=y* would match resources where foo is *x* or *y*)

ne inequality, or set exclusion

lt select resources where the field's value is less than `{value}`

le select resources where the field's value is less than or equal to `{value}`

gt select resources where the field's value is greater than `{value}`

ge select resources where the field's value is greater than or equal to `{value}`

contains select resources where the field's value contains `{value}`

For example:

- `http://build.example.org/api/v2/builder?name__lt=cccc`
- `http://build.example.org/api/v2/buildsets?complete__eq=false`

Boolean values can be given as `on`/`off`, `true`/`false`, `yes`/`no`, or `1`/`0`.

Sorting

Collection responses may be ordered with the `order` query parameter. This parameter takes a field name to sort on, optionally prefixed with `-` to reverse the sort. The parameter can appear multiple times, and will be sorted lexically with the fields arranged in the given order. For example:

- `http://build.example.org/api/v2/buildrequest?order=builderid&order=buildrequestid`

Pagination

Collection responses may be paginated with the `offset` and `limit` query parameters. The offset is the 0-based index of the first result to included, after filtering and sorting. The limit is the maximum number of results to return. Some resource types may impose a maximum on the limit parameter; be sure to check the resulting links to determine whether further data is available. For example:

- `http://build.example.org/api/v2/buildrequest?order=builderid&limit=10`
- `http://build.example.org/api/v2/buildrequest?order=builderid&offset=20&limit=10`

Controlling

Data API control operations are handled by POST requests using a simplified form of JSONRPC 2.0 (http://www.jsonrpc.org/specification). The JSONRPC "method" is mapped to the data API "action", and the parameters are passed to that application.

The following parts of the protocol are not supported:

- positional parameters
- batch requests

Requests are sent as an HTTP POST, containing the request JSON in the body. The content-type header must be `application/json`.

A simple example:

```
POST http://build.example.org/api/v2/scheduler/4
--> {"jsonrpc": "2.0", "method": "force", "params": {"revision": "abcd", "branch":
↪"dev"}, "id": 843}
<-- {"jsonrpc": "2.0", "result": {"buildsetid": 44}, "id": 843}
```

Authentication

Authentication to the REST API is performed in the same manner as authentication to the main web interface. Once credentials have been established, a cookie will be set, which must be sent to the buildbot REST API with every request thereafter. For those buildbot instances using OAuth2 authentication providers, access tokens can be used for automated access. For example, GitHub's personal access tokens can be used to access the buildbot as a github user without needing to store the username and password of the user. To use an OAuth2 access token, send a *GET* request to the */auth/login* with the *token* URL parameter set to the access token that the OAuth2 provider has given you. A *python* example using *requests* is shown below, where we first authenticate with our OAuth2 access token, and then are able to request otherwise shielded endpoints:

```
import requests
s = requests.Session()
s.get("https://<buildbot_url>/auth/login", params={"token": OAUTH_TOKEN})
builders = s.get("https://<buildbot_url>/api/v2/builders").json()
```

Raml Specs

The Data API is documented in RAML 1.0 format (https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md). RAML describes and documents all our data, rest, and javascript APIs in a format that can be easily manipulated by human and machines.

Build

resource type: `build`

Attributes

- **`buildid`** (*`integer`*) – the unique ID of this build
- **`number`** (*`integer`*) – the number of this build (sequential for a given builder)
- **`builderid`** (*`integer`*) – id of the builder for this build
- **`buildrequestid`** (*`integer`*) – build request for which this build was performed, or None if no such request exists
- **`workerid`** (*`integer`*) – the worker this build ran on
- **`masterid`** (*`integer`*) – the master this build ran on
- **`started_at`** (*`date`*) – time at which this build started
- **`complete`** (*`boolean`*) – true if this build is complete Note that this is a calculated field (from complete_at != None). Ordering by this field is not optimized by the database layer.
- **`complete_at?`** (*`date`*) – time at which this build was complete, or None if it's still running
- **`properties?`** (*`sourcedproperties`*) – a dictionary of properties attached to build.

- **results?** (*integer*) – the results of the build (see *Build Result Codes*), or None if not complete
- **state_string** (*string*) – a string giving detail on the state of the build.

example

```
{
    "builderid": 10,
    "buildid": 100,
    "buildrequestid": 13,
    "workerid": 20,
    "complete": false,
    "complete_at": null,
    "masterid": 824,
    "number": 1,
    "results": null,
    "started_at": 1451001600,
    "state_string": "created",
    "properties": {}
}
```

This resource type describes completed and in-progress builds. Much of the contextual data for a build is associated with the build request, and through it the buildset.

Note: *properties*

This properties dict is only filled out if the *properties filterspec* is set.

Meaning that, *property filter* allows one to request the Builds DATA API like so:

- api/v2/builds?property=propKey1&property=propKey2 (returns Build's properties which match given keys)
- api/v2/builds?property=* (returns all Build's properties)
- api/v2/builds?propKey1&property=propKey2&limit=30 (filters combination)

Important: When combined with `field` filter, to get properties, one should ensure **properties** `field` is set.

- api/v2/builds?field=buildid&field=properties&property=workername&property=user

---

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.builds.`**Build**

newBuild(*builderid*, *buildrequestid*, *workerid*)

Parameters

- **builderid** (*integer*) – builder performing this build
- **buildrequstid** (*integer*) – build request being built
- **workerid** (*integer*) – worker on which this build is performed

Returns (buildid, number) via Deferred

Create a new build resource and return its ID. The state strings for the new build will be set to 'starting'.

setBuildStateString(*buildid*, *state_string*)

Parameters

- **buildid** (*integer*) – the build to modify
- **state_string** (*unicode*) – new state string for this build

Replace the existing state strings for a build with a new list.

finishBuild(*buildid*, *results*)

Parameters

- **buildid** (*integer*) – the build to modify
- **results** (*integer*) – the build's results

Mark the build as finished at the current time, with the given results.

Endpoints

path: /builders/{builderid}/builds

Path Keys **builderid** (*number*) – the ID of the builder

This path selects all builds for a builder (can return lots of data!)

GET

returns *collection* of *build*

path: /builders/{builderid}/builds/{build_number}

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder

This path selects a specific build by builderid, buildnumber

GET

returns *collection* of *build*

POST with method: /builders/{builderid}/builds/{build_number}:stop

Body keys

- **method** (*string*) – must be `stop`
- **reason** (*string*) – The reason why the build was stopped
- **results** (*integer*) – optionally results value override (default CANCELLED)

POST with method: /builders/{builderid}/builds/{build_number}:rebuild

Body keys **method** (*string*) – must be `rebuild`

path: /buildrequests/{buildrequestid}/builds

Path Keys **buildrequestid** (*number*) – the id of the buildrequest

GET

returns *collection* of *build*

path: /builds

GET

returns *collection* of *build*

path: /builds/{buildid}

Path Keys **buildid** (*number*) – the id of the build

This path selects a build by id

GET

returns *collection* of *build*

POST with method: /builds/{buildid}:stop

Body keys

- **method** (*string*) – must be `stop`
- **reason** (*string*) – The reason why the build was stopped

POST with method: /builds/{buildid}:rebuild

Body keys **method** (*string*) – must be `rebuild`

builder

resource type: builder

Attributes

- **builderid** (*integer*) – the ID of this builder
- **description?** (*string*) – The description for that builder
- **masterids[]** (*integer*) – the ID of the masters this builder is running on
- **name** (*identifier*) – builder name
- **tags[]** (*string*) – list of tags for this builder

This resource type describes a builder.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.builders.`**Builder**

updateBuilderList(*masterid*, *builderNames*)

Parameters

- **masterid** (*integer*) – this master's master ID
- **builderNames** (*list*) – list of names of currently-configured builders (unicode strings)

Returns Deferred

Record the given builders as the currently-configured set of builders on this master. Masters should call this every time the list of configured builders changes.

Endpoints

path: /builders

This path selects all builders

GET

returns *collection* of *builder*

path: /builders/{builderid}

Path Keys **builderid** (*number*) – the ID of the builder

This path selects a builder by builderid

GET

returns *collection* of *builder*

path: /masters/{masterid}/builders

Path Keys **masterid** (*number*) – the id of the master

This path selects all builders of a given master

GET

returns *collection* of *builder*

path: /masters/{masterid}/builders/{builderid}

Path Keys

- **masterid** (*number*) – the id of the master
- **builderid** (*number*) – the id of the builder

This path selects one builder by id of a given master

GET

returns *collection* of *builder*

buildrequest

resource type: buildrequest

Attributes

- **buildrequestid** (*integer*) – the unique ID of this buildrequest
- **builderid** (*integer*) – the id of the builder linked to this buildrequest
- **buildsetid** (*integer*) – the id of the buildset that contains this buildrequest
- **claimed** (*boolean*) – True if this buildrequest has been claimed. Note that this is a calculated field (from claimed_at != None). Ordering by this field is not optimized by the database layer.
- **claimed_at?** (*date*) – time at which this build has last been claimed. None if this buildrequest has never been claimed or has been unclaimed

- **claimed_by_masterid?** (*integer*) – the id of the master that claimed this buildrequest. None if this buildrequest has never been claimed or has been unclaimed
- **complete** (*boolean*) – true if this buildrequest is complete
- **complete_at?** (*date*) – time at which this buildrequest was completed, or None if it's still running
- **priority** (*integer*) – the priority of this buildrequest
- **results?** (*integer*) – the results of this buildrequest (see *Build Result Codes*), or None if not complete
- **submitted_at** (*date*) – time at which this buildrequest were submitted
- **waited_for** (*boolean*) – True if the entity that triggered this buildrequest is waiting for it to complete. Should be used by an (unimplemented so far) clean shutdown to only start br that are waited_for.

This resource type describes completed and in-progress buildrequests. Much of the contextual data for a buildrequest is associated with the buildset that contains this buildrequest.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.buildrequests.`**BuildRequest**

claimBuildRequests(*brids*, *claimed_at=None*, *_reactor=twisted.internet.reactor*)

Parameters

- **brids** (*list(integer)*) – list of buildrequest id to claim
- **claimed_at** (*datetime*) – date and time when the buildrequest is claimed
- **_reactor** (*twisted.internet.interfaces.IReactorTime*) – reactor used to get current time if `claimed_at` is None

Returns (boolean) whether claim succeeded or not

Claim a list of buildrequests

reclaimBuildRequests(*brids*, *_reactor=twisted.internet.reactor*)

Parameters

- **brids** (*list(integer)*) – list of buildrequest id to reclaim
- **_reactor** (*twisted.internet.interfaces.IReactorTime*) – reactor used to get current time

Returns (boolean) whether reclaim succeeded or not

Reclaim a list of buildrequests

unclaimBuildRequests(*brids*)

Parameters **brids** (*list(integer)*) – list of buildrequest id to unclaim

Unclaim a list of buildrequests

completeBuildRequests(*brids*, *results*, *complete_at=None*, *_reactor=twisted.internet.reactor*)

Parameters

- **brids** (*list (integer)*) – list of buildrequest id to complete
- **results** (*integer*) – the results of the buildrequest (see *Build Result Codes*)
- **complete_at** (*datetime*) – date and time when the buildrequest is completed
- **_reactor** (*twisted.internet.interfaces.IReactorTime*) – reactor used to get current time, if `complete_at` is None

Complete a list of buildrequest with the `results` status

unclaimExpiredRequests(*old*, *_reactor=twisted.internet.reactor*)

Parameters

- **old** (*integer*) – time in seconds considered for getting unclaimed buildrequests
- **_reactor** (*twisted.internet.interfaces.IReactorTime*) – reactor used to get current time

Unclaim the previously claimed buildrequests that are older than `old` seconds and that were never completed

Endpoints

path: /builders/{builderid}/buildrequests

Path Keys **builderid** (*number*) – the ID of the builder

This path selects all buildrequests for a given builder (can return lots of data!)

GET

returns *collection* of *buildrequest*

path: /buildrequests

GET

returns *collection* of *buildrequest*

path: /buildrequests/{buildrequestid}

Path Keys **buildrequestid** (*number*) – the id of the buildrequest

GET

returns *collection* of *buildrequest*

POST with method: /buildrequests/{buildrequestid}:cancel

Body keys

- **method** (*string*) – must be `cancel`
- **reason** (*string*) – The reason why the buildrequest was cancelled

buildset

resource type: buildset

Attributes

- **bsid** (*integer*) – the ID of this buildset

- **complete** (*boolean*) – true if all of the build requests in this buildset are complete
- **complete_at?** (*integer*) – the time this buildset was completed, or None if not complete
- **external_idstring?** (*string*) – an identifier that external applications can use to identify a submitted buildset; can be None
- **parent_buildid?** (*integer*) – optional build id that is the parent for this buildset
- **parent_relationship?** (*string*) – relationship identifier for the parent, this is is configured relationship between the parent build, and the childs buildsets
- **reason** (*string*) – the reason this buildset was scheduled
- **results?** (*integer*) – the results of the buildset (see *Build Result Codes*), or None if not complete
- **sourcestamps[]** (*sourcestamp*) – the sourcestamps for this buildset; each element is a valid *sourcestamp* entity
- **submitted_at** (*integer*) – the time this buildset was submitted

A buildset gathers build requests that were scheduled at the same time, and which share a source stamp, properties, and so on.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.buildsets.`**Buildset**

addBuildset(*scheduler=None*, *sourcestamps=[]*, *reason=''*, *properties={}*, *builderids=[]*, *external_idstring=None*, *parent_buildid=None*, *parent_relationship=None*)

Parameters

- **scheduler** (*string*) – the name of the scheduler creating this buildset
- **sourcestamps** (*list*) – sourcestamps for the new buildset; see below
- **reason** (*unicode*) – the reason for this build
- **properties** (*dictionary with unicode keys and (source, property value) values*) – properties to set on this buildset
- **builderids** (*list*) – names of the builderids for which build requests should be created
- **external_idstring** (*unicode*) – arbitrary identifier to recognize this buildset later
- **parent_buildid** (*int*) – optional build id that is the parent for this buildset
- **parent_relationship** (*unicode*) – relationship identifier for the parent, this is is configured relationship between the parent build, and the childs buildsets

Returns (buildset id, dictionary mapping builder ids to build request ids) via Deferred

Create a new buildset and corresponding buildrequests based on the given parameters. This is the low-level interface for scheduling builds.

Each sourcestamp in the list of sourcestamps can be given either as an integer, assumed to be a sourcestamp ID, or a dictionary of keyword arguments to be passed to `findSourceStampId`.

maybeBuildsetComplete(*bsid*)

Parameters **bsid** (*integer*) – buildset that may be complete

Returns Deferred

This method should be called when a build request is finished. It checks the given buildset to see if all of its buildrequests are finished. If so, it updates the status of the buildset and send the appropriate messages.

Endpoints

path: /buildsets

this path selects all buildsets

GET

returns *collection* of *buildset*

path: /buildsets/{bsid}

Path Keys **bsid** (*identifier*) – the id of the buildset

this path selects a buildset by id

GET

returns *collection* of *buildset*

change

resource type: change

Attributes

- **changeid** (*integer*) – the ID of this change
- **author** (*string*) – the author of the change in "name", "name <email>" or just "email" (with @) format
- **branch?** (*string*) – branch on which the change took place, or none for the "default branch", whatever that might mean
- **category?** (*string*) – user-defined category of this change, or none
- **codebase** (*string*) – codebase in this repository
- **comments** (*string*) – user comments for this change (aka commit)
- **files[]** (*string*) – list of source-code filenames changed
- **parent_changeids[]** (*integer*) – The ID of the parents. The data api allow for several parents, but the core buildbot does not yet support
- **project** (*string*) – user-defined project to which this change corresponds
- **properties** (*sourcedproperties*) – user-specified properties for this change, represented as an object mapping keys to tuple (value, source)
- **repository** (*string*) – repository where this change occurred
- **revision?** (*string*) – revision for this change, or none if unknown
- **revlink?** (*string*) – link to a web view of this change

- **sourcestamp** (*sourcestamp*) – the sourcestamp resouce for this change
- **when_timestamp** (*integer*) – time of the change

A change resource represents a change to the source code monitored by Buildbot.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.changes.`**Change**

addChange(*files=None*, *comments=None*, *author=None*, *revision=None*, *when_timestamp=None*, *branch=None*, *category=None*, *revlink=''*, *properties={}*, *repository=''*, *codebase=None*, *project=''*, *src=None*)

Parameters

- **files** (*list of unicode strings*) – a list of filenames that were changed
- **comments** (*unicode*) – user comments on the change
- **author** (*unicode*) – the author of this change
- **revision** (*unicode*) – the revision identifier for this change
- **when_timestamp** (*integer*) – when this change occurred (seconds since the epoch), or the current time if None
- **branch** (*unicode*) – the branch on which this change took place
- **category** (*unicode*) – category for this change
- **revlink** (*string*) – link to a web view of this revision
- **properties** (*dictionary with unicode keys and simple values (JSON-able)*) – properties to set on this change. Note that the property source is *not* included in this dictionary.
- **repository** (*unicode*) – the repository in which this change took place
- **project** (*unicode*) – the project this change is a part of
- **src** (*unicode*) – source of the change (vcs or other)

Returns the ID of the new change, via Deferred

Add a new change to Buildbot. This method is the interface between change sources and the rest of Buildbot.

All parameters should be passed as keyword arguments.

All parameters labeled 'unicode' must be unicode strings and not bytestrings. Filenames in `files`, and property names, must also be unicode strings. This is tested by the fake implementation.

Endpoints

path: /builds/{buildid}/changes

Path Keys **buildid** (*number*) – the id of the build

This path selects all changes tested by a build

GET

returns *collection* of *change*

path: /changes

This path selects **all** changes. On a reasonably loaded master, this can quickly return a very large result, taking minutes to process. A specific query configuration is optimized which allows to get the recent changes: `order:-changeid&limit=<n>`

GET

returns *collection* of *change*

path: /changes/{changeid}

Path Keys **changeid** (*number*) – the id of a change

this path selects one change by id

GET

returns *collection* of *change*

path: /sourcestamps/{ssid}/changes

Path Keys **ssid** (*number*) – the id of the sourcestamp

This path selects all changes associated to one sourcestamp

GET

returns *collection* of *change*

changesource

resource type: changesource

Attributes

- **changesourceid** (*integer*) – the ID of this changesource
- **master?** (*master*) – the master on which this worker is running, or None if it is inactive
- **name** (*string*) – name of this changesource

A changesource generates change objects, for example in response to an update in some repository. A particular changesource (by name) runs on at most one master at a time.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.changesources.`**ChangeSource**

findChangeSourceId(*name*)

Parameters **name** (*string*) – changesource name

Returns changesource ID via Deferred

Get the ID for the given changesource name, inventing one if necessary.

trySetChangeSourceMaster(*changesourceid*, *masterid*)

Parameters

- **changesourceid** (*integer*) – changesource ID to try to claim
- **masterid** (*integer*) – this master's master ID

Returns `True` or `False`, via Deferred

Try to claim the given scheduler for the given master and return `True` if the scheduler is to be activated on that master.

Endpoints

path: /changesources

This path selects all changesource.

GET

returns *collection* of *changesource*

path: /changesources/{changesourceid}

Path Keys **changesourceid** (*number*) – the id of a changesource

This path selects one changesource given its id.

GET

returns *collection* of *changesource*

path: /masters/{masterid}/changesources

Path Keys **masterid** (*number*) – the id of the master

This path selects all changesources for a given master

GET

returns *collection* of *changesource*

path: /masters/{masterid}/changesources/{changesourceid}

Path Keys **masterid** (*number*) – the id of the master

This path selects one changesource by id for a given master

GET

returns *collection* of *changesource*

forcescheduler

resource type: forcescheduler

Attributes

- **all_fields[]** (*object*) –
- **builder_names[]** (*identifier*) – names of the builders that this scheduler can trigger
- **button_name** (*string*) – label of the button to use in the UI
- **label** (*string*) – label of this scheduler to be displayed in the ui

- **name** (*identifier*) – name of this scheduler

A forcescheduler initiates builds, via a formular in the web UI. At the moment, forceschedulers must be defined on all the masters where a web ui is configured. A particular forcescheduler runs on the master where the web request was sent.

Note: This datatype and associated endpoints will be deprecated when bug #2673 (http://trac.buildbot.net/ticket/2673) will be resolved.

Endpoints

path: /builders/{builderid}/forceschedulers

Path Keys **builderid** (*number*) – the ID of the builder

This path selects all force-schedulers for a given builder

GET

returns *collection* of *forcescheduler*

path: /forceschedulers

This path selects all forceschedulers.

GET

returns *collection* of *forcescheduler*

path: /forceschedulers/{schedulername}

Path Keys **schedulername** (*identifier*) – the name of a scheduler

This path selects all changesource.

GET

returns *collection* of *forcescheduler*

POST with method: /forceschedulers/{schedulername}:force

Body keys

- **method** (*string*) – must be `force`
- **owner** (*string*) – The user who wants to create the buildrequest
- **[]** – content of the forcescheduler parameter is dependent on the configuration of the forcescheduler

identifier

resource type: identifier

Logs

resource type: log

Attributes

- **complete** (*boolean*) – true if this log is complete and will not generate additional logchunks
- **logid** (*integer*) – the unique ID of this log
- **name** (*string*) – the name of this log (e.g., `err.html`)
- **num_lines** (*integer*) – total number of line of this log
- **slug** (*identifier*) – the "slug", suitable for use in a URL, of this log (e.g., `err_html`)
- **stepid** (*integer*) – id of the step containing this log
- **type** (*identifier*) – log type, identified by a single ASCII letter; see *logchunk* for details.

`example`

```
{
    "logid": 60,
    "name": "stdio",
    "slug": "stdio",
    "stepid": 50,
    "complete": false,
    "num_lines": 0,
    "type": "s"
}
```

A log represents a stream of textual output from a step. The actual output is encoded as a sequence of *logchunk* resources. In-progress logs append logchunks as new data is added to the end, and event subscription allows a client to "follow" the log.

Each log has a "slug" which is unique within the step, and which can be used in paths. The slug is generated by *addLog* based on the name, using *forceIdentifier* and *incrementIdentifier* to guarantee uniqueness.

Todo

event: `build.$buildid.step.$number.log.$logid.newlog`
The log has just started. Logs are started when they are created, so this also indicates the creation of a new log.

event: `build.$buildid.step.$number.log.$logid.complete`
The log is complete.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.logs.`**`Log`**

`addLog`(*stepid*, *name*, *type*)

Parameters

- **stepid** (*integer*) – stepid containing this log
- **name** (*string*) – name for the log

Raises **`KeyError`** – if a log by the given name already exists

Returns logid via Deferred

Create a new log and return its ID. The name need not be unique. This method will generate a unique slug based on the name.

appendLog(logid, content):

Parameters

- **logid** (*integer*) – the log to which content should be appended
- **content** (*unicode*) – the content to append

Append the given content to the given log. The content must end with a newline. All newlines in the content should be UNIX-style (`\n`).

finishLog(*logid*)

Parameters **logid** (*integer*) – the log to finish

Mark the log as complete.

compressLog(*logid*)

Parameters **logid** (*integer*) – the log to compress

Compress the given log, after it is finished. This operation may take some time.

Endpoints

path: /builders/{builderid}/builds/{build_number}/steps/{step_name}/logs

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder
- **step_name** (*identifier*) – the slug name of the step

This path selects all logs for the given step.

GET

returns *collection* of *log*

path: /builders/{builderid}/builds/{build_number}/steps/{step_name}/logs/{log_slug}

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder
- **step_name** (*identifier*) – the slug name of the step
- **log_slug** (*identifier*) – the slug name of the log

GET

returns *collection* of *log*

path: /builders/{builderid}/builds/{build_number}/steps/{step_number}/logs

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder

- **step_number** (*number*) – the number of the step

This path selects all log of a a specific step

GET

returns *collection* of *log*

path: /builders/{builderid}/builds/{build_number}/steps/{step_number}/logs/{log_slug}

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder
- **step_number** (*number*) – the number of the step
- **log_slug** (*identifier*) – the slug name of the log

This path selects one log of a a specific step

GET

returns *collection* of *log*

path: /builds/{buildid}/steps/{step_number_or_name}/logs

Path Keys

- **buildid** (*number*) – the id of the build
- **| number step_number_or_name** (*identifier*) – the name or number of the step

This path selects all logs of a step of a build

GET

returns *collection* of *log*

path: /builds/{buildid}/steps/{step_number_or_name}/logs/{log_slug}

Path Keys

- **buildid** (*number*) – the id of the build
- **| number step_number_or_name** (*identifier*) – the name or number of the step
- **log_slug** (*identifier*) – the slug name of the log

This path selects one log of a a specific step

GET

returns *collection* of *log*

path: /logs/{logid}

Path Keys **logid** (*number*) – the id of the log

This path selects one log

GET

returns *collection* of *log*

path: /steps/{stepid}/logs

Path Keys **stepid** (*number*) – the id of the step

This path selects all logs for the given step.

GET

returns *collection* of *log*

path: /steps/{stepid}/logs/{log_slug}

Path Keys

- **stepid** (*number*) – the id of the step
- **log_slug** (*identifier*) – the slug name of the log

GET

returns *collection* of *log*

logchunk

resource type: logchunk

Attributes

- **content** (*string*) – content of the chunk
- **firstline** (*integer*) – zero-based line number of the first line in this chunk
- **logid** (*integer*) – the ID of log containing this chunk

A logchunk represents a contiguous sequence of lines in a logfile. Logs are not individually addressable in the data API; instead, they must be requested by line number range. In a strict REST sense, many logchunk resources will contain the same line.

The chunk contents is represented as a single unicode string. This string is the concatenation of each newline terminated-line.

Each log has a type, as identified by the "type" field of the corresponding *log*. While all logs are sequences of unicode lines, the type gives additional information fo interpreting the contents. The defined types are:

- `t` – text, a simple sequence of lines of text
- `s` – stdio, like text but with each line tagged with a stream
- `h` – HTML, represented as plain text

In the stream type, each line is prefixed by a character giving the stream type for that line. The types are `i` for input, `o` for stdout, `e` for stderr, and `h` for header. The first three correspond to normal UNIX standard streams, while the header stream contains metadata produced by Buildbot itself.

The `offset` and `limit` parameters can be used to select the desired lines. These are specified as query parameters via the REST interface, or as arguments to the `get` method in Python. The result will begin with line `offset` (so the resulting `firstline` will be equal to the given `offset`), and will contain up to `limit` lines.

Following example will get the first 100 lines of a log:

```
from buildbot.data import resultspec
first_100_lines = yield self.master.data.get(("logs", log['logid'], "contents"),
    resultSpec=resultspec.ResultSpec(limit=100))
```

Following example will get the last 100 lines of a log:

```
from buildbot.data import resultspec
last_100_lines = yield self.master.data.get(("logs", log['logid'], "contents"),
    resultSpec=resultspec.ResultSpec(offset=log['num_lines']-100))
```

Note: There is no event for a new chunk. Instead, the log resource is updated when new chunks are added, with the new number of lines. Consumers can then request those lines, if desired.

Update Methods

Log chunks are updated via *log*.

Endpoints

path: /builders/{builderid}/builds/{build_number}/steps/{step_name}/logs/{log_slug}/contents

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder
- **step_name** (*identifier*) – the slug name of the step
- **log_slug** (*identifier*) – the slug name of the log

GET

returns *collection* of *logchunk*

path: /builders/{builderid}/builds/{build_number}/steps/{step_number}/logs/{log_slug}/conten

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder
- **step_number** (*number*) – the number of the step
- **log_slug** (*identifier*) – the slug name of the log

GET

returns *collection* of *logchunk*

path: /builds/{buildid}/steps/{step_number_or_name}/logs/{log_slug}/contents

Path Keys

- **buildid** (*number*) – the id of the build
- **| number step_number_or_name** (*identifier*) – the name or number of the step
- **log_slug** (*identifier*) – the slug name of the log

GET

returns *collection* of *logchunk*

path: /logs/{logid}/contents

Path Keys **logid** (*number*) – the id of the log

GET

returns *collection* of *logchunk*

path: /steps/{stepid}/logs/{log_slug}/contents

Path Keys

- **stepid** (*number*) – the id of the step
- **log_slug** (*identifier*) – the slug name of the log

GET

returns *collection* of *logchunk*

master

resource type: master

Attributes

- **active** (*boolean*) – true if the master is active
- **last_active** (*date*) – time this master was last marked active
- **masterid** (*integer*) – the ID of this master
- **name** (*string*) – master name (in the form "hostname:basedir")

This resource type describes buildmasters in the buildmaster cluster.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.masters.`**Master**

masterActive(*name*, *masterid*)

Parameters

- **name** (*unicode*) – the name of this master (generally `hostname:basedir`)
- **masterid** (*integer*) – this master's master ID

Returns Deferred

Mark this master as still active. This method should be called at startup and at least once per minute. The master ID is acquired directly from the database early in the master startup process.

expireMasters()

Returns Deferred

Scan the database for masters that have not checked in for ten minutes. This method should be called about once per minute.

masterStopped(*name*, *masterid*)

Parameters

- **name** (*unicode*) – the name of this master
- **masterid** (*integer*) – this master's master ID

Returns Deferred

Mark this master as inactive. Masters should call this method before completing an expected shutdown, and on startup. This method will take care of deactivating or removing configuration resources like builders and schedulers as well as marking lost builds and build requests for retry.

Endpoints

path: /builders/{builderid}/masters

Path Keys **builderid** (*number*) – the ID of the builder

This path selects all masters supporting a given builder

GET

returns *collection* of *master*

path: /builders/{builderid}/{masterid}

Path Keys

- **builderid** (*number*) – the ID of the builder
- **masterid** (*number*) – the id of the master

This path selects a master by id filtered by given builderid

GET

returns *collection* of *master*

path: /masters

This path selects all masters

GET

returns *collection* of *master*

path: /masters/{masterid}

Path Keys **masterid** (*number*) – the id of the master

This path selects one master given its id

GET

returns *collection* of *master*

patch

resource type: patch

Attributes

- **patchid** (*integer*) – the unique ID of this patch
- **body** (*string*) – patch body as a binary string

- **level** (*integer*) – patch level - the number of directory names to strip from filenames in the patch
- **subdir** (*string*) – subdirectory in which patch should be applied
- **author?** (*string*) – patch author, or None
- **comment?** (*string*) – patch comment, or None

This resource type describes a patch. Patches have unique IDs, but only appear embedded in sourcestamps, so those IDs are not especially useful.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.patches.`**Patch**
(no update methods)

rootlink

resource type: rootlink

Attributes **name** (*string*) –

Endpoints

path: /

GET

returns *collection* of *rootlink*

scheduler

resource type: scheduler

Attributes

- **master?** (*master*) – the master on which this scheduler is running, or None if it is inactive
- **name** (*string*) – name of this scheduler
- **schedulerid** (*integer*) – the ID of this scheduler

A scheduler initiates builds, often in response to changes from change sources. A particular scheduler (by name) runs on at most one master at a time.

Note: This data type and associated endpoints is planned to be merged with forcescheduler data type when bug #2673 (http://trac.buildbot.net/ticket/2673) will be resolved.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.schedulers.`**Scheduler**

findSchedulerId(*name*)

Parameters **name** (*string*) – scheduler name

Returns scheduler ID via Deferred

Get the ID for the given scheduler name, inventing one if necessary.

trySetSchedulerMaster(*schedulerid*, *masterid*)

Parameters

- **schedulerid** (*integer*) – scheduler ID to try to claim
- **masterid** (*integer*) – this master's master ID

Returns `True` or `False`, via Deferred

Try to claim the given scheduler for the given master and return `True` if the scheduler is to be activated on that master.

Endpoints

path: /masters/{masterid}/schedulers

Path Keys **masterid** (*number*) – the id of the master

This path selects all schedulers for a given master

GET

returns *collection* of *scheduler*

path: /masters/{masterid}/schedulers/{schedulerid}

Path Keys

- **masterid** (*number*) – the id of the master
- **schedulerid** (*number*) – the id of the scheduler

This path selects one scheduler by id for a given master

GET

returns *collection* of *scheduler*

path: /schedulers

This path selects all schedulers

GET

returns *collection* of *scheduler*

path: /schedulers/{schedulerid}

Path Keys **schedulerid** (*number*) – the id of the scheduler

This path selects one scheduler by id

GET

returns *collection* of *scheduler*

sourcedproperties

resource type: `sourcedproperties`

Attributes **`[]`** (*object*) – Each key of this map is the name of a defined property The value consist on a couple (source, value)

user-specified properties for this change, represented as an object mapping keys to tuple (value, source)

Properties are present in several data resources, but have a separate endpoints, because they can represent a large dataset.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.properties.`**`Properties`**

`setBuildProperty`(*buildid*, *name*, *value*, *source*)

Parameters

- **`buildid`** (*integer*) – build ID
- **`name`** (*unicode*) – Name of the property to set
- **`value`** (*Any JSON-able type is accepted (lists, dicts, strings and numbers)*) – Value of the property
- **`source`** (*unicode*) – Source of the property to set

Set a build property. If no property with that name exists in that build, a new property will be created.

`setBuildProperties`(*buildid*, *props*)

Parameters

- **`buildid`** (*integer*) – build ID
- **`props`** (*IProperties*) – Name of the property to set

Synchronise build properties with the db. This sends only one event in the end of the sync, and only if properties changed. The event contains only the updated properties, for network efficiency reasons.

Endpoints

path: `/builds/{buildid}/properties`

Path Keys **`buildid`** (*number*) – the id of the build

This path selects all properties of a build

`GET`

returns *collection* of *sourcedproperties*

path: `/buildsets/{bsid}/properties`

Path Keys **`bsid`** (*identifier*) – the id of the buildset

This path selects all properties of a buildset. Buildset properties is part of the initial properties of a build.

GET

returns *collection* of *sourcedproperties*

sourcestamp

resource type: `sourcestamp`

Attributes

- **`ssid`** (*integer*) – the ID of this sourcestamp

  Note: For legacy reasons, the abbreviated name `ssid` is used instead of canonical `sourcestampid`. This might change in the future (bug #3509 (http://trac.buildbot.net/ticket/3509)).

- **`branch?`** (*string*) – code branch, or none for the "default branch", whatever that might mean
- **`codebase`** (*string*) – revision for this sourcestamp, or none if unknown
- **`created_at`** (*date*) – the timestamp when this sourcestamp was created
- **`patch?`** (*patch*) – the patch for this sourcestamp, or none
- **`project`** (*string*) – user-defined project to which this sourcestamp corresponds
- **`repository`** (*string*) – repository where this sourcestamp occurred
- **`revision?`** (*string*) – revision for this sourcestamp, or none if unknown

A source stamp represents a particular version of the source code. Absolute sourcestamps specify this completely, while relative sourcestamps (with revision = None) specify the latest source at the current time. Source stamps can also have patches; such stamps describe the underlying revision with the given patch applied.

Note that, depending on the underlying version-control system, the same revision may describe different code in different branches (e.g., SVN) or may be independent of the branch (e.g., Git).

The `created_at` timestamp can be used to indicate the first time a sourcestamp was seen by Buildbot. This provides a reasonable default ordering for sourcestamps when more reliable information is not available.

Endpoints

path: `/sourcestamps`

This path selects all sourcestamps (can return lots of data!)

GET

returns *collection* of *sourcestamp*

path: `/sourcestamps/{ssid}`

Path Keys **`ssid`** (*number*) – the id of the sourcestamp

This path selects one sourcestamp by id

GET

returns *collection* of *sourcestamp*

spec

resource type: spec

Attributes

- **path** (*string*) –
- **plural** (*string*) –
- **type** (*string*) –
- **type_spec** (*object*) –

Endpoints

path: /application.spec

GET

returns *collection* of *spec*

step

resource type: step

Attributes

- **stepid** (*integer*) – the unique ID of this step
- **buildid** (*integer*) – id of the build containing this step
- **complete** (*boolean*) – true if this step is complete
- **complete_at?** (*date*) – time at which this step was complete, or None if it's still running
- **hidden** (*boolean*) – true if the step should not be displayed
- **name** (*identifier*) – the step name, unique within the build
- **number** (*integer*) – the number of this step (sequential within the build)
- **results?** (*integer*) – the results of the step (see *Build Result Codes*), or None if not complete
- **started_at?** (*date*) – time at which this step started, or None if it hasn't started yet
- **state_string** (*string*) – a string giving detail on the state of the build. The first is usually one word or phrase; the remainder are sized for one-line display.
- **urls[]** – a list of URLs associated with this step.

This resource type describes a step in a build. Steps have unique IDs, but are most commonly accessed by name in the context of their containing builds.

Update Methods

All update methods are available as attributes of `master.data.updates`.

class `buildbot.data.steps.`**Step**

newStep(*buildid*, *name*)

Parameters

- **buildid** (*integer*) – buildid containing this step
- **name** (50-character *identifier*) – name for the step

Returns (stepid, number, name) via Deferred

Create a new step and return its ID, number, and name. Note that the name may be different from the requested name, if that name was already in use. The state strings for the new step will be set to 'pending'.

startStep(*stepid*)

Parameters **stepid** (*integer*) – the step to modify

Start the step.

setStepStateString(*stepid*, *state_string*)

Parameters

- **stepid** (*integer*) – the step to modify
- **state_string** (*unicode*) – new state strings for this step

Replace the existing state string for a step with a new list.

addStepURL(stepid, name, url):

Parameters

- **stepid** (*integer*) – the step to modify
- **name** (*string*) – the url name
- **url** (*string*) – the actual url

Returns None via deferred

Add a new url to a step. The new url is added to the list of urls.

finishStep(*stepid*, *results*, *hidden*)

Parameters

- **stepid** (*integer*) – the step to modify
- **results** (*integer*) – the step's results
- **hidden** (*boolean*) – true if the step should not be displayed

Mark the step as finished at the current time, with the given results.

Endpoints

path: /builders/{builderid}/builds/{build_number}/steps

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder

This path selects all steps for the given build.

GET

returns *collection* of *step*

path: /builders/{builderid}/builds/{build_number}/steps/{step_name}

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder
- **step_name** (*identifier*) – the slug name of the step

This path selects a specific step for the given build.

GET

returns *collection* of *step*

path: /builders/{builderid}/builds/{build_number}/steps/{step_number}

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder
- **step_number** (*number*) – the number of the step

This path selects a specific step given its step number

GET

returns *collection* of *step*

path: /builds/{buildid}/steps

Path Keys **buildid** (*number*) – the id of the build

This path selects all steps of a build

GET

returns *collection* of *step*

path: /builds/{buildid}/steps/{step_number_or_name}

Path Keys

- **buildid** (*number*) – the id of the build
- **| number step_number_or_name** (*identifier*) – the name or number of the step

This path selects one step of a build

GET

returns *collection* of *step*

worker

resource type: `worker`

Attributes

- **`workerid`** (*`integer`*) – the ID of this worker
- **`configured_on[]`** – list of builders on masters this worker is configured on
- **`connected_to[]`** – list of masters this worker is attached to
- **`name`** (*`string`*) – the name of the worker
- **`workerinfo`** (*`object`*) – information about the worker.

  The worker information can be any JSON-able object. In practice, it contains the following keys, based on information provided by the worker:

  - `admin` (the admin information)
  - `host` (the name of the host)
  - `access_uri` (the access URI)
  - `version` (the version on the worker)

A worker resource represents a worker to the source code monitored by Buildbot.

The contents of the `connected_to` and `configured_on` attributes are sensitive to the context of the request. If a builder or master is specified in the path, then only the corresponding connections and configurations are included in the result.

