Cyclid Documentation

Release 0.3.1

Cyclid

Jul 15, 2017

User Documentation

This is the documentation for the Cyclid CI system. It covers everything from installing & configuring Cyclid, to writing Cyclid jobs and using the Cyclid command line client.

This documentation and Cyclid itself are Open Source; if you think this documentation could be improved, we welcome contributions.

CHAPTER 1

Contents

Cyclid Server

Installing

These instructions cover installing & configuring your own Cyclid server. If you don't want to go through the trouble of doing that, you might want to try out the *Cyclid Vagrant instance* instead.

Prerequisites

1. A machine running Linux. Cyclid has been developed to run on Ubuntu 14.04, but other distributions and versions should work just as well.
2. Ruby 2.x. Cyclid has been developed against Ruby 2.3 but any 2.x version should work. If you use Ubuntu you might want to to consider installing Ruby 2.3 from the Brightbox RubyNG PPA.
3. A SQL database. Cyclid uses ActiveRecord, so in theory any database that is supported by ActiveRecord should work. I use MySQL so these instructions reflect that. You'll need a new (empty) database and a user which has create & update permissions.

4. A cloud provider, or container platform, that is supported by Cyclid or an available plugin.

Dependencies

Cyclid itself has some dependencies. Some it can install & configure itself, some will require you to install & configure them.

The Cyclid Server has the following additional dependencies

- Sidekiq
- Redis server

Sidekiq is used to run jobs in the background, and in turn depends on Redis to queue data. Cyclid does not use Redis directly.

Application server

You'll probably want to run Cyclid behind a reverse proxy, through an application server. These instructions cover installing & configuring Unicorn and Nginx with Cyclid.

Installing on Ubuntu

These instructions assume you are installing Cyclid on an Ubuntu 14.04 machine, with the Brightbox RubyNG PPA enabled, and that both the Cyclid API server & Cyclid UI server will run alongside each other on the same server.

Install dependencies

Install the dependencies for installing the Gems that Cyclid requires:

```
$ sudo apt-get install ruby2.3 ruby2.3-dev build-essential cmake mysql-client␣
↪libmysqlclient-dev redis-server
$ sudo mkdir -p /var/lib/cyclid /var/run/cyclid
```

Install Cyclid

```
$ sudo gem install cyclid
```

Rubygems will install the Ruby dependencies for Cyclid automatically, which may include building native extensions for some Gems.

You will also need to install any dependencies for your preferred database. For example, if you are using a MySQL database, we recommend the `mysql2` adaptor:

```
$ sudo gem install mysql2
```

If you're not using a MySQL database you should consult the documentation for the appropriate ActiveRecord adapter for information on which dependencies you'll need to install.

Create the Cyclid configuration file

```
$ sudo mkdir /etc/cyclid
$ sudo vim /etc/cyclid/config
```

The configuration file specifies the database connection and a few configuration options to use:

```
server:
  database: mysql2://<username>:<password>@<server>/<database>
  log: stderr
  dispatcher: local
  builder: <Your preferred Builder plugin>
```

Replace the MySQL username, password & hostname with ones suitable for your database server. The user should have create & update privileges. The database should already exist on the server (I.e. `create database <database>;` should already have been run).

See the complete *documentation for the configuration file* for more information.

Create the Cyclid database

Warning: `cyclid-db-init` will drop any existing data from your database! Do NOT run `cyclid-db-init` if you have an existing database, as you WILL lose your data.

```
$ cyclid-db-init
```

The database schema will be populated and the initial Admin user & organization will be created. The initialization process will create a random password & HMAC secret for the Admin user and print them out at the end E.g.

```
Admin secret: fe150f3939ed0419f32f8079482380f5cc54885a381904c15d861e8dc5989286
Admin password: 9u%Y5ySl
```

Make a note of the secret & password as you will require them to log in to Cyclid!