Endpoints

path: `/builders/{builderid}/workers`

Path Keys **`builderid`** (*`number`*) – the ID of the builder

This path selects all workers configured for a given builder

`GET`

`returns` *`collection`* of *`worker`*

path: `/builders/{builderid}/workers/{name}`

Path Keys

- **`builderid`** (*`number`*) – the ID of the builder
- **`name`** (*`identifier`*) – the name of the worker

This path selects a worker by name filtered by given builderid

`GET`

`returns` *`collection`* of *`worker`*

path: `/builders/{builderid}/workers/{workerid}`

Path Keys

- **`builderid`** (*`number`*) – the ID of the builder
- **`workerid`** (*`number`*) – the id of the worker

This path selects a worker by id filtered by given builderid

GET

returns *collection* of *worker*

path: /masters/{masterid}/builders/{builderid}/workers

Path Keys

- **masterid** (*number*) – the id of the master
- **builderid** (*number*) – the id of the builder

This path selects all workers for a given builder and a given master

GET

returns *collection* of *worker*

path: /masters/{masterid}/builders/{builderid}/workers/{name}

Path Keys

- **masterid** (*number*) – the id of the master
- **builderid** (*number*) – the id of the builder
- **name** (*identifier*) – the name of the worker

This path selects one workers by name for a given builder and a given master

GET

returns *collection* of *worker*

path: /masters/{masterid}/builders/{builderid}/workers/{workerid}

Path Keys

- **masterid** (*number*) – the id of the master
- **builderid** (*number*) – the id of the builder
- **workerid** (*number*) – the id of the worker

This path selects one workers by name for a given builder and a given master

GET

returns *collection* of *worker*

path: /masters/{masterid}/workers

Path Keys **masterid** (*number*) – the id of the master

This path selects all workers for a given master

GET

returns *collection* of *worker*

path: /masters/{masterid}/workers/{name}

Path Keys

- **masterid** (*number*) – the id of the master
- **name** (*identifier*) – the name of the worker

This path selects one worker by name for a given master

GET

returns *collection* of *worker*

path: /masters/{masterid}/workers/{workerid}

Path Keys

- **masterid** (*number*) – the id of the master
- **workerid** (*number*) – the id of the worker

This path selects one worker by id for a given master

GET

returns *collection* of *worker*

path: /workers

this path selects all workers

GET

returns *collection* of *worker*

path: /workers/{name_or_id}

Path Keys | number name_or_id (*identifier*) – the name or id of a worker

this path selects worker by name or id

GET

returns *collection* of *worker*

Raw endpoints

Raw endpoints allow to download content in their raw format (i.e. not within a json glue). The `content-disposition` http header is set, so that the browser knows which file to store the content to.

path: /builders/{builderid}/builds/{build_number}/steps/{step_name}/logs/{log_slug}/raw

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder
- **step_name** (*identifier*) – the slug name of the step
- **log_slug** (*identifier*) – the slug name of the log

This endpoint allows to get the raw logs for downloading into a file. This endpoint does not provide paging capabilities. For stream log types, the type line header characters are dropped. 'text/plain' is used as the mime type except for html logs, where 'text/html' is used. The 'slug' is used as the filename for the resulting download. Some browsers are appending `".txt"` or `".html"` to this filename according to the mime-type.

path: /builders/{builderid}/builds/{build_number}/steps/{step_number}/logs/{log_slug}/raw

Path Keys

- **builderid** (*number*) – the ID of the builder
- **build_number** (*number*) – the number of the build within the builder
- **step_number** (*number*) – the number of the step

- **log_slug** (*identifier*) – the slug name of the log

This path downloads the whole log

path: /builds/{buildid}/steps/{step_number_or_name}/logs/{log_slug}/raw

Path Keys

- **buildid** (*number*) – the id of the build
- **| number step_number_or_name** (*identifier*) – the name or number of the step
- **log_slug** (*identifier*) – the slug name of the log

This path downloads the whole log

path: /logs/{logid}/raw

Path Keys **logid** (*number*) – the id of the log

This path downloads the whole log

path: /steps/{stepid}/logs/{log_slug}/raw

Path Keys

- **stepid** (*number*) – the id of the step
- **log_slug** (*identifier*) – the slug name of the log

This path downloads the whole log

Raml spec verbatim

Sometimes Raml is just clearer than formatted text.

```
#%RAML 1.0
title: Buildbot Web API
version: v2
mediaType: application/json
traits:
    bbget:
        responses:
            200:
                body:
                    application/json:
                        type: responseObjects.libraries.types.<<bbtype>>
            404:
                body:
                    text/plain:
                        example: "not found"
    bbpost:
        body:
            type: <<reqtype>>
        responses:
            200:
                body:
                    application/json:
                        type: <<resptype>>
            404:
                body:
                    text/plain:
```

```
                        example: "not found"
    bbgetraw:
        responses:
            200:
                headers:
                    content-disposition:
                        description: content disposition header allows browser to␣
↪save log file with proper filename
                        example: attachment; filename=stdio
                body:
                    text/html:
                        description: "html data if the object is html"
                    text/plain:
                        description: "plain text data if the object is text"

types:
    build: !include types/build.raml
    builder: !include types/builder.raml
    buildrequest: !include types/buildrequest.raml
    buildset: !include types/buildset.raml
    worker: !include types/worker.raml
    change: !include types/change.raml
    changesource: !include types/changesource.raml
    forcescheduler: !include types/forcescheduler.raml
    identifier: !include types/identifier.raml
    log: !include types/log.raml
    logchunk: !include types/logchunk.raml
    master: !include types/master.raml
    rootlink: !include types/rootlink.raml
    scheduler: !include types/scheduler.raml
    sourcedproperties: !include types/sourcedproperties.raml
    sourcestamp: !include types/sourcestamp.raml
    patch: !include types/patch.raml
    spec: !include types/spec.raml
    step: !include types/step.raml
/:
    get:
        is:
        - bbget: {bbtype: rootlink}
/application.spec:
    get:
        is:
        - bbget: {bbtype: spec}
/builders:
    description: This path selects all builders
    get:
        is:
        - bbget: {bbtype: builder}
    /{builderid}:
        uriParameters:
            builderid:
                type: number
                description: the ID of the builder
        description: This path selects a builder by builderid
        get:
            is:
            - bbget: {bbtype: builder}
        /forceschedulers:
```

```
            description: This path selects all force-schedulers for a given builder
            get:
                is:
                - bbget: {bbtype: forcescheduler}
        /buildrequests:
            description: This path selects all buildrequests for a given builder (can␣
↪return lots of data!)
            get:
                is:
                - bbget: {bbtype: buildrequest}
        /builds:
            description: This path selects all builds for a builder (can return lots␣
↪of data!)
            get:
                is:
                - bbget: {bbtype: build}
            /{build_number}:
                uriParameters:
                    build_number:
                        type: number
                        description: the number of the build within the builder
                description: This path selects a specific build by builderid,␣
↪buildnumber
                get:
                    is:
                    - bbget: {bbtype: build}
                /actions/stop:
                    post:
                        description: |
                            stops one build.
                        body:
                            application/json:
                                properties:
                                    reason:
                                        type: string
                                        required: false
                                        description: The reason why the build was␣
↪stopped
                                    results:
                                        type: integer
                                        required: false
                                        description: optionally results value␣
↪override (default CANCELLED)
                /actions/rebuild:
                    post:
                        description: |
                            rebuilds one build.
                        body:
                            application/json:
                                description: no parameter are needed
                /steps:
                    description: This path selects all steps for the given build.
                    get:
                        is:
                        - bbget: {bbtype: step}
                    /{step_name}:
                        uriParameters:
                            step_name:
```

```
                                type: identifier
                                description: the slug name of the step
                        description: This path selects a specific step for the given␣
↪build.
                        get:
                            is:
                            - bbget: {bbtype: step}
                        /logs:
                            description: This path selects all logs for the given␣
↪step.
                            get:
                                is:
                                - bbget: {bbtype: log}
                            /{log_slug}:
                                uriParameters:
                                    log_slug:
                                        type: identifier
                                        description: the slug name of the log
                                get:
                                    description: |
                                        This path selects a specific log in the given␣
↪step.
                                    is:
                                    - bbget: {bbtype: log}
                                /contents:
                                    get:
                                        description: |
                                            This path selects chunks from a specific␣
↪log in the given step.
                                        is:
                                        - bbget: {bbtype: logchunk}
                                /raw:
                                    get:
                                        description: |
                                            This endpoint allows to get the raw logs␣
↪for downloading into a file.

                                            This endpoint does not provide paging␣
↪capabilities.

                                            For stream log types, the type line␣
↪header characters are dropped.

                                            'text/plain' is used as the mime type␣
↪except for html logs, where 'text/html' is used.

                                            The 'slug' is used as the filename for␣
↪the resulting download. Some browsers are appending ``".txt"`` or ``".html"`` to␣
↪this filename according to the mime-type.
                                        is:
                                        - bbgetraw:

                    /{step_number}:
                        uriParameters:
                            step_number:
                                type: number
                                description: the number of the step
                        description: This path selects a specific step given its step␣
↪number
                        get:
                            is:
                            - bbget: {bbtype: step}
```

```
                        /logs:
                            description: This path selects all log of a a specific␣
↪step
                            get:
                                is:
                                - bbget: {bbtype: log}
                            /{log_slug}:
                                uriParameters:
                                    log_slug:
                                        type: identifier
                                        description: the slug name of the log
                                description: This path selects one log of a a␣
↪specific step
                                get:
                                    is:
                                    - bbget: {bbtype: log}
                                /contents:
                                    get:
                                        description: |
                                            This path selects chunks from a specific␣
↪log in the given step.
                                        is:
                                        - bbget: {bbtype: logchunk}
                                /raw:
                                    get:
                                        description: |
                                            This path downloads the whole log
                                        is:
                                        - bbgetraw:

        /workers:
            description: |
                This path selects all workers configured for a given builder
            get:
                is:
                - bbget: {bbtype: worker}
            /{name}:
                description: |
                    This path selects a worker by name filtered by given builderid
                uriParameters:
                    name:
                        type: identifier
                        description: the name of the worker
                get:
                    is:
                    - bbget: {bbtype: worker}
            /{workerid}:
                description: |
                    This path selects a worker by id filtered by given builderid
                uriParameters:
                    workerid:
                        type: number
                        description: the id of the worker
                get:
                    is:
                    - bbget: {bbtype: worker}
        /masters:
            description: |
```

```
                This path selects all masters supporting a given builder
            get:
                is:
                - bbget: {bbtype: master}

        /{masterid}:
            uriParameters:
                masterid:
                    type: number
                    description: the id of the master
            description: |
                This path selects a master by id filtered by given builderid
            get:
                is:
                - bbget: {bbtype: master}
/buildrequests:
    /{buildrequestid}:
        uriParameters:
            buildrequestid:
                type: number
                description: the id of the buildrequest
        get:
            is:
            - bbget: {bbtype: buildrequest}
        /builds:
            get:
                is:
                - bbget: {bbtype: build}
        /actions/cancel:
            post:
                description: |
                    Cancel one buildrequest.
                    If necessary, this will stop the builds generated by the␣
↪buildrequest, including triggered builds.
                body:
                    application/json:
                        properties:
                            reason:
                                type: string
                                required: false
                                description: The reason why the buildrequest was␣
↪cancelled
    get:
        is:
        - bbget: {bbtype: buildrequest}
/builds:
    get:
        is:
        - bbget: {bbtype: build}
    /{buildid}:
        description: |
            This path selects a build by id
        uriParameters:
            buildid:
                type: number
                description: the id of the build
        get:
            is:
```

```
        - bbget: {bbtype: build}
    /actions/stop:
        post:
            description: |
                stops one build.
            body:
                application/json:
                    properties:
                        reason:
                            type: string
                            required: false
                            description: The reason why the build was stopped
    /actions/rebuild:
        post:
            description: |
                rebuilds one build.
            body:
                application/json:
                    description: no parameter are needed
    /changes:
        description: |
            This path selects all changes tested by a build
        get:
            is:
            - bbget: {bbtype: change}
    /properties:
        description: |
            This path selects all properties of a build
        get:
            is:
            - bbget: {bbtype: sourcedproperties}
    /steps:
        description: |
            This path selects all steps of a build
        get:
            is:
            - bbget: {bbtype: step}
        /{step_number_or_name}:
            uriParameters:
                step_number_or_name:
                    type: identifier | number
                    description: the name or number of the step
            description: |
                This path selects one step of a build
            get:
                is:
                - bbget: {bbtype: step}
            /logs:
                description: |
                    This path selects all logs of a step of a build
                get:
                    is:
                    - bbget: {bbtype: log}
                /{log_slug}:
                    uriParameters:
                        log_slug:
                            type: identifier
                            description: the slug name of the log
```

```
                        description: This path selects one log of a a specific step
                        get:
                            is:
                            - bbget: {bbtype: log}
                        /contents:
                            get:
                                description: |
                                    This path selects chunks from a specific log in␣
↪the given step.
                                is:
                                - bbget: {bbtype: logchunk}
                        /raw:
                            get:
                                description: |
                                    This path downloads the whole log
                                is:
                                - bbgetraw:
/buildsets:
    description: this path selects all buildsets
    get:
        is:
        - bbget: {bbtype: buildset}
    /{bsid}:
        description: this path selects a buildset by id
        uriParameters:
            bsid:
                type: identifier
                description: the id of the buildset
        get:
            is:
            - bbget: {bbtype: buildset}
        /properties:
            description: |
                This path selects all properties of a buildset.
                Buildset properties is part of the initial properties of a build.
            get:
                is:
                - bbget: {bbtype: sourcedproperties}
/workers:
    description: this path selects all workers
    get:
        is:
        - bbget: {bbtype: worker}
    /{name_or_id}:
        description: this path selects worker by name or id
        uriParameters:
            name_or_id:
                type: identifier | number
                description: the name or id of a worker
        get:
            is:
            - bbget: {bbtype: worker}

/changes:
    description: |
        This path selects **all** changes.
        On a reasonabily loaded master, this can quickly return a very large result,␣
↪taking minutes to process.
```

```
        A specific query configuration is optimized which allows to get the recent␣
↪changes: ``order:-changeid&limit=<n>``
    get:
        is:
        - bbget: {bbtype: change}
    /{changeid}:
        description: this path selects one change by id
        uriParameters:
            changeid:
                type: number
                description: the id of a change
        get:
            is:
            - bbget: {bbtype: change}

/changesources:
    description: |
        This path selects all changesource.
    get:
        is:
        - bbget: {bbtype: changesource}
    /{changesourceid}:
        uriParameters:
            changesourceid:
                type: number
                description: the id of a changesource
        description: |
            This path selects one changesource given its id.
        get:
            is:
            - bbget: {bbtype: changesource}
/forceschedulers:
    description: |
        This path selects all forceschedulers.
    get:
        is:
        - bbget: {bbtype: forcescheduler}

    /{schedulername}:
        description: |
            This path selects all changesource.
        uriParameters:
            schedulername:
                type: identifier
                description: the name of a scheduler
        get:
            is:
            - bbget: {bbtype: forcescheduler}

        /actions/force:
            post:
                description: |
                    Triggers the forcescheduler
                body:
                    application/json:
                        properties:
                            owner:
                                type: string
```

```
                                required: false
                                description: The user who wants to create the␣
↪buildrequest
                            '[]':
                                description: content of the forcescheduler parameter␣
↪is dependent on the configuration of the forcescheduler
/logs/{logid}:
    uriParameters:
        logid:
            type: number
            description: the id of the log
    description: This path selects one log
    get:
        is:
        - bbget: {bbtype: log}
    /contents:
        get:
            description: |
                This path selects chunks from a specific log
            is:
            - bbget: {bbtype: logchunk}
    /raw:
        get:
            description: |
                This path downloads the whole log
            is:
            - bbgetraw:
/masters:
    description: This path selects all masters
    get:
        is:
        - bbget: {bbtype: master}
    /{masterid}:
        description: This path selects one master given its id
        uriParameters:
            masterid:
                type: number
                description: the id of the master
        get:
            is:
            - bbget: {bbtype: master}
        /builders:
            description: This path selects all builders of a given master
            get:
                is:
                - bbget: {bbtype: builder}
            /{builderid}:
                description: This path selects one builder by id of a given master
                uriParameters:
                    builderid:
                        type: number
                        description: the id of the builder
                get:
                    is:
                    - bbget: {bbtype: builder}
                /workers:
                    description: This path selects all workers for a given builder␣
↪and a given master
```

```
                    get:
                        is:
                        - bbget: {bbtype: worker}
                    /{name}:
                        description: This path selects one workers by name for a␣
↪given builder and a given master
                        uriParameters:
                            name:
                                type: identifier
                                description: the name of the worker
                        get:
                            is:
                            - bbget: {bbtype: worker}
                    /{workerid}:
                        description: This path selects one workers by name for a␣
↪given builder and a given master
                        uriParameters:
                            workerid:
                                type: number
                                description: the id of the worker
                        get:
                            is:
                            - bbget: {bbtype: worker}
        /workers:
            description: This path selects all workers for a given master
            get:
                is:
                - bbget: {bbtype: worker}
            /{name}:
                description: This path selects one worker by name for a given master
                uriParameters:
                    name:
                        type: identifier
                        description: the name of the worker
                get:
                    is:
                    - bbget: {bbtype: worker}
            /{workerid}:
                description: This path selects one worker by id for a given master
                uriParameters:
                    workerid:
                        type: number
                        description: the id of the worker
                get:
                    is:
                    - bbget: {bbtype: worker}
        /changesources:
            description: This path selects all changesources for a given master
            get:
                is:
                - bbget: {bbtype: changesource}
            /{changesourceid}:
                description: This path selects one changesource by id for a given␣
↪master
                get:
                    is:
                    - bbget: {bbtype: changesource}
        /schedulers:
```

```
            description: This path selects all schedulers for a given master
            get:
                is:
                - bbget: {bbtype: scheduler}
            /{schedulerid}:
                description: This path selects one scheduler by id for a given master
                uriParameters:
                    schedulerid:
                        type: number
                        description: the id of the scheduler
                get:
                    is:
                    - bbget: {bbtype: scheduler}
/schedulers:
    description: This path selects all schedulers
    get:
        is:
        - bbget: {bbtype: scheduler}
    /{schedulerid}:
        uriParameters:
            schedulerid:
                type: number
                description: the id of the scheduler
        description: This path selects one scheduler by id
        get:
            is:
            - bbget: {bbtype: scheduler}
/sourcestamps:
    description: This path selects all sourcestamps (can return lots of data!)
    get:
        is:
        - bbget: {bbtype: sourcestamp}
    /{ssid}:
        description: This path selects one sourcestamp by id
        uriParameters:
            ssid:
                type: number
                description: the id of the sourcestamp
        get:
            is:
            - bbget: {bbtype: sourcestamp}
        /changes:
            description: This path selects all changes associated to one sourcestamp
            get:
                is:
                - bbget: {bbtype: change}
/steps:
    /{stepid}:
        description: This path selects one step by id
        uriParameters:
            stepid:
                type: number
                description: the id of the step
        /logs:
            description: This path selects all logs for the given step.
            get:
                is:
                - bbget: {bbtype: log}
```

```
            /{log_slug}:
                uriParameters:
                    log_slug:
                        type: identifier
                        description: the slug name of the log
                get:
                    description: |
                        This path selects a specific log in the given step.
                    is:
                    - bbget: {bbtype: log}
                /contents:
                    get:
                        description: |
                            This path selects chunks from a specific log in the given␣
↪step.
                        is:
                        - bbget: {bbtype: logchunk}
                /raw:
                    get:
                        description: |
                            This path downloads the whole log
                        is:
                        - bbgetraw:
```

Data API

The data layer combines access to stored state and messages, ensuring consistency between them, and exposing a well-defined API that can be used both internally and externally. Using caching and the clock information provided by the db and mq layers, this layer ensures that its callers can easily receive a dump of current state plus changes to that state, without missing or duplicating messages.

Sections

The data api is divided into four sections:

- getters - fetching data from the db API, and
- subscriptions - subscribing to messages from the mq layer;
- control - allows state to be changed in specific ways by sending appropriate messages (e.g., stopping a build); and
- updates - direct updates to state appropriate messages.

The getters and subscriptions are exposed everywhere. Access to the control section should be authenticated at higher levels, as the data layer does no authentication. The updates section is for use only by the process layer.

The interfaces for all sections but the updates sections are intended to be language-agnostic. That is, they should be callable from JavaScript via HTTP, or via some other interface added to Buildbot after the fact.

Getter

The getter section can get either a single resource, or a list of resources. Getting a single resource requires a resource identifier (a tuple of strings) and a set of options to support automatic expansion of links to other resources (thus

saving round-trips). Lists are requested with a partial resource identifier (a tuple of strings) and an optional set of filter options. In some cases, certain filters are implicit in the path, e.g., the list of buildsteps for a particular build.

Subscriptions

Message subscriptions can be made to anything that can be listed or gotten from the getter sections, using the same resource identifiers. Options and explicit filters are not supported - a message contains only the most basic information about a resource, and a list subscription results in a message for every new resource of the desired type. Implicit filters are supported.

Control

The control sections defines a set of actions that cause Buildbot to behave in a certain way, e.g., rebuilding a build or shutting down a worker. Actions correspond to a particular resource, although sometimes that resource is the root resource (an empty tuple).

Updates

The updates section defines a free-form set of methods that Buildbot's process implementation calls to update data. Most update methods both modify state via the db API and send a message via the mq API. Some are simple wrappers for these APIs, while others contain more complex logic, e.g., building a source stamp set for a collection of changes. This section is the proper place to put common functionality, e.g., rebuilding builds or assembling buildsets.

Concrete Interfaces

Python Interface

Within the buildmaster process, the root of the data API is available at *self.master.data*, which is a *DataConnector* instance.

class buildbot.data.connector.**DataConnector**

This class implements the root of the data API. Within the buildmaster process, the data connector is available at *self.master.data*. The first three sections are implemented with the `get` and *control* methods, respectively, while the updates section is implemented using the `updates` attribute. The `path` arguments to these methods should always be tuples. Integer arguments can be presented as either integers or strings that can be parsed by `int`; all other arguments must be strings.

get(path, filters=None, fields=None, order=None, limit=None, offset=None):

Parameters

- **path** (*tuple*) – A tuple of path elements representing the API path to fetch. Numbers can be passed as strings or integers.
- **filters** – result spec filters
- **fields** – result spec fields
- **order** – result spec order
- **limit** – result spec limit
- **offset** – result spec offset

Raises *InvalidPathError*

Returns a resource or list via Deferred, or None

This method implements the getter section. Depending on the path, it will return a single resource or a list of resources. If a single resource is not specified, it returns `None`.

The `filters`, `fields`, `order`, `limit`, and `offset` are passed to the *ResultSpec* constructor.

The return value is composed of simple Python objects - lists, dicts, strings, numbers, and None.

getEndpoint(*path*)

Parameters **path** (*tuple*) – A tuple of path elements representing the API path. Numbers can be passed as strings or integers.

Raises *InvalidPathError*

Returns tuple of endpoint and a dictionary of keyword arguments from the path

Get the endpoint responsible for the given path, along with any arguments extracted from the path. This can be used by callers that need access to information from the endpoint beyond that returned from `get`.

produceEvent(*rtype*, *msg*, *event*)

Parameters

- **rtype** – the name identifying a resource type
- **msg** – a dictionary describing the msg to send
- **event** – the event to produce

This method implements the production of an event, for the rtype identified by its name string. Usually, this is the role of the data layer to produce the events inside the update methods. For the potential use cases where it would make sense to solely produce an event, and not update data, please use this API, rather than directly call mq. It ensures the event is sent to all the routingkeys specified by eventPathPatterns.

control(*action*, *args*, *path*)

Parameters

- **action** – a short string naming the action to perform
- **args** – dictionary containing arguments for the action
- **path** (*tuple*) – A tuple of path elements representing the API path. Numbers can be passed as strings or integers.

Raises *InvalidPathError*

Returns a resource or list via Deferred, or None

This method implements the control section. Depending on the path, it may return a new created resource.

allEndpoints()

Returns list of endpoint specifications

This method returns the deprecated API spec. Please use *Raml Specs* instead.

rtypes

This object has an attribute named for each resource type, named after the singular form (e.g., *self.master.data.builder*). These attributes allow resource types to access one another for purposes of coordination. They are *not* intended for external access – all external access to the data API should be via the methods above or update methods.

Updates

The updates section is available at *self.master.data.updates*, and contains a number of ad-hoc methods needed by the process modules.

Note: The update methods are implemented in resource type classes, but through some initialization-time magic, all appear as attributes of `self.master.data.updates`.

The update methods are found in the resource type pages.

Exceptions

exception `buildbot.data.exceptions.`**`DataException`**
This is a base class for all other Data API exceptions.

exception `buildbot.data.exceptions.`**`InvalidPathError`**
The path argument was invalid or unknown.

exception `buildbot.data.exceptions.`**`InvalidOptionError`**
A value in the `options` argument was invalid or ill-formed.

exception `buildbot.data.exceptions.`**`SchedulerAlreadyClaimedError`**
Identical to *`SchedulerAlreadyClaimedError`*.

Web Interface

The HTTP interface is implemented by the `buildbot.www` package, as configured by the user. Part of that configuration is a base URL, which is considered a prefix for all paths mentioned here.

See *Base web application* for more information.

Extending the Data API

The data API may be extended in various ways: adding new endpoints, new fields to resource types, new update methods, or entirely new resource types. In any case, you should only extend the API if you plan to submit the extensions to be merged into Buildbot itself. Private API extensions are strongly discouraged.

Adding Resource Types

You'll need to use both plural and singular forms of the resource type; in this example, we'll use 'pub' and 'pubs'. You can also follow an existing file, like master/buildbot/data/changes.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/data/changes.py), to see when to use which form.

In `master/buildbot/data/pubs.py`, create a subclass of *`ResourceType`*:

```
from buildbot.data import base

class Pub(base.ResourceType):
    name = "pub"
    endpoints = []
    keyFields = ['pubid']
```

```
    class EntityType(types.Entity):
        pubid = types.Integer()
        name = types.String()
        num_taps = types.Integer()
        closes_at = types.Integer()

    entityType = EntityType(name)
```

class buildbot.data.base.**ResourceType**

name

Type string

The singular, lower-cased name of the resource type. This becomes the first component in message routing keys.

plural

Type string

The plural, lower-cased name of the resource type. This becomes the key containing the data in REST responses.

endpoints

Type list

Subclasses should set this to a list of endpoint classes for this resource type.

eventPathPatterns

Type str

This attribute should list the message routes where events should be sent, encoded as a REST like endpoint:

```
pub/:pubid
```

In the example above, a call to `produceEvent({'pubid': 10, 'name': 'Winchester'}, 'opened')` would result in a message with routing key `('pub', '10', 'opened')`.

Several paths can be specified in order to be consistent with rest endpoints.

entityType

Type *buildbot.data.types.Entity*

The entity type describes the types of all of the fields in this particular resource type. See *buildbot.data.types.Entity* and *Adding Fields to Resource Types*.

The parent class provides the following methods

getEndpoints()

Returns a list of *Endpoint* instances

This method returns a list of the endpoint instances associated with the resource type.

The base method instantiates each class in the *endpoints* attribute. Most subclasses can simply list *Endpoint* subclasses in `endpoints`.

produceEvent(*msg*, *event*)

Parameters

- **msg** (*dict*) – the message body
- **event** (*string*) – the name of the event that has occurred

This is a convenience method to produce an event message for this resource type. It formats the routing key correctly and sends the message, thereby ensuring consistent routing-key structure.

Like all Buildbot source files, every resource type module must have corresponding tests. These should thoroughly exercise all update methods.

All resource types must be documented in the Buildbot documentation and linked from the bottom of this file (master/docs/developer/data.rst (https://github.com/buildbot/buildbot/blob/master/master/docs/developer/data.rst)).

Adding Endpoints

Each resource path is implemented as an *Endpoint* instance. In most cases, each instance is of a different class, but this is not required.

The data connector's `get` and *control* methods both take a `path` argument that is used to look up the corresponding endpoint. The path matching is performed by *buildbot.util.pathmatch*, and supports automatically extracting variable fields from the path. See that module's description for details.

class `buildbot.data.base.`**Endpoint**

pathPatterns

Type string

This attribute defines the path patterns which incoming paths must match to select this endpoint. Paths are specified as URIs, and can contain variables as parsed by *buildbot.util.pathmatch.Matcher*. Multiple paths are separated by whitespace.

For example, the following specifies two paths with the second having a single variable:

```
pathPatterns = """
    /bugs
    /component/i:component_name/bugs
"""
```

rootLinkName

Type string

If set, then the first path pattern for this endpoint will be included as a link in the root of the API. This should be set for any endpoints that begin an explorable tree.

isCollection

Type boolean

If true, then this endpoint returns collections of resources.

isRaw

Type boolean

If true, then this endpoint returns raw resource.

Raw resources are used to get the data not encoded in JSON via the rest API. In the REST principles, this should be done via another endpoint, and not via a query parameter. The get() method from endpoint should return following data structure:

```
{
    "raw": u"raw data to be sent to the http client",
    "mime-type": u"<mime-type>",
    "filename": u"filename_to_be_used_in_content_disposition_attachement_
↪header"
}
```

get (*options*, *resultSpec*, *kwargs*)

Parameters

- **options** (*dict*) – model-specific options
- **resultSpec** – a *ResultSpec* instance describing the desired results
- **kwargs** (*dict*) – fields extracted from the path

Returns data via Deferred

Get data from the endpoint. This should return either a list of dictionaries (for list endpoints), a dictionary, or None (both for details endpoints). The endpoint is free to handle any part of the result spec. When doing so, it should remove the relevant configuration from the spec. See below.

Any result spec configuration that remains on return will be applied automatically.

control (*action*, *args*, *kwargs*)

Parameters

- **action** – a short string naming the action to perform
- **args** – dictionary containing arguments for the action
- **kwargs** – fields extracted from the path

Continuing the pub example, a simple endpoint would look like this:

```
class PubEndpoint(base.Endpoint):
    pathPattern = ('pub', 'i:pubid')

    def get(self, resultSpec, kwargs):
        return self.master.db.pubs.getPub(kwargs['pubid'])
```

Endpoint implementations must have unit tests. An endpoint's path should be documented in the `.rst` file for its resource type.

The initial pass at implementing any endpoint should just ignore the `resultSpec` argument to `get`. After that initial pass, the argument can be used to optimize certain types of queries. For example, if the resource type has many resources, but most real-life queries use the result spec to filter out all but a few resources from that group, then it makes sense for the endpoint to examine the result spec and allow the underlying DB API to do that filtering.

When an endpoint handles parts of the result spec, it must remove those parts from the spec before it returns. See the documentation for *ResultSpec* for methods to do so.

Note that endpoints must be careful not to alter the order of the filtering applied for a result spec. For example, if an endpoint implements pagination, then it must also completely implement filtering and ordering, since those operations precede pagination in the result spec application.

Adding Messages

Message types are defined in master/buildbot/test/util/validation.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/tes via the `message` module-level value. This is a dictionary of `MessageValidator` objects, one for each message

type. The message type is determined from the first atom of its routing key. The `events` dictionary lists the possible last atoms of the routing key. It should be identical to the attribute of the ResourceType with the same name.

Adding Update Methods

Update methods are for use by the Buildbot process code, and as such are generally designed to suit the needs of that code. They generally encapsulate logic common to multiple users (e.g., creating buildsets), and finish by performing modifications in the database and sending a corresponding message. In general, Buildbot does not depend on timing of either the database or message broker, so the order in which these operations are initiated is not important.

Update methods are considered part of Buildbot's user-visible interface, and as such incompatible changes should be avoided wherever possible. Instead, either add a new method (and potentially re-implement existing methods in terms of the new method) or add new, optional parameters to an existing method. If an incompatible change is unavoidable, it should be described clearly in the release notes.

Update methods are implemented as methods of *ResourceType* subclasses, decorated with `@base.updateMethod`:

`buildbot.data.base.`**`updateMethod`**(*f*)
A decorator for *ResourceType* subclass methods, indicating that the method should be copied to `master.data.updates`.

Returning to the pub example:

```
class PubResourceType(base.ResourceType):
    # ...
    @base.updateMethod
    @defer.inlineCallbacks
    def setPubTapList(self, pubid, beers):
        pub = yield self.master.db.pubs.getPub(pubid)
        # ...
        self.produceMessage(pub, 'taps-updated')
```

Update methods should be documented in master/docs/developer/data.rst (https://github.com/buildbot/buildbot/blob/master/master/docs/
They should be thoroughly tested with unit tests. They should have a fake implentation in master/buildbot/test/fake/fakedata.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/fake/fakedata.py).
That fake implementation should be tested to match the real implementation in master/buildbot/test/unit/test_data_connector.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/unit/test_data_conne

Adding Fields to Resource Types

The details of the fields of a resource type are rigorously enforced at several points in the Buildbot tests. The enforcement is performed by the *buildbot.data.types* module.

The module provides a number of type classes for basic and compound types. Each resource type class defines its entity type in its *entityType* class attribute. Other resource types may refer to this class attribute if they embed an entity of that type.

The types are used both for tests, and by the REST interface to properly decode user-supplied query parameters.

Basic Types

class `buildbot.data.types.`**`Integer`**
An integer.

```
myid = types.Integer()
```

class buildbot.data.types.**String**
A string. Strings must always be Unicode.

```
name = types.String()
```

class buildbot.data.types.**Binary**
A binary bytestring.

```
data = types.Binary()
```

class buildbot.data.types.**Boolean**
A boolean value.

```
complete = types.Boolean()
```

class buildbot.data.types.**Identifier**(*length*)
An identifier; see *Identifier*. The constructor argument specifies the maximum length.

```
ident = types.Identifier(25)
```

Compound Types

class buildbot.data.types.**NoneOk**(*nestedType*)
Either the nested type, or None.

```
category = types.NoneOk(types.String())
```

class buildbot.data.types.**List**(*of*)
An list of objects. The named constructor argument `of` specifies the type of the list elements.

```
tags = types.List(of=types.String())
```

class buildbot.data.types.**SourcedProperties**
A data structure representing properties with their sources, in the form `{name: (value, source)}`. The property name and source must be Unicode, and the value must be JSON-able.

```
props = types.SourcedProperties()
```

Entity Type

class buildbot.data.types.**Entity**(*name*)
A data resource is represented by a dictionary with well-known keys. To define those keys and their values, subclass the *Entity* class within your ResourceType class and include each field as an attribute:

```
class MyStuff(base.ResourceType):
    name = "mystuff"
    # ...
    class EntityType(types.Entity):
        myid = types.Integer()
        name = types.String()
        data = types.Binary()
```

```
        complete = types.Boolean()
        ident = types.Identifier(25)
        category = types.NoneOk(types.String())
        tags = types.List(of=types.String())
        props = types.SourcedProperties()
```

Then instantiate the class with the resource type name:

```
entityType = EntityType(name)
```

To embed another entity type, reference its entityType class attribute:

```
class EntityType(types.Entity):
    # ...
    master = masters.Master.entityType
```

Data Model

The data api enforces a strong and well-defined model on Buildbot's data. This model is influenced by REST, in the sense that it defines resources, representations for resources, and identifiers for resources. For each resource type, the API specifies

- the attributes of the resource and their types (e.g., changes have a string specifying their project);
- the format of links to other resources (e.g., buildsets to sourcestamp sets);
- the paths relating to the resource type;
- the format of routing keys for messages relating to the resource type;
- the events that can occur on that resource (e.g., a buildrequest can be claimed); and
- options and filters for getting resources.

Some resource type attributes only appear in certain formats, as noted in the documentation for the resource types. In general, messages do not include any optional attributes, nor links.

Paths are given here separated by slashes, with key names prefixed by `:` and described below. Similarly, message routing keys given here are separated by dots, with key names prefixed by `$`. The translation to tuples and other formats should be obvious.

All strings in the data model are unicode strings.

Database

As of version 0.8.0, Buildbot has used a database as part of its storage backend. This section describes the database connector classes, which allow other parts of Buildbot to access the database. It also describes how to modify the database schema and the connector classes themselves.

Database Overview

All access to the Buildbot database is mediated by database connector classes. These classes provide a functional, asynchronous interface to other parts of Buildbot, and encapsulate the database-specific details in a single location in the codebase.

The connector API, defined below, is a stable API in Buildbot, and can be called from any other component. Given a master `master`, the root of the database connectors is available at `master.db`, so, for example, the state connector's `getState` method is `master.db.state.getState`.

The connectors all use SQLAlchemy Core (http://www.sqlalchemy.org/docs/index.html) to achieve (almost) database-independent operation. Note that the SQLAlchemy ORM is not used in Buildbot. Database queries are carried out in threads, and report their results back to the main thread via Twisted Deferreds.

Schema

The database schema is maintained with SQLAlchemy-Migrate (https://github.com/openstack/sqlalchemy-migrate). This package handles the details of upgrading users between different schema versions.

The schema itself is considered an implementation detail, and may change significantly from version to version. Users should rely on the API (below), rather than performing queries against the database itself.

API

types

Identifier

An "identifier" is a nonempty unicode string of limited length, containing only ASCII alphanumeric characters along with - (dash) and _ (underscore), and not beginning with a digit Wherever an identifier is used, the documentation will give the maximum length in characters. The function *buildbot.util.identifiers.isIdentifier* is useful to verify a well-formed identifier.

buildrequests

exception `buildbot.db.buildrequests.`**`AlreadyClaimedError`**
Raised when a build request is already claimed, usually by another master.

exception `buildbot.db.buildrequests.`**`NotClaimedError`**
Raised when a build request is not claimed by this master.

class `buildbot.db.buildrequests.`**`BuildRequestsConnectorComponent`**
This class handles the complex process of claiming and unclaiming build requests, based on a polling model: callers poll for unclaimed requests with *getBuildRequests*, then attempt to claim the requests with *claimBuildRequests*. The claim can fail if another master has claimed the request in the interim.

An instance of this class is available at `master.db.buildrequests`.

Build requests are indexed by an ID referred to as a *brid*. The contents of a request are represented as build request dictionaries (brdicts) with keys

- `buildrequestid`
- `buildsetid`
- `builderid`
- `buildername`
- `priority`
- `claimed` (boolean, true if the request is claimed)

- `claimed_at` (datetime object, time this request was last claimed)
- `claimed_by_masterid` (integer, the id of the master that claimed this buildrequest)
- `complete` (boolean, true if the request is complete)
- `complete_at` (datetime object, time this request was completed)
- `submitted_at` (datetime object, time this request was completed)
- `results` (integer result code)
- `waited_for` (boolean)

getBuildRequest(*brid*)

Parameters **brid** – build request id to look up

Returns brdict or `None`, via Deferred

Get a single BuildRequest, in the format described above. This method returns `None` if there is no such buildrequest. Note that build requests are not cached, as the values in the database are not fixed.

getBuildRequests(*buildername=None*, *complete=None*, *claimed=None*, *bsid=None*, *branch=None*, *repository=None*)

Parameters

- **buildername** (*string*) – limit results to buildrequests for this builder
- **complete** – if true, limit to completed buildrequests; if false, limit to incomplete buildrequests; if `None`, do not limit based on completion.
- **claimed** – see below
- **bsid** – see below
- **repository** – the repository associated with the sourcestamps originating the requests
- **branch** – the branch associated with the sourcestamps originating the requests

Returns list of brdicts, via Deferred

Get a list of build requests matching the given characteristics.

Pass all parameters as keyword parameters to allow future expansion.

The `claimed` parameter can be `None` (the default) to ignore the claimed status of requests; `True` to return only claimed builds, `False` to return only unclaimed builds, or a `master ID` to return only builds claimed by a particular master instance. A request is considered unclaimed if its `claimed_at` column is either NULL or 0, and it is not complete. If `bsid` is specified, then only build requests for that buildset will be returned.

A build is considered completed if its `complete` column is 1; the `complete_at` column is not consulted.

claimBuildRequests(*brids*[, *claimed_at=XX*])

Parameters

- **brids** (*list*) – ids of buildrequests to claim
- **claimed_at** (*datetime*) – time at which the builds are claimed

Returns Deferred

Raises *AlreadyClaimedError*

Try to "claim" the indicated build requests for this buildmaster instance. The resulting deferred will fire normally on success, or fail with *AlreadyClaimedError* if *any* of the build requests are already claimed by another master instance. In this case, none of the claims will take effect.

If `claimed_at` is not given, then the current time will be used.

As of 0.8.5, this method can no longer be used to re-claim build requests. All given ID's must be unclaimed. Use *reclaimBuildRequests* to reclaim.

Note: On database backends that do not enforce referential integrity (e.g., SQLite), this method will not prevent claims for nonexistent build requests. On database backends that do not support transactions (MySQL), this method will not properly roll back any partial claims made before an *AlreadyClaimedError* is generated.

---

reclaimBuildRequests(*brids*)

Parameters **brids** (*list*) – ids of buildrequests to reclaim

Returns Deferred

Raises *AlreadyClaimedError*

Re-claim the given build requests, updating the timestamp, but checking that the requests are owned by this master. The resulting deferred will fire normally on success, or fail with *AlreadyClaimedError* if *any* of the build requests are already claimed by another master instance, or don't exist. In this case, none of the reclaims will take effect.

unclaimBuildRequests(*brids*)

Parameters **brids** (*list*) – ids of buildrequests to unclaim

Returns Deferred

Release this master's claim on all of the given build requests. This will not unclaim requests that are claimed by another master, but will not fail in this case. The method does not check whether a request is completed.

completeBuildRequests(*brids*, *results*[, *complete_at=XX*])

Parameters

- **brids** (*integer*) – build request IDs to complete
- **results** (*integer*) – integer result code
- **complete_at** (*datetime*) – time at which the buildset was completed

Returns Deferred

Raises *NotClaimedError*

Complete a set of build requests, all of which are owned by this master instance. This will fail with *NotClaimedError* if the build request is already completed or does not exist. If `complete_at` is not given, the current time will be used.

unclaimExpiredRequests(*old*)

Parameters **old** (*int*) – number of seconds after which a claim is considered old

Returns Deferred

Find any incomplete claimed builds which are older than `old` seconds, and clear their claim information.

This is intended to catch builds that were claimed by a master which has since disappeared. As a side effect, it will log a message if any requests are unclaimed.

builds

class buildbot.db.builds.**BuildsConnectorComponent**

This class handles builds. One build record is created for each build performed by a master. This record contains information on the status of the build, as well as links to the resources used in the build: builder, master, worker, etc.

An instance of this class is available at `master.db.builds`.

Builds are indexed by *buildid* and their contents represented as *builddicts* (build dictionaries), with the following keys:

- `id` (the build ID, globally unique)
- `number` (the build number, unique only within the builder)
- `builderid` (the ID of the builder that performed this build)
- `buildrequestid` (the ID of the build request that caused this build)
- `workerid` (the ID of the worker on which this build was performed)
- `masterid` (the ID of the master on which this build was performed)
- `started_at` (datetime at which this build began)
- `complete_at` (datetime at which this build finished, or None if it is ongoing)
- `state_string` (short string describing the build's state)
- `results` (results of this build; see *Build Result Codes*)

getBuild(*buildid*)

Parameters **buildid** (*integer*) – build id

Returns Build dictionary as above or `None`, via Deferred

Get a single build, in the format described above. Returns `None` if there is no such build.

getBuildByNumber(*builderid*, *number*)

Parameters

- **builder** (*integer*) – builder id
- **number** (*integer*) – build number within that builder

Returns Build dictionary as above or `None`, via Deferred

Get a single build, in the format described above, specified by builder and number, rather than build id. Returns `None` if there is no such build.

getPrevSuccessfulBuild(*builderid*, *number*, *ssBuild*)

Parameters

- **builderid** (*integer*) – builder to get builds for
- **number** (*integer*) – the current build number. Previous build will be taken from this number
- **ssBuild** (*list*) – the list of sourcestamps for the current build number

Returns None or a build dictionary

Returns the last successful build from the current build number with the same repository/repository/codebase

getBuilds(*builderid=None*, *buildrequestid=None*, *complete=None*)

Parameters

- **builderid** (*integer*) – builder to get builds for
- **buildrequestid** (*integer*) – buildrequest to get builds for
- **complete** (*boolean*) – if not None, filters results based on completeness

Returns list of build dictionaries as above, via Deferred

Get a list of builds, in the format described above. Each of the parameters limit the resulting set of builds.

addBuild(*builderid*, *buildrequestid*, *workerid*, *masterid*, *state_string*)

Parameters

- **builderid** (*integer*) – builder to get builds for
- **buildrequestid** (*integer*) – build request id
- **workerid** (*integer*) – worker performing the build
- **masterid** (*integer*) – master performing the build
- **state_string** (*unicode*) – initial state of the build

Returns tuple of build ID and build number, via Deferred

Add a new build to the db, recorded as having started at the current time. This will invent a new number for the build, unique within the context of the builder.

setBuildStateString(buildid, state_string):

Parameters

- **buildid** (*integer*) – build id
- **state_string** (*unicode*) – updated state of the build

Returns Deferred

Update the state strings for the given build.

finishBuild(*buildid*, *results*)

Parameters

- **buildid** (*integer*) – build id
- **results** (*integer*) – build result

Returns Deferred

Mark the given build as finished, with `complete_at` set to the current time.

Note: This update is done unconditionally, even if the build is already finished.

---

getBuildProperties(*buildid*)

Parameters **buildid** – build ID

Returns dictionary mapping property name to `value, source`, via Deferred

Return the properties for a build, in the same format they were given to *addBuild*.

Note that this method does not distinguish a non-existent build from a build with no properties, and returns `{}` in either case.

setBuildProperty(*buildid*, *name*, *value*, *source*)

Parameters

- **buildid** (*integer*) – build ID
- **name** (*string*) – Name of the property to set
- **value** – Value of the property
- **source** (*string*) – Source of the Property to set

Returns Deferred

Set a build property. If no property with that name existed in that build, a new property will be created.

steps

class buildbot.db.steps.**StepsConnectorComponent**

This class handles the steps performed within the context of a build. Within a build, each step has a unique name and a unique, 0-based number.

An instance of this class is available at `master.db.steps`.

Builds are indexed by *stepid* and their contents represented as *stepdicts* (step dictionaries), with the following keys:

- `id` (the step ID, globally unique)
- `number` (the step number, unique only within the build)
- `name` (the step name, an 50-character *identifier* unique only within the build)
- `buildid` (the ID of the build containing this step)
- `started_at` (datetime at which this step began)
- `complete_at` (datetime at which this step finished, or None if it is ongoing)
- `state_string` (short string describing the step's state)
- `results` (results of this step; see *Build Result Codes*)
- `urls` (list of URLs produced by this step. Each urls is stored as a dictionary with keys *name* and *url*)
- `hidden` (true if the step should be hidden in status displays)

getStep(*stepid=None*, *buildid=None*, *number=None*, *name=None*)

Parameters

- **stepid** (*integer*) – the step id to retrieve
- **buildid** (*integer*) – the build from which to get the step
- **number** (*integer*) – the step number
- **name** (50-character *identifier*) – the step name

Returns stepdict via Deferred

Get a single step. The step can be specified by

- `stepid` alone;
- `buildid` and `number`, the step number within that build; or
- `buildid` and `name`, the unique step name within that build.

getSteps(*buildid*)

Parameters **buildid** (*integer*) – the build from which to get the step

Returns list of stepdicts, sorted by number, via Deferred

Get all steps in the given build, in order by number.

addStep(*self*, *buildid*, *name*, *state_string*)

Parameters

- **buildid** (*integer*) – the build to which to add the step
- **name** (50-character *identifier*) – the step name
- **state_string** (*unicode*) – the initial state of the step

Returns tuple of step ID, step number, and step name, via Deferred

Add a new step to a build. The given name will be used if it is unique; otherwise, a unique numerical suffix will be appended.

setStepStateString(stepid, state_string):

Parameters

- **stepid** (*integer*) – step ID
- **state_string** (*unicode*) – updated state of the step

Returns Deferred

Update the state string for the given step.

finishStep(*stepid*, *results*, *hidden*)

Parameters

- **stepid** (*integer*) – step ID
- **results** (*integer*) – step result
- **hidden** (*bool*) – true if the step should be hidden

Returns Deferred

Mark the given step as finished, with `complete_at` set to the current time.

Note: This update is done unconditionally, even if the steps are already finished.

---

addURL(*self*, *stepid*, *name*, *url*)

Parameters

- **stepid** (*integer*) – the stepid to add the url.
- **name** (*string*) – the url name
- **url** (*string*) – the actual url

Returns None via deferred

Add a new url to a step. The new url is added to the list of urls.

logs

class buildbot.db.logs.**LogsConnectorComponent**

This class handles log data. Build steps can have zero or more logs. Logs are uniquely identified by name within a step.

Information about a log, apart from its contents, is represented as a dictionary with the following keys, referred to as a *logdict*:

- `id` (log ID, globally unique)
- `stepid` (step ID, indicating the containing step)
- `name` free-form name of this log
- `slug` (50-identifier for the log, unique within the step)
- `complete` (true if the log is complete and will not receive more lines)
- `num_lines` (number of lines in the log)
- `type` (log type; see below)

Each log has a type that describes how to interpret its contents. See the *logchunk* resource type for details.

A log is contains a sequence of newline-separated lines of unicode. Log line numbering is zero-based.

Each line must be less than 64k when encoded in UTF-8. Longer lines will be truncated, and a warning logged.

Lines are stored internally in "chunks", and optionally compressed, but the implementation hides these details from callers.

getLog(*logid*)

Parameters **logid** (*integer*) – ID of the requested log

Returns logdict via Deferred

Get a log, identified by logid.

getLogBySlug(*stepid*, *slug*)

Parameters

- **stepid** (*integer*) – ID of the step containing this log
- **slug** – slug of the logfile to retrieve

Returns logdict via Deferred

Get a log, identified by name within the given step.

getLogs(*stepid*)

Parameters **stepid** (*integer*) – ID of the step containing the desired logs

Returns list of logdicts via Deferred

Get all logs within the given step.

getLogLines(*logid*, *first_line*, *last_line*)

Parameters

- **logid** (*integer*) – ID of the log
- **first_line** – first line to return
- **last_line** – last line to return

Returns see below

Get a subset of lines for a logfile.

The return value, via Deferred, is a concatenation of newline-terminated strings. If the requested last line is beyond the end of the logfile, only existing lines will be included. If the log does not exist, or has no associated lines, this method returns an empty string.

addLog(*stepid*, *name*, *type*)

Parameters

- **stepid** (*integer*) – ID of the step containing this log
- **name** (*string*) – name of the logfile
- **slug** (*50-character identifier*) – slug (unique identifier) of the logfile
- **type** (*string*) – log type (see above)

Raises **KeyError** – if a log with the given slug already exists in the step

Returns ID of the new log, via Deferred

Add a new log file to the given step.

appendLog(*logid*, *content*)

Parameters

- **logid** (*integer*) – ID of the requested log
- **content** (*string*) – new content to be appended to the log

Returns tuple of first and last line numbers in the new chunk, via Deferred

Append content to an existing log. The content must end with a newline. If the given log does not exist, the method will silently do nothing.

It is not safe to call this method more than once simultaneously for the same `logid`.

finishLog(*logid*)

Parameters **logid** (*integer*) – ID of the log to mark complete

Returns Deferred

Mark a log as complete.

Note that no checking for completeness is performed when appending to a log. It is up to the caller to avoid further calls to `appendLog` after `finishLog`.

compressLog(*logid*)

Parameters **logid** (*integer*) – ID of the log to compress

Returns Deferred

Compress the given log. This method performs internal optimizations of a log's chunks to reduce the space used and make read operations more efficient. It should only be called for finished logs. This method may take some time to complete.

buildsets

class buildbot.db.buildsets.**BuildsetsConnectorComponent**

This class handles getting buildsets into and out of the database. Buildsets combine multiple build requests that were triggered together.

An instance of this class is available at `master.db.buildsets`.

Buildsets are indexed by *bsid* and their contents represented as *bsdicts* (buildset dictionaries), with keys

- `bsid`
- `external_idstring` (arbitrary string for mapping builds externally)
- `reason` (string; reason these builds were triggered)
- `sourcestamps` (list of sourcestamps for this buildset, by ID)
- `submitted_at` (datetime object; time this buildset was created)
- `complete` (boolean; true if all of the builds for this buildset are complete)
- `complete_at` (datetime object; time this buildset was completed)
- `results` (aggregate result of this buildset; see *Build Result Codes*)

addBuildset(*sourcestamps*, *reason*, *properties*, *builderids*, *external_idstring=None*, *parent_buildid=None*, *parent_relationship=None*)

Parameters

- **sourcestamps** (*list*) – sourcestamps for the new buildset; see below
- **reason** (*short unicode string*) – reason for this buildset
- **properties** (*dictionary, where values are tuples of (value, source)*) – properties for this buildset
- **builderids** (*list of int*) – builderids specified by this buildset
- **external_idstring** (*unicode string*) – external key to identify this buildset; defaults to None
- **submitted_at** (*datetime*) – time this buildset was created; defaults to the current time
- **parent_buildid** (*int*) – optional build id that is the parent for this buildset
- **parent_relationship** (*unicode*) – relationship identifier for the parent, this is is configured relationship between the parent build, and the childs buildsets

Returns buildset ID and buildrequest IDs, via a Deferred

Add a new Buildset to the database, along with BuildRequests for each builder, returning the resulting bsid via a Deferred. Arguments should be specified by keyword.

Each sourcestamp in the list of sourcestamps can be given either as an integer, assumed to be a sourcestamp ID, or a dictionary of keyword arguments to be passed to `findSourceStampId`.

The return value is a tuple `(bsid, brids)` where `bsid` is the inserted buildset ID and `brids` is a dictionary mapping builderids to build request IDs.

completeBuildset(*bsid*, *results*[, *complete_at=XX*])

Parameters

- **bsid** (*integer*) – buildset ID to complete

- **results** (*integer*) – integer result code
- **complete_at** (*datetime*) – time the buildset was completed

Returns Deferred

Raises `KeyError` if the buildset does not exist or is already complete

Complete a buildset, marking it with the given `results` and setting its `completed_at` to the current time, if the `complete_at` argument is omitted.

getBuildset(*bsid*)

Parameters **bsid** – buildset ID

Returns bsdict, or `None`, via Deferred

Get a bsdict representing the given buildset, or `None` if no such buildset exists.

Note that buildsets are not cached, as the values in the database are not fixed.

getBuildsets(*complete=None*)

Parameters **complete** – if true, return only complete buildsets; if false, return only incomplete buildsets; if `None` or omitted, return all buildsets

Returns list of bsdicts, via Deferred

Get a list of bsdicts matching the given criteria.

getRecentBuildsets(count=None, branch=None, repository=None, complete=None):

Parameters

- **count** (*integer*) – maximum number of buildsets to retrieve (required).
- **branch** (*string*) – optional branch name. If specified, only buildsets affecting such branch will be returned.
- **repository** (*string*) – optional repository name. If specified, only buildsets affecting such repository will be returned.
- **complete** (`Boolean`) – if true, return only complete buildsets; if false, return only incomplete buildsets; if `None` or omitted, return all buildsets

Returns list of bsdicts, via Deferred

Get "recent" buildsets, as defined by their `submitted_at` times.

getBuildsetProperties(*buildsetid*)

Parameters **bsid** – buildset ID

Returns dictionary mapping property name to `value, source`, via Deferred

Return the properties for a buildset, in the same format they were given to *addBuildset*.

Note that this method does not distinguish a nonexistent buildset from a buildset with no properties, and returns `{}` in either case.

workers

class `buildbot.db.workers.`**WorkersConnectorComponent**

This class handles Buildbot's notion of workers. The worker information is returned as a dictionary:

- `id`
- `name` - the name of the worker
- `workerinfo` - worker information as dictionary
- `connected_to` - a list of masters, by ID, to which this worker is currently connected. This list will typically contain only one master, but in unusual circumstances the same worker may appear to be connected to multiple masters simultaneously.
- `configured_on` - a list of master-builder pairs, on which this worker is configured. Each pair is represented by a dictionary with keys `buliderid` and `masterid`.

The worker information can be any JSON-able object. See *worker* for more detail.

findWorkerId(*name=name*)

Parameters **name** (*50-character identifier*) – worker name

Returns worker ID via Deferred

Get the ID for a worker, adding a new worker to the database if necessary. The worker information for a new worker is initialized to an empty dictionary.

getWorkers(*masterid=None*, *builderid=None*)

Parameters

- **masterid** (*integer*) – limit to workers configured on this master
- **builderid** (*integer*) – limit to workers configured on this builder

Returns list of worker dictionaries, via Deferred

Get a list of workers. If either or both of the filtering parameters either specified, then the result is limited to workers configured to run on that master or builder. The `configured_on` results are limited by the filtering parameters as well. The `connected_to` results are limited by the `masterid` parameter.

getWorker(*workerid=None*, *name=None*, *masterid=None*, *builderid=None*)

Parameters

- **name** (*string*) – the name of the worker to retrieve
- **workerid** (*integer*) – the ID of the worker to retrieve
- **masterid** (*integer*) – limit to workers configured on this master
- **builderid** (*integer*) – limit to workers configured on this builder

Returns info dictionary or None, via Deferred

Looks up the worker with the given name or ID, returning `None` if no matching worker is found. The `masterid` and `builderid` arguments function as they do for *getWorkers*.

workerConnected(*workerid*, *masterid*, *workerinfo*)

Parameters

- **workerid** (*integer*) – the ID of the worker
- **masterid** (*integer*) – the ID of the master to which it connected
- **workerinfo** (*dict*) – the new worker information dictionary

Returns Deferred

Record the given worker as attached to the given master, and update its cached worker information. The supplied information completely replaces any existing information.

workerDisconnected(*workerid*, *masterid*)

Parameters

- **workerid** (*integer*) – the ID of the worker
- **masterid** (*integer*) – the ID of the master to which it connected

Returns Deferred

Record the given worker as no longer attached to the given master.

workerConfigured(*workerid*, *masterid*, *builderids*)

Parameters

- **workerid** (*integer*) – the ID of the worker
- **masterid** (*integer*) – the ID of the master to which it configured
- **of integer builderids** (*list*) – the ID of the builders to which it is configured

Returns Deferred

Record the given worker as being configured on the given master and for given builders. This method will also remove any other builder that were configured previously for same (worker, master) combination.

deconfigureAllWorkersForMaster(*masterid*)

Parameters **masterid** (*integer*) – the ID of the master to which it configured

Returns Deferred

Unregister all the workers configured to a master for given builders. This shall happen when master disabled or before reconfiguration

changes

class buildbot.db.changes.**ChangesConnectorComponent**

This class handles changes in the buildbot database, including pulling information from the changes sub-tables.

An instance of this class is available at `master.db.changes`.

Changes are indexed by *changeid*, and are represented by a *chdict*, which has the following keys:

- `changeid` (the ID of this change)
- `parent_changeids` (list of ID; change's parents)
- `author` (unicode; the author of the change)
- `files` (list of unicode; source-code filenames changed)
- `comments` (unicode; user comments)
- `is_dir` (deprecated)
- `links` (list of unicode; links for this change, e.g., to web views, review)
- `revision` (unicode string; revision for this change, or `None` if unknown)
- `when_timestamp` (datetime instance; time of the change)
- `branch` (unicode string; branch on which the change took place, or `None` for the "default branch", whatever that might mean)
- `category` (unicode string; user-defined category of this change, or `None`)

- `revlink` (unicode string; link to a web view of this change)
- `properties` (user-specified properties for this change, represented as a dictionary mapping keys to (value, source))
- `repository` (unicode string; repository where this change occurred)
- `project` (unicode string; user-defined project to which this change corresponds)

getParentChangeIds(*branch*, *repository*, *project*, *codebase*)

Parameters

- **branch** (*unicode string*) – the branch of the change
- **repository** (*unicode string*) – the repository in which this change took place
- **project** (*unicode string*) – the project this change is a part of
- **codebase** (*unicode string*) –

return the last changeID which matches the repository/project/codebase

addChange(*author=None*, *files=None*, *comments=None*, *is_dir=0*, *links=None*, *revision=None*, *when_timestamp=None*, *branch=None*, *category=None*, *revlink=''*, *properties={}*, *repository=''*, *project=''*, *uid=None*)

Parameters

- **author** (*unicode string*) – the author of this change
- **files** – a list of filenames that were changed
- **comments** – user comments on the change
- **is_dir** – deprecated
- **links** (*list of unicode strings*) – a list of links related to this change, e.g., to web viewers or review pages
- **revision** (*unicode string*) – the revision identifier for this change
- **when_timestamp** (*datetime instance or None*) – when this change occurred, or the current time if None
- **branch** (*unicode string*) – the branch on which this change took place
- **category** (*unicode string*) – category for this change (arbitrary use by Buildbot users)
- **revlink** (*unicode string*) – link to a web view of this revision
- **properties** (*dictionary*) – properties to set on this change, where values are tuples of (value, source). At the moment, the source must be `'Change'`, although this may be relaxed in later versions.
- **repository** (*unicode string*) – the repository in which this change took place
- **project** (*unicode string*) – the project this change is a part of
- **uid** (*integer*) – uid generated for the change author

Returns new change's ID via Deferred

Add a Change with the given attributes to the database, returning the changeid via a Deferred. All arguments should be given as keyword arguments.

The `project` and `repository` arguments must be strings; `None` is not allowed.

getChange(*changeid*, *no_cache=False*)

Parameters

- **changeid** – the id of the change instance to fetch
- **no_cache** (*boolean*) – bypass cache and always fetch from database

Returns chdict via Deferred

Get a change dictionary for the given changeid, or `None` if no such change exists.

getChangeUids(*changeid*)

Parameters **changeid** – the id of the change instance to fetch

Returns list of uids via Deferred

Get the userids associated with the given changeid.

getRecentChanges(*count*)

Parameters **count** – maximum number of instances to return

Returns list of dictionaries via Deferred, ordered by changeid

Get a list of the `count` most recent changes, represented as dictionaries; returns fewer if that many do not exist.

Note: For this function, "recent" is determined by the order of the changeids, not by `when_timestamp`. This is most apparent in DVCS's, where the timestamp of a change may be significantly earlier than the time at which it is merged into a repository monitored by Buildbot.

---

getChanges()

Returns list of dictionaries via Deferred

Get a list of the changes, represented as dictionaries; changes are sorted, and paged using generic data query options

getChangesCount()

Returns list of dictionaries via Deferred

Get the number changes, that the query option would return if no paging option where set

getLatestChangeid()

Returns changeid via Deferred

Get the most-recently-assigned changeid, or `None` if there are no changes at all.

getChangesForBuild(*buildid*)

Parameters **buildid** – ID of the build

Returns list of dictionaries via Deferred

Get the "blame" list of changes for a build.

getChangeFromSSid(*sourcestampid*)

Parameters **sourcestampid** – ID of the sourcestampid

Returns chdict via Deferred

returns the change dictionary related to the sourcestamp ID.

changesources

exception buildbot.db.changesources.**ChangeSourceAlreadyClaimedError**
Raised when a changesource request is already claimed by another master.

class buildbot.db.changesources.**ChangeSourcesConnectorComponent**
This class manages the state of the Buildbot changesources.

An instance of this class is available at `master.db.changesources`.

Changesources are identified by their changesourceid, which can be objtained from *findChangeSourceId*.

Changesources are represented by dictionaries with the following keys:

- `id` - changesource's ID
- `name` - changesource's name
- `masterid` - ID of the master currently running this changesource, or None if it is inactive

Note that this class is conservative in determining what changesources are inactive: a changesource linked to an inactive master is still considered active. This situation should never occur, however; links to a master should be deleted when it is marked inactive.

findChangeSourceId(*name*)

Parameters **name** – changesource name

Returns changesource ID via Deferred

Return the changesource ID for the changesource with this name. If such a changesource is already in the database, this returns the ID. If not, the changesource is added to the database and its ID returned.

setChangeSourceMaster(*changesourceid*, *masterid*)

Parameters

- **changesourceid** – changesource to set the master for
- **masterid** – new master for this changesource, or None

Returns Deferred

Set, or unset if `masterid` is None, the active master for this changesource. If no master is currently set, or the current master is not active, this method will complete without error. If the current master is active, this method will raise `ChangeSourceAlreadyClaimedError`.

getChangeSource(*changesourceid*)

Parameters **changesourceid** – changesource ID

Returns changesource dictionary or None, via Deferred

Get the changesource dictionary for the given changesource.

getChangeSources(*active=None*, *masterid=None*)

Parameters

- **active** (*boolean*) – if specified, filter for active or inactive changesources
- **masterid** (*integer*) – if specified, only return changesources attached associated with this master

Returns list of changesource dictionaries in unspecified order

Get a list of changesources.

If `active` is given, changesources are filtered according to whether they are active (true) or inactive (false). An active changesource is one that is claimed by an active master.

If `masterid` is given, the list is restricted to schedulers associated with that master.

schedulers

exception `buildbot.db.schedulers.`**`SchedulerAlreadyClaimedError`**
Raised when a scheduler request is already claimed by another master.

class `buildbot.db.schedulers.`**`SchedulersConnectorComponent`**
This class manages the state of the Buildbot schedulers. This state includes classifications of as-yet un-built changes.

An instance of this class is available at `master.db.schedulers`.

Schedulers are identified by their schedulerid, which can be objtained from *`findSchedulerId`*.

Schedulers are represented by dictionaries with the following keys:

- `id` - scheduler's ID
- `name` - scheduler's name
- `masterid` - ID of the master currently running this scheduler, or None if it is inactive

Note that this class is conservative in determining what schedulers are inactive: a scheduler linked to an inactive master is still considered active. This situation should never occur, however; links to a master should be deleted when it is marked inactive.

`classifyChanges`(*objectid*, *classifications*)

Parameters

- **`schedulerid`** – ID of the scheduler classifying the changes
- **`classifications`** (*`dictionary`*) – mapping of changeid to boolean, where the boolean is true if the change is important, and false if it is unimportant

Returns Deferred

Record the given classifications. This method allows a scheduler to record which changes were important and which were not immediately, even if the build based on those changes will not occur for some time (e.g., a tree stable timer). Schedulers should be careful to flush classifications once they are no longer needed, using *`flushChangeClassifications`*.

`flushChangeClassifications`(*objectid*, *less_than=None*)

Parameters

- **`schedulerid`** – ID of the scheduler owning the flushed changes
- **`less_than`** – (optional) lowest changeid that should *not* be flushed

Returns Deferred

Flush all scheduler_changes for the given scheduler, limiting to those with changeid less than `less_than` if the parameter is supplied.

`getChangeClassifications`(*objectid*[, *branch*])

Parameters

- **schedulerid** (*integer*) – ID of scheduler to look up changes for
- **branch** (*string or None (for default branch)*) – (optional) limit to changes with this branch

Returns dictionary via Deferred

Return the classifications made by this scheduler, in the form of a dictionary mapping changeid to a boolean, just as supplied to *classifyChanges*.

If `branch` is specified, then only changes on that branch will be given. Note that specifying `branch=None` requests changes for the default branch, and is not the same as omitting the `branch` argument altogether.

findSchedulerId(*name*)

Parameters **name** – scheduler name

Returns scheduler ID via Deferred

Return the scheduler ID for the scheduler with this name. If such a scheduler is already in the database, this returns the ID. If not, the scheduler is added to the database and its ID returned.

setSchedulerMaster(*schedulerid*, *masterid*)

Parameters

- **schedulerid** – scheduler to set the master for
- **masterid** – new master for this scheduler, or None

Returns Deferred

Set, or unset if `masterid` is None, the active master for this scheduler. If no master is currently set, or the current master is not active, this method will complete without error. If the current master is active, this method will raise `SchedulerAlreadyClaimedError`.

getScheduler(*schedulerid*)

Parameters **schedulerid** – scheduler ID

Returns scheduler dictionary or None via Deferred

Get the scheduler dictionary for the given scheduler.

getSchedulers(*active=None*, *masterid=None*)

Parameters

- **active** (*boolean*) – if specified, filter for active or inactive schedulers
- **masterid** (*integer*) – if specified, only return schedulers attached associated with this master

Returns list of scheduler dictionaries in unspecified order

Get a list of schedulers.

If `active` is given, schedulers are filtered according to whether they are active (true) or inactive (false). An active scheduler is one that is claimed by an active master.

If `masterid` is given, the list is restricted to schedulers associated with that master.

sourcestamps

class buildbot.db.sourcestamps.**SourceStampsConnectorComponent**

This class manages source stamps, as stored in the database. A source stamp uniquely identifies a particular version a single codebase. Source stamps are identified by their ID. It is safe to use sourcestamp ID equality as a proxy for source stamp equality. For example, all builds of a particular version of a codebase will share the same sourcestamp ID. This equality does not extend to patches: two sourcestamps generated with exactly the same patch will have different IDs.

Relative source stamps have a `revision` of None, meaning "whatever the latest is when this sourcestamp is interpreted". While such source stamps may correspond to a wide array of revisions over the lifetime of a buildbot install, they will only ever have one ID.

An instance of this class is available at `master.db.sourcestamps`.

- `ssid`
- `branch` (branch, or `None` for default branch)
- `revision` (revision, or `None` to indicate the latest revision, in which case this is a relative source stamp)
- `patchid` (ID of the patch)
- `patch_body` (body of the patch, or `None`)
- `patch_level` (directory stripping level of the patch, or `None`)
- `patch_subdir` (subdirectory in which to apply the patch, or `None`)
- `patch_author` (author of the patch, or `None`)
- `patch_comment` (comment for the patch, or `None`)
- `repository` (repository containing the source; never `None`)
- `project` (project this source is for; never `None`)
- `codebase` (codebase this stamp is in; never `None`)
- `created_at` (timestamp when this stamp was first created)

Note that the patch body is a bytestring, not a unicode string.

findSourceStampId(branch=None, revision=Node, repository=None, project=None, patch_body=None, patch_level=None, patch_author=None, patch_comment=None, patch_subdir=None):

Parameters

- **branch** (*unicode string or None*) –
- **revision** (*unicode string or None*) –
- **repository** (*unicode string or None*) –
- **project** (*unicode string or None*) –
- **codebase** (*unicode string (required)*) –
- **patch_body** (*unicode string or None*) – patch body
- **patch_level** (*integer or None*) – patch level
- **patch_author** (*unicode string or None*) – patch author

- **patch_comment** (*unicode string or None*) – patch comment
- **patch_subdir** (*unicode string or None*) – patch subdir

Returns ssid, via Deferred

Create a new SourceStamp instance with the given attributes, or find an existing one. In either case, return its ssid. The arguments all have the same meaning as in an ssdict.

If a new SourceStamp is created, its `created_at` is set to the current time.

getSourceStamp(*ssid*)

Parameters

- **ssid** – sourcestamp to get
- **no_cache** (*boolean*) – bypass cache and always fetch from database

Returns ssdict, or `None`, via Deferred

Get an ssdict representing the given source stamp, or `None` if no such source stamp exists.

getSourceStamps()

Returns list of ssdict, via Deferred

Get all sourcestamps in the database. You probably don't want to do this! This method will be extended to allow appropriate filtering.

getSourceStampsForBuild(*buildid*)

Parameters **buildid** – build ID

Returns list of ssdict, via Deferred

Get sourcestamps related to a build.

state

class buildbot.db.state.**StateConnectorComponent**

This class handles maintaining arbitrary key/value state for Buildbot objects. Each object can store arbitrary key/value pairs, where the values are any JSON-encodable value. Each pair can be set and retrieved atomically.

Objects are identified by their (user-visible) name and their class. This allows, for example, a `nightly_smoketest` object of class `NightlyScheduler` to maintain its state even if it moves between masters, but avoids cross-contaminating state between different classes of objects with the same name.

Note that "class" is not interpreted literally, and can be any string that will uniquely identify the class for the object; if classes are renamed, they can continue to use the old names.

An instance of this class is available at `master.db.state`.

Objects are identified by *objectid*.

getObjectId(*name*, *class_name*)

Parameters

- **name** – name of the object
- **class_name** – object class name

Returns the objectid, via a Deferred.

Get the object ID for this combination of a name and a class. This will add a row to the 'objects' table if none exists already.

getState(*objectid*, *name*[, *default*])

Parameters

- **objectid** – objectid on which the state should be checked
- **name** – name of the value to retrieve
- **default** – (optional) value to return if `name` is not present

Returns state value via a Deferred

Raises **KeyError** – if `name` is not present and no default is given

Raises TypeError if JSON parsing fails

Get the state value for key `name` for the object with id `objectid`.

setState(*objectid*, *name*, *value*)

Parameters

- **objectid** – the objectid for which the state should be changed
- **name** – the name of the value to change
- **value** (*JSON-able value*) – the value to set
- **returns** – value actually written via Deferred

Raises TypeError if JSONification fails

Set the state value for `name` for the object with id `objectid`, overwriting any existing value. In case of two racing writes, the first (as per db rule) one wins, the seconds returns the value from the first.

atomicCreateState(*objectid*, *name*, *thd_create_callback*)

Parameters

- **objectid** – the objectid for which the state should be created
- **name** – the name of the value to create
- **thd_create_callback** – the function to call from thread to create the value if non-existent. (returns JSON-able value)
- **returns** – Deferred

Raises TypeError if JSONification fails

Atomically creates the state value for `name` for the object with id `objectid`, If there is an existing value, returns that instead. This implementation ensures the state is created only once for the whole cluster.

Those 3 methods have their threaded equivalent, `thdGetObjectId`, `thdGetState`, `thdSetState` that are intended to run in synchronous code, (e.g master.cfg environment)

users

class `buildbot.db.users.`**UsersConnectorComponent**

This class handles Buildbot's notion of users. Buildbot tracks the usual information about users – username and password, plus a display name.

The more complicated task is to recognize each user across multiple interfaces with Buildbot. For example, a user may be identified as 'djmitche' in Subversion, 'dustin@v.igoro.us' in Git, and 'dustin' on IRC. To support this functionality, each user as a set of attributes, keyed by type. The `findUserByAttr` method uses these attributes to match users, adding a new user if no matching user is found.

Users are identified canonically by *uid*, and are represented by *usdicts* (user dictionaries) with keys

- `uid`
- `identifier` (display name for the user)
- `bb_username` (buildbot login username)
- `bb_password` (hashed login password)

All attributes are also included in the dictionary, keyed by type. Types colliding with the keys above are ignored.

findUserByAttr(*identifier*, *attr_type*, *attr_data*)

Parameters

- **identifier** – identifier to use for a new user
- **attr_type** – attribute type to search for and/or add
- **attr_data** – attribute data to add

Returns userid via Deferred

Get an existing user, or add a new one, based on the given attribute.

This method is intended for use by other components of Buildbot to search for a user with the given attributes.

Note that `identifier` is *not* used in the search for an existing user. It is only used when creating a new user. The identifier should be based deterministically on the attributes supplied, in some fashion that will seem natural to users.

For future compatibility, always use keyword parameters to call this method.

getUser(*uid*)

Parameters

- **uid** – user id to look up
- **no_cache** (*boolean*) – bypass cache and always fetch from database

Returns usdict via Deferred

Get a usdict for the given user, or `None` if no matching user is found.

getUserByUsername(*username*)

Parameters **username** (*string*) – username portion of user credentials

Returns usdict or None via deferred

Looks up the user with the bb_username, returning the usdict or `None` if no matching user is found.

getUsers()

Returns list of partial usdicts via Deferred

Get the entire list of users. User attributes are not included, so the results are not full userdicts.

updateUser(*uid=None*, *identifier=None*, *bb_username=None*, *bb_password=None*, *attr_type=None*, *attr_data=None*)

Parameters

- **uid** (*int*) – the user to change
- **identifier** (*string*) – (optional) new identifier for this user
- **bb_username** (*string*) – (optional) new buildbot username
- **bb_password** (*string*) – (optional) new hashed buildbot password
- **attr_type** (*string*) – (optional) attribute type to update
- **attr_data** (*string*) – (optional) value for `attr_type`

Returns Deferred

Update information about the given user. Only the specified attributes are updated. If no user with the given uid exists, the method will return silently.

Note that `bb_password` must be given if `bb_username` appears; similarly, `attr_type` requires `attr_data`.

removeUser(*uid*)

Parameters **uid** (*int*) – the user to remove

Returns Deferred

Remove the user with the given uid from the database. This will remove the user from any associated tables as well.

identifierToUid(*identifier*)

Parameters **identifier** (*string*) – identifier to search for

Returns uid or `None`, via Deferred

Fetch a uid for the given identifier, if one exists.

masters

class `buildbot.db.masters.`**MastersConnectorComponent**

This class handles tracking the buildmasters in a multi-master configuration. Masters "check in" periodically. Other masters monitor the last activity times, and mark masters that have not recently checked in as inactive.

Masters are represented by master dictionaries with the following keys:

- `id` – the ID of this master
- `name` – the name of the master (generally of the form `hostname:basedir`)
- `active` – true if this master is running
- `last_active` – time that this master last checked in (a datetime object)

findMasterId(*name*)

Parameters **name** (*unicode*) – name of this master

Returns master id via Deferred

Return the master ID for the master with this master name (generally `hostname:basedir`). If such a master is already in the database, this returns the ID. If not, the master is added to the database, with `active=False`, and its ID returned.

setMasterState(*masterid*, *active*)

Parameters

- **masterid** (*integer*) – the master to check in
- **active** (*boolean*) – whether to mark this master as active or inactive

Returns boolean via Deferred

Mark the given master as active or inactive, returning true if the state actually changed. If `active` is true, the `last_active` time is updated to the current time. If `active` is false, then any links to this master, such as schedulers, will be deleted.

getMaster(*masterid*)

Parameters **masterid** (*integer*) – the master to check in

Returns Master dict or None via Deferred

Get the indicated master.

getMasters()

Returns list of Master dicts via Deferred

Get a list of the masters, represented as dictionaries; masters are sorted and paged using generic data query options

setAllMastersActiveLongTimeAgo()

Returns None via Deferred

This method is intended to be call by upgrade-master, and will effectively force housekeeping on all masters at next startup. This method is not intended to be called outside of housekeeping scripts.

builders

class buildbot.db.builders.**BuildersConnectorComponent**

This class handles the relationship between builder names and their IDs, as well as tracking which masters are configured for this builder.

Builders are represented by master dictionaries with the following keys:

- `id` – the ID of this builder
- `name` – the builder name, a 20-character *identifier*
- `masterids` – the IDs of the masters where this builder is configured (sorted by id)

findBuilderId(*name*)

Parameters **name** (20-character *identifier*) – name of this builder

Returns builder id via Deferred

Return the builder ID for the builder with this builder name. If such a builder is already in the database, this returns the ID. If not, the builder is added to the database.

addBuilderMaster(*builderid=None*, *masterid=None*)

Parameters

- **builderid** (*integer*) – the builder
- **masterid** (*integer*) – the master

Returns Deferred

Add the given master to the list of masters on which the builder is configured. This will do nothing if the master and builder are already associated.

removeBuilderMaster(*builderid=None*, *masterid=None*)

Parameters

- **builderid** (*integer*) – the builder
- **masterid** (*integer*) – the master

Returns Deferred

Remove the given master from the list of masters on which the builder is configured.

getBuilder(*builderid*)

Parameters **builderid** (*integer*) – the builder to check in

Returns Builder dict or None via Deferred

Get the indicated builder.

getBuilders(*masterid=None*)

Parameters **masterid** (*integer*) – ID of the master to which the results should be limited

Returns list of Builder dicts via Deferred

Get all builders (in unspecified order). If `masterid` is given, then only builders configured on that master are returned.

Writing Database Connector Methods

The information above is intended for developers working on the rest of Buildbot, and treating the database layer as an abstraction. The remainder of this section describes the internals of the database implementation, and is intended for developers modifying the schema or adding new methods to the database layer.

Warning: It's difficult to change the database schema significantly after it has been released, and very disruptive to users to change the database API. Consider very carefully the future-proofing of any changes here!

The DB Connector and Components

class buildbot.db.connector.**DBConnector**

The root of the database connectors, `master.db`, is a *DBConnector* instance. Its main purpose is to hold reference to each of the connector components, but it also handles timed cleanup tasks.

If you are adding a new connector component, import its module and create an instance of it in this class's constructor.

class buildbot.db.base.**DBConnectorComponent**

This is the base class for connector components.

There should be no need to override the constructor defined by this base class.

db

A reference to the *DBConnector*, so that connector components can use e.g., `self.db.pool` or `self.db.model`. In the unusual case that a connector component needs access to the master, the easiest path is `self.db.master`.

checkLength(*col*, *value*)
For use by subclasses to check that 'value' will fit in 'col', where 'col' is a table column from the model. Ignore this check for database engines that either provide this error themselves (postgres) or that do not enforce maximum-length restrictions (sqlite)

findSomethingId(*self*, *tbl*, *whereclause*, *insert_values*, *_race_hook=None*)
Find (using C{whereclause}) or add (using C{insert_values) a row to C{table}, and return the resulting ID.

hashColumns(**args*)
Hash the given values in a consistent manner: None is represented as xf5, an invalid unicode byte; strings are converted to utf8; and integers are represented by their decimal expansion. The values are then joined by '0' and hashed with sha1.

doBatch(*batch*, *batch_n=500*)
returns an Iterator that batches stuff in order to not push to many thing in a single request. Especially sqlite has 999 limit on argument it can take in a requests.

Direct Database Access

The connectors all use SQLAlchemy Core (http://www.sqlalchemy.org/docs/index.html) as a wrapper around database client drivers. Unfortunately, SQLAlchemy is a synchronous library, so some extra work is required to use it in an asynchronous context like Buildbot. This is accomplished by deferring all database operations to threads, and returning a Deferred. The `Pool` class takes care of the details.

A connector method should look like this:

```
def myMethod(self, arg1, arg2):
    def thd(conn):
        q = ... # construct a query
        for row in conn.execute(q):
            ... # do something with the results
        return ... # return an interesting value
    return self.db.pool.do(thd)
```

Picking that apart, the body of the method defines a function named `thd` taking one argument, a `Connection` object. It then calls `self.db.pool.do`, passing the `thd` function. This function is called in a thread, and can make blocking calls to SQLAlchemy as desired. The `do` method will return a Deferred that will fire with the return value of `thd`, or with a failure representing any exceptions raised by `thd`.

The return value of `thd` must not be an SQLAlchemy object - in particular, any `ResultProxy` objects must be parsed into lists or other data structures before they are returned.

Warning: As the name `thd` indicates, the function runs in a thread. It should not interact with any other part of Buildbot, nor with any of the Twisted components that expect to be accessed from the main thread – the reactor, Deferreds, etc.

Queries can be constructed using any of the SQLAlchemy core methods, using tables from *Model*, and executed with the connection object, `conn`.

Note: SQLAlchemy requires the use of a syntax that is forbidden by pep8. If in where clauses you need to select rows where a value is NULL, you need to write (*tbl.c.value == None*). This form is forbidden by pep8 which requires the use of *is None* instead of *== None*. As sqlalchemy is using operator overloading to implement pythonic SQL statements, and *is* operator is not overloadable, we need to keep the == operators. In order to solve this issue, buildbot

uses *buildbot.db.NULL* constant, which is *None*. So instead of writting *tbl.c.value == None*, please write *tbl.c.value == NULL*)

---

class buildbot.db.pool.**DBThreadPool**

do(*callable*, ...)

Returns Deferred

Call `callable` in a thread, with a `Connection` object as first argument. Returns a deferred that will fire with the results of the callable, or with a failure representing any exception raised during its execution.

Any additional positional or keyword arguments are passed to `callable`.

do_with_engine(*callable*, ...)

Returns Deferred

Similar to *do*, call `callable` in a thread, but with an `Engine` object as first argument.

This method is only used for schema manipulation, and should not be used in a running master.

Database Schema

Database connector methods access the database through SQLAlchemy, which requires access to Python objects representing the database tables. That is handled through the model.

class buildbot.db.model.**Model**

This class contains the canonical description of the buildbot schema, It is presented in the form of SQLAlchemy `Table` instances, as class variables. At runtime, the model is available at `master.db.model`, so for example the `buildrequests` table can be referred to as `master.db.model.buildrequests`, and columns are available in its `c` attribute.

The source file, master/buildbot/db/model.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/db/model.py), contains comments describing each table; that information is not replicated in this documentation.

Note that the model is not used for new installations or upgrades of the Buildbot database. See *Modifying the Database Schema* for more information.

metadata

The model object also has a `metadata` attribute containing a `MetaData` instance. Connector methods should not need to access this object. The metadata is not bound to an engine.

The *Model* class also defines some migration-related methods:

is_current()

Returns boolean via Deferred

Returns true if the current database's version is current.

upgrade()

Returns Deferred

Upgrades the database to the most recent schema version.

Caching

Connector component methods that get an object based on an ID are good candidates for caching. The *cached* decorator makes this automatic:

buildbot.db.base.**cached**(*cachename*)

Parameters **cache_name** – name of the cache to use

A decorator for "getter" functions that fetch an object from the database based on a single key. The wrapped method will only be called if the named cache does not contain the key.

The wrapped function must take one argument (the key); the wrapper will take a key plus an optional `no_cache` argument which, if true, will cause it to invoke the underlying method even if the key is in the cache.

The resulting method will have a `cache` attribute which can be used to access the underlying cache.

In most cases, getter methods return a well-defined dictionary. Unfortunately, Python does not handle weak references to bare dictionaries, so components must instantiate a subclass of `dict`. The whole assembly looks something like this:

```
class ThDict(dict):
    pass

class ThingConnectorComponent(base.DBConnectorComponent):

    @base.cached('thdicts')
    def getThing(self, thid):
        def thd(conn):
            ...
            thdict = ThDict(thid=thid, attr=row.attr, ...)
            return thdict
        return self.db.pool.do(thd)
```

Tests

It goes without saying that any new connector methods must be fully tested!

You will also want to add an in-memory implementation of the methods to the fake classes in `master/buildbot/test/fake/fakedb.py`. Non-DB Buildbot code is tested using these fake implementations in order to isolate that code from the database code, and to speed-up tests.

The keys and types used in the return value from a connector's `get` methods are described in master/buildbot/test/util/validation.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/util/validation.py), via the `dbdict` module-level value. This is a dictionary of `DictValidator` objects, one for each return value.

These values are used within test methods like this:

```
rv = yield self.db.masters.getMaster(7)
validation.verifyDbDict(self, 'masterdict', rv)
```

Modifying the Database Schema

Changes to the schema are accomplished through migration scripts, supported by SQLAlchemy-Migrate (https://github.com/openstack/sqlalchemy-migrate). In fact, even new databases are created with the migration scripts – a new database is a migrated version of an empty database.

The schema is tracked by a version number, stored in the `migrate_version` table. This number is incremented for each change to the schema, and used to determine whether the database must be upgraded. The master will refuse to run with an out-of-date database.

To make a change to the schema, first consider how to handle any existing data. When adding new columns, this may not be necessary, but table refactorings can be complex and require caution so as not to lose information.

Create a new script in master/buildbot/db/migrate/versions (https://github.com/buildbot/buildbot/blob/master/master/buildbot/db/migrate following the numbering scheme already present. The script should have an `update` method, which takes an engine as a parameter, and upgrades the database, both changing the schema and performing any required data migrations. The engine passed to this parameter is "enhanced" by SQLAlchemy-Migrate, with methods to handle adding, altering, and dropping columns. See the SQLAlchemy-Migrate documentation for details.

Next, modify master/buildbot/db/model.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/db/model.py) to represent the updated schema. Buildbot's automated tests perform a rudimentary comparison of an upgraded database with the model, but it is important to check the details - key length, nullability, and so on can sometimes be missed by the checks. If the schema and the upgrade scripts get out of sync, bizarre behavior can result.

Also, adjust the fake database table definitions in master/buildbot/test/fake/fakedb.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/fake/fakedb.py) according to your changes.

Your upgrade script should have unit tests. The classes in master/buildbot/test/util/migration.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/util/migration.py) make this straightforward. Unit test scripts should be named e.g., `test_db_migrate_versions_015_remove_bad_master_objectid.py`.

The master/buildbot/test/integration/test_upgrade.py (https://github.com/buildbot/buildbot/blob/master/master/buildbot/test/integration/t also tests upgrades, and will confirm that the resulting database matches the model. If you encounter implicit indexes on MySQL, that do not appear on SQLite or Postgres, add them to `implied_indexes` in `master/buidlbot/db/model.py`.

Foreign key checking

PostgreSQL and SQlite db backends are checking the foreign keys consistency. bug #2248 (http://trac.buildbot.net/ticket/2248) needs to be fixed so that we can support foreign key checking for MySQL.

To maintain consistency with real db, fakedb can check the foreign key consistency of your test data. For this, just enable it with:

```
self.db = fakedb.FakeDBConnector(self.master, self)
self.db.checkForeignKeys = True
```

Note that tests that only use fakedb do not really need foreign key consistency, even if this is a good practice to enable it in new code.

Database Compatibility Notes

Or: "If you thought any database worked right, think again"

Because Buildbot works over a wide range of databases, it is generally limited to database features present in all supported backends. This section highlights a few things to watch out for.

In general, Buildbot should be functional on all supported database backends. If use of a backend adds minor usage restrictions, or cannot implement some kinds of error checking, that is acceptable if the restrictions are well-documented in the manual.

The metabuildbot tests Buildbot against all supported databases, so most compatibility errors will be caught before a release.

Index Length in MySQL

MySQL only supports about 330-character indexes. The actual index length is 1000 bytes, but MySQL uses 3-byte encoding for UTF8 strings. This is a longstanding bug in MySQL - see "Specified key was too long; max key length is 1000 bytes" with utf8 (http://bugs.mysql.com/bug.php?id=4541). While this makes sense for indexes used for record lookup, it limits the ability to use unique indexes to prevent duplicate rows.

InnoDB only supports indexes up to 255 unicode characters, which is why all indexed columns are limited to 255 characters in Buildbot.

Transactions in MySQL

Unfortunately, use of the MyISAM storage engine precludes real transactions in MySQL. `transaction.commit()` and `transaction.rollback()` are essentially no-ops: modifications to data in the database are visible to other users immediately, and are not reverted in a rollback.

Referential Integrity in SQLite and MySQL

Neither MySQL nor SQLite enforce referential integrity based on foreign keys. Postgres does enforce, however. If possible, test your changes on Postgres before committing, to check that tables are added and removed in the proper order.

Subqueries in MySQL

MySQL's query planner is easily confused by subqueries. For example, a DELETE query specifying id's that are IN a subquery will not work. The workaround is to run the subquery directly, and then execute a DELETE query for each returned id.

If this weakness has a significant performance impact, it would be acceptable to conditionalize use of the subquery on the database dialect.

Too Many Variables in SQLite

Sqlite has a limitation on the number of variables it can use. This limitation is usually SQLITE_LIMIT_VARIABLE_NUMBER=999 (http://www.sqlite.org/c3ref/c_limit_attached.html#sqlitelimitvariablenumber). There is currently no way with pysqlite to query the value of this limit. The C-api `sqlite_limit` is just not bound to the python.

When you hit this problem, you will get error like the following:

```
sqlalchemy.exc.OperationalError: (OperationalError) too many SQL variables
u'DELETE FROM scheduler_changes WHERE scheduler_changes.changeid IN (?, ?, ?, ......
↪tons of ?? and IDs .... 9363, 9362, 9361)
```

You can use the method `doBatch` in order to write batching code in a consistent manner.

Testing migrations with real databases

By default Buildbot test suite uses SQLite database for testings database migrations. To use other database set `BUILDBOT_TEST_DB_URL` environment variable to value in SQLAlchemy database URL specification (http://docs.sqlalchemy.org/en/latest/core/engines.html#database-urls).

For example, to run tests with file-based SQLite database you can start tests in the following way:

```
BUILDBOT_TEST_DB_URL=sqlite:////tmp/test_db.sqlite trial buildbot.test
```

Run databases in Docker

Docker (https://www.docker.com/) allows to easily install and configure different databases locally in containers.

To run tests with PostgreSQL:

```
# Install psycopg.
pip install psycopg2
# Start container with PostgreSQL 9.5.
# It will listen on port 15432 on localhost.
sudo docker run --name bb-test-postgres -e POSTGRES_PASSWORD=password \
    -p 127.0.0.1:15432:5432 -d postgres:9.5
# Start interesting tests
BUILDBOT_TEST_DB_URL=postgresql://postgres:password@localhost:15432/postgres \
    trial buildbot.test
```

To run tests with MySQL:

```
# Install mysqlclient
pip install mysqlclient
# Start container with MySQL 5.5.
# It will listen on port 13306 on localhost.
sudo docker run --name bb-test-mysql -e MYSQL_ROOT_PASSWORD=password \
    -p 127.0.0.1:13306:3306 -d mysql:5.5
# Start interesting tests
BUILDBOT_TEST_DB_URL=mysql+mysqldb://root:password@127.0.0.1:13306/mysql \
    trial buildbot.test
```

Messaging and Queues

As of version 0.9.0, Buildbot uses a message-queueing structure to handle asynchronous notifications in a distributed fashion. This avoids, for the most part, the need for each master to poll the database, allowing masters to react to events as they happen.

Overview

Buildbot is structured as a hybrid state- and event-based application, which will probably offend adherents of either pattern. In particular, the most current state is stored in the *Database*, while any changes to the state are announced in the form of a message. The content of the messages is sufficient to reconstruct the updated state, allowing external processes to represent "live" state without polling the database.

This split nature immediately brings to light the problem of synchronizing the two interfaces. Queueing systems can introduce queueing delays as messages propagate. Likewise, database systems may introduce a delay between committed modifications and the modified data appearing in queries; for example, with MySQL master/slave replication, there can be several seconds' delay before a slave is updated.

Buildbot's MQ connector simply relays messages, and makes no attempt to coordinate the timing of those messages with the corresponding database updates. It is up to higher layers to apply such coordination.

Connector API

All access to the queueing infrastructure is mediated by an MQ connector. The connector's API is defined below. The connector itself is always available as `master.mq`, where `master` is the current `BuildMaster` instance. The connector API is quite simple. It is loosely based on AMQP, although simplified because there is only one exchange (see *Queue Schema*).

All messages include a "routing key", which is a tuple of *7-bit ascii* strings describing the content of the message. By convention, the first element of the tuple gives the type of the data in the message. The last element of the tuple describes the event represented by the message. The remaining elements of the tuple describe attributes of the data in the message that may be useful for filtering; for example, buildsets may usefully be filtered on buildsetids. The topics and associated message types are described below in *Message Schema*.

Filters are also specified with tuples. For a filter to match a routing key, it must have the same length, and each element of the filter that is not None must match the corresponding routing key element exactly.

class `buildbot.mq.base.`**MQConnector**

This is an abstract parent class for MQ connectors, and defines the interface. It should not be instantiated directly. It is a subclass of *buildbot.util.service.AsyncService*, and subclasses can override service methods to start and stop the connector.

produce(*routing_key*, *data*)

Parameters

- **routing_key** (*tuple*) – the routing key for this message
- **data** – JSON-serializable body of the message

This method produces a new message and queues it for delivery to any associated consumers.

The routing key and data should match one of the formats given in *Message Schema*.

The method returns immediately; the caller will not receive any indication of a failure to transmit the message, although errors will be displayed in `twistd.log`.

startConsuming(*callback*, *filter*[, *persistent_name=name*])

Parameters

- **callback** – callable to invoke for matching messages
- **filter** (*tuple*) – filter for routing keys of interest
- **persistent_name** – persistent name for this consumer

Returns a *QueueRef* instance via Deferred

This method will begin consuming messages matching the filter, invoking `callback` for each message. See above for the format of the filter.

The callback will be invoked with two arguments: the message's routing key and the message body, as a Python data structure. It may return a Deferred, but no special processing other than error handling will be applied to that Deferred. In particular, note that the callback may be invoked a second time before the Deferred from the first invocation fires.

A message is considered delivered as soon as the callback is invoked - there is no support for acknowledgements or re-queueing unhandled messages.

Note that the timing of messages is implementation-dependent. It is not guaranteed that messages sent before the *startConsuming* method completes will be received. In fact, because the registration process may not be immediate, even messages sent after the method completes may not be received.

If `persistent_name` is given, then the consumer is assumed to be persistent, and consumption can be resumed with the given name. Messages that arrive when no consumer is active are queued and will be delivered when a consumer becomes active.

waitUntilEvent (*filter*, *check_callback*)

Parameters

- **filter** (*tuple*) – filter for routing keys of interest
- **check_callback** (*function*) – a callback which check if the event has already happened

Returns a Deferred that fires when the event has been received, and contain tuple (routing_key, value) representing the event

This method is a helper which returns a defer that fire when a certain event has occured. This is useful for waiting the end of a build or disconnection of a worker. You shall make sure when using this method that this event will happen in the future, and take care of race conditions. For that caller must provide a check_callback which will check of the event has already occured. The whole race-condition-free process is:

•Register to event

•Check if it has already happened

•If not wait for the event

•Unregister from event

class buildbot.mq.base.**QueueRef**

The *QueueRef* returned (via Deferred) from *startConsuming* can be used to stop consuming messages when they are no longer needed. Users should be *very* careful to ensure that consumption is terminated in all cases.

stopConsuming()

Stop invoking the `callback` passed to *startConsuming*. This method can be called multiple times for the same *QueueRef* instance without harm.

After the first call to this method has returned, the callback will not be invoked.

Implementations

Several concrete implementations of the MQ connector exist. The simplest is intended for cases where only one master exists, similar to the SQLite database support. The remainder use various existing queueing applications to support distributed communications.

Simple

class buildbot.mq.simple.**SimpleMQ**

The *SimpleMQ* class implements a local equivalent of a message-queueing server. It is intended for Buildbot installations with only one master.

Wamp

class buildbot.mq.wamp.**WampMQ**

The *WampMQ* class implements message-queueing using a wamp router. This class translates the semantics of

the buildbot mq api to the semantics of the wamp messaging system. The message route is translated to a wamp topic by joining with dot and prefixing with buildbot namespace. Example message that is sent via wamp is:

```
topic = "org.buildbot.mq.builds.1.new"
data = {
    'builderid': 10,
    'buildid': 1,
    'buildrequestid': 13,
    'workerid': 20,
    'complete': False,
    'complete_at': None,
    'masterid': 824,
    'number': 1,
    'results': None,
    'started_at': 1,
    'state_string': u'created'
}
```

class buildbot.wamp.connector.**WampConnector**

The *WampConnector* class implements a buildbot service for wamp. It is managed outside of the mq module as this protocol can also be reused for worker protocol. The connector support queuing of requests until the wamp connection is created, but do not support disconnection and reconnection. Reconnection will be supported as part of a next release of AutobahnPython (https://github.com/crossbario/autobahn-python/issues/295). There is a chicken and egg problem at the buildbot initialization phasis, so the produce messages are actually not sent with deferred.

Queue Schema

Buildbot uses a particularly simple architecture: in AMQP terms, all messages are sent to a single topic exchange, and consumers define anonymous queues bound to that exchange.

In future versions of Buildbot, some components (e.g., schedulers) may use durable queues to ensure that messages are not lost when one or more masters are disconnected.

Message Schema

This section describes the general structure messages. The specific routing keys and content of each message are described in the relevant sub-section of *Data API*.

Routing Keys

Routing keys are a sequence of strings, usually written with dot separators. Routing keys are represented with variables when one or more of the words in the key are defined by the content of the message. For example, `buildset.$bsid` describes routing keys such as `buildset.1984`, where 1984 is the ID of the buildset described by the message body. Internally, keys are represented as tuples of strings.

Body Format

Message bodies are encoded in JSON. The top level of each message is an object (a dictionary).

Most simple Python types - strings, numbers, lists, and dictionaries - are mapped directly to the corresponding JSON types. Timestamps are represented as seconds since the UNIX epoch in message bodies.

Cautions

Message ordering is generally maintained by the backend implementations, but this should not be depended on. That is, messages originating from the same master are *usually* delivered to consumers in the order they were produced. Thus, for example, a consumer can expect to see a build request claimed before it is completed. That said, consumers should be resilient to messages delivered out of order, at the very least by scheduling a "reload" from state stored in the database when messages arrive in an invalid order.

Unit tests should be used to ensure this resiliency.

Some related messages are sent at approximately the same time. Due to the non-blocking nature of message delivery, consumers should *not* assume that subsequent messages in a sequence remain queued. For example, upon receipt of a `buildset.$bsid.new` message, it is already too late to try to subscribe to the associated build requests messages, as they may already have been consumed.

Schema Changes

Future versions of Buildbot may add keys to messages, or add new messages. Consumers should expect unknown keys and, if using wildcard topics, unknown messages.

Python3 compatibility

A good place to start looking for advice to ensure that any code is compatible with both Python-3.x and Python2.6/2.7 is too look at the python-future cheat sheet (http://python-future.org/compatible_idioms.html) . Buildbot uses the python-future library to ensure compatibility with both Python2.6/2.7 and Python3.x.

Imports

All `__future__` import have to happen at the top of the module, anything else is seen as a syntax error. All imports from the python-future package should happen below `__future__` imports, but before any other.

Yes:

```
from __future__ import print_function
from builtins import str
```

No:

```
from twisted.application import internet
from twisted.spread import pb
from builtins import str
from __future__ import print_function
```

Dictionaries

In python3, `dict.iteritems` is no longer available. While `dict.items()` does exist, it can be memory intensive in python2. For this reason, please use the python.future function `iteritems()`.

Example:

```
d = {"cheese": 4, "bread": 5, "milk": 1}
for item, num in d.iteritems():
    print("We have {} {}".format(num, item))
```

should be written as:

```
from future.utils import iteritems
d = {"cheese": 4, "bread": 5, "milk": 1}
for item, num in iteritems(d):
    print("We have {} {}".format(num, item))
```

This also applies to the similar methods `dict.itervalues()` and `dict.values()`, which have an equivalent `itervalues()`.

If a list is required, please use `list(iteritems(dict))`. This is for compatibility with the six library.

For iterating over dictionary keys, please use `for key in dict:`. For example:

```
d = {"cheese": 4, "bread": 5, "milk": 1}
for item in d:
    print("We have {}".format(item))
```

Similarly when you want a list of keys:

```
keys = list(d)
```

New-style classes

All classes in Python3 are newstyle, so any classes added to the code base must therefore be new-style. This is done by inheriting `object`

Old-style:

```
class Foo:
    def __init__(self, bar)
        self.bar = bar
```

new-style:

```
class Foo(object):
    def __init__(self, bar)
        self.bar = bar
```

When creating new-style classes, it is advised to import `object` from the builtins module. The reasoning for this can be read in the python-future documentation (http://python-future.org/changelog.html#newobject-base-object-defines-fallback-py2-compatible-special-methods)

Strings

Note: This has not yet been implemented in the current code base, and will not be strictly adhered to yet. But it is important to keep in mind when writing code, that there is a strict distinction between bytestrings and unicode in Python3'

In python2, there is only one type of string. It can be both unicode and bytestring. In python3, this is no longer the case. For this reasons all string must be marked with either `u''` or `b''` to signify if the string is a unicode string or a bytestring respectively

Example:

```
u'this is a unicode string, a string for humans to read'
b'This is a bytestring, a string for computers to read'
```

Exceptions

All exceptions should be written with the `as` statement. Before:

```
try:
    number = 5 / 0
except ZeroDivisionError, err:
    print(err.msg)
```

After:

```
try:
    number = 5/0
except ZeroDivisionError as err:
    print(err.msg)
```

Basestring

In Python2 there is a basestring type, which both str and unicode inherit. In Python3, only unicode should be of this type, while bytestrings are `type(byte)`.

For this reason, we use a builtin form python future. Before:

```
s = "this is a string"
if(isinstance(basestring)):
    print "This line will run"
```

After:

```
from builtins import str
unicode_s = u"this is a unicode string"
byte_s = b"this is a bytestring"

if(isinstance(unicode_s, str)):
    print("This line will print")
if(isinstance(byte_s, str):
    print("this line will not print")
```

Print statements

Print statements are gone in python3. Please import `from __future__ import print_function` at the very top of the module to enable use of python3 style print functions

Division

Integer division is slightly different in Python3. `//` is integer division and `/` is floating point division. For this reason, we use `division` from the future module. Before:

```
2 / 3 = 0
```

After:

```
from __future__ import division

2 / 3 = 1.5
2 // 3 = 0
```

Types

The types standard library has changed in Python3. Please make sure to read the official documentation (https://docs.python.org/3.3/library/types.html) for the library and adapt accordingly

Classes

The sections contained here document classes that can be used or subclassed.

Note: Some of this information duplicates information available in the source code itself. Consider this information authoritative, and the source code a demonstration of the current implementation which is subject to change.

Builds

The *Build* class represents a running build, with associated steps.

Build

class buildbot.process.build.**Build**

buildid
The ID of this build in the database.

getSummaryStatistic(*name*, *summary_fn*, *initial_value=None*)

Parameters

- **name** – statistic name to summarize
- **summary_fn** – callable with two arguments that will combine two values
- **initial_value** – first value to pass to `summary_fn`

Returns summarized result

This method summarizes the named step statistic over all steps in which it exists, using `combination_fn` and `initial_value` to combine multiple results into a single result. This translates to a call to Python's `reduce`:

```
return reduce(summary_fn, step_stats_list, initial_value)
```

getUrl()

Returns Url as string

Returns url of the build in the UI. Build must be started. This is useful for customs steps.

Workers

The *Worker* class represents a worker, which may or may not be connected to the master. Instances of this class are created directly in the Buildbot configuration file.

Worker

class buildbot.worker.**Worker**

workerid
The ID of this worker in the database.

BuildFactory

BuildFactory Implementation Note

The default `BuildFactory`, provided in the `buildbot.process.factory` module, contains an internal list of *BuildStep specifications*: a list of `(step_class, kwargs)` tuples for each. These specification tuples are constructed when the config file is read, by asking the instances passed to `addStep` for their subclass and arguments.

To support config files from Buildbot version 0.7.5 and earlier, `addStep` also accepts the `f.addStep(shell.Compile, command=["make","build"])` form, although its use is discouraged because then the `Compile` step doesn't get to validate or complain about its arguments until build time. The modern pass-by-instance approach allows this validation to occur while the config file is being loaded, where the admin has a better chance of noticing problems.

When asked to create a `Build`, the `BuildFactory` puts a copy of the list of step specifications into the new `Build` object. When the `Build` is actually started, these step specifications are used to create the actual set of `BuildSteps`, which are then executed one at a time. This serves to give each Build an independent copy of each step.

Each step can affect the build process in the following ways:

- If the step's `haltOnFailure` attribute is `True`, then a failure in the step (i.e. if it completes with a result of `FAILURE`) will cause the whole build to be terminated immediately: no further steps will be executed, with the exception of steps with `alwaysRun` set to `True`. `haltOnFailure` is useful for setup steps upon which the rest of the build depends: if the CVS checkout or **`./configure`** process fails, there is no point in trying to compile or test the resulting tree.
- If the step's `alwaysRun` attribute is `True`, then it will always be run, regardless of if previous steps have failed. This is useful for cleanup steps that should always be run to return the build directory or worker into a good state.

- If the `flunkOnFailure` or `flunkOnWarnings` flag is set, then a result of `FAILURE` or `WARNINGS` will mark the build as a whole as `FAILED`. However, the remaining steps will still be executed. This is appropriate for things like multiple testing steps: a failure in any one of them will indicate that the build has failed, however it is still useful to run them all to completion.
- Similarly, if the `warnOnFailure` or `warnOnWarnings` flag is set, then a result of `FAILURE` or `WARNINGS` will mark the build as having `WARNINGS`, and the remaining steps will still be executed. This may be appropriate for certain kinds of optional build or test steps. For example, a failure experienced while building documentation files should be made visible with a `WARNINGS` result but not be serious enough to warrant marking the whole build with a `FAILURE`.

In addition, each `Step` produces its own results, may create logfiles, etc. However only the flags described above have any effect on the build as a whole.

The pre-defined `BuildSteps` like `CVS` and `Compile` have reasonably appropriate flags set on them already. For example, without a source tree there is no point in continuing the build, so the `CVS` class has the `haltOnFailure` flag set to `True`. Look in `buildbot/steps/*.py` to see how the other `Steps` are marked.

Each `Step` is created with an additional `workdir` argument that indicates where its actions should take place. This is specified as a subdirectory of the worker's base directory, with a default value of `build`. This is only implemented as a step argument (as opposed to simply being a part of the base directory) because the CVS/SVN steps need to perform their checkouts from the parent directory.

BuildSetSummaryNotifierMixin

Some status notifiers will want to report the status of all builds all at once for a particular buildset, instead of reporting each build individually as it finishes. In order to do this, the status notifier must wait for all builds to finish, collect their results, and then report a kind of summary on all of the collected results. The act of waiting for and collecting the results of all of the builders is implemented via `BuildSetSummaryNotifierMixin`, to be subclassed by a status notification implementation.

BuildSetSummaryNotifierMixin

buildbot.status.buildset.BuildSetSummaryNotifierMixin::

This class provides some helper methods for implementing a status notification that provides notifications for all build results for a buildset at once.

This class provides the following methods:

buildbot.status.buildset.**summarySubscribe**()

Call this to start receiving *sendBuildSetSummary* callbacks. Typically this will be called from the subclass's `startService` method.

buildbot.status.buildset.**summaryUnsubscribe**()

Call this to stop receiving *sendBuildSetSummary* callbacks. Typically this will be called from the subclass's `stopService` method.

The following methods are hooks to be implemented by the subclass.

buildbot.status.buildset.**sendBuildSetSummary**(*buildset*, *builds*)

Parameters

- **buildset** – A `BuildSet` object
- **builds** – A list of `Build` objects

This method must be implemented by the subclass. This method is called when all of the builds for a buildset have finished, and it should initiate sending a summary status for the buildset.

The following attributes must be provided by the subclass.

buildbot.status.buildset.**master**
This must point to the BuildMaster object.

Change Sources

ChangeSource

class buildbot.changes.base.**ChangeSource**
This is the base class for change sources.

Subclasses should override the inherited *activate* and *deactivate* methods if necessary to handle initialization and shutdown.

Change sources which are active on every master should, instead, override startService and stopService.

ReconfigurablePollingChangeSource

class buildbot.changes.base.**ReconfigurablePollingChangeSource**
This is a subclass of *ChangeSource* which adds polling behavior. Its constructor accepts the pollInterval and pollAtLaunch arguments as documented for most built-in change sources.

Subclasses should override the poll method. This method may return a Deferred. Calls to poll will not overlap.

PollingChangeSource

class buildbot.changes.base.**PollingChangeSource**
This is a legacy class for polling change sources not yet ported to the :py:class::*BuildbotService* component lifecycle. Do not use for new code.

RemoteCommands

Most of the action in build steps consists of performing operations on the worker. This is accomplished via *RemoteCommand* and its subclasses. Each represents a single operation on the worker.

Most data is returned to a command via updates. These updates are described in detail in *Updates*.

RemoteCommand

class buildbot.process.remotecommand.**RemoteCommand**(*remote_command*, *args*, *collectStdout=False*, *ignore_updates=False*, *decodeRC=dict(0)*, *stdioLogName='stdio'*)

Parameters

- **remote_command** (*string*) – command to run on the worker
- **args** (*dictionary*) – arguments to pass to the command
- **collectStdout** – if True, collect the command's stdout

- **ignore_updates** – true to ignore remote updates
- **decodeRC** – dictionary associating `rc` values to buildsteps results constants (e.g. `SUCCESS`, `FAILURE`, `WARNINGS`)
- **stdioLogName** – name of the log to which to write the command's stdio

This class handles running commands, consisting of a command name and a dictionary of arguments. If true, `ignore_updates` will suppress any updates sent from the worker.

This class handles updates for `stdout`, `stderr`, and `header` by appending them to s stdio logfile named by the `stdioLogName` parameter. Steps that run multiple commands and want to separate those commands' stdio streams can use this parameter.

It handles updates for `rc` by recording the value in its `rc` attribute.

Most worker-side commands, even those which do not spawn a new process on the worker, generate logs and an `rc`, requiring this class or one of its subclasses. See *Updates* for the updates that each command may send.

active
True if the command is currently running

run(*step*, *remote*)

Parameters

- **step** – the buildstep invoking this command
- **remote** – a reference to the remote `WorkerForBuilder` instance

Returns Deferred

Run the command. Call this method to initiate the command; the returned Deferred will fire when the command is complete. The Deferred fires with the *RemoteCommand* instance as its value.

interrupt(*why*)

Parameters **why** (*Twisted Failure*) – reason for interrupt

Returns Deferred

This method attempts to stop the running command early. The Deferred it returns will fire when the interrupt request is received by the worker; this may be a long time before the command itself completes, at which time the Deferred returned from *run* will fire.

results()

Returns results constant

This method checks the `rc` against the decodeRC dictionary, and returns results constant

didFail()

Returns bool

This method returns True if the results() function returns FAILURE

The following methods are invoked from the worker. They should not be called directly.

remote_update(*updates*)

Parameters **updates** – new information from the worker

Handles updates from the worker on the running command. See *Updates* for the content of the updates. This class splits the updates out, and handles the `ignore_updates` option, then calls *remoteUpdate* to process the update.

remote_complete(*failure=None*)

Parameters **failure** – the failure that caused the step to complete, or None for success

Called by the worker to indicate that the command is complete. Normal completion (even with a nonzero `rc`) will finish with no failure; if `failure` is set, then the step should finish with status *EXCEPTION*.

These methods are hooks for subclasses to add functionality.

remoteUpdate(*update*)

Parameters **update** – the update to handle

Handle a single update. Subclasses must override this method.

remoteComplete(*failure*)

Parameters **failure** – the failure that caused the step to complete, or None for success

Returns Deferred

Handle command completion, performing any necessary cleanup. Subclasses should override this method. If `failure` is not None, it should be returned to ensure proper processing.

logs
A dictionary of `LogFile` instances representing active logs. Do not modify this directly – use *useLog* instead.

rc
Set to the return code of the command, after the command has completed. For compatibility with shell commands, 0 is taken to indicate success, while nonzero return codes indicate failure.

stdout
If the `collectStdout` constructor argument is true, then this attribute will contain all data from stdout, as a single string. This is helpful when running informational commands (e.g., `svnversion`), but is not appropriate for commands that will produce a large amount of output, as that output is held in memory.

To set up logging, use *useLog* or *useLogDelayed* before starting the command:

useLog(*log*, *closeWhenFinished=False*, *logfileName=None*)

Parameters

- **log** – the `LogFile` instance to add to.
- **closeWhenFinished** – if true, call `finish` when the command is finished.
- **logfileName** – the name of the logfile, as given to the worker. This is `stdio` for standard streams.

Route log-related updates to the given logfile. Note that `stdio` is not included by default, and must be added explicitly. The `logfileName` must match the name given by the worker in any `log` updates.

useLogDelayed(*logfileName*, *activateCallback*, *closeWhenFinished=False*)

Parameters

- **logfileName** – the name of the logfile, as given to the worker. This is `stdio` for standard streams.
- **activateCallback** – callback for when the log is added; see below
- **closeWhenFinished** – if true, call `finish` when the command is finished.

Similar to *useLog*, but the logfile is only actually added when an update arrives for it. The callback, `activateCallback`, will be called with the *RemoteCommand* instance when the first update for the log is delivered. It should return the desired log instance, optionally via a Deferred.

With that finished, run the command using the inherited *run* method. During the run, you can inject data into the logfiles with any of these methods:

addStdout(*data*)

Parameters **data** – data to add to the logfile

Returns Deferred

Add stdout data to the `stdio` log.

addStderr(*data*)

Parameters **data** – data to add to the logfile

Returns Deferred

Add stderr data to the `stdio` log.

addHeader(*data*)

Parameters **data** – data to add to the logfile

Returns Deferred

Add header data to the `stdio` log.

addToLog(*logname*, *data*)

Parameters

- **logname** – the logfile to receive the data
- **data** – data to add to the logfile

Returns Deferred

Add data to a logfile other than `stdio`.

class `buildbot.process.remotecommand.`**RemoteShellCommand**(*workdir*, *command*, *env=None*, *want_stdout=True*, *want_stderr=True*, *timeout=20*60*, *maxTime=None*, *sigtermTime=None*, *logfiles={}*, *usePTY=None*, *logEnviron=True*, *collectStdio=False*)

Parameters

- **workdir** – directory in which command should be executed, relative to the builder's basedir.
- **command** (*string or list*) – shell command to run
- **want_stdout** – If false, then no updates will be sent for stdout.
- **want_stderr** – If false, then no updates will be sent for stderr.
- **timeout** – Maximum time without output before the command is killed.
- **maxTime** – Maximum overall time from the start before the command is killed.
- **sigtermTime** – Try to kill the command with SIGTERM and wait for sigtermTime seconds before firing SIGKILL. If None, SIGTERM will not be fired.

- **env** – A dictionary of environment variables to augment or replace the existing environment on the worker.
- **logfiles** – Additional logfiles to request from the worker.
- **usePTY** – True to use a PTY, false to not use a PTY; the default value is False.
- **logEnviron** – If false, do not log the environment on the worker.
- **collectStdout** – If True, collect the command's stdout.

Most of the constructor arguments are sent directly to the worker; see *shell* for the details of the formats. The `collectStdout` parameter is as described for the parent class.

If shell command contains passwords they can be hidden from log files by passing them as tuple in command argument. Eg. `['print', ('obfuscated', 'password', 'dummytext')]` is logged as `['print', 'dummytext']`.

This class is used by the *ShellCommand* step, and by steps that run multiple customized shell commands.

BuildSteps

There are a few parent classes that are used as base classes for real buildsteps. This section describes the base classes. The "leaf" classes are described in *Build Steps*.

See *Writing New BuildSteps* for a guide to implementing new steps.

BuildStep

class buildbot.process.buildstep.**BuildStep**(*name*, *description*, *descriptionDone*, *descriptionSuffix*, *locks*, *haltOnFailure*, *flunkOnWarnings*, *flunkOnFailure*, *warnOnWarnings*, *warnOnFailure*, *alwaysRun*, *progressMetrics*, *useProgress*, *doStepIf*, *hideStepIf*)

All constructor arguments must be given as keyword arguments. Each constructor parameter is copied to the corresponding attribute.

name
The name of the step. Note that this value may change when the step is started, if the existing name was not unique.

stepid
The ID of this step in the database. This attribute is not set until the step starts.

description
The description of the step.

descriptionDone
The description of the step after it has finished.

descriptionSuffix
Any extra information to append to the description.

locks
List of locks for this step; see *Interlocks*.

progressMetrics
List of names of metrics that should be used to track the progress of this build, and build ETA's for users. This is generally set in the

useProgress
If true (the default), then ETAs will be calculated for this step using progress metrics. If the step is known to have unpredictable timing (e.g., an incremental build), then this should be set to false.

doStepIf
A callable or bool to determine whether this step should be executed. See *Common Parameters* for details.

hideStepIf
A callable or bool to determine whether this step should be shown in the waterfall and build details pages. See *Common Parameters* for details.

The following attributes affect the behavior of the containing build:

haltOnFailure
If true, the build will halt on a failure of this step, and not execute subsequent tests (except those with `alwaysRun`).

flunkOnWarnings
If true, the build will be marked as a failure if this step ends with warnings.

flunkOnFailure
If true, the build will be marked as a failure if this step fails.

warnOnWarnings
If true, the build will be marked as warnings, or worse, if this step ends with warnings.

warnOnFailure
If true, the build will be marked as warnings, or worse, if this step fails.

alwaysRun
If true, the step will run even if a previous step halts the build with `haltOnFailure`.

logEncoding
The log encoding to use for logs produced in this step, or None to ues the global default. See *Log Handling*.

rendered
At the beginning of the step, the renderable attributes are rendered against the properties. There is a slight delay however when those are not yet rendered, which lead to weird and difficult to reproduce bugs. To address this problem, a `rendered` attribute is available for methods that could be called early in the buildstep creation.

results
This is the result (a code from *buildbot.process.results*) of the step. This attribute only exists after the step is finished, and should only be used in *getResultSummary*.

A few important pieces of information are not available when a step is constructed, and are added later. These are set by the following methods; the order in which these methods are called is not defined.

setBuild(*build*)

> **Parameters** **build** – the *Build* instance controlling this step.

This method is called during setup to set the build instance controlling this worker. Subclasses can override this to get access to the build object as soon as it is available. The default implementation sets the *build* attribute.

build
The build object controlling this step.

setWorker(*build*)

> **Parameters** **build** – the *Worker* instance on which this step will run.

Similarly, this method is called with the worker that will run this step. The default implementation sets the *worker* attribute.

worker
The worker that will run this step.

workdir
Implemented as a property. Workdir where actions of the step are happening. The workdir is by order of priority

- workdir of the step, if defined via constructor argument
- workdir of the BuildFactory (itself defaults to 'build').

BuildFactory workdir can be a function of sourcestamp. See *Factory Workdir Functions*

setDefaultWorkdir(*workdir*)

Parameters **workdir** – the default workdir, from the build

Note: This method is deprecated and should not be used anymore, as workdir is calculated automatically via a property

---

setupProgress()
This method is called during build setup to give the step a chance to set up progress tracking. It is only called if the build has *useProgress* set. There is rarely any reason to override this method.

Execution of the step itself is governed by the following methods and attributes.

startStep(*remote*)

Parameters **remote** – a remote reference to the worker-side `WorkerForBuilderPb` instance

Returns Deferred

Begin the step. This is the build's interface to step execution. Subclasses should override *run* to implement custom behaviors.

run()

Returns result via Deferred

Execute the step. When this method returns (or when the Deferred it returns fires), the step is complete. The method's return value must be an integer, giving the result of the step – a constant from *buildbot.process.results*. If the method raises an exception or its Deferred fires with failure, then the step will be completed with an EXCEPTION result. Any other output from the step (logfiles, status strings, URLs, etc.) is the responsibility of the `run` method.

Subclasses should override this method. Do *not* call *finished* or *failed* from this method.

start()

Returns `None` or *SKIPPED*, optionally via a Deferred.

Begin the step. BuildSteps written before Buildbot-0.9.0 often override this method instead of *run*, but this approach is deprecated.

When the step is done, it should call *finished*, with a result – a constant from *buildbot.process.results*. The result will be handed off to the *Build*.

If the step encounters an exception, it should call *failed* with a Failure object.

If the step decides it does not need to be run, *start* can return the constant *SKIPPED*. In this case, it is not necessary to call *finished* directly.

finished(*results*)

Parameters **results** – a constant from *results*

A call to this method indicates that the step is finished and the build should analyze the results and perhaps proceed to the next step. The step should not perform any additional processing after calling this method. This method must only be called from the (deprecated) *start* method.

failed(*failure*)

Parameters **failure** – a `Failure` instance

Similar to *finished*, this method indicates that the step is finished, but handles exceptions with appropriate logging and diagnostics.

This method handles *BuildStepFailed* specially, by calling `finished(FAILURE)`. This provides subclasses with a shortcut to stop execution of a step by raising this failure in a context where *failed* will catch it. This method must only be called from the (deprecated) *start* method.

interrupt(*reason*)

Parameters **reason** (string or `Failure`) – why the build was interrupted

This method is used from various control interfaces to stop a running step. The step should be brought to a halt as quickly as possible, by cancelling a remote command, killing a local process, etc. The step must still finish with either *finished* or *failed*.

The `reason` parameter can be a string or, when a worker is lost during step processing, a `ConnectionLost` failure.

The parent method handles any pending lock operations, and should be called by implementations in subclasses.

stopped
If false, then the step is running. If true, the step is not running, or has been interrupted.

A step can indicate its up-to-the-moment status using a short summary string. These methods allow step subclasses to produce such summaries.

updateSummary()
Update the summary, calling *getCurrentSummary* or *getResultSummary* as appropriate. New-style build steps should call this method any time the summary may have changed. This method is debounced, so even calling it for every log line is acceptable.

getCurrentSummary()

Returns dictionary, optionally via Deferred

Returns a dictionary containing status information for a running step. The dictionary can have a `step` key with a unicode value giving a summary for display with the step. This method is only called while the step is running.

New-style build steps should override this method to provide a more interesting summary than the default `u"running"`.

getResultSummary()

Returns dictionary, optionally via Deferred

Returns a dictionary containing status information for a completed step. The dictionary can have keys `step` and `build`, each with unicode values. The `step` key gives a summary for display with the step, while the `build` key gives a summary for display with the entire build. The latter should be used sparingly,

and include only information that the user would find relevant for the entire build, such as a number of test failures. Either or both keys can be omitted.

This method is only called while the step is finished. The step's result is available in `self.results` at that time.

New-style build steps should override this method to provide a more interesting summary than the default, or to provide any build summary information.

describe(*done=False*)

Parameters **done** – If true, the step is finished.

Returns list of strings

Describe the step succinctly. The return value should be a sequence of short strings suitable for display in a horizontally constrained space.

Note: Be careful not to assume that the step has been started in this method. In relatively rare circumstances, steps are described before they have started. Ideally, unit tests should be used to ensure that this method is resilient.

Note: This method is not called for new-style steps. Instead, override *getCurrentSummary* and *getResultSummary*.

Build steps have statistics, a simple key/value store of data which can later be aggregated over all steps in a build. Note that statistics are not preserved after a build is complete.

hasStatistic(*stat*)

Parameters **stat** (*string*) – name of the statistic

Returns True if the statistic exists on this step

getStatistic(*stat*, *default=None*)

Parameters

- **stat** (*string*) – name of the statistic
- **default** – default value if the statistic does not exist

Returns value of the statistic, or the default value

getStatistics()

Returns a dictionary of all statistics for this step

setStatistic(*stat*, *value*)

Parameters

- **stat** (*string*) – name of the statistic
- **value** – value to assign to the statistic

Returns value of the statistic

Build steps support progress metrics - values that increase roughly linearly during the execution of the step, and can thus be used to calculate an expected completion time for a running step. A metric may be a count of lines logged, tests executed, or files compiled. The build mechanics will take care of translating this progress information into an ETA for the user.

setProgress(*metric*, *value*)

Parameters

- **metric** (*string*) – the metric to update
- **value** (*integer*) – the new value for the metric

Update a progress metric. This should be called by subclasses that can provide useful progress-tracking information.

The specified metric name must be included in *progressMetrics*.

The following methods are provided as utilities to subclasses. These methods should only be invoked after the step is started.

workerVersion(*command*, *oldversion=None*)

Parameters

- **command** (*string*) – command to examine
- **oldversion** – return value if the worker does not specify a version

Returns string

Fetch the version of the named command, as specified on the worker. In practice, all commands on a worker have the same version, but passing `command` is still useful to ensure that the command is implemented on the worker. If the command is not implemented on the worker, *workerVersion* will return `None`.

Versions take the form `x.y` where `x` and `y` are integers, and are compared as expected for version numbers.

Buildbot versions older than 0.5.0 did not support version queries; in this case, *workerVersion* will return `oldVersion`. Since such ancient versions of Buildbot are no longer in use, this functionality is largely vestigial.

workerVersionIsOlderThan(*command*, *minversion*)

Parameters

- **command** (*string*) – command to examine
- **minversion** – minimum version

Returns boolean

This method returns true if `command` is not implemented on the worker, or if it is older than `minversion`.

checkWorkerHasCommand(*command*)

Parameters **command** (*string*) – command to examine

This method raise `WorkerTooOldError` if `command` is not implemented on the worker

getWorkerName()

Returns string

Get the name of the worker assigned to this step.

Most steps exist to run commands. While the details of exactly how those commands are constructed are left to subclasses, the execution of those commands comes down to this method:

runCommand(*command*)

Parameters **command** – *RemoteCommand* instance

Returns Deferred

This method connects the given command to the step's worker and runs it, returning the Deferred from *run*.

The *BuildStep* class provides methods to add log data to the step. Subclasses provide a great deal of user-configurable functionality on top of these methods. These methods can be called while the step is running, but not before.

addLog(*name*, *type="s"*, *logEncoding=None*)

Parameters

- **name** – log name
- **type** – log type; see *logchunk*
- **logEncoding** – the log encoding, or None to use the step or global default (see *Log Handling*)

Returns *Log* instance via Deferred

Add a new logfile with the given name to the step, and return the log file instance.

getLog(*name*)

Parameters **name** – log name

Raises

- **KeyError** – if there is no such log
- **KeyError** – if no such log is defined

Returns *Log* instance

Return an existing logfile, previously added with *addLog*. Note that this return value is synchronous, and only available after *addLog*'s deferred has fired.

addCompleteLog(*name*, *text*)

Parameters

- **name** – log name
- **text** – content of the logfile

Returns Deferred

This method adds a new log and sets `text` as its content. This is often useful to add a short logfile describing activities performed on the master. The logfile is immediately closed, and no further data can be added.

If the logfile's content is a bytestring, it is decoded with the step's log encoding or the global default log encoding. To add a logfile with a different character encoding, perform the decode operation directly and pass the resulting unicode string to this method.

addHTMLLog(*name*, *html*)

Parameters

- **name** – log name
- **html** – content of the logfile

Returns Deferred

Similar to *addCompleteLog*, this adds a logfile containing pre-formatted HTML, allowing more expressiveness than the text format supported by *addCompleteLog*.

addLogObserver(*logname*, *observer*)

Parameters

- **logname** – log name
- **observer** – log observer instance

Add a log observer for the named log. The named log need not have been added already: the observer will be connected when the log is added.

See *Adding LogObservers* for more information on log observers.

addLogWithFailure(*why*, *logprefix=''*)

Parameters

- **why** (*Failure*) – the failure to log
- **logprefix** – prefix for the log name

Returns Deferred

Add log files displaying the given failure, named `<logprefix>err.text` and `<logprefix>err.html`.

addLogWithException(*why*, *logprefix=''*)

Parameters

- **why** (*Exception*) – the exception to log
- **logprefix** – prefix for the log name

Returns Deferred

Similar to `addLogWithFailure`, but for an Exception instead of a Failure.

Along with logs, build steps have an associated set of links that can be used to provide additional information for developers. Those links are added during the build with this method:

addURL(*name*, *url*)

Parameters

- **name** – URL name
- **url** – the URL

Add a link to the given `url`, with the given `name` to displays of this step. This allows a step to provide links to data that is not available in the log files.

LoggingBuildStep

class buildbot.process.buildstep.**LoggingBuildStep**(*name*, *locks*, *haltOnFailure*, *flunkOnWarnings*, *flunkOnFailure*, *warnOnWarnings*, *warnOnFailure*, *alwaysRun*, *progressMetrics*, *useProgress*, *doStepIf*, *hideStepIf*)

The remaining arguments are passed to the *BuildStep* constructor.

Warning: Subclasses of this class are always old-style steps. As such, this class will be removed after Buildbot-0.9.0. Instead, subclass *BuildStep* and mix in *ShellMixin* to get similar behavior.

This subclass of *BuildStep* is designed to help its subclasses run remote commands that produce standard I/O logfiles. It:

•tracks progress using the length of the stdout logfile

•provides hooks for summarizing and evaluating the command's result

•supports lazy logfiles

•handles the mechanics of starting, interrupting, and finishing remote commands

•detects lost workers and finishes with a status of *RETRY*

logfiles

The logfiles to track, as described for *ShellCommand*. The contents of the class-level `logfiles` attribute are combined with those passed to the constructor, so subclasses may add log files with a class attribute:

```
class MyStep(LoggingBuildStep):
    logfiles = dict(debug='debug.log')
```

Note that lazy logfiles cannot be specified using this method; they must be provided as constructor arguments.

setupLogsRunCommandAndProcessResults(cmd, stdioLog=None, closeLogWhenFinished=True, er

Parameters

- **command** – the *RemoteCommand* instance to start
- **stdioLog** – an optional *Log* object where the stdout of the command will be stored.
- **closeLogWhenFinished** – a boolean
- **logfiles** – optional dictionary see *ShellCommand*
- **lazylogfiles** – optional boolean see *ShellCommand*

Returns step result from *buildbot.process.results*

Note:

This method permits an optional `errorMessages` parameter, allowing errors detected early in the command process to be logged. It will be removed, and its use is deprecated.

Handle all of the mechanics of running the given command. This sets up all required logfiles, and calls the utility hooks described below.

Subclasses should use that method if they want to launch multiple commands in a single step. One could use that method, like for example

```
@defer.inlineCallbacks
def run(self):
    cmd = RemoteCommand(...)
    res = yield self.setupLogRunCommandAndProcessResults(cmd)
    if res == results.SUCCESS:
         cmd = RemoteCommand(...)
         res = yield self.setupLogRunCommandAndProcessResults(cmd)
    defer.returnValue(res)
```

To refine the status output, override one or more of the following methods. The *LoggingBuildStep* implementations are stubs, so there is no need to call the parent method.

commandComplete(*command*)

Parameters **command** – the just-completed remote command

This is a general-purpose hook method for subclasses. It will be called after the remote command has finished, but before any of the other hook functions are called.

evaluateCommand(*command*)

Parameters **command** – the just-completed remote command

Returns step result from *buildbot.process.results*

This hook should decide what result the step should have.

CommandMixin

The *runCommand* method can run a *RemoteCommand* instance, but it's no help in building that object or interpreting the results afterward. This mixin class adds some useful methods for running commands.

This class can only be used in new-style steps.

class buildbot.process.buildstep.**CommandMixin**

Some remote commands are simple enough that they can boil down to a method call. Most of these take an `abandonOnFailure` argument which, if true, will abandon the entire buildstep on command failure. This is accomplished by raising *BuildStepFailed*.

These methods all write to the `stdio` log (generally just for errors). They do not close the log when finished.

runRmdir(*dir*, *abandonOnFailure=True*)

Parameters

- **dir** – directory to remove
- **abndonOnFailure** – if true, abandon step on failure

Returns Boolean via Deferred

Remove the given directory, using the `rmdir` command. Returns False on failure.

runMkdir(*dir*, *abandonOnFailure=True*)

Parameters

- **dir** – directory to create
- **abndonOnFailure** – if true, abandon step on failure

Returns Boolean via Deferred

Create the given directory and any parent directories, using the `mkdir` command. Returns False on failure.

pathExists(*path*)

Parameters **path** – path to test

Returns Boolean via Deferred

Determine if the given path exists on the worker (in any form - file, directory, or otherwise). This uses the `stat` command.

runGlob(*path*)

Parameters **path** – path to test

Returns list of filenames

Get the list of files matching the given path pattern on the worker. This uses Python's `glob` module. If the `runGlob` method fails, it aborts the step.

getFileContentFromWorker(*path*, *abandonOnFailure=False*)

Parameters **path** – path of the file to download from worker

Returns string via deferred (content of the file)

Get the content of a file on the worker.

ShellMixin

Most Buildbot steps run shell commands on the worker, and Buildbot has an impressive array of configuration parameters to control that execution. The `ShellMixin` mixin provides the tools to make running shell commands easy and flexible.

This class can only be used in new-style steps.

class `buildbot.process.buildstep.`**ShellMixin**

This mixin manages the following step configuration parameters, the contents of which are documented in the manual. Naturally, all of these are renderable.

command

workdir

env

want_stdout

want_stderr

usePTY

logfiles

lazylogfiles

timeout

maxTime

logEnviron

interruptSignal

sigtermTime

initialStdin

decodeRC

setupShellMixin(*constructorArgs*, *prohibitArgs=[]*)

Parameters

- **constructorArgs** (*dict*) – constructor keyword arguments
- **prohibitArgs** (*list*) – list of recognized arguments to reject

Returns keyword arguments destined for *BuildStep*

This method is intended to be called from the shell constructor, passed any keyword arguments not otherwise used by the step. Any attributes set on the instance already (e.g., class-level attributes) are used as defaults. Attributes named in `prohibitArgs` are rejected with a configuration error.

The return value should be passed to the *BuildStep* constructor.

makeRemoteShellCommand(*collectStdout=False*, *collectStderr=False*, ***overrides*)

Parameters

- **collectStdout** – if true, the command's stdout wil be available in `cmd.stdout` on completion
- **collectStderr** – if true, the command's stderr wil be available in `cmd.stderr` on completion
- **overrides** – overrides arguments that might have been passed to *setupShellMixin*

Returns *RemoteShellCommand* instance via Deferred

This method constructs a *RemoteShellCommand* instance based on the instance attributes and any supplied overrides. It must be called while the step is running, as it examines the worker capabilities before creating the command. It takes care of just about everything:

•Creating log files and associating them with the command

•Merging environment configuration

•Selecting the appropriate workdir configuration

All that remains is to run the command with *runCommand*.

The `ShellMixin` class implements *getResultSummary*, returning a summary of the command. If no command was specified or run, it falls back to the default `getResultSummary` based on `descriptionDone`. Subclasses can override this method to return a more appropriate status.

Exceptions

exception `buildbot.process.buildstep.`**BuildStepFailed**

This exception indicates that the buildstep has failed. It is useful as a way to skip all subsequent processing when a step goes wrong. It is handled by *BuildStep.failed*.

BaseScheduler

class `buildbot.schedulers.base.`**BaseScheduler**

This is the base class for all Buildbot schedulers. See *Writing Schedulers* for information on writing new schedulers.

__init__(*name*, *builderNames*, *properties={}*, *codebases={'':{}}*)

Parameters

- **name** – (positional) the scheduler name
- **builderName** – (positional) a list of builders, by name, for which this scheduler can queue builds
- **properties** – a dictionary of properties to be added to queued builds
- **codebases** – the codebase configuration for this scheduler (see user documentation)

Initializes a new scheduler.

The scheduler configuration parameters, and a few others, are available as attributes:

name
This scheduler's name.

builderNames

Type list

Builders for which this scheduler can queue builds.

codebases

Type dict

The codebase configuration for this scheduler.

properties

Type Properties instance

Properties that this scheduler will attach to queued builds. This attribute includes the `scheduler` property.

schedulerid

Type integer

The ID of this scheduler in the `schedulers` table.

Subclasses can consume changes by implementing *gotChange* and calling *startConsumingChanges* from `startActivity`.

startConsumingChanges(*self*, *fileIsImportant=None*, *change_filter=None*, *onlyImportant=False*)

Parameters

- **fileIsImportant** (*callable*) – a callable provided by the user to distinguish important and unimportant changes
- **change_filter** (`buildbot.changes.filter.ChangeFilter` instance) – a filter to determine which changes are even considered by this scheduler, or `None` to consider all changes
- **onlyImportant** (*boolean*) – If True, only important changes, as specified by fileIsImportant, will be added to the buildset.

Returns Deferred

Subclasses should call this method when becoming active in order to receive changes. The parent class will take care of filtering the changes (using `change_filter`) and (if `fileIsImportant` is not None) classifying them.

gotChange(*change*, *important*)

Parameters

- **change** (`Change`) – the new change
- **important** (*boolean*) – true if the change is important

Returns Deferred

This method is called when a change is received. Schedulers which consume changes should implement this method.

If the `fileIsImportant` parameter to `startConsumingChanges` was None, then all changes are considered important. It is guaranteed that the `codebase` of the change is one of the scheduler's codebase.

Note: The `change` instance will instead be a change resource in later versions.

The following methods are available for subclasses to queue new builds. Each creates a new buildset with a build request for each builder.

addBuildsetForSourceStamps(*self*, *sourcestamps=[]*, *waited_for=False*, *reason=''*, *external_idstring=None*, *properties=None*, *builderNames=None*)

Parameters

- **sourcestamps** (*list*) – a list of full sourcestamp dictionaries or sourcestamp IDs
- **waited_for** (*boolean*) – if true, this buildset is being waited for (and thus should continue during a clean shutdown)
- **reason** (*string*) – reason for the build set
- **external_idstring** (*string*) – external identifier for the buildset
- **properties** (`Properties` instance) – properties - in addition to those in the scheduler configuration - to include in the buildset
- **builderNames** (*list*) – a list of builders for the buildset, or None to use the scheduler's configured `builderNames`

Returns (buildset ID, buildrequest IDs) via Deferred

Add a buildset for the given source stamps. Each source stamp must be specified as a complete source stamp dictionary (with keys `revision`, `branch`, `project`, `repository`, and `codebase`), or an integer `sourcestampid`.

The return value is a tuple. The first tuple element is the ID of the new buildset. The second tuple element is a dictionary mapping builder name to buildrequest ID.

addBuildsetForSourceStampsWithDefaults(*reason*, *sourcestamps*, *waited_for=False*, *properties=None*, *builderNames=None*)

Parameters

- **reason** (*string*) – reason for the build set
- **sourcestamps** (*list*) – partial list of source stamps to build
- **waited_for** (*boolean*) – if true, this buildset is being waited for (and thus should continue during a clean shutdown)
- **properties** (`Properties` instance) – properties - in addition to those in the scheduler configuration - to include in the buildset
- **builderNames** (*list*) – a list of builders for the buildset, or None to use the scheduler's configured `builderNames`

Returns (buildset ID, buildrequest IDs) via Deferred, as for *addBuildsetForSourceStamps*

Create a buildset based on the supplied sourcestamps, with defaults applied from the scheduler's configuration.

The `sourcestamps` parameter is a list of source stamp dictionaries, giving the required parameters. Any unspecified values, including sourcestamps from unspecified codebases, will be filled in from the scheduler's configuration. If `sourcestamps` is None, then only the defaults will be used. If `sourcestamps` includes sourcestamps for codebases not configured on the scheduler, they will be included anyway, although this is probably a sign of an incorrect configuration.

addBuildsetForChanges(*waited_for=False*, *reason=''*, *external_idstring=None*, *changeids=[]*, *builderNames=None*, *properties=None*)

Parameters

- **waited_for** (*boolean*) – if true, this buildset is being waited for (and thus should continue during a clean shutdown)
- **reason** (*string*) – reason for the build set
- **external_idstring** (*string*) – external identifier for the buildset
- **changeids** (*list*) – changes from which to construct the buildset
- **builderNames** (*list*) – a list of builders for the buildset, or None to use the scheduler's configured `builderNames`
- **properties** (`Properties` instance) – properties - in addition to those in the scheduler configuration - to include in the buildset

Returns (buildset ID, buildrequest IDs) via Deferred, as for *addBuildsetForSourceStamps*

Add a buildset for the given changes (`changeids`). This will take sourcestamps from the latest of any changes with the same codebase, and will fill in sourcestamps for any codebases for which no changes are included.

The active state of the scheduler is tracked by the following attribute and methods.

active
True if this scheduler is active

activate()

Returns Deferred

Subclasses should override this method to initiate any processing that occurs only on active schedulers. This is the method from which to call `startConsumingChanges`, or to set up any timers or message subscriptions.

deactivate()

Returns Deferred

Subclasses should override this method to stop any ongoing processing, or wait for it to complete. The method's returned Deferred should not fire until the processing is complete.

The state-manipulation methods are provided by *buildbot.util.state.StateMixin*. Note that no locking of any sort is performed between these two functions. They should *only* be called by an active scheduler.

getState(*name*[, *default*])

Parameters

- **name** – state key to fetch
- **default** – default value if the key is not present

Returns Deferred

This calls through to *buildbot.db.state.StateConnectorComponent.getState*, using the scheduler's objectid.

setState(*name*, *value*)

Parameters

- **name** – state key
- **value** – value to set for the key

Returns Deferred

This calls through to *buildbot.db.state.StateConnectorComponent.setState*, using the scheduler's objectid.

ForceScheduler

The force scheduler has a symbiotic relationship with the web application, so it deserves some further description.

Parameters

The force scheduler comes with a fleet of parameter classes. This section contains information to help users or developers who are interested in adding new parameter types or hacking the existing types.

class buildbot.schedulers.forceshed.**BaseParameter**(*name*, *label*, *regex*, ***kwargs*)

This is the base implementation for most parameters, it will check validity, ensure the arg is present if the *required* attribute is set, and implement the default value. It will finally call *updateFromKwargs* to process the string(s) from the HTTP POST.

The *BaseParameter* constructor converts all keyword arguments into instance attributes, so it is generally not necessary for subclasses to implement a constructor.

For custom parameters that set properties, one simple customization point is *getFromKwargs*:

getFromKwargs(*kwargs*)

Parameters **kwargs** – a dictionary of the posted values

Given the passed-in POST parameters, return the value of the property that should be set.

For more control over parameter parsing, including modifying sourcestamps or changeids, override the `updateFromKwargs` function, which is the function that `ForceScheduler` invokes for processing:

updateFromKwargs(*master*, *properties*, *changes*, *sourcestamps*, *collector*, *kwargs*)

Parameters

- **master** – the `BuildMaster` instance
- **properties** – a dictionary of properties
- **changes** – a list of changeids that will be used to build the SourceStamp for the forced builds
- **sourcestamps** – the SourceStamp dictionary that will be passed to the build; some parameters modify sourcestamps rather than properties.
- **collector** – a `buildbot.schedulers.forcesched.ValidationErrorCollector` object, which is used by nestedParameter to collect errors from its childs
- **kwargs** – a dictionary of the posted values

This method updates `properties`, `changes`, and/or `sourcestamps` according to the request. The default implementation is good for many simple uses, but can be overridden for more complex purposes.

When overriding this function, take all parameters by name (not by position), and include an `**unused` catch-all to guard against future changes.

The remaining attributes and methods should be overridden by subclasses, although *BaseParameter* provides appropriate defaults.

name
The name of the parameter. This corresponds to the name of the property that your parameter will set. This name is also used internally as identifier for http POST arguments

label
The label of the parameter, as displayed to the user. This value can contain raw HTML.

fullName()
A fully-qualified name that uniquely identifies the parameter in the scheduler. This name is used internally as the identifier for HTTP POST arguments. It is a mix of *name* and the parent's *name* (in the case of nested parameters). This field is not modifiable.

type
A string identifying the type that the parameter conforms to. It is used by the angular application to find which angular directive to use for showing the form widget. The available values are visible in www/base/src/app/common/directives/forcefields/forcefields.directive.coffee (https://github.com/buildbot/buildbot/blob/master/www/base/src/app/common/directives/forcefields/forcefields.directiv

Examples of how to create a custom parameter widgets are available in the buildbot source code in directories:
- www/codeparameter (https://github.com/buildbot/buildbot/blob/master/www/codeparameter)
- www/nestedexample (https://github.com/buildbot/buildbot/blob/master/www/nestedexample)

default
The default value to use if there is no user input. This is also used to fill in the form presented to the user.

required
If true, an error will be shown to user if there is no input in this field

multiple
If true, this parameter represents a list of values (e.g. list of tests to run)

regex
A string that will be compiled as a regex and used to validate the string value of this parameter. If None, then no validation will take place.

parse_from_args(*l*)
return the list of object corresponding to the list or string passed default function will just call *parse_from_arg* with the first argument

parse_from_arg(*s*)
return the object corresponding to the string passed default function will just return the unmodified string

Nested Parameters

The `NestedParameter` class is a container for parameters. The original motivating purpose for this feature is the multiple-codebase configuration, which needs to provide the user with a form to control the branch (et al) for each codebase independently. Each branch parameter is a string field with name 'branch' and these must be disambiguated.

In Buildbot nine, this concept has been extended to allow grouping different parameters into UI containers. Details of the available layouts is described in *NestedParameter*.

Each of the child parameters mixes in the parent's name to create the fully qualified `fullName`. This allows, for example, each of the 'branch' fields to have a unique name in the POST request. The *NestedParameter* handles adding this extra bit to the name to each of the children. When the *kwarg* dictionary is posted back, this class also converts the flat POST dictionary into a richer structure that represents the nested structure.

As illustration, if the nested parameter has the name 'foo', and has children 'bar1' and 'bar2', then the POST will have entries like "foo.bar1" and "foo.bar2". The nested parameter will translate this into a dictionary in the 'kwargs' structure, resulting in something like:

```
kwargs = {
    # ...
    'foo': {
        'bar1': '...',
        'bar2': '...'
    }
}
```

Arbitrary nesting is allowed and results in a deeper dictionary structure.

Nesting can also be used for presentation purposes. If the name of the `NestedParameter` is empty, the nest is "anonymous" and does not mangle the child names. However, in the HTML layout, the nest will be presented as a logical group.

IRenderable

buildbot.interfaces.IRenderable::

Providers of this class can be "rendered", based on available properties, when a build is started.

getRenderingFor(*iprops*)

Parameters **iprops** – the `IProperties` provider supplying the properties of the build.

Returns the interpretation of the given properties, optionally in a Deferred.

IProperties

buildbot.interfaces.IProperties::

Providers of this interface allow get and set access to a build's properties.

getProperty(*propname*, *default=None*)

Get a named property, returning the default value if the property is not found.

hasProperty(*propname*)

Determine whether the named property exists.

setProperty(*propname*, *value*, *source*)

Set a property's value, also specifying the source for this value.

getProperties()

Get a `buildbot.process.properties.Properties` instance. The interface of this class is not finalized; where possible, use the other `IProperties` methods.

ResultSpecs

Result specifications are used by the *Data API* to describe the desired results of a `get` call. They can be used to filter, sort and paginate the contents of collections, and to limit the fields returned for each item.

Python calls to `get` call can pass a *`ResultSpec`* instance directly. Requests to the HTTP REST API are converted into instances automatically.

Implementers of Data API endpoints can ignore result specifications entirely, except where efficiency suffers. Any filters, sort keys, and so on still present after the endpoint returns its result are applied generically. *ResultSpec* instances are mutable, so endpoints that do apply some of the specification can remove parts of the specification.

Result specifications are applied in the following order:

- Field Selection (fields)
- Filters
- Order
- Pagination (limit/offset)
- Properties

Only fields & properties are applied to non-collection results. Endpoints processing a result specification should take care to replicate this behavior.

class buildbot.data.resultspec.**ResultSpec**

A result specification has the following attributes, which should be treated as read-only:

filters
A list of *Filter* instances to be applied. The result is a logical AND of all filters.

fields
A list of field names that should be included, or None for no sorting. if the field names all begin with −, then those fields will be omitted and all others included.

order
A list of field names to sort on. if any field name begins with −, then the ordering on that field will be in reverse.

limit
The maximum number of collection items to return.

offset
The 0-based index of the first collection item to return.

properties

A list of *Property* instances to be applied. The result is a logical AND of all properties.

All of the attributes can be supplied as constructor keyword arguments.

Endpoint implementations may call these methods to indicate that they have processed part of the result spec. A subsequent call to *apply* will then not waste time re-applying that part.

popProperties()
If a property exists, return its values list and remove it from the result spec.

popFilter(*field*, *op*)
If a filter exists for the given field and operator, return its values list and remove it from the result spec.

popBooleanFilter(*field*)
If a filter exists for the field, remove it and return the expected value (True or False); otherwise return None. This method correctly handles odd cases like field__ne=false.

popStringFilter(*field*)
If one string filter exists for the field, remove it and return the expected value (as string); otherwise return None.

popIntegerFilter(*field*)
If one integer filter exists for the field, remove it and return the expected value (as integer); otherwise return None. raises ValueError if the field is not convertible to integer.

removePagination()
Remove the pagination attributes (*limit* and *offset*) from the result spec. And endpoint that calls this method should return a `ListResult` instance with its pagination attributes set appropriately.

removeOrder()
Remove the order attribute.

popField(*field*)
Remove a single field from the *fields* attribute, returning True if it was present. Endpoints can use this in conditionals to avoid fetching particularly expensive fields from the DB API.

The following method is used internally to apply any remaining parts of a result spec that are not handled by the endpoint.

apply(*data*)
Apply the result specification to the data, returning a transformed copy of the data. If the data is a collection, then the result will be a `ListResult` instance.

class buildbot.data.resultspec.**Filter**(*field*, *op*, *values*)

Parameters

- **field** (*string*) – the field to filter on
- **op** (*string*) – the comparison operator (e.g., "eq" or "gt")
- **values** (*list*) – the values on the right side of the operator

A filter represents a limitation of the items from a collection that should be returned.

Many operators, such as "gt", only accept one value. Others, such as "eq" or "ne", can accept multiple values. In either case, the values must be passed as a list.

class buildbot.data.resultspec.**Property**(*values*)

Parameters **values** (*list*) – the values on the right side of the operator (`eq`)

A property represents an item of a foreign table.

In either case, the values must be passed as a list.

Protocols

To exchange information over the network between master and worker we need to use protocol.

buildbot.worker.protocols.base provide interfaces to implement wrappers around protocol specific calls, so other classes which use them do not need to know about protocol calls or handle protocol specific exceptions.

class buildbot.worker.protocols.base.**Listener**(*master*)

Parameters **master** – `buildbot.master.BuildMaster` instance

Responsible for spawning Connection instances and updating registrations. Protocol-specific subclasses are instantiated with protocol-specific parameters by the buildmaster during startup.

class buildbot.worker.protocols.base.**Connection**(*master*, *worker*)
Represents connection to single worker

proxies
Dictionary containing mapping between `Impl` classes and `Proxy` class for this protocol This may be overridden by subclass to declare its proxy implementations

createArgsProxies(*args*)

Returns shallow copy of args dictionary with proxies instead of impls

Helper method that will use *proxies*, and replace Impl objects by specific Proxy counterpart.

notifyOnDisconnect(*cb*)

Parameters **cb** – callback

Returns buildbot.util.subscriptions.Subscription

Register a callback to be called if worker gets disconnected

loseConnection()
Close connection

remotePrint(*message*)

Parameters **message** (*string*) – message for worker

Returns Deferred

Print message to worker log file

remoteGetWorkerInfo()

Returns Deferred

Get worker information, commands and version, put them in dictionary then return back

remoteSetBuilderList(*builders*)

Parameters **builders** (List) – list with wanted builders

Returns Deferred containing PB references XXX

Take list with wanted builders and send them to worker, return list with created builders

remoteStartCommand(*remoteCommand*, *builderName*, *commandId*, *commandName*, *args*)

Parameters

- **remoteCommand** – *RemoteCommandImpl* instance
- **builderName** (*string*) – self explanatory
- **commandId** (*string*) – command number
- **commandName** (*string*) – command which will be executed on worker
- **args** (List) – arguments for that command

Returns Deferred

Start command on worker

remoteShutdown()

Returns Deferred

Shutdown the worker, causing its process to halt permanently.

remoteStartBuild(*builderName*)
:param builderName name of the builder for which the build is starting :returns: Deferred

Just starts build

remoteInterruptCommand(*builderName*, *commandId*, *why*)

Parameters

- **builderName** (*string*) – self explanatory
- **commandId** (*string*) – command number

- **why** (*string*) – reason to interrupt

Returns Deferred

Interrupt the command executed on builderName with given commandId on worker, print reason "why" to worker logs

Following classes are describing the worker -> master part of the protocol.

In order to support old workers, we must make sure we do not change the current pb protocol. This is why we implement a `Impl vs Proxy` methods. All the objects that are referenced from the workers for remote calls have an `Impl` and a `Proxy` base classes in this module.

`Impl` classes are subclassed by buildbot master, and implement the actual logic for the protocol api. `Proxy` classes are implemented by the worker/master protocols, and implements the demux and de-serialization of protocol calls.

On worker sides, those proxy objects are replaced by a proxy object having a single method to call master side methodss:

class `buildbot.worker.protocols.base.`**`workerProxyObject`**

`callRemote`(*message*, **args*, ***kw*)
calls the method `"remote_" + message` on master side

class `buildbot.worker.protocols.base.`**`RemoteCommandImpl`**
Represents a RemoteCommand status controller

`remote_update`(*updates*)

Parameters **`updates`** – dictionary of updates

Called when the workers has updates to the current remote command

possible keys for updates are:

- `stdout`: Some logs where captured in remote command's stdout. value: `<data> as string`
- `stderr`: Some logs where captured in remote command's stderr. value: `<data> as string`
- `header`: remote command's header text. value: `<data> as string`
- `log`: one of the watched logs has received some text. value: `(<logname> as string, <data> as string)`
- `rc`: Remote command exited with a return code. value: `<rc> as integer`
- `elapsed`: Remote command has taken <elapsed> time. value: `<elapsed seconds> as float`
- `stat`: sent by the `stat` command with the result of the os.stat, converted to a tuple. value: `<stat> as tuple`
- `files`: sent by the `glob` command with the result of the glob.glob. value: `<files> as list of string`
- `got_revision`: sent by the source commands with the revision checked out. value: `<revision> as string`
- `repo_downloaded`: sent by the `repo` command with the list of patches downloaded by repo. value: `<downloads> as list of string`

class `buildbot.worker.protocols.base.`**`FileWriterImpl`**
Class used to implement data transfer between worker and master

class buildbot.worker.protocols.base.**FileReaderImpl**(*object*)

remote_read(*maxLength*)

Parameters **maxLength** – maximum length of the data to send

Returns data read

called when worker needs more data

remote_close()

Called when master should close the file

WorkerManager

WorkerRegistration

class buildbot.worker.manager.**WorkerRegistration**(*master*, *worker*)

Represents single Worker registration

unregister()

Remove registartion for *worker*

update(*worker_config*, *global_config*)

Parameters

- **worker_config** (*Worker*) – new Worker instance
- **global_config** (*MasterConfig*) – Buildbot config

Update the registration in case the port or password has changed.

NOTE: You should invoke this method after calling *WorkerManager.register(worker)*

WorkerManager

class buildbot.worker.manager.**WorkerManager**(*master*)

Handle Worker registrations for mulitple protocols

register(*worker*)

Parameters **worker** (*Worker*) – new Worker instance

Returns *WorkerRegistration*

Creates *WorkerRegistration* instance.

NOTE: You should invoke *.update()* on returned WorkerRegistration instance

Logs

class buildbot.process.log.**Log**

This class handles write-only access to log files from running build steps. It does not provide an interface for reading logs - such access should occur directly through the Data API.

Instances of this class can only be created by the *addLog* method of a build step.

name

The name of the log.

type
The type of the log, represented as a single character. See *logchunk* for details.

logid
The ID of the logfile.

decoder
A callable used to decode bytestrings. See *logEncoding*.

subscribe(*receiver*)

Parameters **receiver** (*callable*) – the function to call

Register `receiver` to be called with line-delimited chunks of log data. The callable is invoked as `receiver(stream, chunk)`, where the stream is indicated by a single character, or None for logs without streams. The chunk is a single string containing an arbitrary number of log lines, and terminated with a newline. When the logfile is finished, `receiver` will be invoked with `None` for both arguments.

The callable cannot return a Deferred. If it must perform some asynchronous operation, it will need to handle its own Deferreds, and be aware that multiple overlapping calls may occur.

Note that no "rewinding" takes place: only log content added after the call to `subscribe` will be supplied to `receiver`.

finish()

Returns Deferred

This method indicates that the logfile is finished. No further additions will be permitted.

In use, callers will receive a subclass with methods appropriate for the log type:

class buildbot.process.log.**TextLog**

addContent(text):

Parameters **text** – log content

Returns Deferred

Add the given data to the log. The data need not end on a newline boundary.

class buildbot.process.log.**HTMLLog**

addContent(text):

Parameters **text** – log content

Returns Deferred

Same as `TextLog.addContent`.

class buildbot.process.log.**StreamLog**
This class handles logs containing three interleaved streams: stdout, stderr, and header. The resulting log maintains data distinguishing these streams, so they can be filtered or displayed in different colors. This class is used to represent the stdio log in most steps.

addStdout(*text*)

Parameters **text** – log content

Returns Deferred

Add content to the stdout stream. The data need not end on a newline boundary.

addStderr(*text*)

Parameters **text** – log content

Returns Deferred

Add content to the stderr stream. The data need not end on a newline boundary.

addHeader(*text*)

Parameters **text** – log content

Returns Deferred

Add content to the header stream. The data need not end on a newline boundary.

LogObservers

class buildbot.process.logobserver.**LogObserver**

This is a base class for objects which receive logs from worker commands as they are produced. It does not provide an interface for reading logs - such access should occur directly through the Data API.

See *Adding LogObservers* for help creating and using a custom log observer.

The three methods that subclasses may override follow. None of these methods may return a Deferred. It is up to the callee to handle any asynchronous operations. Subclasses may also override the constructor, with no need to call *LogObserver*'s constructor.

outReceived(data):

Parameters **data** (*unicode*) – received data

This method is invoked when a "chunk" of data arrives in the log. The chunk contains one or more newline-terminated unicode lines. For stream logs (e.g., stdio), output to stderr generates a call to errReceived, instead.

errReceived(data):

Parameters **data** (*unicode*) – received data

This method is similar to outReceived, but is called for output to stderr.

headerReceived(data):

Parameters **data** (*unicode*) – received data

This method is similar to outReceived, but is called for header output.

finishReceived()

This method is invoked when the observed log is finished.

class buildbot.process.logobserver.**LogLineObserver**

This subclass of *LogObserver* calls its subclass methods once for each line, instead of once per chunk.

outLineReceived(line):

Parameters **line** (*unicode*) – received line, without newline

Like outReceived, this is called once for each line of output received. The argument does not contain the trailing newline character.

errLineReceived(line):

Parameters **line** (*unicode*) – received line, without newline

Similar to outLineReceived, but for stderr.

headerLineReceived(line):

Parameters **line** (*unicode*) – received line, without newline

Similar to `outLineReceived`, but for header output..

finishReceived()
This method, inherited from *LogObserver*, is invoked when the observed log is finished.

class buildbot.process.logobserver.**LineConsumerLogObserver**
This subclass of *LogObserver* takes a generator function and "sends" each line to that function. This allows consumers to be written as stateful Python functions, e.g.,

```
def logConsumer(self):
    while True:
        stream, line = yield
        if stream == 'o' and line.startswith('W'):
            self.warnings.append(line[1:])

def __init__(self):
    ...
    self.warnings = []
    self.addLogObserver('stdio', logobserver.LineConsumerLogObserver(self.
↪logConsumer))
```

Each `yield` expression evaluates to a tuple of (stream, line), where the stream is one of 'o', 'e', or 'h' for stdout, stderr, and header, respectively. As with any generator function, the `yield` expression will raise a `GeneratorExit` exception when the generator is complete. To do something after the log is finished, just catch this exception (but then re-raise it or return)

```
def logConsumer(self):
    while True:
        try:
            stream, line = yield
            if stream == 'o' and line.startswith('W'):
                self.warnings.append(line[1:])
        except GeneratorExit:
            self.warnings.sort()
            return
```

Warning: This use of generator functions is a simple Python idiom first described in PEP 342 (https://www.python.org/dev/peps/pep-0342/). It is unrelated to the generators used in `inlineCallbacks`. In fact, consumers of this type are incompatible with asynchronous programming, as each line must be processed immediately.

class buildbot.process.logobserver.**BufferLogObserver**(*wantStdout=True*, *wantStderr=False*)

Parameters

- **wantStdout** (*boolean*) – true if stdout should be buffered
- **wantStderr** (*boolean*) – true if stderr should be buffered

This subclass of *LogObserver* buffers stdout and/or stderr for analysis after the step is complete. This can cause excessive memory consumption if the output is large.

getStdout()

Returns unicode string

Return the accumulated stdout.

getStderr()

Returns unicode string

Return the accumulated stderr.

Authentication

class buildbot.www.auth.**AuthBase**

This class is the base class for all authentication methods. All authentications are not done at the same level, so several optional methods are available. This class implements default implementation. The login session is stored via twisted's request.getSession(), and detailed used information is stored in request.getSession().user_info. The session information is then sent to the UI via the config constant (in the user attribute of config)

userInfoProvider

Authentication modules are responsible for providing user information as detailed as possible. When there is a need to get additional information from another source, a userInfoProvider can optionally be specified.

reconfigAuth(*master*, *new_config*)

Parameters

- **master** – the reference to the master
- **new_config** – the reference to the new configuration

Reconfigure the authentication module. In the base class, this simply sets self.master.

maybeAutoLogin(*request*)

Parameters **request** – the request object

Returns Deferred

This method is called when /config.js is fetched. If the authentication method supports automatic login, e.g., from a header provided by a frontend proxy, this method handles the login.

If it succeeds, the method sets request.getSession().user_info. If the login fails unexpectedly, it raises resource.Error. The default implementation simply returns without setting user_info.

getLoginResource()

Return the resource representing /auth/login.

getLogout()

Return the resource representing /auth/logout.

updateUserInfo(*request*)

Parameters **request** – the request object

Separate entrypoint for getting user information. This is a mean to call self.userInfoProvider if provided.

class buildbot.www.auth.**UserInfoProviderBase**

Class that can be used, to get more info for the user like groups, in a separate database.

getUserInfo(*username*)

returns the user infos, from the username used for login (via deferred)

returns a `dict` with following keys:

- `email`: email address of the user
- `full_name`: Full name of the user, like "Homer Simpson"
- `groups`: groups the user belongs to, like ["duff fans", "dads"]

class `buildbot.www.oauth2.`**`OAuth2Auth`**

OAuth2Auth implements oauth2 2 phases authentication. With this method `/auth/login` is called twice. Once without argument. It should return the URL the browser has to redirect in order to perform oauth2 authentication, and authorization. Then the oauth2 provider will redirect to `/auth/login?code=???`, and buildbot web server will verify the code using the oauth2 provider.

Typical login process is:

- UI calls the `/auth/login` api, and redirect the browser to the returned oauth2 provider url
- oauth2 provider shows a web page with a form for the user to authenticate, and ask the user the permission for buildbot to access its account.
- oauth2 provider redirects the browser to `/auth/login?code=???`
- OAuth2Auth module verifies the code, and get the user's additional information
- buildbot UI is reloaded, with the user authenticated.

This implementation is using requests (http://docs.python-requests.org/en/latest/) subclasses must override following class attributes: * `name` Name of the oauth plugin * `faIcon` font awesome class to use for login button logo * `resourceEndpoint` URI of the resource where the authentication token is used * `authUri` URI the browser is pointed to to let the user enter creds * `tokenUri` URI to verify the browser code and get auth token * `authUriAdditionalParams` Additional parameters for the authUri * `tokenUriAdditionalParams` Additional parameters for the tokenUri

`getUserInfoFromOAuthClient`(*self*, *c*)

This method is called after a successful authentication to get additional information about the user from the oauth2 provider.

Avatars

Buildbot's avatar support associate a small image with each user.

class `buildbot.www.avatar.`**`AvatarBase`**

Class that can be used, to get more the avatars for the users. This can be used for the authenticated users, but also for the users referenced by changes.

`getUserAvatar`(*self*, *email*, *size*, *defaultAvatarUrl*)

returns the user's avatar, from the user's email (via deferred). If the data is directly available, this function returns a tuple `(mime_type, picture_raw_data)`. If the data is available in another URL, this function can raise `resource.Redirect(avatar_url)`, and the web server will redirect to the avatar_url.

Web Server Classes

Most of the source in master/buildbot/www (https://github.com/buildbot/buildbot/blob/master/master/buildbot/www) is self-explanatory. However, a few classes and methods deserve some special mention.

Resources

class buildbot.www.resource.**Redirect**(*url*)
This is a subclass of Twisted Web's Error. If this is raised within *asyncRenderHelper*, the user will be redirected to the given URL.

class buildbot.www.resource.**Resource**
This class specializes the usual Twisted Web Resource class.

It adds support for resources getting notified when the master is reconfigured.

needsReconfig
If True, *reconfigResource* will be called on reconfig.

reconfigResource(*new_config*)

Parameters **new_config** – new *MasterConfig* instance

Returns Deferred if desired

Reconfigure this resource.

It's surprisingly difficult to render a Twisted Web resource asynchronously. This method makes it quite a bit easier:

asyncRenderHelper(*request*, *callable*, *writeError=None*)

Parameters

- **request** – the request instance
- **callable** – the render function
- **writeError** – optional callable for rendering errors

This method will call callable, which can return a Deferred, with the given request. The value returned from this callable will be converted to an HTTP response. Exceptions, including Error subclasses, are handled properly. If the callable raises *Redirect*, the response will be a suitable HTTP 302 redirect.

Use this method as follows:

```
def render_GET(self, request):
    return self.asyncRenderHelper(request, self.renderThing)
```

CHAPTER 4

Release Notes

Buildbot 0.9.5 (2017-03-18)

Bug fixes

- Fix issue with compressing empty log
- Fix issue with db being closed by wrong thread
- Fix issue with buildbot_worker not closing file handles when using the transfer steps
- Fix issue with buildbot requesting too many permissions from GitHub's OAuth
- Fix `HTTPStep` to accept `json` as keyword argument.
- Updated `OpenStackLatentWorker` to use keystoneauth1 so it will support latest python-novaclient packages.
- Include `RpmLint` step in steps plugins.

Core Features

- Experimental support for Python 3.5 and 3.6. Note that complete support depends on fixes to be released in Twisted 17.2.0.
- New experimental *Secret Management* framework, which allows to securely declare secrets, reusable in your steps.
- New *Writing Dashboards with Flask or Bottle* plugin, which allows to write custom dashboard with traditional server side web frameworks.
- Added `AnyControlEndpointMatcher` and `EnableSchedulerEndpointMatcher` for better configurability of the access control. If you have access control to your Buildbot, it is recommended you add `AnyControlEndpointMatcher` at the end of your access control configuration.
- Schedulers can now be toggled on and off from the UI. Useful for temporarily disabling periodic timers.

Components Features

- `FileUpload` now supports setting the url title text that is visible in the web UI. `FileUpload` now supports custom *description* and *descriptionDone* text.
- *`EC2LatentWorker`* now provides instance id as the *instance* property enabling use of the AWS toolkit.
- Add GitHub pull request Poller to list of available changesources.
- `OAuth2LoginResource` now supports the *token* URL parameter. If a user wants to authenticate through OAuth2 with a pre- generated token (such as the *access_token* provided by GitHub) it can be passed to */auth/login* as the *token* URL parameter and the user will be authenticated to buildbot with those credentials.
- New reporter *`GitHubCommentPush`* can comment on GitHub PRs
- `GitPoller` now supports polling tags in a git repository.
- `MultipleFilUpload` now supports the *glob* parameter. If *glob* is set to *True* all *workersrcs* parameters will be run through *glob* and the result will be uploaded to *masterdest*
- Changed `OpenStackLatentWorker` to default to v2 of the Nova API. The novaclient package has had a deprecation warning about v1.1 and would use v2 anyway.

Deprecations and Removals

- `master/contrib` and `worker/contrib` directories have been moved to their own repository at https://github.com/buildbot/buildbot-contrib/

Buildbot `0.9.4 (2017-02-08)`

Database upgrade

A database upgrade is necessary for this release (see *`upgrade-master`*).

Bug fixes

- Like for `buildbot start`, `buildbot upgrade-master` will now erase an old pidfile if the process is not live anymore instead of just failing.
- Change properties 'value' changed from String(1024) to Text. Requires upgrade master. (bug #3197 (http://trac.buildbot.net/ticket/3197))
- When using REST API, it is now possible to filter and sort in descending order at the same time.
- Fix issue with *`HttpStatusPush`* raising `datetime is not JSON serializable` error.
- Fix issue with log viewer not properly rendering color codes.
- Fixed log viewer selection and copy-paste for Firefox (bug #3662 (http://trac.buildbot.net/ticket/3662)).
- Fix issue with `DelayedCalled` already called, and worker missing notification email never received.
- *`schedulers`* and *`change_source`* are now properly taking configuration change in account with `buildbot reconfig`.
- `setuptools` is now explicitly marked as required. The dependency was previously implicit.

- *buildbotNetUsageData* now uses `requests` if available and will default to HTTP if a bogus SSL implementation is found. It will also correctly send information about the platform type.

Features

- Buildbot now uses JWT (https://en.wikipedia.org/wiki/JSON_Web_Token) to store its web UI Sessions. Sessions now persist upon buildbot restart. Sessions are shared between masters. Session expiration time is configurable with `c['www']['cookie_expiration_time']` see *www*.
- Builders page has been optimized and can now be displayed with 4 http requests whatever is the builder count (previously, there was one http request per builder).
- Builder and Worker page build list now have the `numbuilds=` option which allows to show more builds.
- Masters page now shows more information about a master (workers, builds, activity timer)
- Workers page improvements:
  - Shows which master the worker is connected to.
  - Shows correctly the list of builders that this master is configured on (not the list of `buildermaster` which nobody cares about).
  - Shows list of builds per worker similar to the builders page.
  - New worker details page displays the list of builds built by this worker using database optimized query.

Deprecations and Removals

- Some deprecated broken *Contrib Scripts* were removed.
- `buildbot.www.hooks.googlecode` has been removed, since the Google Code service has been shut down.
- `buildbot.util.json` has been deprecated in favor of the standard library `json`. `simplejson` will not be used anymore if found in the virtualenv.

Buildbot `0.9.3 ( 2017-01-11 )`

Bug fixes

- Fix *BitbucketStatusPush* `ep should start with /` assertion error.
- Fix duplicate worker use case, where a worker with the same name would make the other worker also disconnect (bug #3656 (http://trac.buildbot.net/ticket/3656))
- `GitPoller: buildPushesWithNoCommits` now rebuilds for a known branch that was updated to an existing commit.
- Fix issue with log viewer not staying at bottom of the log when loading log lines.
- Fixed *addBuildURLs* in `Trigger` to use results from triggered builds to include in the URL name exposed by API.
- Fix *Wamp* *mq* support by removing `debug`, `debug_amp` and `debug_app` from the *mq* config, which is not available in latest version of Python Autobahn (http://autobahn.ws). You can now use `wamp_debug_level` option instead.

- fix issue with factory.workdir AttributeError are not properly reported.

Features

- Optimize the memory consumption of the log compression process. Buildbot do not load the whole log into memory anymore. This should improve a lot buildbot memory footprint.
- Changed the build page so that the preview of the logs are shown in live. It is a preview means the last lines of log. How many lines is configurable per user in the user settings page.
- Log viewer line numbers are no longer selectable, so that it is easier to copy paste.
- *DockerLatentWorker* accepts now renderable Dockerfile
- *Renderer* function can now return `IRenderable` objects.
- new *SetProperties* which allows to generate and transform properties separately.
- Handle new workers in *windows_service.py* script.
- Sort the builders in the waterfall view by name instead of ID.

Buildbot 0.9.2 (2016-12-13)

Bug fixes

- Fix *GitHubAuth* to retrieve all organizations instead of only those publicly available.
- Fixed *ref* to point to *branch* instead of commit *sha* in `GitlabStatusPush`
- *IRC* `maybeColorize` is able to highlight single words and stop colorization at the end. The previous implementation only stopped colorization but not boldface.
- fix compatibility issue with mysql5 (do not set default value for TEXT column).
- Fixed *addChange* in *Change* to use the *revlink* configuration option to generate the revlink.
- fix threading issue in *DockerLatentWorker*

Features

- Implement *BitbucketAuth*.
- New *GerritEventLogPoller* poller to poll Gerrit changes via http API.
- New *GerritVerifyStatusPush* can send multiple review status for the same Gerrit change.
- *IRC* appends the builder URL to a successful/failed build if available
- *MailNotifier* now accepts `useSmtps` parameter for initiating connection over an SSL/TLS encrypted connection (SMTPS)
- New support for `Mesos` and Marathon (https://mesosphere.github.io/marathon/) via *MarathonLatentWorker*. `Marathon` is a production-grade container orchestration platform for Mesosphere's Data- center Operating System (DC/OS) and Apache `Mesos`.
- `password` in *DockerLatentWorker* and *HyperLatentWorker*, can be None. In that case, they will be auto-generated from random number.

- *StashStatusPush* now accepts `key`, `buildName`, `endDescription`, `startDescription`, and `verbose` parameters to control the JSON sent to Stash.
- Buildbot can now be configured to deny read access to REST api resources based on authorization rules.

CHAPTER 5

Older Release Notes

Release Notes for Buildbot `0.9.1`

The following are the release notes for Buildbot `0.9.1`. This version was released on November 1, 2016.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Features

- Add support for hyper.sh via `HyperLatentWorker` Hyper (https://hyper.sh) is a CaaS solution for hosting docker container in the cloud, billed to the second. It forms a very cost efficient solution to run your CI in the cloud.
- The *`Trigger`* step now supports `unimportantSchedulerNames`
- add a UI button to allow to cancel the whole queue for a builder
- Buildbot log viewer now support 256 colors ANSI codes
- new *`GitHub`* which correctly checkout the magic branch like `refs/pull/xx/merge`.
- `MailNotifier` now supports a *schedulers* constructor argument that allows you to send mail only for builds triggered by the specified list of schedulers.
- `MailNotifier` now supports a *branches* constructor argument that allows you to send mail only for builds triggered by the specified list of branches.
- Optimization of the data api filtering, sorting and paging, speeding up a lot the UI when the master has lots of builds.
- *`GerritStatusPush`* now accepts a `notify` parameter to control who gets emailed by Gerrit.
- Add a `format_fn` parameter to the `HttpStatusPush` reporter to customize the information being pushed.
- Latent Workers can now start in parallel.

  - The build started by latent worker will be created while the latent worker is substantiated.
  - Latent Workers will now report startup issues in the UI.
- **Workers will be temporarily put in quarantine in case of build preparation issues.** This avoids master and database overload in case of bad worker configuration. The quarantine is implemented with an exponential back-off timer.
- Master Stop will now stop all builds, and wait for all workers to properly disconnect. Previously, the worker connections was stopped, which incidentally made all their builds marked retried. Now, builds started with a `Triggereable` scheduler will be cancelled, while other builds will be retried. The master will make sure that all latent workers are stopped.
- The `MessageFormatter` class also allows inline-templates with the `template` parameter.
- The `MessageFormatter` class allows custom mail's subjects with the `subject` and `subject_name` parameters.
- The `MessageFormatter` class allows extending the context given to the Templates via the `ctx` parameter.
- The new `MessageFormatterMissingWorker` class allows to customize the message sent when a worker is missing.
- The *`OpenStackLatentWorker`* worker now supports rendering the block device parameters. The `volume_size` parameter will be automatically calculated if it is `None`.

Fixes

- fix the UI to allow to cancel a buildrequest (bug #3582 (http://trac.buildbot.net/ticket/3582))
- *`GitHub`* change hook now correctly use the refs/pull/xx/merge branch for testing PRs.
- Fix the UI to better adapt to different screen width (bug #3614 (http://trac.buildbot.net/ticket/3614))
- Don't log `AlreadyClaimedError`. They are normal in case of *`Trigger`* cancelling, and in a multimaster configuration.
- Fix issues with worker disconnection. When a worker disconnects, its current buildstep must be interrupted and the buildrequests should be retried.
- Fix the worker missing email notification.
- Fix issue with worker builder list not being updated in UI when buildmaster is reconfigured (bug #3629 (http://trac.buildbot.net/ticket/3629))

Changes for Developers

Features

- New `SharedService` can be used by steps, reporters, etc to implement per master resource limit.
- New `HTTPClientService` can be used by steps, reporters, etc to implement HTTP client. This class will automatically choose between treq (https://pypi.python.org/pypi/treq) and txrequests (https://pypi.python.org/pypi/txrequests), whichever is installed, in order to access HTTP servers. This class comes with a fake implementation helping to write unit tests.
- All HTTP reporters have been ported to `HTTPClientService`

Fixes

Deprecations, Removals, and Non-Compatible Changes

- By default, non-distinct commits received via `buildbot.status.web.hooks.github.GitHubEventHandler` now get recorded as a `Change`. In this way, a commit pushed to a branch that is not being watched (e.g. a dev branch) will still get acted on when it is later pushed to a branch that is being watched (e.g. master). In the past, such a commit would get ignored and not built because it was non-distinct. To disable this behavior and revert to the old behavior, install a `ChangeFilter` that checks the `github_distinct` property:

```
ChangeFilter(filter_fn=lambda c: c.properties.getProperty('github_distinct'))
```

- setup.py 'scripts' have been converted to console_scripts entry point. This makes them more portable and compatible with wheel format. Most consequences are for the windows users:
  - `buildbot.bat` does not exist anymore, and is replaced by `buildbot.exe`, which is generated by the console_script entrypoint.
  - `buildbot_service.py` is replaced by `buildbot_windows_service.exe`, which is generated by the console_script entrypoint As this script has been written in 2006, has only inline documentation and no unit tests, it is not guaranteed to be working. Please help improving the windows situation.
- The `user` and `password` parameters of the `HttpStatusPush` reporter have been deprecated in favor of the `auth` parameter.
- The `template_name` parameter of the `MessageFormatter` class has been deprecated in favor of `template_filename`.

Worker

Fixes

Changes for Developers

Deprecations, Removals, and Non-Compatible Changes

- The worker now requires at least Twisted 10.2.0.
- setup.py 'scripts' have been converted to console_scripts entry point. This makes them more portable and compatible with wheel format. Most consequences are for the windows users:
  - `buildbot_worker.bat` does not exist anymore, and is replaced by `buildbot_worker.exe`, which is generated by the console_script entrypoint.
  - `buildbot_service.py` is replaced by `buildbot_worker_windows_service.exe`, which is generated by the console_script entrypoint As this script has been written in 2006, has only inline documentation and no unit tests, it is not guaranteed to be working. Please help improving the windows situation.
- `AbstractLatentWorker` is now in `buildbot.worker.latent` instead of `buildbot.worker.base`.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0..v0.9.1
```

Release Notes for Buildbot 0.9.0

The following are the release notes for Buildbot 0.9.0. This version was released on October 6, 2016.

This is a concatenation of important changes done between 0.8.12 and 0.9.0. This does not contain details of the bug fixes related to the nine beta and rc period. This document was written during the very long period of nine development. It might contain some incoherencies, *please* help us and report them on irc or trac.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

This version represents a refactoring of Buildbot into a consistent, well-defined application composed of loosely coupled components. The components are linked by a common database backend and a messaging system. This allows components to be distributed across multiple build masters. It also allows the rendering of complex web status views to be performed in the browser, rather than on the buildmasters.

The branch looks forward to committing to long-term API compatibility, but does not reach that goal. The Buildbot-0.9.x series of releases will give the new APIs time to "settle in" before we commit to them. Commitment will wait for Buildbot-1.0.0 (as per http://semver.org). Once Buildbot reaches version 1.0.0, upgrades will become much easier for users.

To encourage contributions from a wider field of developers, the web application is designed to look like a normal AngularJS application. Developers familiar with AngularJS, but not with Python, should be able to start hacking on the web application quickly. The web application is "pluggable", so users who develop their own status displays can package those separately from Buildbot itself.

Other goals:

- An approachable HTTP REST API, with real time event features used by the web application but available for any other purpose.
- A high degree of coverage by reliable, easily-modified tests.
- "Interlocking" tests to guarantee compatibility. For example, the real and fake DB implementations must both pass the same suite of tests. Then no unseen difference between the fake and real implementations can mask errors that will occur in production.

Requirements

The `buildbot` package requires Python 2.7 – Python 2.5 and 2.6 are no longer supported. The `buildbot-slave` package requires Python 2.6 or higher – Python 2.4 and 2.5 are no longer supported.

No additional software or systems, aside from some minor Python packages, are required.

But the devil is in the details:

- If you want to do web *development*, or *build* the `buildbot-www` package, you'll need Node. It's an Angular app, and that's how such apps are developed. We've taken pains to not make either a requirement for users - you can simply 'pip install' `buildbot-www` and be on your way. This is the case even if you're hacking on the Python side of Buildbot.
- For a single master, nothing else is required.

Note for distro package maintainers: The npm dependency hell

In order to *build* the `buildbot-www` package, you'll need Node.

Node has a very specific package manager named npm, which has the interesting property of allowing different version of package to co-exist in the same environment. The node ecosystem also has the habit of creating packages for a few line of code.

Buildbot UI uses the node ecosystem to build its javascript UI.

The buildsystem that we use is called guanlecoja (https://www.npmjs.com/package/guanlecoja), which is just an integration of various javascript build tools.

Through npm dependency hell, guanlecoja is depending on *625* npm packages/versions. We do not advise you to try and package all those npm *build* dependencies. They are *not* required in order to *run* buildbot.

We do release pre-built packages in the form of the wheel (http://pythonwheels.com/) format on pypi. Those wheels contain the full python source code, and prebuilt javascript source code.

Depending on distro maintainers feedback, we *could* also release source tarballs with prebuilt javascript, but those would be pypi packages with different names, e.g. `buildbot_www_prebuilt.0.9.0.tar.gz`.

Another option would be to package a guanlecoja (https://www.npmjs.com/package/guanlecoja) that would embed all its dependencies inside one package.

Detailed requirements

see *Requirements*

Features

Buildbot-0.9.0 introduces the *Data API*, a consistent and scalable method for accessing and updating the state of the Buildbot system. This API replaces the existing, ill-defined Status API, which has been removed. Buildbot-0.9.0 introduces new *WWW Server* Interface using websocket for realtime updates. Buildbot code that interacted with the Status API (a substantial portion!) has been rewritten to use the Data API. Individual features and improvements to the Data API are not described on this page.

- Buildbot now supports plugins. They allow Buildbot to be extended by using components distributed independently from the main code. They also provide for a unified way to access all components. When previously the following construction was used:

```
from buildbot.kind.other.bits import ComponentClass

... ComponentClass ...
```

  the following construction achieves the same result:

```
from buildbot.plugins import kind

... kind.ComponentClass ...
```

  Kinds of components that are available this way are described in *Plugin Infrastructure in Buildbot*.

  Note: While the components can be still directly imported as `buildbot.kind.other.bits`, this might not be the case after Buildbot v1.0 is released.

- Both the P4 source step and P4 change source support ticket-based authentication.
- OpenStack latent slaves now support block devices as a bootable volume.
- Add new *Cppcheck* step.
- Add a new *Docker latent Workers*.
- Add a new configuration for creating custom services in out-of-tree CI systems or plugins. See *buildbot.util.service.BuildbotService*
- Add `try_ssh` configuration file setting and `--ssh` command line option for the try tool to specify the command to use for connecting to the build master.
- GitHub change hook now supports application/json format.
- Add support for dynamically adding steps during a build. See *Dynamic Build Factories*.
- *GitPoller* now supports detecting new branches
- *Git* supports an "origin" option to give a name to the remote repo.
- Mercurial hook was updated and modernized. It is no longer necessary to fork. One can now extend PYTHONPATH via the hook configuration. Among others, it permits to use a buildbot virtualenv instead of installing buildbot in all the system. Added documentation inside the hook. Misc. clean-up and reorganization in order to make the code a bit more readable.
- UI templates can now be customizable. You can provide html or jade overrides to the www plugins, to customize the UI
- The irc command `hello` now returns 'Hello' in a random language if invoked more than once.
- *Triggerable* now accepts a `reason` parameter.
- *GerritStatusPush* now accepts a `builders` parameter.
- *StatusPush* callback now receives build results (success/failure/etc) with the `buildFinished` event.
- There's a new renderable type, *Transform*.
- `GitPoller` now has a `buildPushesWithNoCommits` option to allow the rebuild of already known commits on new branches.
- Add GitLab authentication plugin for web UI. See *buildbot.www.oauth2.GitLabAuth*.
- *CMake* build step is added. It provides a convenience interface to CMake (https://cmake.org/cmake/help/latest/) build system.
- MySQL InnoDB tables are now supported.
- `HttpStatusPush` has been ported to reporter API.
- *StashStatusPush* has been ported to reporter API.
- `GithubStatusPush` has been ported to reporter API.
- *summaryCB* of *GerritStatusPush* now gets not only pre-processed information but the actual build as well.
- EC2LatentWorker supports VPCs, instance profiles, and advanced volume mounts.
- New steps for Visual Studio 2015 (VS2015, VC14, and MsBuild14).
- The *P4* step now obfuscates the password in status logs.
- Added support for specifying the depth of a shallow clone in *Git*.

- *OpenStackLatentWorker* now uses a single novaclient instance to not require re-authentication when starting or stopping instances.
- Buildbot UI introduces branch new Authentication, and Authorizations framework.

  Please look at their respective guide in *WWW Server*
- `buildbot stop` now waits for complete buildmaster stop by default.
- New `--no-wait` argument for `buildbot stop` which allows not to wait for complete master shutdown.
- New `LocalWorker` worker to run a worker in the master process, requires `buildbot-worker` package installed.
- *GerritStatusPush* now includes build properties in the `startCB` and `reviewCB` functions. `startCB` now must return a dictionary.
- add tool to send usage data to buildbot.net *buildbotNetUsageData*
- new *GitHub* which correctly checkout the magic branch like `refs/pull/xx/merge`.
- Enable parallel builds with Visual Studio and MSBuild.

Reporters

Status plugins have been moved into the `reporters` namespace. Their API has slightly to changed in order to adapt to the new data API. See respective documentation for details.

- *GerritStatusPush* renamed to `GerritStatusPush`
- `MailNotifier` renamed to *MailNotifier*
- `MailNotifier` argument `messageFormatter` should now be a `MessageFormatter`, due to removal of data api, custom message formaters need to be rewritten.
- `MailNotifier` argument `previousBuildGetter` is not supported anymore
- `Gerrit` supports specifying an SSH identity file explicitly.
- Added StashStatusPush status hook for Atlassian Stash
- *MailNotifier* no longer forces SSL 3.0 when `useTls` is true.
- *GerritStatusPush* callbacks slightly changed signature, and include a master reference instead of a status reference.
- new *GitLabStatusPush* to report builds results to GitLab.
- new *HipchatStatusPush* to report build results to Hipchat.

Fixes

- Buildbot is now compatible with SQLAlchemy 0.8 and higher, using the newly-released SQLAlchemy-Migrate.
- The version check for SQLAlchemy-Migrate was fixed to accept more version string formats.
- The *HTTPStep* step's requeset parameters are now renderable.
- With Git(), force the updating submodules to ensure local changes by the build are overwitten. This both ensures more consistent builds and avoids errors when updating submodules.
- Buildbot is now compatible with Gerrit v2.6 and higher.

  To make this happen, the return result of `reviewCB` and `summaryCB` callback has changed from

```
(message, verified, review)
```

to

```
{'message': message,
 'labels': {'label-name': value,
            ...
            }
}
```

The implications are:

- there are some differences in behaviour: only those labels that were provided will be updated
- Gerrit server must be able to provide a version, if it can't the *GerritStatusPush* will not work

---

Note: If you have an old style `reviewCB` and/or `summaryCB` implemented, these will still work, however there could be more labels updated than anticipated.

More detailed information is available in *GerritStatusPush* section.

- *P4Source*'s `server_tz` parameter now works correctly.
- The `revlink` in changes produced by the Bitbucket hook now correctly includes the `changes/` portion of the URL.
- *PBChangeSource*'s git hook master/contrib/git_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/git_buildbot.py) now supports git tags

  A pushed git tag generates a change event with the `branch` property equal to the tag name. To schedule builds based on buildbot tags, one could use something like this:

```
c['schedulers'].append(
   SingleBranchScheduler(name='tags',
      change_filter=filter.ChangeFilter(
         branch_re='v[0-9]+\.[0-9]+\.[0-9]+(?:-pre|rc[0-9]+|p[0-9]+)?')
      treeStableTimer=None,
      builderNames=['tag_build']))
```

- Missing "name" and "email" properties received from Gerrit are now handled properly
- Fixed bug which made it impossible to specify the project when using the BitBucket dialect.
- The *PyLint* step has been updated to understand newer output.
- Fixed SVN master-side source step: if a SVN operation fails, the repository end up in a situation when a manual intervention is required. Now if SVN reports such a situation during initial check, the checkout will be clobbered.
- The build properties are now stored in the database in the `build_properties` table.
- The list of changes in the build page now displays all the changes since the last sucessful build.
- GitHub change hook now correctly responds to ping events.
- GitHub change hook now correctly use the refs/pull/xx/merge branch for testing PRs.
- `buildbot.steps.http` steps now correctly have `url` parameter renderable
- When no arguments are used `buildbot checkconfig` now uses `buildbot.tac` to locate the master config file.

- *buildbot.util.flatten* now correctly flattens arbitrarily nested lists. *buildbot.util.flattened_iterator* provides an iterable over the collection which may be more efficient for extremely large lists.
- The *PyFlakes* and *PyLint* steps no longer parse output in Buildbot log headers (bug #3337 (http://trac.buildbot.net/ticket/3337)).
- *GerritChangeSource* is now less verbose by default, and has a `debug` option to enable the logs.
- *P4Source* no longer relies on the perforce server time to poll for new changes.
- The commit message for a change from *P4Source* now matches what the user typed in.
- Fix incompatibility with MySQL-5.7 (bug #3421 (http://trac.buildbot.net/ticket/3421))
- Fix incompatibility with postgresql driver psycopg2 (bug #3419 (http://trac.buildbot.net/ticket/3419), further regressions will be caught by travis)
- Made `Interpolate` safe for deepcopy or serialization/deserialization
- sqlite access is serialized in order to improve stability (bug #3565 (http://trac.buildbot.net/ticket/3565))

Deprecations, Removals, and Non-Compatible Changes

- Seamless upgrading between buildbot 0.8.12 and buildbot 0.9.0 is not supported. Users should start from a clean install but can reuse their config according to the *Upgrading to Nine* guide.
- *BonsaiPoller* is removed.
- `buildbot.ec2buildslave` is removed; use `buildbot.buildslave.ec2` instead.
- `buildbot.libvirtbuildslave` is removed; use `buildbot.buildslave.libvirt` instead.
- *buildbot.util.flatten* flattens lists and tuples by default (previously only lists). Additionally, flattening something that isn't the type to flatten has different behaviour. Previously, it would return the original value. Instead, it now returns an array with the original value as the sole element.
- `buildbot.tac` does not support `print` statements anymore. Such files should now use `print` as a function instead (see https://docs.python.org/3.0/whatsnew/3.0.html#print-is-a-function for more details). Note that this applies to both python2.x and python3.x runtimes.
- Deprecated `workdir` property has been removed, `builddir` property should be used instead.
- To support MySQL InnoDB, the size of six VARCHAR(256) columns `changes.(author, branch, category, name); object_state.name; user.identifier` was reduced to VARCHAR(255).
- **`StatusPush` has been removed from buildbot.** Please use the much simpler `HttpStatusPush` instead.
- Worker changes described in below worker section will probably impact a buildbot developer who uses undocumented '*slave*' API. Undocumented APIs have been replaced without failover, so any custom code that uses it shall be updated with new undocumented API.
- Web server does not provide /png and /redirect anymore (bug #3357 (http://trac.buildbot.net/ticket/3357)). This functionality is used to implement build status images. This should be easy to implement if you need it. One could port the old image generation code, or implement a redirection to http://shields.io/.
- Support of worker-side `usePTY` was removed from `buildbot-worker`. `usePTY` argument was removed from `WorkerForBuilder` and `Worker` classes.
- html is no longer permitted in 'label' attributes of forcescheduler parameters.
- `public_html` directory is not created anymore in `buildbot create-master` (it's not used for some time already). Documentation was updated with suggestions to use third party web server for serving static file.

- `usePTY` default value has been changed from `slave-config` to `None` (use of `slave-config` will still work).
- `/json` web status was removed. *Data API* should be used instead.

WebStatus

The old, clunky WebStatus has been removed. You will like the new interface! RIP WebStatus, you were a good friend.

remove it and replace it with *`www configuration`*.

Requirements

- Support for python 2.6 was dropped from the master.
- Buildbot's tests now require at least Mock-0.8.0.
- SQLAlchemy-Migrate-0.6.1 is no longer supported.
- Builder names are now restricted to unicode strings or ASCII bytestrings. Encoded bytestrings are not accepted.

Steps

- New-style steps are now the norm, and support for old-style steps is deprecated. Upgrade your steps to new-style now, as support for old-style steps will be dropped after Buildbot-0.9.0. See *New-Style Build Steps* for details.
  - Status strings for old-style steps could be supplied through a wide variety of conflicting means (`describe`, `description`, `descriptionDone`, `descriptionSuffix`, `getText`, and `setText`, to name just a few). While all attempts have been made to maintain compatibility, you may find that the status strings for old-style steps have changed in this version. To fix steps that call `setText`, try setting the `descriptionDone` attribute directly, instead – or just rewrite the step in the new style.
- Old-style *source* steps (imported directly from `buildbot.steps.source`) are no longer supported on the master.
- The monotone source step got an overhaul and can now better manage its database (initialize and/or migrate it, if needed). In the spirit of monotone, buildbot now always keeps the database around, as it's an append-only database.

Changes and Removals

- Buildslave names must now be 50-character *identifier*. Note that this disallows some common characters in bulidslave names, including spaces, `/`, and `..`
- Builders now have "tags" instead of a category. Builders can have multiple tags, allowing more flexible builder displays.
- *`ForceScheduler`* has the following changes:
  - The default configuration no longer contains four `AnyPropertyParameter` instances.
  - Configuring `codebases` is now mandatory, and the deprecated `branch`, `repository`, `project`, `revision` are not supported anymore in *`ForceScheduler`*
  - `buildbot.schedulers.forcesched.BaseParameter.updateFromKwargs` now takes a `collector` parameter used to collect all validation errors

- *Periodic*, *Nightly* and *NightlyTriggerable* have the following changes:
  - The *Periodic* and *Nightly* schedulers can now consume changes and use `onlyIfChanged` and `createAbsoluteTimestamps`.
  - All "timed" schedulers now handle `codebases` the same way. Configuring `codebases` is strongly recommended. Using the `branch` parameter is discouraged.
- Logs are now stored as Unicode strings, and thus must be decoded properly from the bytestrings provided by shell commands. By default this encoding is assumed to be UTF-8, but the *logEncoding* parameter can be used to select an alternative. Steps and individual logfiles can also override the global default.
- The PB status service uses classes which have now been removed, and anyway is redundant to the REST API, so it has been removed. It has taken the following with it:
  - `buildbot statuslog`
  - `buildbot statusgui` (the GTK client)
  - `buildbot debugclient`

  The `PBListener` status listener is now deprecated and does nothing. Accordingly, there is no external access to status objects via Perspective Broker, aside from some compatibility code for the try scheduler.

  The `debugPassword` configuration option is no longer needed and is thus deprecated.
- The undocumented and un-tested `TinderboxMailNotifier`, designed to send emails suitable for the abandoned and insecure Tinderbox tool, has been removed.
- Buildslave info is no longer available via *Interpolate* and the `SetSlaveInfo` buildstep has been removed.
- The undocumented `path` parameter of the *MasterShellCommand* buildstep has been renamed `workdir` for better consistency with the other steps.
- The name and source of a Property have to be unicode or ascii string.
- Property values must be serializable in JSON.
- *IRC* has the following changes:
  - categories parameter is deprecated and removed. It should be replaced with tags=[cat]
  - noticeOnChannel parameter is deprecated and removed.
- workdir behavior has been unified:
  - `workdir` attribute of steps is now a property in *BuildStep*, and choose the workdir given following priority:
    - workdir of the step, if defined
    - workdir of the builder (itself defaults to 'build')
    - setDefaultWorkdir() has been deprecated, but is now behaving the same for all the steps: Setting self.workdir if not already set
- *Trigger* now has a `getSchedulersAndProperties` method that can ve overridden to support dynamic triggering.
- `master.cfg is now parsed from a thread. Previously it was run in the main thread, and thus slowing down the master in case of big config, or network access done to generate the config.
- *SVNPoller*'s svnurl parameter has been changed to repourl.
- Providing Latent AWS EC2 credentials by the `.ec2/aws_id` file is deprecated: Use the standard `.aws/credentials` file, instead.

Changes for Developers

- Botmaster no longer service parent for workers. Service parent functionality has been transferred to WorkerManager. It should be noted Botmaster no longer has a `slaves` field as it was moved to WorkerManager.
- The sourcestamp DB connector now returns a `patchid` field.
- Buildbot no longer polls the database for jobs. The `db_poll_interval` configuration parameter and the *db* key of the same name are deprecated and will be ignored.
- The interface for adding changes has changed. The new method is `master.data.updates.addChange` (implemented by `addChange`), although the old interface (`master.addChange`) will remain in place for a few versions. The new method:
  - returns a change ID, not a Change instance;
  - takes its `when_timestamp` argument as epoch time (UNIX time), not a datetime instance; and
  - does not accept the deprecated parameters `who`, `isdir`, `is_dir`, and `when`.
  - requires that all strings be unicode, not bytestrings.

  Please adjust any custom change sources accordingly.
- A new build status, CANCELLED, has been added. It is used when a step or build is deliberately cancelled by a user.
- This upgrade will delete all rows from the `buildrequest_claims` table. If you are using this table for analytical purposes outside of Buildbot, please back up its contents before the upgrade, and restore it afterward, translating object IDs to scheduler IDs if necessary. This translation would be very slow and is not required for most users, so it is not done automatically.
- All of the schedulers DB API methods now accept a schedulerid, rather than an objectid. If you have custom code using these methods, check your code and make the necessary adjustments.
- The `addBuildsetForSourceStamp` method has become `addBuildsetForSourceStamps`, and its signature has changed. The `addBuildsetForSourceStampSetDetails` method has become `addBuildsetForSourceStampsWithDefaults`, and its signature has changed. The `addBuildsetForSourceStampDetails` method has been removed. The `addBuildsetForLatest` method has been removed. It is equivalent to `addBuildsetForSourceStampDetails` with `sourcestamps=None`. These methods are not yet documented, and their interface is not stable. Consult the source code for details on the changes.
- The `preStartConsumingChanges` and `startTimedSchedulerService` hooks have been removed.
- The triggerable schedulers `trigger` method now requires a list of sourcestamps, rather than a dictionary.
- The `SourceStamp` class is no longer used. It remains in the codebase to support loading data from pickles on upgrade, but should not be used in running code.
- The `BuildRequest` class no longer has full `source` or `sources` attributes. Use the data API to get this information (which is associated with the buildset, not the build request) instead.
- The undocumented `BuilderControl` method `submitBuildRequest` has been removed.
- The debug client no longer supports requesting builds (the `requestBuild` method has been removed). If you have been using this method in production, consider instead creating a new change source, using the *ForceScheduler*, or using one of the try schedulers.
- The `buildbot.misc.SerializedInvocation` class has been removed; use *buildbot.util.debounce.method* instead.

- The `progress` attributes of both *`buildbot.process.buildstep.BuildStep`* and *`buildbot.process.build.Build`* have been removed. Subclasses should only be accessing the progress-tracking mechanics via the *`buildbot.process.buildstep.BuildStep.setProgress`* method.
- The *`BuilderConfig`* `nextSlave` keyword argument takes a callable. This callable now receives `BuildRequest` instance in its signature as 3rd parameter. **For retro-compatibility, all callable taking only 2 parameters will still work**.
- properties object is now directly present in build, and not in build_status. This should not change much unless you try to access your properties via step.build.build_status. Remember that with PropertiesMixin, you can access properties via getProperties on the steps, and on the builds objects.

Slaves/Workers

Transition to "worker" terminology

Since version 0.9.0 of Buildbot "slave"-based terminology is deprecated in favor of "worker"-based terminology.

For details about public API changes see *Transition to "worker" terminology*, and *Release Notes for Buildbot 0.9.0b8* release notes.

- The `buildbot-slave` package has been renamed to `buildbot-worker`.
- Buildbot now requires import to be sorted using isort (https://isort.readthedocs.io/en/stable/). Please run `make isort` before creating a PR or use any available editor plugin in order to reorder your imports.

Requirements

- `buildbot-worker` requires Python 2.6

Features

- The Buildbot worker now includes the number of CPUs in the information it supplies to the master on connection. This value is autodetected, but can be overridden with the `--numcpus` argument to `buildslave create-worker`.
- The `DockerLatentWorker` image attribute is now renderable (can take properties in account).
- The `DockerLatentWorker` sets environment variables describing how to connect to the master. Example dockerfiles can be found in master/contrib/docker (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/docker).
- `DockerLatentWorker` now has a `hostconfig` parameter that can be used to setup host configuration when creating a new container.
- `DockerLatentWorker` now has a `networking_config` parameter that can be used to setup container networks.
- The `DockerLatentWorker` `volumes` attribute is now renderable.

Fixes

Changes for Developers

- EC2 Latent Worker upgraded from `boto2` to `boto3`.

Deprecations, Removals, and Non-Compatible Changes

- buildmaster and worker no longer supports old-style source steps.
- On Windows, if a *ShellCommand* step in which `command` was specified as a list is executed, and a list element is a string consisting of a single pipe character, it no longer creates a pipeline. Instead, the pipe character is passed verbatim as an argument to the program, like any other string. This makes command handling consistent between Windows and Unix-like systems. To have a pipeline, specify `command` as a string.
- Support for python 2.6 was dropped from the master.
- `public_html` directory is not created anymore in `buildbot create-master` (it's not used for some time already). Documentation was updated with suggestions to use third party web server for serving static file.
- `usePTY` default value has been changed from `slave-config` to `None` (use of `slave-config` will still work).
- `GithubStatusPush` reporter was renamed to *GitHubStatusPush*.
- Worker commands version bumped to 3.0.
- Master/worker protocol has been changed:
  - `slave_commands` key in worker information was renamed to `worker_commands`.
  - `getSlaveInfo` remote method was renamed to `getWorkerInfo`.
  - `slave-config` value of `usePTY` is not supported anymore.
  - `slavesrc` command argument was renamed to `workersrc` in `uploadFile` and `uploadDirectory` commands.
  - `slavedest` command argument was renamed to `workerdest` in `downloadFile` command.
  - Previously deprecated `WorkerForBuilder.remote_shutdown()` remote command has been removed.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.8.12..v0.9.0
```

Release Notes for Buildbot `0.9.0rc4`

The following are the release notes for Buildbot `0.9.0rc4`. This version was released on September 28, 2016.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Fixes

- Fix the UI to better adapt to different screen width (bug #3614 (http://trac.buildbot.net/ticket/3614))
- Add more REST api documentation (document `/raw` endpoints, and `POST` actions)

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0rc3..v0.9.0rc4
```

Release Notes for Buildbot `0.9.0rc3`

The following are the release notes for Buildbot `0.9.0rc3`. This version was released on September 14, 2016.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Features

- add tool to send usage data to buildbot.net *`buildbotNetUsageData`*

Fixes

- Publish python module buildbot.buildslave in the dist files
- Upgrade to guanlecoja 0.7 (for compatibility with node6)
- Fix invocation of trial on windows, with twisted 16+
- Fix rare issue which makes buildbot throw a exception when there is a sourcestamp with no change for a particular codebase.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0rc2..v0.9.0rc3
```

Release Notes for Buildbot `0.9.0rc2`

The following are the release notes for Buildbot `0.9.0rc2`. This version was released on August 23, 2016.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Features

- add a UI button to allow to cancel the whole queue for a builder

Fixes

- fix the UI to allow to cancel a buildrequest (bug #3582 (http://trac.buildbot.net/ticket/3582))
- Fix BitbucketPullrequestPoller change detection
- Fix customization for template_type in email reporter
- fix DockerLatent integration of volumes mounting
- misc doc fixes
- fix buildbot not booting when builder tags contains duplicates
- `forcesched`: fix owner parameter when no authentication is used
- REST: fix problem with twisted 16 error reporting
- CORS: format errors according to API type
- Dockerfiles fix and upgrade Ubuntu to 16.04
- Fixes #3430 Increased size of builder identifier from 20 to 50 (brings it in line to size of steps and workers in same module).
- Fix missing VS2015 entry_points
- removed the restriction on twisted < 16.3.0 now that autobahn 0.16.0 fixed the issue

Changes for Developers

Features

Fixes

Deprecations, Removals, and Non-Compatible Changes

- remove repo from worker code (obsoleted by repo master source step)

Worker

Fixes

Changes for Developers

Deprecations, Removals, and Non-Compatible Changes

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0rc1..v0.9.0rc2
```

Release Notes for Buildbot 0.9.0rc1

The following are the release notes for Buildbot 0.9.0rc1.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Features

- new *HipchatStatusPush* to report build results to Hipchat.
- new steps for Visual Studio 2015 (VS2015, VC14, and MsBuild14).
- The *P4* step now obfuscates the password in status logs.
- Added support for specifying the depth of a shallow clone in *Git*.
- *OpenStackLatentWorker* now uses a single novaclient instance to not require re-authentication when starting or stopping instances.
- The `dist` parameter in *RpmBuild* is now renderable.
- new *BitbucketStatusPush* to report build results to a Bitbucket Cloud repository.

Fixes

- *GerritStatusPush* now includes build properties in the `startCB` and `reviewCB` functions. `startCB` now must return a dictionary.
- Fix TypeError exception with `HgPoller` if `usetimestamps=False` is used (bug #3562 (http://trac.buildbot.net/ticket/3562))
- Fix recovery upon master unclean kill or crash (bug #3564 (http://trac.buildbot.net/ticket/3564))
- sqlite access is serialized in order to improve stability (bug #3565 (http://trac.buildbot.net/ticket/3565))
- Docker latent worker has been fixed (bug #3571 (http://trac.buildbot.net/ticket/3571))

Changes for Developers

Features

Fixes

Deprecations, Removals, and Non-Compatible Changes

- Support for python 2.6 was dropped from the master.
- `public_html` directory is not created anymore in `buildbot create-master` (it's not used for some time already). Documentation was updated with suggestions to use third party web server for serving static file.
- `usePTY` default value has been changed from `slave-config` to `None` (use of `slave-config` will still work).
- `GithubStatusPush` reporter was renamed to *GitHubStatusPush*.

Worker

Deprecations, Removals, and Non-Compatible Changes

- The `buildbot-slave` package has finished being renamed to `buildbot-worker`.

Worker

Fixes

- `runGlob()` uses the correct remote protocol for both `CommandMixin` and `ComposititeStepMixin`.
- Rename `glob()` to `runGlob()` in `CommandMixin`

Changes for Developers

- EC2 Latent Worker upgraded from `boto2` to `boto3`.

Deprecations, Removals, and Non-Compatible Changes

- Worker commands version bumped to 3.0.
- Master/worker protocol has been changed:
  - `slave_commands` key in worker information was renamed to `worker_commands`.
  - `getSlaveInfo` remote method was renamed to `getWorkerInfo`.
  - `slave-config` value of `usePTY` is not supported anymore.
  - `slavesrc` command argument was renamed to `workersrc` in `uploadFile` and `uploadDirectory` commands.
  - `slavedest` command argument was renamed to `workerdest` in `downloadFile` command.
  - Previously deprecated `WorkerForBuilder.remote_shutdown()` remote command has been removed.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0b9..v0.9.0rc1
```

Note that Buildbot-0.8.11 was never released.

Release Notes for Buildbot 0.9.0b9

The following are the release notes for Buildbot 0.9.0b9 This version was released on May 10, 2016.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Features

- new *GitLabStatusPush* to report builds results to GitLab.
- `buildbot stop` now waits for complete buildmaster stop by default.
- New `--no-wait` argument for `buildbot stop` which allows not to wait for complete master shutdown.
- Builder page is now sorted by builder name
- LogViewer page now supports ANSI color codes, and is displayed white on black.

Changes for Developers

- Speed improvements for integration tests by use of SynchronousTestCase, and in-memory sqlite.
- Buildbot now requires import to be sorted using isort (https://isort.readthedocs.io/en/stable/). Please run `make isort` before creating a PR or use any available editor plugin in order to reorder your imports.

Fixes

- OpenStackLatentWorker uses the novaclient API correctly now.
- The *MsBuild4* and *MsBuild12* steps work again (bug #2878 (http://trac.buildbot.net/ticket/2878)).
- Scheduler changes are now identified by serviceid instead of objectid (bug #3532 (http://trac.buildbot.net/ticket/3532))
- Make groups optional in LdapUserInfo (bug #3511 (http://trac.buildbot.net/ticket/3511))
- Buildbot nine do not write pickles anymore in the master directory
- Fix build page to not display build urls, but rather directly the build-summary, which already contain the URL.
- UI Automatically reconnect on disconnection from the websocket. (bug #3462 (http://trac.buildbot.net/ticket/3462))

Deprecations, Removals, and Non-Compatible Changes

- The buildmaster now requires at least Twisted-14.0.1.
- The web ui has upgrade its web components dependencies to latest versions (https://github.com/buildbot/guanlecoja-ui/tree/master#changelog). This can impact web-ui plugin.
- Web server does not provide /png and /redirect anymore (bug #3357 (http://trac.buildbot.net/ticket/3357)). This functionality is used to implement build status images. This should be easy to implement if you need it. One could port the old image generation code, or implement a redirection to http://shields.io/.
- Support of worker-side `usePTY` was removed from `buildbot-worker`. `usePTY` argument was removed from `WorkerForBuilder` and `Worker` classes.
- html is no longer permitted in 'label' attributes of forcescheduler parameters.
- `LocalWorker` now requires `buildbot-worker` package, instead of `buildbot-slave`.
- *Collapse Request Functions* now takes master as first argument. The previous callable contained too few data in order to be really usable. As collapseRequests has never been released outside of beta, backward compatibility with previous release has **not** been implemented.

- This is the last version of buildbot nine which supports python 2.6 for the master. Next version will drop python 2.6 support.

Worker

Fixes

- `buildbot-worker` script now outputs message to terminal.
- Windows helper script now called `buildbot-worker.bat` (was `buildbot_worker.bat`, notice underscore), so that `buildbot-worker` command can be used in virtualenv both on Windows and POSIX systems.

Changes for Developers

- `SLAVEPASS` environment variable is not removed in default-generated `buildbot.tac`. Environment variables are cleared in places where they are used (e.g. in Docker Latent Worker contrib scripts).
- Master-part handling has been removed from `buildbot-worker` log watcher (bug #3482 (http://trac.buildbot.net/ticket/3482)).
- `WorkerDetectedError` exception type has been removed.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0b8..v0.9.0b9
```

Release Notes for Buildbot 0.9.0b8

The following are the release notes for Buildbot 0.9.0b8 This version was released on April 11, 2016.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Features

- `GitPoller` now has a `buildPushesWithNoCommits` option to allow the rebuild of already known commits on new branches.
- Add GitLab authentication plugin for web UI. See *`buildbot.www.oauth2.GitLabAuth`*.
- `DockerLatentWorker` now has a `hostconfig` parameter that can be used to setup host configuration when creating a new container.
- `DockerLatentWorker` now has a `networking_config` parameter that can be used to setup container networks.
- The `DockerLatentWorker` `volumes` attribute is now renderable.
- *`CMake`* build step is added. It provides a convenience interface to CMake (https://cmake.org/cmake/help/latest/) build system.

- MySQL InnoDB tables are now supported.
- `HttpStatusPush` has been ported to reporter API.
- *`StashStatusPush`* has been ported to reporter API.
- `GithubStatusPush` has been ported to reporter API.
- *summaryCB* of *`GerritStatusPush`* now gets not only pre-processed information but the actual build as well.
- EC2LatentWorker supports VPCs, instance profiles, and advanced volume mounts.

Fixes

- Fix loading *`LdapUserInfo`* plugin and its documentation (bug #3371 (http://trac.buildbot.net/ticket/3371)).
- Fix deprecation warnings seen with docker-py >= 1.4 when passing arguments to `docker_client.start()`.
- `GitHubEventHandler` now uses the `['repository']['html_url']` key in the webhook payload to populate `repository`, as the previously used `['url']` and `['clone_url']` keys had a different format between push and pull requests and GitHub and GitHub Enterprise instances.
- Fix race condition where log compression could lead to empty log results in reporter api
- Error while applying db upgrade is now properly reported in the buildbot upgrade-master command line.
- Made `Interpolate` safe for deepcopy or serialization/deserialization
- Optimized UI REST requests for child builds and change page.
- Fix `DockerLatentWorker` use of *volume* parameter, they now propely manage *src:dest* syntax.
- Fix `DockerLatentWorker` to properly create properties so that docker parameters can be renderable.
- Lock down autobahn version for python 2.6 (note that autobahn and twisted are no longer supporting 2.6, and thus do not receive security fixes anymore).
- Fix docs and example to always use port 8020 for the web ui.

Deprecations, Removals, and Non-Compatible Changes

- Deprecated `workdir` property has been removed, `builddir` property should be used instead.
- To support MySQL InnoDB, the size of six VARCHAR(256) columns `changes.(author, branch, category, name); object_state.name; user.identifier` was reduced to VARCHAR(255).
- **`StatusPush` has been removed from buildbot.** Please use the much simpler `HttpStatusPush` instead.

Changes for Developers

Worker changes described in below worker section will probably impact a buildbot developer who uses undocumented '*slave*' API. Undocumented APIs have been replaced without failover, so any custom code that uses it shall be updated with new undocumented API.

Worker

Package *buildbot-slave* is being renamed *buildbot-worker*. As the work is not completely finished, neither *buildbot-slave==0.9.0b8* or *buildbot-worker==0.9.0b8* have been released.

You can safely use any version of *buildbot-slave* with *buildbot==0.9.0b8*, either *buildbot-slave==0.8.12* or *buildbot-slave==0.9.0b7*.

Transition to "worker" terminology

Since version 0.9.0 of Buildbot "slave"-based terminology is deprecated in favor of "worker"-based terminology.

For details about public API changes see *Transition to "worker" terminology*.

API changes done without providing fallback:

Old name	Ne
`buildbot.buildslave.manager`	*bu*
`buildbot.buildslave.manager.BuildslaveRegistration`	*bu*
`buildbot.buildslave.manager.BuildslaveRegistration.buildslave`	bu
`buildbot.buildslave.manager.BuildslaveManager`	*bu*
`buildbot.buildslave.manager.BuildslaveManager.slaves`	bu
`buildbot.buildslave.manager.BuildslaveManager.getBuildslaveByName`	bu
`buildbot.buildslave.docker.DockerLatentBuildSlave`	*bu*
`buildbot.buildslave.local.LocalBuildSlave`	bu
`buildbot.buildslave.local.LocalBuildSlave.LocalBuildSlaveFactory`	bu
`buildbot.buildslave.local.LocalBuildSlave.remote_slave`	bu
`buildbot.buildslave.base` module with all contents	bu
`buildbot.buildslave.AbstractBuildSlave.updateSlave`	bu
`buildbot.buildslave.AbstractBuildSlave.slavebuilders`	bu
`buildbot.buildslave.AbstractBuildSlave.updateSlaveStatus`	bu
`buildbot.buildslave.AbstractLatentBuildSlave.updateSlave`	bu
`buildbot.buildslave.BuildSlave.slave_status`	bu
`buildbot.config.MasterConfig.load_slaves`	lo
`buildbot.master.BuildMaster.buildslaves`	bu
`buildbot.process.build.Build.slavebuilder`	wo
`buildbot.process.build.Build.setSlaveEnvironment`	se
`buildbot.process.build.Build.slaveEnvironment`	wo
`buildbot.process.build.Build.getSlaveCommandVersion`	ge
`buildbot.process.build.Build.setupSlaveBuilder`	se
`buildbot.process.builder.Build.canStartWithSlavebuilder`	ca
`buildbot.process.slavebuilder.AbstractSlaveBuilder.getSlaveCommandVersion`	bu
`buildbot.process.slavebuilder.AbstractSlaveBuilder.attached` method argument `slave` was renamed	wo
`buildbot.buildslave.AbstractBuildSlave.slave_commands`	bu
`buildbot.buildslave.AbstractBuildSlave.slave_environ`	bu
`buildbot.buildslave.AbstractBuildSlave.slave_basedir`	bu
`buildbot.buildslave.AbstractBuildSlave.slave_system`	bu
`buildbot.buildslave.AbstractBuildSlave.buildslaveid`	bu
`buildbot.buildslave.AbstractBuildSlave.addSlaveBuilder`	bu
`buildbot.buildslave.AbstractBuildSlave.removeSlaveBuilder`	bu
`buildbot.buildslave.AbstractBuildSlave.messageReceivedFromSlave`	bu
`buildbot.process.slavebuilder.LatentSlaveBuilder` constructor positional argument `slave` was renamed	wo

Table 5.1 – continued f

Old name	Ne
`buildbot.process.buildrequestdistributor.BasicBuildChooser.nextSlave`	ne
`buildbot.process.buildrequestdistributor.BasicBuildChooser.slavepool`	wo
`buildbot.process.buildrequestdistributor.BasicBuildChooser.preferredSlaves`	pr
`buildbot.process.buildrequestdistributor.BasicBuildChooser.rejectedSlaves`	re
`buildbot.steps.shell.ShellCommand.slaveEnvironment` (Note: this variable is renderable)	bu
`buildbot.status.slave`	bu
`buildbot.status.slave.SlaveStatus`	bu
`buildbot.interfaces.IStatusReceiver.slaveConnected` with all implementations	bu
`buildbot.interfaces.IStatusReceiver.slaveDisconnected` with all implementations	bu
`buildbot.status.master.Status.slaveConnected`	bu
`buildbot.status.master.Status.slaveDisconnected`	bu
`buildbot.status.master.Status.slavePaused`	bu
`buildbot.status.master.Status.slaveUnpaused`	bu
`buildbot.status.master.Status.buildslaves`	bu
`buildbot.status.base.StatusReceiverBase.slavePaused`	bu
`buildbot.status.base.StatusReceiverBase.slaveUnpaused`	bu
`buildbot.interfaces.IStatus.getSlaveNames` with all implementations	bu
`buildbot.interfaces.IStatus.getSlave` with all implementations	bu
`buildbot.interfaces.IBuildStatus.getSlavename` with all implementations	bu
`buildbot.status.build.BuildStatus.setSlavename`	bu
`buildbot.status.build.BuildStatus.slavename`	bu
`buildbot.interfaces.IBuilderStatus.getSlaves` with all implementations	bu
`buildbot.status.builder.BuilderStatus.slavenames`	bu
`buildbot.status.builder.BuilderStatus.setSlavenames`	bu
`buildbot.process.botmaster.BotMaster.slaveLost`	bu
`buildbot.process.botmaster.BotMaster.getBuildersForSlave`	bu
`buildbot.process.botmaster.BotMaster.maybeStartBuildsForSlave`	bu
`buildbot.locks.RealSlaveLock`	bu
`buildbot.locks.RealSlaveLock.maxCountForSlave`	bu
`buildbot.protocols.base.Connection` constructor positional argument `buildslave` was renamed	wo
`buildbot.protocols.base.Connection.buidslave`	bu
`buildbot.protocols.base.Connection.remoteGetSlaveInfo`	bu
`buildbot.protocols.pb.Connection` constructor positional argument `buildslave` was renamed	wo

Other changes done without providing fallback:

- Functions argument `buildslaveName` renamed to `workerName`.
- Loop variables, local variables, helper functions:

Old name	New name
`s`	`w` or `worker`
`sl`	`w` or `worker`
`bs` ("buildslave")	`w`
`sb`	`wfb` ("worker for builder")
`bs1()`, `bs2()`	`w1()`, `w2()`
`bslave`	`worker`
`BS1_NAME`, `BS1_ID`, `BS1_INFO`	`W1_NAME`, `W1_ID`, `W1_INFO`

- In `buildbot.config.BuilderConfig.getConfigDict` result `'slavenames'` key changed to `'workernames'`; `'slavebuilddir'` key changed to `'workerbuilddir'`; `'nextSlave'` key changed to `'nextWorker'`.

- `buildbot.process.builder.BuilderControl.ping` now generates `["ping", "no worker"]` event, instead of `["ping", "no slave"]`.
- `buildbot.plugins.util.WorkerChoiceParameter` (previously `BuildslaveChoiceParameter`) label was changed from `Build slave` to `Worker`.
- `buildbot.plugins.util.WorkerChoiceParameter` (previously `BuildslaveChoiceParameter`) default name was changed from `slavename` to `workername`.
- `buildbot.status.builder.SlaveStatus` fallback was removed. `SlaveStatus` was moved to `buildbot.status.builder.slave` previously, and now it's `buildbot.status.worker.WorkerStatus`.
- `buildbot.status.status_push.StatusPush` events generation changed (this module will be completely removed in 0.9.x):
  - instead of `slaveConnected` with data `slave=...` now generated `workerConnected` event with data `worker=...`;
  - instead of `slaveDisconnected` with data `slavename=...` now generated `workerDisconnected` with data `workername=...`;
  - instead of `slavePaused` with data `slavename=...` now generated `workerPaused` event with data `workername=...`;
  - instead of `slaveUnpaused` with data `slavename=...` now generated `workerUnpaused` event with data `workername=...`;
- `buildbot.status.build.BuildStatus.asDict` returns worker name under `'worker'` key, instead of `'slave'` key.
- `buildbot.status.builder.BuilderStatus.asDict` returns worker names under `'workers'` key, instead of `'slaves'` key.
- Definitely privately used "slave"-named variables and attributes were renamed, including tests modules, classes and methods.

Database

Database API changes done without providing fallback.

Old name	New name
buildbot.db.buildslaves.BuildslavesConnectorComponent.getBuildslaves (rewritten in nine) and buildbot.db.buildslaves.BuildslavesConnectorComponent.getBuildslave (introduced in nine) results uses instead of 'slaveinfo' key	'workerinfo' key
buildbot.db.model.Model.buildslaves	buildbot.db.model.Model.workers
buildbot.db.model.Model.configured_buildslaves	buildbot.db.model.Model.configured_workers
buildbot.db.model.Model.connected_buildslaves	buildbot.db.model.Model.connected_workers
buildbot.db.buildslaves.BuildslavesConnectorComponent.findBuildslaveId (introduced in nine)	*buildbot.db.workers.WorkersConnectorComponent.findWorkerId*
buildbot.db.buildslaves.BuildslavesConnectorComponent.deconfigureAllBuidslavesForMaster (introduced in nine, note typo Buidslaves)	*buildbot.db.workers.WorkersConnectorComponent.deconfigureAllWorkersForMaster*
buildbot.db.buildslaves.BuildslavesConnectorComponent.buildslaveConfigured (introduced in nine)	*buildbot.db.workers.WorkersConnectorComponent.workerConfigured*
buildbot.db.buildslaves.BuildslavesConnectorComponent.buildslaveConfigured method argument buildslaveid was renamed (introduced in nine)	workerid
buildbot.db.buildslaves.BuildslavesConnectorComponent.getBuildslave	*buildbot.db.workers.WorkersConnectorComponent.getWorker*
buildbot.db.buildslaves.BuildslavesConnectorComponent.getBuildslaves method argument _buildslaveid was renamed (introduced in nine)	_workerid
buildbot.db.buildslaves.BuildslavesConnectorComponent.buildslaveConnected (introduced in nine)	*buildbot.db.workers.WorkersConnectorComponent.workerConnected*
buildbot.db.buildslaves.BuildslavesConnectorComponent.buildslaveConnected method argument slaveinfo was renamed (introduced in nine)	workerinfo
buildbot.db.buildslaves.BuildslavesConnectorComponent.buildslaveConnected method argument buildslaveid was renamed (introduced in nine)	workerid
buildbot.db.buildslaves.BuildslavesConnectorComponent.buildslaveDisconnected (introduced in nine)	*buildbot.db.workers.WorkersConnectorComponent.workerDisconnected*
buildbot.db.buildslaves.BuildslavesConnectorComponent.buildslaveDisconnected method argument buildslaveid was renamed (introduced in nine)	workerid
buildbot.db.builds.BuildsConnectorComponent.getBuilds method argument buildslaveid was renamed (introduced in nine)	workerid
buildbot.db.builds.BuildsConnectorComponent.addBuild method argument buildslaveid was renamed	workerid

Data API

Python API changes:

Old name	New name
`buildbot.data.buildslaves`	`workers`
`buildbot.data.buildslaves.BuildslaveEndpoint`	`WorkerEndpoint`
`buildbot.data.buildslaves.BuildslavesEndpoint`	`WorkersEndpoint`
`buildbot.data.buildslaves.Buildslave`	`Worker`
`buildbot.data.buildslaves.Buildslave.buildslaveConfigured`	`workerConfigured`
`buildbot.data.buildslaves.Buildslave.findBuildslaveId`	`findWorkerId`
`buildbot.data.buildslaves.Buildslave.buildslaveConnected`	`workerConnected`
`buildbot.data.buildslaves.Buildslave.buildslaveDisconnected`	`workerDisconnected`
`buildbot.data.buildslaves.Buildslave.deconfigureAllBuidslavesForMaster`	`deconfigureAllWorkersForMaster`
`buildslaveid` in function arguments and API specification	`workerid`
`slaveinfo` in function arguments and API specification	`workerinfo`

Changed REST endpoints:

Old name	New name
`/buildslaves`	`/workers`
`/buildslaves/n:buildslaveid`	`/workers/n:workerid`
`/buildslaves/n:buildslaveid/builds`	`/workers/n:workerid/builds`
`/buildslaves/:buildslaveid/builds/:buildid`	`/workers/:workerid/builds/:buildid`
`/masters/n:masterid/buildslaves`	`/masters/n:masterid/workers`
`/masters/n:masterid/buildslaves/n:buildslaveid`	`/masters/n:masterid/workers/n:workerid`
`/masters/n:masterid/builders/n:builderid/buildslaves`	`/masters/n:masterid/builders/n:builderid/workers`
`/masters/n:masterid/builders/n:builderid/buildslaves/n:buildslaveid`	`/masters/n:masterid/builders/n:builderid/workers/n:workerid`
`/builders/n:builderid/buildslaves`	`/builders/n:builderid/workers`
`/builders/n:builderid/buildslaves/n:buildslaveid`	`/builders/n:builderid/workers/n:workerid`

Changed REST object keys:

Old name	New name
`buildslaveid`	`workerid`
`slaveinfo`	`workerinfo`
`buildslave`	`worker`
`buildslaves`	`workers`

`data_module` version bumped from `1.2.0` to `2.0.0`.

Web UI

In base web UI (`www/base`) and Material Design web UI (`www/md_base`) all "slave"-named messages and identifiers were renamed to use "worker" names and new REST API endpoints.

MQ layer

`buildslaveid` key in messages were replaced with `workerid`.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0b7..v0.9.0b8
```

Release Notes for Buildbot 0.9.0b7

The following are the release notes for Buildbot 0.9.0b7 This version was released on February 14, 2016.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Features

Fixes

- Fix incompatibility with MySQL-5.7 (bug #3421 (http://trac.buildbot.net/ticket/3421))
- Fix incompatibility with postgresql driver psycopg2 (bug #3419 (http://trac.buildbot.net/ticket/3419), further regressions will be caught by travis)
- Fix regressions in forcescheduler UI (bug #3416 (http://trac.buildbot.net/ticket/3416), bug #3418 (http://trac.buildbot.net/ticket/3418), bug #3422 (http://trac.buildbot.net/ticket/3422))

Deprecations, Removals, and Non-Compatible Changes

- The `buildbot` Python dist now (finally) requires SQLAlchemy-0.8.0 or later and SQLAlchemy-Migrate-0.9.0 or later. While the old pinned versions (0.7.10 and 0.7.2, respectively) still work, this compatibility is no longer tested and this configuration should be considered deprecated.

Changes for Developers

Slave

Features

Fixes

Deprecations, Removals, and Non-Compatible Changes

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0b6..v0.9.0b7
```

Release Notes for Buildbot 0.9.0b6

The following are the release notes for Buildbot 0.9.0b6 This version was released on January 20, 2016.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Features

- Builders ui page has improved tag filtering capabilities
- Home page enhanced with the list of recent builds sorted by builder
- *IRC* reporter has been partially ported to work on data api.

Fixes

- better stability and reliability in the UI thanks to switch to buildbot data-module
- fix irc

Changes for Developers

- properties object is now directly present in build, and not in build_status. This should not change much unless you try to access your properties via step.build.build_status. Remember that with PropertiesMixin, you can access properties via getProperties on the steps, and on the builds objects.
- *Javascript Data Module* is now integrated, which sets a definitive API for accessing buildbot data in angularJS UI.

Slave

Features

- The `DockerLatentBuildSlave` image attribute is now renderable (can take properties in account).
- The `DockerLatentBuildSlave` sets environment variables describing how to connect to the master. Example dockerfiles can be found in master/contrib/docker (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/docker).

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0b5..v0.9.0b6
```

Release Notes for Buildbot 0.9.0b5

The following are the release notes for Buildbot 0.9.0b5. This version was released on October 21, 2015.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

This version addresses http://trac.buildbot.net/wiki/SecurityAlert090b4 by preventing dissemination of hook information via the web UI.

This also reverts the addition of the frontend data service in 0.9.0b4, as that contained many bugs. It will be re-landed in a subsequent release.

Slave

No changes.

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0b4..0.9.0b5
```

Release Notes for Buildbot 0.9.0b4

The following are the release notes for Buildbot 0.9.0b4 This version was released on October 20, 2015.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

This version is very similar to 0.9.0b3, re-released due to issues with PyPI uploads.

Changes for Developers

- The data API's `startConsuming` method has been removed. Instead of calling this method with a data API path, call `self.master.mq.startConsuming` with an appropriate message routing pattern.

Slave

No changes since 0.9.0b3.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0b3..v0.9.0b4
```

Release Notes for Buildbot 0.9.0b3

The following are the release notes for Buildbot 0.9.0b3. This version was released on October 18, 2015.

See *Upgrading to Nine* for a guide to upgrading from 0.8.x to 0.9.x

Master

Features

- The irc command `hello` now returns 'Hello' in a random language if invoked more than once.
- *`Triggerable`* now accepts a `reason` parameter.
- *`GerritStatusPush`* now accepts a `builders` parameter.
- *StatusPush* callback now receives build results (success/failure/etc) with the `buildFinished` event.
- There's a new renderable type, *Transform*.
- Buildbot now supports wamp as a mq backend. This allows to run a multi-master configuration. See *MQ Specification*.

Fixes

- The *`PyFlakes`* and *`PyLint`* steps no longer parse output in Buildbot log headers (bug #3337 (http://trac.buildbot.net/ticket/3337)).
- *`GerritChangeSource`* is now less verbose by default, and has a `debug` option to enable the logs.
- *`P4Source`* no longer relies on the perforce server time to poll for new changes.
- The commit message for a change from *`P4Source`* now matches what the user typed in.

Deprecations, Removals, and Non-Compatible Changes

- The `buildbot.status.results` module no longer exists and has been renamed to *`buildbot.process.results`*.

Slave

Features

- The Buildbot slave now includes the number of CPUs in the information it supplies to the master on connection. This value is autodetected, but can be overridden with the `--numcpus` argument to `buildslave create-slave`.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0b2..v0.9.0b3
```

Release Notes for Buildbot 0.9.0b2

The following are the release notes for Buildbot 0.9.0b2. Buildbot 0.9.0b2 was released on August, 2 2015.

Master

Features

- Mercurial hook was updated and modernized. It is no longer necessary to fork. One can now extend PYTHONPATH via the hook configuration. Among others, it permits to use a buildbot virtualenv instead of installing buildbot in all the system. Added documentation inside the hook. Misc. clean-up and reorganization in order to make the code a bit more readable.
- UI templates can now be customizable. You can provide html or jade overrides to the www plugins, to customize the UI
- UI side bar is now fixed by default for large screens.

Fixes

- Fix setup for missing www.hooks module
- Fix setup to install only on recents version of pip (>=1.4). This prevents unexpected upgrade to nine from people who just use `pip install -U buildbot`
- Fix a crash in the git hook.
- Add checks to enforce slavenames are identifiers.

Deprecations, Removals, and Non-Compatible Changes

Changes for Developers

- The *`BuilderConfig`* `nextSlave` keyword argument takes a callable. This callable now receives `BuildRequest` instance in its signature as 3rd parameter. **For retro-compatibility, all callable taking only 2 parameters will still work**.
- Data api provides a way to query the build list per slave.
- Data api provides a way to query some build properties in a build list.

Slave

- `buildbot-slave` now requires Python 2.6

Features

- Schedulers: the `codebases` parameter can now be specified in a simple list-of-strings form.

Fixes

- Fix two race conditions in the integration tests

Deprecations, Removals, and Non-Compatible Changes

- Providing Latent AWS EC2 credentials by the `.ec2/aws_id` file is deprecated: Use the standard `.aws/credentials` file, instead.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.9.0b1..v0.9.0b2
```

Release Notes for Buildbot 0.9.0b1

The following are the release notes for Buildbot 0.9.0b1. Buildbot 0.9.0b1 was released on the 25th of June, 2015.

Master

This version represents a refactoring of Buildbot into a consistent, well-defined application composed of loosely coupled components. The components are linked by a common database backend and a messaging system. This allows components to be distributed across multiple build masters. It also allows the rendering of complex web status views to be performed in the browser, rather than on the buildmasters.

The branch looks forward to committing to long-term API compatibility, but does not reach that goal. The Buildbot-0.9.x series of releases will give the new APIs time to "settle in" before we commit to them. Commitment will wait for Buildbot-1.0.0 (as per http://semver.org). Once Buildbot reaches version 1.0.0, upgrades will become much easier for users.

To encourage contributions from a wider field of developers, the web application is designed to look like a normal AngularJS application. Developers familiar with AngularJS, but not with Python, should be able to start hacking on the web application quickly. The web application is "pluggable", so users who develop their own status displays can package those separately from Buildbot itself.

Other goals:

- An approachable HTTP REST API, used by the web application but available for any other purpose.
- A high degree of coverage by reliable, easily-modified tests.
- "Interlocking" tests to guarantee compatibility. For example, the real and fake DB implementations must both pass the same suite of tests. Then no unseen difference between the fake and real implementations can mask errors that will occur in production.

Requirements

The `buildbot` package requires Python 2.6 or higher – Python 2.5 is no longer supported. The `buildbot-slave` package requires Python 2.5 or higher – Python 2.4 is no longer supported.

No additional software or systems, aside from some minor Python packages, are required.

But the devil is in the details:

- If you want to do web *development*, or *build* the `buildbot-www` package, you'll need Node. It's an Angular app, and that's how such apps are developed. We've taken pains to not make either a requirement for users - you

can simply 'pip install' `buildbot-www` and be on your way. This is the case even if you're hacking on the Python side of Buildbot.

- For a single master, nothing else is required.

Minor Python Packages

- Buildbot requires at least Twisted-11.0.0.
- Buildbot works python-dateutil >= 1.5

Known Limitations of 0.9.0b1

The following feature will be implemented for Buildbot 0.9.1 Milestone.

- Multimaster is not supported as of Buildbot 0.9.0. http://trac.buildbot.net/ticket/2644
- Not all status plugin are converted to the new reporter API. Only email and Gerrit reporters are fully supported. Irc support is limited, and not converted to reporter api http://trac.buildbot.net/ticket/2648

Features

Buildbot-0.9.0 introduces the *Data API*, a consistent and scalable method for accessing and updating the state of the Buildbot system. This API replaces the existing, ill-defined Status API, which has been removed. Buildbot-0.9.0 introduces new *WWW Server* Interface using websocket for realtime updates. Buildbot code that interacted with the Status API (a substantial portion!) has been rewritten to use the Data API. Individual features and improvements to the Data API are not described on this page.

- Buildbot now supports plugins. They allow Buildbot to be extended by using components distributed independently from the main code. They also provide for a unified way to access all components. When previously the following construction was used:

```
from buildbot.kind.other.bits import ComponentClass

... ComponentClass ...
```

the following construction achieves the same result:

```
from buildbot.plugins import kind

... kind.ComponentClass ...
```

Kinds of components that are available this way are described in *Plugin Infrastructure in Buildbot*.

Note: While the components can be still directly imported as `buildbot.kind.other.bits`, this might not be the case after Buildbot v1.0 is released.

- Both the P4 source step and P4 change source support ticket-based authentication.
- OpenStack latent slaves now support block devices as a bootable volume.
- Add new *Cppcheck* step.
- Add a new *Docker latent BuildSlave*.

- Add a new configuration for creating custom services in out-of-tree CI systems or plugins. See *buildbot.util.service.BuildbotService*
- Add `try_ssh` configuration file setting and `--ssh` command line option for the try tool to specify the command to use for connecting to the build master.
- GitHub change hook now supports application/json format.
- Add support for dynamically adding steps during a build. See *Dynamic Build Factories*.
- *GitPoller* now supports detecting new branches
- *Git* supports an "origin" option to give a name to the remote repo.

Reporters

Status plugins have been moved into the `reporters` namespace. Their API has slightly to changed in order to adapt to the new data API. See respective documentation for details.

- *GerritStatusPush* renamed to `GerritStatusPush`
- `MailNotifier` renamed to *MailNotifier*
- `MailNotifier` argument `messageFormatter` should now be a `MessageFormatter`, due to removal of data api, custom message formaters need to be rewritten.
- `MailNotifier` argument `previousBuildGetter` is not supported anymore
- `Gerrit` supports specifying an SSH identity file explicitly.
- Added StashStatusPush status hook for Atlassian Stash
- *MailNotifier* no longer forces SSL 3.0 when `useTls` is true.
- *GerritStatusPush* callbacks slightly changed signature, and include a master reference instead of a status reference.
- `GitHubStatus` now accepts a `context` parameter to be passed to the GitHub Status API.
- Buildbot UI introduces branch new Authentication, and Authorizations framework.

  Please look at their respective guide in *WWW Server*

Fixes

- Buildbot is now compatible with SQLAlchemy 0.8 and higher, using the newly-released SQLAlchemy-Migrate.
- The version check for SQLAlchemy-Migrate was fixed to accept more version string formats.
- The *HTTPStep* step's requeset parameters are now renderable.
- With Git(), force the updating submodules to ensure local changes by the build are overwitten. This both ensures more consistent builds and avoids errors when updating submodules.
- Buildbot is now compatible with Gerrit v2.6 and higher.

  To make this happen, the return result of `reviewCB` and `summaryCB` callback has changed from

```
(message, verified, review)
```

  to

```
{'message': message,
 'labels': {'label-name': value,
            ...
            }
}
```

The implications are:

- there are some differences in behaviour: only those labels that were provided will be updated
- Gerrit server must be able to provide a version, if it can't the *GerritStatusPush* will not work

---

Note: If you have an old style `reviewCB` and/or `summaryCB` implemented, these will still work, however there could be more labels updated than anticipated.

More detailed information is available in *GerritStatusPush* section.

- *P4Source*'s `server_tz` parameter now works correctly.
- The `revlink` in changes produced by the Bitbucket hook now correctly includes the `changes/` portion of the URL.
- *PBChangeSource*'s git hook master/contrib/git_buildbot.py (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/git_buildbot.py) now supports git tags

  A pushed git tag generates a change event with the `branch` property equal to the tag name. To schedule builds based on buildbot tags, one could use something like this:

```
c['schedulers'].append(
   SingleBranchScheduler(name='tags',
      change_filter=filter.ChangeFilter(
         branch_re='v[0-9]+\.[0-9]+\.[0-9]+(?:-pre|rc[0-9]+|p[0-9]+)?')
      treeStableTimer=None,
      builderNames=['tag_build']))
```

- Missing "name" and "email" properties received from Gerrit are now handled properly
- Fixed bug which made it impossible to specify the project when using the BitBucket dialect.
- The *PyLint* step has been updated to understand newer output.
- Fixed SVN master-side source step: if a SVN operation fails, the repository end up in a situation when a manual intervention is required. Now if SVN reports such a situation during initial check, the checkout will be clobbered.
- The build properties are now stored in the database in the `build_properties` table.
- The list of changes in the build page now displays all the changes since the last sucessful build.
- GitHub change hook now correctly responds to ping events.
- `buildbot.steps.http` steps now correctly have `url` parameter renderable
- When no arguments are used `buildbot checkconfig` now uses `buildbot.tac` to locate the master config file.
- *buildbot.util.flatten* now correctly flattens arbitrarily nested lists. *buildbot.util.flattened_iterator* provides an iterable over the collection which may be more efficient for extremely large lists.

Deprecations, Removals, and Non-Compatible Changes

- *BonsaiPoller* is removed.
- `buildbot.ec2buildslave` is removed; use `buildbot.buildslave.ec2` instead.
- `buildbot.libvirtbuildslave` is removed; use `buildbot.buildslave.libvirt` instead.
- *buildbot.util.flatten* flattens lists and tuples by default (previously only lists). Additionally, flattening something that isn't the type to flatten has different behaviour. Previously, it would return the original value. Instead, it now returns an array with the original value as the sole element.
- `buildbot.tac` does not support `print` statements anymore. Such files should now use `print` as a function instead (see https://docs.python.org/3.0/whatsnew/3.0.html#print-is-a-function for more details). Note that this applies to both python2.x and python3.x runtimes.

WebStatus

The old, clunky WebStatus has been removed. You will like the new interface! RIP WebStatus, you were a good friend.

remove it and replace it with *`www configuration`*.

Requirements

- Buildbot's tests now require at least Mock-0.8.0.
- SQLAlchemy-Migrate-0.6.1 is no longer supported.
- Builder names are now restricted to unicode strings or ASCII bytestrings. Encoded bytestrings are not accepted.

Steps

- New-style steps are now the norm, and support for old-style steps is deprecated. Such support will be removed in the next release.
  - Status strings for old-style steps could be supplied through a wide variety of conflicting means (`describe`, `description`, `descriptionDone`, `descriptionSuffix`, `getText`, and `setText`, to name just a few). While all attempts have been made to maintain compatibility, you may find that the status strings for old-style steps have changed in this version. To fix steps that call `setText`, try setting the `descriptionDone` attribute directly, instead – or just rewrite the step in the new style.
- Old-style *source* steps (imported directly from `buildbot.steps.source`) are no longer supported on the master.
- The monotone source step got an overhaul and can now better manage its database (initialize and/or migrate it, if needed). In the spirit of monotone, buildbot now always keeps the database around, as it's an append-only database.

Changes and Removals

- Buildslave names must now be 50-character *identifier*. Note that this disallows some common characters in bulidslave names, including spaces, `/`, and `..`

- Builders now have "tags" instead of a category. Builders can have multiple tags, allowing more flexible builder displays.
- *ForceScheduler* has the following changes:
  - The default configuration no longer contains four `AnyPropertyParameter` instances.
  - Configuring `codebases` is now mandatory, and the deprecated `branch`, `repository`, `project`, `revision` are not supported anymore in *ForceScheduler*
  - `buildbot.schedulers.forcesched.BaseParameter.updateFromKwargs` now takes a `collector` parameter used to collect all validation errors
- *Periodic*, *Nightly* and *NightlyTriggerable* have the following changes:
  - The *Periodic* and *Nightly* schedulers can now consume changes and use `onlyIfChanged` and `createAbsoluteTimestamps`.
  - All "timed" schedulers now handle `codebases` the same way. Configuring `codebases` is strongly recommended. Using the `branch` parameter is discouraged.
- Logs are now stored as Unicode strings, and thus must be decoded properly from the bytestrings provided by shell commands. By default this encoding is assumed to be UTF-8, but the *logEncoding* parameter can be used to select an alternative. Steps and individual logfiles can also override the global default.
- The PB status service uses classes which have now been removed, and anyway is redundant to the REST API, so it has been removed. It has taken the following with it:
  - `buildbot statuslog`
  - `buildbot statusgui` (the GTK client)
  - `buildbot debugclient`

  The `PBListener` status listener is now deprecated and does nothing. Accordingly, there is no external access to status objects via Perspective Broker, aside from some compatibility code for the try scheduler.

  The `debugPassword` configuration option is no longer needed and is thus deprecated.
- The undocumented and un-tested `TinderboxMailNotifier`, designed to send emails suitable for the abandoned and insecure Tinderbox tool, has been removed.
- Buildslave info is no longer available via *Interpolate* and the `SetSlaveInfo` buildstep has been removed.
- The undocumented `path` parameter of the *MasterShellCommand* buildstep has been renamed `workdir` for better consistency with the other steps.
- The name and source of a Property have to be unicode or ascii string.
- Property values must be serializable in JSON.
- *IRC* has the following changes:
  - categories parameter is deprecated and removed. It should be replaced with tags=[cat]
  - noticeOnChannel parameter is deprecated and removed.
- workdir behavior has been unified:
  - `workdir` attribute of steps is now a property in *BuildStep*, and choose the workdir given following priority:
    - workdir of the step, if defined
    - workdir of the builder (itself defaults to 'build')

    * setDefaultWorkdir() has been deprecated, but is now behaving the same for all the steps: Setting self.workdir if not already set

- *Trigger* now has a `getSchedulersAndProperties` method that can ve overridden to support dynamic triggering.
- `` `master.cfg `` is now parsed from a thread. Previously it was run in the main thread, and thus slowing down the master in case of big config, or network access done to generate the config.
- *SVNPoller*'s svnurl parameter has been changed to repourl.

Changes for Developers

- Botmaster no longer service parent for buildslaves. Service parent functionality has been transferred to BuildslaveManager. It should be noted Botmaster no longer has a `slaves` field as it was moved to BuildslaveManager.
- The sourcestamp DB connector now returns a `patchid` field.
- Buildbot no longer polls the database for jobs. The `db_poll_interval` configuration parameter and the *db* key of the same name are deprecated and will be ignored.
- The interface for adding changes has changed. The new method is `master.data.updates.addChange` (implemented by `addChange`), although the old interface (`master.addChange`) will remain in place for a few versions. The new method:
  - returns a change ID, not a Change instance;
  - takes its `when_timestamp` argument as epoch time (UNIX time), not a datetime instance; and
  - does not accept the deprecated parameters `who`, `isdir`, `is_dir`, and `when`.
  - requires that all strings be unicode, not bytestrings.

  Please adjust any custom change sources accordingly.
- A new build status, CANCELLED, has been added. It is used when a step or build is deliberately cancelled by a user.
- This upgrade will delete all rows from the `buildrequest_claims` table. If you are using this table for analytical purposes outside of Buildbot, please back up its contents before the upgrade, and restore it afterward, translating object IDs to scheduler IDs if necessary. This translation would be very slow and is not required for most users, so it is not done automatically.
- All of the schedulers DB API methods now accept a schedulerid, rather than an objectid. If you have custom code using these methods, check your code and make the necessary adjustments.
- The `addBuildsetForSourceStamp` method has become `addBuildsetForSourceStamps`, and its signature has changed. The `addBuildsetForSourceStampSetDetails` method has become `addBuildsetForSourceStampsWithDefaults`, and its signature has changed. The `addBuildsetForSourceStampDetails` method has been removed. The `addBuildsetForLatest` method has been removed. It is equivalent to `addBuildsetForSourceStampDetails` with `sourcestamps=None`. These methods are not yet documented, and their interface is not stable. Consult the source code for details on the changes.
- The `preStartConsumingChanges` and `startTimedSchedulerService` hooks have been removed.
- The triggerable schedulers' `trigger` method now requires a list of sourcestamps, rather than a dictionary.
- The `SourceStamp` class is no longer used. It remains in the codebase to support loading data from pickles on upgrade, but should not be used in running code.

- The `BuildRequest` class no longer has full `source` or `sources` attributes. Use the data API to get this information (which is associated with the buildset, not the build request) instead.
- The undocumented `BuilderControl` method `submitBuildRequest` has been removed.
- The debug client no longer supports requesting builds (the `requestBuild` method has been removed). If you have been using this method in production, consider instead creating a new change source, using the *`ForceScheduler`*, or using one of the try schedulers.
- The `buildbot.misc.SerializedInvocation` class has been removed; use *`buildbot.util.debounce.method`* instead.
- The `progress` attributes of both *`buildbot.process.buildstep.BuildStep`* and *`buildbot.process.build.Build`* have been removed. Subclasses should only be accessing the progress-tracking mechanics via the *`buildbot.process.buildstep.BuildStep.setProgress`* method.

Slave

Features

Fixes

Deprecations, Removals, and Non-Compatible Changes

- buildmaster and buildslave no longer supports old-style source steps.
- On Windows, if a *`ShellCommand`* step in which `command` was specified as a list is executed, and a list element is a string consisting of a single pipe character, it no longer creates a pipeline. Instead, the pipe character is passed verbatim as an argument to the program, like any other string. This makes command handling consistent between Windows and Unix-like systems. To have a pipeline, specify `command` as a string.

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.8.10..v0.9.0b1
```

Release Notes for Buildbot 0.8.11

The following are the release notes for Buildbot 0.8.11. This version was released on the 20th of April, 2015.

Master

Requirements:

- Buildbot works python-dateutil >= 1.5

Features

- GitHub change hook now supports application/json format.
- Buildbot is now compatible with Gerrit v2.6 and higher.

  To make this happen, the return result of `reviewCB` and `summaryCB` callback has changed from

```
(message, verified, review)
```

  to

```
{'message': message,
 'labels': {'label-name': value,
            ...
            }
}
```

  The implications are:

  - there are some differences in behaviour: only those labels that were provided will be updated
  - Gerrit server must be able to provide a version, if it can't the `GerritStatusPush` will not work

  Note: If you have an old style `reviewCB` and/or `summaryCB` implemented, these will still work, however there could be more labels updated than anticipated.

  More detailed information is available in `GerritStatusPush` section.

- Buildbot now supports plugins. They allow Buildbot to be extended by using components distributed independently from the main code. They also provide for a unified way to access all components. When previously the following construction was used:

```
from buildbot.kind.other.bits import ComponentClass

... ComponentClass ...
```

  the following construction achieves the same result:

```
from buildbot.plugins import kind

... kind.ComponentClass ...
```

  Kinds of components that are available this way are described in *Plugin Infrastructure in Buildbot*.

  Note: While the components can be still directly imported as `buildbot.kind.other.bits`, this might not be the case after Buildbot v1.0 is released.

- *GitPoller* now supports detecting new branches
- *MasterShellCommand* now renders the `path` argument.
- `ShellMixin`: the `workdir` can now be overridden in the call to `makeRemoteShellCommand`.
- GitHub status target now allows to specify a different base URL for the API (usefule for GitHub enterprise installations). This feature requires *txgithub* of version 0.2.0 or better.

- GitHub change hook now supports payload validation using shared secret, see the GitHub hook documentation for details.
- Added StashStatusPush status hook for Atlassian Stash
- Builders can now have multiple "tags" associated with them. Tags can be used in various status classes as filters (eg, on the waterfall page).
- `MailNotifier` no longer forces SSL 3.0 when `useTls` is true.
- GitHub change hook now supports function as codebase argument.
- GitHub change hook now supports pull_request events.
- `Trigger`: the `getSchedulersAndProperties` customization method has been backported from Nine. This provides a way to dynamically specify which schedulers (and the properties for that scheduler) to trigger at runtime.

Fixes

- GitHub change hook now correctly responds to ping events.
- `buildbot.steps.http` steps now correctly have `url` parameter renderable
- *`MasterShellCommand`* now correctly logs the working directory where it was run.
- With Git(), force the updating submodules to ensure local changes by the build are overwitten. This both ensures more consistent builds and avoids errors when updating submodules.
- With Git(), make sure 'git submodule sync' is called before 'git submodule update' to update stale remote urls (bug #2155 (http://trac.buildbot.net/ticket/2155)).

Deprecations, Removals, and Non-Compatible Changes

- The builder parameter "category" is deprecated and is replaced by a parameter called "tags".

Changes for Developers

- `Trigger`: `createTriggerProperties` now takes one argument (the properties to generate).
- `Trigger`: `getSchedulers` method is no longer used and was removed.

Slave

Features

Fixes

Deprecations, Removals, and Non-Compatible Changes

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.8.10..532cf49
```

Release Notes for Buildbot 0.8.10

The following are the release notes for Buildbot 0.8.10. Buildbot 0.8.10 was released on the 2nd of December, 2014.

Master

Features

- Both the P4 source step and P4 change source support ticket-based authentication.
- Clickable 'categories' links added in 'Waterfall' page (web UI).

Fixes

- Buildbot is now compatible with SQLAlchemy 0.8 and higher, using the newly-released SQLAlchemy-Migrate.
- The *HTTPStep* step's request parameters are now renderable.
- Fixed content spoofing vulnerabilities (bug #2589 (http://trac.buildbot.net/ticket/2589)).
- Fixed cross-site scripting in status_json (bug #2943 (http://trac.buildbot.net/ticket/2943)).
- *GerritStatusPush* supports specifying an SSH identity file explicitly.
- Fixed bug which made it impossible to specify the project when using the BitBucket dialect.
- Fixed SVN master-side source step: if a SVN operation fails, the repository end up in a situation when a manual intervention is required. Now if SVN reports such a situation during initial check, the checkout will be clobbered.
- Fixed master-side source steps to respect the specified timeout when removing files.

Deprecations, Removals, and Non-Compatible Changes

Changes for Developers

Slave

Features

Fixes

Deprecations, Removals, and Non-Compatible Changes

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.8.9..eight
```

Release Notes for Buildbot 0.8.9

The following are the release notes for Buildbot 0.8.9. Buildbot 0.8.9 was released on 14 June, 2014.

Master

Features

- **The following optional parameters have been added to `EC2LatentBuildSlave`**
  - Boolean parameter `spot_instance`, default False, creates a spot instance.
  - Float parameter `max_spot_price` defines the maximum bid for a spot instance.
  - List parameter `volumes`, takes a list of (volume_id, mount_point) tuples.
  - String parameter `placement` is appended to the `region` parameter, e.g. `region='us-west-2', placement='b'` will result in the spot request being placed in us-west-2b.
  - Float parameter `price_multiplier` specifies the percentage bid above the 24-hour average spot price.
  - Dict parameter `tags` specifies AWS tags as key/value pairs to be applied to new instances.

  With `spot_instance=True`, an `EC2LatentBuildSlave` will attempt to create a spot instance with the provided spot price, placement, and so on.
- The web hooks now include support for Bitbucket, GitLab and Gitorious.
- The GitHub webhook has been updated to work with v3 of the GitHub webhook API.
- The GitHub webhook can now optionally ignore non-distinct commits (bug #1861 (http://trac.buildbot.net/ticket/1861)).
- The *`HgPoller`* and *`GitPoller`* now split filenames on newlines, rather than whitespace, so files containing whitespace are handled correctly.
- Add 'pollAtLaunch' flag for polling change sources. This allows a poller to poll immediately on launch and get changes that occurred while it was down.
- Added the *`BitbucketPullrequestPoller`* changesource.
- The *`GitPoller`* can now be configured to poll all available branches (pull request 1010 (https://github.com/buildbot/buildbot/pull/1010)).
- The *`P4Source`* changesource now supports Perforce servers in a different timezone than the buildbot master (pull request 728 (https://github.com/buildbot/buildbot/pull/728)).
- Each Scheduler type can now take a 'reason' argument to customize the reason it uses for triggered builds.
- A new argument `createAbsoluteSourceStamps` has been added to *`SingleBranchScheduler`* for use with multiple codebases.
- A new argument `createAbsoluteSourceStamps` has been added to *`Nightly`* for use with multiple codebases.
- The *`Periodic`* scheduler now supports codebases.
- The *`ForceScheduler`* now takes a `buttonName` argument to specify the name of the button on the force-build form.
- Master side source checkout steps now support patches (bug #2098 (http://trac.buildbot.net/ticket/2098)). The *`Git`* and *`Mercurial`* steps use their inbuilt commands to apply patches (bug #2563 (http://trac.buildbot.net/ticket/2563)).
- Master side source checkout steps now support retry option (bug #2465 (http://trac.buildbot.net/ticket/2465)).

- Master-side source checkout steps now respond to the "stop build" button (bug #2356 (http://trac.buildbot.net/ticket/2356)).
- *Git* source checkout step now supports reference repositories.
- The *Git* step now uses the *git clean* option *-f* twice, to also remove untracked directories managed by another git repository. See bug #2560 (http://trac.buildbot.net/ticket/2560).
- The `branch` and `codebase` arguments to the *Git* step are now renderable.
- Gerrit integration with *Git* Source step on master side (bug #2485 (http://trac.buildbot.net/ticket/2485)).
- *P4* source step now supports more advanced options.
- The master-side *SVN* step now supports authentication for mode=export, fixing bug #2463 (http://trac.buildbot.net/ticket/2463).
- The *SVN* step will now canonicalize URL's before matching them for better accuracy.
- The *SVN* step now obfuscates the password in status logs, fixing bug #2468 (http://trac.buildbot.net/ticket/2468).
- *SVN* source step and ShellCommand now support password obfuscation. (bug #2468 (http://trac.buildbot.net/ticket/2468) and bug #1748 (http://trac.buildbot.net/ticket/1748)).
- *CVS* source step now checks for "sticky dates" from a previous checkout before updating an existing source directory.
- :*Repo* now supports a `depth` flag when initializing the repo. This controls the amount of git history to download.
- The `manifestBranch` of the bb:step:*Repo* step is now renderable
- New source step *Monotone* added on master side.
- New source step *Darcs* added on master side.
- A new *Robocopy* step is available for Windows builders (pull request 728 (https://github.com/buildbot/buildbot/pull/728)).
- The attributes `description`, `descriptionDone` and `descriptionSuffix` have been moved from `ShellCommand` to its superclass `BuildStep` so that any class that inherits from `BuildStep` can provide a suitable description of itself.
- A new `FlattenList` Renderable has been added which can flatten nested lists.
- Added new build steps for *VC12*, *VS2013* and *MsBuild12*.
- The `mode` parameter of the VS steps is now renderable (bug #2592 (http://trac.buildbot.net/ticket/2592)).
- The *HTTPStep* step can make arbitrary HTTP requests from the master, allowing communication with external APIs. This new feature requires the optional `txrequests` and `requests` Python packages.
- A new *MultipleFileUpload* step was added to allow uploading several files (or directories) in a single step.
- Information about the buildslaves (admin, host, etc) is now persisted in the database and available even if the slave is not connected.
- Buildslave info can now be retrieved via *Interpolate* and a new `SetSlaveInfo` buildstep.
- The `GNUAutotools` factory now has a reconf option to run autoreconf before `./configure`.
- Builder configurations can now include a `description`, which will appear in the web UI to help humans figure out what the builder does.
- The WebStatus builder page can now filter pending/current/finished builds by property parameters of the form `?property.<name>=<value>`.

- The WebStatus `StatusResourceBuilder` page can now take the `maxsearch` argument
- The WebStatus has a new authz "view" action that allows you to require users to logged in to view the WebStatus.
- The WebStatus now shows revisions (+ codebase) where it used to simply say "multiple rev".
- The Console view now supports codebases.
- **The web UI for Builders has been updated:**
  - shows the build 'reason' and 'interested users'
  - shows sourcestamp information for builders that use multiple codebases (instead of the generic "multiple rev" placeholder that was shown before).
- The waterfall and atom/rss feeds can be filtered with the `project` url paramter.
- The WebStatus `Authorization` support now includes a `view` action which can be used to restrict read-only access to the Buildbot instance.
- The web status now has options to cancel some or all pending builds.
- The WebStatus now interprets ANSI color codes in stdio output.
- It is now possible to select categories to show in the waterfall help
- The web status now automatically scrolls output logs (pull request 1078 (https://github.com/buildbot/buildbot/pull/1078)).
- The web UI now supports a PNG Status Resource that can be accessed publicly from for example README.md files or wikis or whatever other resource. This view produces an image in PNG format with information about the last build for the given builder name or whatever other build number if is passed as an argument to the view.
- Revision links for commits on SouceForge (Allura) are now automatically generated.
- The 'Rebuild' button on the web pages for builds features a dropdown to choose whether to rebuild from exact revisions or from the same sourcestamps (ie, update branch references)
- Build status can be sent to GitHub. Depends on txgithub package. See *`GitHubStatusPush`* and GitHub Commit Status (https://github.com/blog/1227-commit-status-api).
- The IRC bot of *`IRC`* will, unless useRevisions is set, shorten long lists of revisions printed when a build starts; it will only show two, and the number of additional revisions included in the build.
- A new argument `summaryCB` has been added to `GerritStatusPush`, to allow sending one review per buildset. Sending a single "summary" review per buildset is now the default if neither `summaryCB` nor `reviewCB` are specified.
- The `comments` field of changes is no longer limited to 1024 characters on MySQL and Postgres. See bug #2367 (http://trac.buildbot.net/ticket/2367) and pull request 736 (https://github.com/buildbot/buildbot/pull/736).
- HTML log files are no longer stored in status pickles (pull request 1077 (https://github.com/buildbot/buildbot/pull/1077))
- Builds are now retried after a slave is lost (pull request 1049 (https://github.com/buildbot/buildbot/pull/1049)).
- The buildbot status client can now access a build properties via the `getProperties` call.
- The `start`, `restart`, and `reconfig` commands will now wait for longer than 10 seconds as long as the master continues producing log lines indicating that the configuration is progressing.
- Added new config option `protocols` which allows to configure multiple protocols on single master.
- RemoteShellCommands can be killed by SIGTERM with the sigtermTime parameter before resorting to SIGKILL (bug #751 (http://trac.buildbot.net/ticket/751)). If the slave's version is less than 0.8.9, the slave will kill the process with SIGKILL regardless of whether sigtermTime is supplied.

- Introduce an alternative way to deploy Buildbot and try the pyflakes tutorial using *Docker*.
- Added zsh and bash tab-completions support for 'buildbot' command.
- An example of a declarative configuration is included in `master/contrib/SimpleConfig.py`, with copious comments.
- Systemd unit files for Buildbot are available in the master/contrib/ (https://github.com/buildbot/buildbot-contrib/blob/master/master/contrib/) directory.
- We've added some extra checking to make sure that you have a valid locale before starting buildbot (#2608).

Forward Compatibility

In preparation for a more asynchronous implementation of build steps in Buildbot 0.9.0, this version introduces support for new-style steps. Existing old-style steps will continue to function correctly in Buildbot 0.8.x releases and in Buildbot 0.9.0, but support will be dropped soon afterward. See *New-Style Build Steps*, below, for guidance on rewriting existing steps in this new style. To eliminate ambiguity, the documentation for this version only reflects support for new-style steps. Refer to the documentation for previous versions for infrormation on old-style steps.

Fixes

- Fixes an issue where *`GitPoller`* sets the change branch to `refs/heads/master` - which isn't compatible with *`Git`* (pull request 1069 (https://github.com/buildbot/buildbot/pull/1069)).
- Fixed an issue where the *`Git`* and *`CVS`* source steps silently changed the `workdir` to `'build'` when the 'copy' method is used.
- The *`CVS`* source step now respects the timeout parameter.
- The *`Git`* step now uses the *git submodule update* option *–init* when updating the submodules of an existing repository, so that it will receive any newly added submodules.
- The web status no longer relies on the current working directory, which is not set correctly by some initscripts, to find the `templates/` directory (bug #2586 (http://trac.buildbot.net/ticket/2586)).
- The Perforce source step uses the correct path separator when the master is on Windows and the build slave is on a POSIX OS (pull request 1114 (https://github.com/buildbot/buildbot/pull/1114)).
- The source steps now correctly interpolate properties in `env`.
- `GerritStatusPush` now supports setting scores with Gerrit 2.6 and newer
- The change hook no longer fails when passing unicode to `change_hook_auth` (pull request 996 (https://github.com/buildbot/buildbot/pull/996)).
- The source steps now correctly interpolate properties in `env`.
- Whitespace is properly handled for StringParameter, so that appropriate validation errors are raised for `required` parameters (pull request 1084 (https://github.com/buildbot/buildbot/pull/1084)).
- Fix a rare case where a buildtep might fail from a GeneratorExit exception (pull request 1063 (https://github.com/buildbot/buildbot/pull/1063)).
- Fixed an issue where UTF-8 data in logs caused RSS feed exceptions (bug #951 (http://trac.buildbot.net/ticket/951)).
- Fix an issue with unescaped author names causing invalid RSS feeds (bug #2596 (http://trac.buildbot.net/ticket/2596)).
- Fixed an issue with pubDate format in feeds.

- Fixed an issue where the step text value could cause a `TypeError` in the build detail page (pull request 1061 (https://github.com/buildbot/buildbot/pull/1061)).
- Fix failures where `git clean` fails but could be clobbered (pull request 1058 (https://github.com/buildbot/buildbot/pull/1058)).
- Build step now correctly fails when the git clone step fails (pull request 1057 (https://github.com/buildbot/buildbot/pull/1057)).
- Fixed a race condition in slave shutdown (pull request 1019 (https://github.com/buildbot/buildbot/pull/1019)).
- Now correctly unsubscribes StatusPush from status updates when reconfiguring (pull request 997 (https://github.com/buildbot/buildbot/pull/997)).
- Fixes parsing git commit messages that are blank.
- *`Git`* no longer fails when work dir exists but isn't a checkout (bug #2531 (http://trac.buildbot.net/ticket/2531)).
- The *haltOnFailure* and *flunkOnFailure* attributes of *`ShellCommand`* are now renderable. (bug #2486 (http://trac.buildbot.net/ticket/2486)).
- The *rotateLength* and *maxRotatedFile* arguments are no longer treated as strings in `buildbot.tac`. This fixes log rotation. The upgrade_master command will notify users if they have this problem.
- Buildbot no longer specifies a revision when pulling from a mercurial (bug #438 (http://trac.buildbot.net/ticket/438)).
- The WebStatus no longer incorrectly refers to fields that might not be visible.
- The GerritChangeSource now sets a default author, fixing an exception that occured when Gerrit didn't report an owner name/email.
- Respects the `RETRY` status when an interrupt occurs.
- Fixes an off-by-one error when the tryclient is finding the current git branch.
- Improve the Mercurial source stamp extraction in the try client.
- Fixes some edge cases in timezone handling for python < `2.7.4` (bug #2522 (http://trac.buildbot.net/ticket/2522)).
- The `EC2LatentBuildSlave` will now only consider available AMI's.
- Fixes a case where the first build runs on an old slave instead of a new one after reconfig (bug #2507 (http://trac.buildbot.net/ticket/2507)).
- The e-mail address validation for the MailNotifier status receiver has been improved.
- The `--db` parameter of `buildbot create-master` is now validated.
- No longer ignores default choice for ForceScheduler list parameters
- Now correctly handles `BuilderConfig(..., mergeRequests=False)` (bug #2555 (http://trac.buildbot.net/ticket/2555)).
- Now excludes changes from sourcestamps when they aren't in the DB (bug #2554 (http://trac.buildbot.net/ticket/2554)).
- Fixes a compatibility issue with HPCloud in the OpenStack latent slave.
- Allow _ as a valid character in JSONP callback names.
- Fix build start time retrieval in the WebStatus grid view.
- Increase the length of the DB fields `changes.comments` and `buildset_properties.property_value`.

Deprecations, Removals, and Non-Compatible Changes

- The slave-side source steps are deprecated in this version of Buildbot, and master-side support will be removed in a future version. Please convert any use of slave-side steps (imported directly from `buildbot.steps.source`, rather than from a specific module like `buildbot.steps.source.svn`) to use master-side steps.
- Both old-style and new-style steps are supported in this version of Buildbot. Upgrade your steps to new-style now, as support for old-style steps will be dropped after Buildbot-0.9.0. See *New-Style Build Steps* for details.
  - The `LoggingBuildStep` class has been deprecated, and support will be removed along with support for old-style steps after the Buildbot-0.9.0 release. Instead, subclass *`BuildStep`* and mix in `ShellMixin` to get similar behavior.
- `slavePortnum` option deprecated, please use `c['protocols']['pb']['port']` to set up PB port
- The `buildbot.process.mtrlogobserver` module have been renamed to `buildbot.steps.mtrlogobserver`.
- The buildmaster now requires at least Twisted-11.0.0.
- The buildmaster now requires at least sqlalchemy-migrate 0.6.1.
- The `hgbuildbot` Mercurial hook has been moved to `contrib/`, and does not work with recent versions of Mercurial and Twisted. The runtimes for these two tools are incompatible, yet `hgbuildbot` attempts to run both in the same Python interpreter. Mayhem ensues.
- The try scheduler's `--connect=ssh` method no longer supports waiting for results (`--wait`).
- The former `buildbot.process.buildstep.RemoteCommand` class and its subclasses are now in `buildbot.process.remotecommand`, although imports from the previous path will continue to work. Similarly, the former `buildbot.process.buildstep.LogObserver` class and its subclasses are now in *`buildbot.process.logobserver`*, although imports from the previous path will continue to work.
- The undocumented BuildStep method `checkDisconnect` is deprecated and now does nothing as the handling of disconnects is now handled in the `failed` method. Any custom steps adding this method as a callback or errback should no longer do so.
- The build step `MsBuild` is now called `MsBuild4` as multiple versions are now supported. An alias is provided so existing setups will continue to work, but this will be removed in a future release.

Changes for Developers

- The `CompositeStepMixin` now provides a `runGlob` method to check for files on the slave that match a given shell-style pattern.
- The `BuilderStatus` now allows you to pass a `filter_fn` argument to `generateBuilds`.

Slave

Features

- Added zsh and bash tab-completions support for 'buildslave' command.
- RemoteShellCommands accept the new sigtermTime parameter from master. This allows processes to be killed by SIGTERM before resorting to SIGKILL (bug #751 (http://trac.buildbot.net/ticket/751))
- Commands will now throw a `ValueError` if mandatory args are not present.
- Added a new remote command `GlobPath` that can be used to call Python's `glob.glob` on the slave.

Fixes

- Fixed an issue when buildstep stop() was raising an exception incorrectly if timeout for buildstep wasn't set or was None (see pull request 753 (https://github.com/buildbot/buildbot/pull/753)) thus keeping watched logfiles open (this prevented their removal on Windows in subsequent builds).
- Fixed a bug in P4 source step where the `timeout` parameter was ignored.
- Fixed a bug in P4 source step where using a custom view-spec could result in failed syncs due to incorrectly generated command-lines.
- The logwatcher will use `/usr/xpg4/bin/tail` on Solaris, it if is available (pull request 1065 (https://github.com/buildbot/buildbot/pull/1065)).

Deprecations, Removals, and Non-Compatible Changes

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.8.8..v0.8.9
```

Release Notes for Buildbot v0.8.8

The following are the release notes for Buildbot v0.8.8 Buildbot v0.8.8 was released on August 22, 2013

Master

Features

- The `MasterShellCommand` step now correctly handles environment variables passed as list.
- The master now poll the database for pending tasks when running buildbot in multi-master mode.
- The algorithm to match build requests to slaves has been rewritten pull request 615 (https://github.com/buildbot/buildbot/pull/615). The new algorithm automatically takes locks into account, and will not schedule a build only to have it wait on a lock. The algorithm also introduces a `canStartBuild` builder configuration option which can be used to prevent a build request being assigned to a slave.
- `buildbot stop` and `buildbot restart` now accept `--clean` to stop or restart the master cleanly (allowing all running builds to complete first).
- The *`IRC`* bot now supports clean shutdown and immediate shutdown by using the command 'shutdown'. To allow the command to function, you must provide *allowShutdown=True*.
- *`CopyDirectory`* has been added.
- `BuildslaveChoiceParameter` has been added to provide a way to explicitly choose a buildslave for a given build.
- default.css now wraps preformatted text by default.
- Slaves can now be paused through the web status.
- The latent buildslave support is less buggy, thanks pull request 646 (https://github.com/buildbot/buildbot/pull/646).

- The `treeStableTimer` for `AnyBranchScheduler` now maintains separate timers for separate branches, codebases, projects, and repositories.
- *SVN* has a new option *preferLastChangedRev=True* to use the last changed revision for `got_revision`
- The build request DB connector method *getBuildRequests* can now filter by branch and repository.
- A new *SetProperty* step has been added in `buildbot.steps.master` which can set a property directly without accessing the slave.
- The new *LogRenderable* step logs Python objects, which can contain renderables, to the logfile. This is helpful for debugging property values during a build.
- 'buildbot try' now has an additional option *–property* option to set properties. Unlike the existing option *–properties* option, this new option supports setting only a single property and therefore allows commas to be included in the property name and value.
- The `Git` step has a new `config` option, which accepts a dict of git configuration options to pass to the low-level git commands. See *Git* for details.
- In *ShellCommand* ShellCommand now validates its arguments during config and will identify any invalid arguments before a build is started.
- The list of force schedulers in the web UI is now sorted by name.
- OpenStack-based Latent Buildslave support was added. See pull request 666 (https://github.com/buildbot/buildbot/pull/666).
- Master-side support for P4 is available, and provides a great deal more flexibility than the old slave-side step. See pull request 596 (https://github.com/buildbot/buildbot/pull/596).
- Master-side support for Repo is available. The step parameters changed to camelCase. `repo_downloads`, and `manifest_override_url` properties are no longer hardcoded, but instead consult as default values via renderables. Renderable are used in favor of callables for `syncAllBranches` and `updateTarball`.
- Builder configurations can now include a `description`, which will appear in the web UI to help humans figure out what the builder does.
- GNUAutoconf and other pre-defined factories now work correctly (bug #2402 (http://trac.buildbot.net/ticket/2402))
- The pubDate in RSS feeds is now rendered correctly (bug #2530 (http://trac.buildbot.net/ticket/2530))

Deprecations, Removals, and Non-Compatible Changes

- The `split_file` function for *SVNPoller* may now return a dictionary instead of a tuple. This allows it to add extra information about a change (such as `project` or `repository`).
- The `workdir` build property has been renamed to `builddir`. This change accurately reflects its content; the term "workdir" means something different. `workdir` is currently still supported for backwards compatibility, but will be removed eventually.
- The `Blocker` step has been removed.
- Several polling ChangeSources are now documented to take a `pollInterval` argument, instead of `pollinterval`. The old name is still supported.
- StatusReceivers' checkConfig method should no longer take an *errors* parameter. It should indicate errors by calling *error*.
- Build steps now require that their name be a string. Previously, they would accept anything, but not behave appropriately.

- The web status no longer displays a potentially misleading message, indicating whether the build can be rebuilt exactly.
- The `SetProperty` step in `buildbot.steps.shell` has been renamed to *`SetPropertyFromCommand`*.
- The EC2 and libvirt latent slaves have been moved to `buildbot.buildslave.ec2` and `buildbot.buildslave.libirt` respectively.
- Pre v0.8.7 versions of buildbot supported passing keyword arguments to `buildbot.process.BuildFactory.addStep`, but this was dropped. Support was added again, while still being deprecated, to ease transition.

Changes for Developers

- Added an optional build start callback to `buildbot.status.status_gerrit.GerritStatusPush` This release includes the fix for bug #2536 (http://trac.buildbot.net/ticket/2536).
- An optional `startCB` callback to *`GerritStatusPush`* can be used to send a message back to the committer. See the linked documentation for details.
- bb:sched:*ChoiceStringParameter* has a new method `getChoices` that can be used to generate content dynamically for Force scheduler forms.

Slave

Features

- The fix for Twisted bug #5079 is now applied on the slave side, too. This fixes a perspective broker memory leak in older versions of Twisted. This fix was added on the master in Buildbot-0.8.4 (see bug #1958 (http://trac.buildbot.net/ticket/1958)).
- The `--nodaemon` option to `buildslave start` now correctly prevents the slave from forking before running.

Deprecations, Removals, and Non-Compatible Changes

Details

For a more detailed description of the changes made in this version, see the git log itself:

```
git log v0.8.7..v0.8.8
```

Release Notes for Buildbot v0.8.7

The following are the release notes for Buildbot v0.8.7. Buildbot v0.8.7 was released on September 22, 2012. Buildbot 0.8.7p1 was released on November 21, 2012.

0.8.7p1

In addition to what's listed below, the 0.8.7p1 release adds the following.

- The `SetPropertiesFromEnv` step now correctly gets environment variables from the slave, rather than those set on the master. Also, it logs the changes made to properties.
- The master-side `Git` source step now doesn't try to clone a branch called `HEAD`. This is what `git` does by default, and specifying it explicitly doesn't work as expected.
- The `Git` step properly deals with the case when there is a file called `FETCH_HEAD` in the checkout.
- Buildbot no longer forks when told not to daemonize.
- Buildbot's startup is now more robust. See bug #1992 (http://trac.buildbot.net/ticket/1992).
- The `Trigger` step uses the provided list of source stamps exactly, if given, instead of adding them to the sourcestamps of the current build. In 0.8.7, they were combined with the source stamps for the current build.
- The `Trigger` step again completely ignores the source stamp of the current build, if `alwaysUseLatest` is set. In 0.8.7, this was mistakenly changed to only ignore the specified revision of the source stamp.
- The `Triggerable` scheduler is again properly passing changes through to the scheduled builds. See bug #2376 (http://trac.buildbot.net/ticket/2376).
- Web change hooks log errors, allowing debugging.
- The `base` change hook now properly decodes the provided date.
- `CVSMailDir` has been fixed.
- Importing `buildbot.test` no longer causes python to exit, if `mock` insn't installed. The fixes `pydoc -k` when buildbot is installed.
- `Mercurial` properly updates to the correct branch, when using `inrepo` branches.
- Buildbot now doesn't fail on invalid UTF-8 in a number of places.
- Many documentation updates and fixes.

Master

Features

- Buildbot now supports building projects composed of multiple codebases. New schedulers can aggregate changes to multiple codebases into source stamp sets (with one source stamp for each codebase). Source steps then check out each codebase as required, and the remainder of the build process proceeds normally. See the *Multiple-Codebase Builds* for details.
  - The format of the `got_revision` property has changed for multi-codebase builds. It is now a dictionary keyed by codebase.
- `Source` and `ShellCommand` steps now have an optional `descriptionSuffix`, a suffix to the `description`/`descriptionDone` values. For example this can help distinguish between multiple `Compile` steps that are applied to different codebases.
- The `Git` step has a new `getDescription` option, which will run `git describe` after checkout normally. See *Git* for details.
- A new interpolation placeholder *Interpolate*, with more regular syntax, is available.
- A new ternary substitution operator `:?` and `:#?` is available with the `Interpolate` class.

- `IRenderable.getRenderingFor` can now return a deferred.
- The Mercurial hook now supports multiple masters. See pull request 436 (https://github.com/buildbot/buildbot/pull/436).
- There's a new poller for Mercurial: *`HgPoller`*.
- The new `HTPasswdAprAuth` uses libaprutil (through ctypes) to validate the password against the hash from the .htpasswd file. This adds support for all hash types htpasswd can generate.
- `GitPoller` has been rewritten. It now supports multiple branches and can share a directory between multiple pollers. It is also more resilient to changes in configuration, or in the underlying repository.
- Added a new property `httpLoginUrl` to `buildbot.status.web.authz.Authz` to render a nice Login link in WebStatus for unauthenticated users if `useHttpHeader` and `httpLoginUrl` are set.
- `ForceScheduler` has been updated:
  - support for multiple *codebases* via the `codebases` parameter
  - `NestedParameter` to provide a logical grouping of parameters.
  - `CodebaseParameter` to set the branch/revision/repository/project for a codebase
  - new HTML/CSS customization points. Each parameter is contained in a `row` with multiple 'class' attributes associated with them (eg, 'force-string' and 'force-nested') as well as a unique id to use with Javascript. Explicit line-breaks have been removed from the HTML generator and are now controlled using CSS.
- The *`SVNPoller`* now supports multiple projects and codebases. See pull request 443 (https://github.com/buildbot/buildbot/pull/443).
- The *`MailNotifier`* now takes a callable to calculate the "previous" build for purposes of determining status changes. See pull request 489 (https://github.com/buildbot/buildbot/pull/489).
- The `copy_properties` parameter, given a list of properties to copy into the new build request, has been deprecated in favor of explicit use of `set_properties`.

Deprecations, Removals, and Non-Compatible Changes

- Buildbot master now requires at least Python-2.5 and Twisted-9.0.0.
- Passing a *`BuildStep`* subclass (rather than instance) to `addStep` is no longer supported. The `addStep` method now takes exactly one argument.
- Buildbot master requires `python-dateutil` version 1.5 to support the Nightly scheduler.
- `ForceScheduler` has been updated to support multiple *codebases*. The branch/revision/repository/project are deprecated; if you have customized these values, simply provide them as `codebases=[CodebaseParameter(name='', ...)]`.
  - The POST URL names for `AnyPropertyParameter` fields have changed. For example, 'property1name' is now 'property1_name', and 'property1value' is now 'property1_value'. Please update any bookmarked or saved URL's that used these fields.
  - `forcesched.BaseParameter` API has changed quite a bit and is no longer backwards compatible. Updating guidelines:
    * `get_from_post` is renamed to `getFromKwargs`
    * `update_from_post` is renamed to `updateFromKwargs`. This function's parameters are now called via named parameters to allow subclasses to ignore values it doesn't use. Subclasses should

add `**unused` for future compatibility. A new parameter `sourcestampset` is provided to allow subclasses to modify the sourcestamp set, and will probably require you to add the `**unused` field.

- The parameters to the callable version of `build.workdir` have changed. Instead of a single sourcestamp, a list of sourcestamps is passed. Each sourcestamp in the list has a different *codebase*
- The undocumented renderable `_ComputeRepositoryURL` is no longer imported to *buildbot.steps.source*. It is still available at `buildbot.steps.source.oldsource`.
- `IProperties.render` now returns a deferred, so any code rendering properties by hand will need to take this into account.
- `baseURL` has been removed in *SVN* to use just `repourl` - see bug #2066 (http://trac.buildbot.net/ticket/2066). Branch info should be provided with `Interpolate`.

```
from buildbot.steps.source.svn import SVN
factory.append(SVN(baseURL="svn://svn.example.org/svn/"))
```

can be replaced with

```
from buildbot.process.properties import Interpolate
from buildbot.steps.source.svn import SVN
factory.append(SVN(repourl=Interpolate("svn://svn.example.org/svn/%(src::branch)s
↪")))
```

and

```
from buildbot.steps.source.svn import SVN
factory.append(SVN(baseURL="svn://svn.example.org/svn/%%BRANCH%%/project"))
```

can be replaced with

```
from buildbot.process.properties import Interpolate
from buildbot.steps.source.svn import SVN
factory.append(SVN(repourl=Interpolate("svn://svn.example.org/svn/%(src::branch)s/
↪project")))
```

and

```
from buildbot.steps.source.svn import SVN
factory.append(SVN(baseURL="svn://svn.example.org/svn/", defaultBranch="branches/
↪test"))
```

can be replaced with

```
from buildbot.process.properties import Interpolate
from buildbot.steps.source.svn import SVN
factory.append(SVN(repourl=Interpolate("svn://svn.example.org/svn/%(src::branch:-
↪branches/test)s")))
```

- The `P4Sync` step, deprecated since 0.8.5, has been removed. The `P4` step remains.
- The `fetch_spec` argument to `GitPoller` is no longer supported. `GitPoller` now only downloads branches that it is polling, so specifies a refspec itself.
- The format of the changes produced by *SVNPoller* has changed: directory pathnames end with a forward slash. This allows the `split_file` function to distinguish between files and directories. Customized split functions may need to be adjusted accordingly.

- *FlattenList* has been deprecated in favor of *Interpolate*. *Interpolate* doesn't handle functions as keyword arguments. The following code using `WithProperties`

```
from buildbot.process.properties import WithProperties
def determine_foo(props):
    if props.hasProperty('bar'):
        return props['bar']
    elif props.hasProperty('baz'):
        return props['baz']
    return 'qux'
WithProperties('%(foo)s', foo=determine_foo)
```

  can be replaced with

```
from zope.interface import implementer
from buildbot.interfaces import IRenderable
from buildbot.process.properties import Interpolate
@implementer(IRenderable)
class determineFoo(object):
    def getRenderingFor(self, props):
        if props.hasProperty('bar'):
            return props['bar']
        elif props.hasProperty('baz'):
            return props['baz']
        return 'qux'
Interpolate('%s(kw:foo)s', foo=determineFoo())
```

Changes for Developers

- `BuildStep.start` can now optionally return a deferred and any errback will be handled gracefully. If you use `inlineCallbacks`, this means that unexpected exceptions and failures raised will be captured and logged and the build shut down normally.
- The helper methods `getState` and `setState` from `BaseScheduler` have been factored into `buildbot.util.state.StateMixin` for use elsewhere.

Slave

Features

Deprecations, Removals, and Non-Compatible Changes

- The `P4Sync` step, deprecated since 0.8.5, has been removed. The `P4` step remains.

Details

For a more detailed description of the changes made in this version, see the Git log itself:

```
git log v0.8.6..v0.8.7
```

Older Versions

Release notes for older versions of Buildbot are available in the master/docs/relnotes/ (https://github.com/buildbot/buildbot/blob/master/master/docs/relnotes/) directory of the source tree. Starting with version 0.8.6, they are also available under the appropriate version at http://buildbot.net/buildbot/docs.

Release Notes for Buildbot v0.8.6p1

The following are the release notes for Buildbot v0.8.6p1. Buildbot v0.8.6 was released on March 11, 2012. Buildbot v0.8.6p1 was released on March 25, 2012.

0.8.6p1

In addition to what's listed below, the 0.8.6p1 release adds the following.

- Builders are no longer displayed in the order they were configured. This was never intended behavior, and will become impossible in the distributed architecture planned for Buildbot-0.9.x. As of 0.8.6p1, builders are sorted naturally: lexically, but with numeric segments sorted numerically.
- Slave properties in the configuration are now handled correctly.
- The web interface buttons to cancel individual builds now appear when configured.
- The ForceScheduler's properties are correctly updated on reconfig - bug #2248 (http://trac.buildbot.net/ticket/2248).
- If a slave is lost while waiting for locks, it is properly cleaned up - bug #2247 (http://trac.buildbot.net/ticket/2247).
- Crashes when adding new steps to a factory in a reconfig are fixed - bug #2252 (http://trac.buildbot.net/ticket/2252).
- MailNotifier AttributeErrors are fixed - bug #2254 (http://trac.buildbot.net/ticket/2254).
- Cleanup from failed builds is improved - bug #2253 (http://trac.buildbot.net/ticket/2253).

Master

- If you are using the GitHub hook, carefully consider the security implications of allowing un-authenticated change requests, which can potentially build arbitrary code. See bug #2186 (http://trac.buildbot.net/ticket/2186).

Deprecations, Removals, and Non-Compatible Changes

- Forced builds now require that a *ForceScheduler* be defined in the Buildbot configuration. For compatible behavior, this should look like:

```
from buildbot.schedulers.forcesched import ForceScheduler
c['schedulers'].append(ForceScheduler(
                        name="force",
                        builderNames=["b1", "b2", ... ]))
```

  Where all of the builder names in the configuration are listed. See the documentation for the *much* more flexible configuration options now available.

- This is the last release of Buildbot that will be compatible with Python 2.4. The next version will minimally require Python-2.5. See bug #2157 (http://trac.buildbot.net/ticket/2157).
- This is the last release of Buildbot that will be compatible with Twisted-8.x.y. The next version will minimally require Twisted-9.0.0. See bug #2182 (http://trac.buildbot.net/ticket/2182).
- `buildbot start` no longer invokes make if a `Makefile.buildbot` exists. If you are using this functionality, consider invoking make directly.
- The `buildbot sendchange` option `--username` has been removed as promised in bug #1711 (http://trac.buildbot.net/ticket/1711).
- StatusReceivers' checkConfig method should now take an additional *errors* parameter and call its *addError* method to indicate errors.
- The Gerrit status callback now gets an additional parameter (the master status). If you use this callback, you will need to adjust its implementation.
- SQLAlchemy-Migrate version 0.6.0 is no longer supported. See *Buildmaster Requirements*.
- Older versions of SQLite which could limp along for previous versions of Buildbot are no longer supported. The minimum version is 3.4.0, and 3.7.0 or higher is recommended.
- The master-side Git step now checks out 'HEAD' by default, rather than master, which translates to the default branch on the upstream repository. See pull request 301 (https://github.com/buildbot/buildbot/pull/301).
- The format of the repository strings created by `hgbuildbot` has changed to contain the entire repository URL, based on the `web.baseurl` value in `hgrc`. To continue the old (incorrect) behavior, set `hgbuildbot.baseurl` to an empty string as suggested in the Buildbot manual.
- Master Side *SVN* Step has been corrected to properly use `--revision` when `alwaysUseLatest` is set to `False` when in the `full` mode. See bug #2194 (http://trac.buildbot.net/ticket/2194)
- Master Side *SVN* Step paramater svnurl has been renamed repourl, to be consistent with other master-side source steps.
- Master Side *Mercurial* step parameter `baseURL` has been merged with `repourl` parameter. The behavior of the step is already controlled by `branchType` parameter, so just use a single argument to specify the repository.
- Passing a *buildbot.process.buildstep.BuildStep* subclass (rather than instance) to `buildbot.process.factory.BuildFactory.addStep` has long been deprecated, and will be removed in version 0.8.7.
- The *hgbuildbot* tool now defaults to the 'inrepo' branch type. Users who do not explicitly set a branch type would previously have seen empty branch strings, and will now see a branch string based on the branch in the repository (e.g., *default*).

Changes for Developers

- The interface for runtime access to the master's configuration has changed considerably. See *Configuration* for more details.
- The DB connector methods `completeBuildset`, `completeBuildRequest`, and `claimBuildRequest` now take an optional `complete_at` parameter to specify the completion time explicitly.
- Buildbot now sports sourcestamp sets, which collect multiple sourcestamps used to generate a single build, thanks to Harry Borkhuis. See pull request 287 (https://github.com/buildbot/buildbot/pull/287).

- Schedulers no longer have a `schedulerid`, but rather an `objectid`. In a related change, the `schedulers` table has been removed, along with the `buildbot.db.schedulers.SchedulersConnectorComponent.getSchedulerId` method.
- The Dependent scheduler tracks its upstream buildsets using `buildbot.db.schedulers.StateConnectorComponent`, so the `scheduler_upstream_buildsets` table has been removed, along with corresponding (undocumented) `buildbot.db.buildsets.BuildsetsConnector` methods.
- Errors during configuration (in particular in `BuildStep` constructors), should be reported by calling *`buildbot.config.error`*.

Features

- The IRC status bot now display build status in colors by default. It is controllable and may be disabled with useColors=False in constructor.
- Buildbot can now take advantage of authentication done by a front-end web server - see pull request 266 (https://github.com/buildbot/buildbot/pull/266).
- Buildbot supports a simple cookie-based login system, so users no longer need to enter a username and password for every request. See the earlier commits in pull request 278 (https://github.com/buildbot/buildbot/pull/278).
- The master-side SVN step now has an *export* method which is similar to *copy*, but the build directory does not contain Subversion metdata. (bug #2078 (http://trac.buildbot.net/ticket/2078))
- `Property` instances will now render any properties in the default value if necessary. This makes possible constructs like

```
command=Property('command', default=Property('default-command'))
```

- Buildbot has a new web hook to handle push notifications from Google Code - see pull request 278 (https://github.com/buildbot/buildbot/pull/278).
- Revision links are now generated by a flexible runtime conversion configured by *`revlink`* - see pull request 280 (https://github.com/buildbot/buildbot/pull/280).
- Shell command steps will now "flatten" nested lists in the `command` argument. This allows substitution of multiple command-line arguments using properties. See bug #2150 (http://trac.buildbot.net/ticket/2150).
- Steps now take an optional `hideStepIf` parameter to suppress the step from the waterfall and build details in the web. (bug #1743 (http://trac.buildbot.net/ticket/1743))
- `Trigger` steps with `waitForFinish=True` now receive a URL to all the triggered builds. This URL is displayed in the waterfall and build details. See bug #2170 (http://trac.buildbot.net/ticket/2170).
- The `master/contrib/fakemaster.py` script allows you to run arbitrary commands on a slave by emulating a master. See the file itself for documentation.
- MailNotifier allows multiple notification modes in the same instance. See bug #2205 (http://trac.buildbot.net/ticket/2205).
- SVNPoller now allows passing extra arguments via argument `extra_args`. See bug #1766 (http://trac.buildbot.net/ticket/1766)

Slave

Deprecations, Removals, and Non-Compatible Changes

- BitKeeper support is in the "Last-Rites" state, and will be removed in the next version unless a maintainer steps forward.

Features

Details

For a more detailed description of the changes made in this version, see the Git log itself:

```
git log buildbot-0.8.5..buildbot-0.8.6
```

Older Versions

Release notes for older versions of Buildbot are available in the master/docs/relnotes/ (https://github.com/buildbot/buildbot/blob/master/master/docs/relnotes/) directory of the source tree, or in the archived documentation for those versions at http://buildbot.net/buildbot/docs.

Note that Buildbot-0.8.11 was never released.

CHAPTER 6

Indices and Tables

- genindex
- `cfg`
- `sched`
- `chsrc`
- `step`
- `reporter`
- `cmdline`
- `msg`
- `event`
- `rtype`
- `rpath`
- `raction`
- modindex
- search

CHAPTER 7

Copyright

Buildmaster Configuration Index

Scheduler Index

Change Source Index

Build Step Index

T

V

Reporter Target Index

Build Worker Index

Command Line Index

Data API Event Index

B

REST/Data API Resource Type Index

REST/Data API Path Index

/

REST/Data API Actions Index

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y