Configure Sidekiq

Create a file that Sidekiq can use to run Cyclid background jobs:

```
$ echo "require 'cyclid/app'" | sudo tee /var/lib/cyclid/sidekiq.rb
```

Now start Sidekiq:

```
$ sudo sidekiq -e production -d -P /var/run/cyclid/sidekiq.pid -L /var/log/sidekiq.
↪log -r /var/lib/cyclid/sidekiq.rb
```

Configuring Unicorn & Nginx

Cyclid is a Ruby Sinatra application and can be run under any Rack application server. The following instructions cover configuring Cyclid to run with the Unicorn application server with Nginx as a reverse proxy.

Prerequisites

Install Nginx & Unicorn

```
$ sudo apt-get install nginx
$ sudo gem install unicorn
```

Create the application directory

```
$ sudo mkdir -p /var/lib/cyclid
```

Create the log directory

```
$ sudo mkdir /var/log/cyclid
```

Configure Unicorn for the Cyclid API server

Create a Rack configuration file for Unicorn

```
$ sudo vim /var/lib/cyclid/config.ru
```

This is a standard Rack configuration file. The most basic configuration for Cyclid is:

```
require 'sinatra'
require 'cyclid/app'

run Cyclid::API::App
```

Configure Unicorn

Create the Unicorn configuration file:

```
$ sudo vim /var/lib/cyclid/unicorn.rb

working_directory "/var/lib/cyclid"
pid "/var/run/unicorn.cyclid-api.pid"

stderr_path "/var/log/cyclid/unicorn.cyclid-api.log"
stdout_path "/var/log/cyclid/unicorn.cyclid-api.log"

listen "/var/run/unicorn.cyclid-api.sock"

worker_processes 4
timeout 10
```

Start Unicorn

```
$ sudo unicorn -D -E production -c /var/lib/cyclid/unicorn.rb
```

Configure Nginx

You must configure Nginx to act as a reverse proxy to Unicorn.

Cyclid API Nginx configuration

```
$ sudo vim /etc/nginx/sites-available/cyclid-api

upstream cyclid-api {
  server unix:/var/run/unicorn.cyclid-api.sock fail_timeout=0;
}

server {
  listen 8361;
  server_name cyclid.example.com;
  root /var/lib/cyclid;

  try_files $uri @cyclid-api;

  location @cyclid-api {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://cyclid-api;
  }

  error_page 500 502 503 504 /500.html;
  client_max_body_size 4G;
  keepalive_timeout 10;
}
```

Replace the `server_name` with something more suitable for your installation. Restart Nginx Enable the Nginx configuration & restart Nginx:

```
$ sudo rm /etc/nginx/sites-enabled/default
$ sudo ln -s /etc/nginx/sites-available/cyclid-api /etc/nginx/sites-enabled/cyclid-api
$ sudo service nginx restart
```

You should now be able to configure your client to connect to the server, using the admin password & secret that were provided when you created the database.

Note: You may want to run Sidekiq & Unicorn under a process supervisor, rather than starting them directly as daemons. We prefer Runit for this but any process supervisor or init scheme should work.

Upgrading

These instructions cover upgrading an existing Cyclid server. You should consult the release notes for the version you new installing for any instructions which are specific to that version.

Upgrade Cyclid

```
$ sudo gem install cyclid
```

Rubygems will install the Ruby dependencies for Cyclid automatically, which may include building native extensions for some Gems.

You should also install any newer dependencies for your preferred database. For example, if you are using a MySQL database, we recommend the `mysql2` adaptor:

```
$ sudo gem install mysql2
```

If you're not using a MySQL database you should consult the documentation for the appropriate ActiveRecord adapter for information on which dependencies you'll need to install.

You should upgrade any Cyclid plugins which you have installed E.g.

```
$ sudo gem install cyclid-example-plugin
```

Migrate the Cyclid database

Warning: You should back up any databses before running `cyclid-db-migrate`!

```
$ cyclid-db-migrate
```

Any required migrations will be applied to the Cyclid database schema. You only need to run `cyclid-db-migrate` once per. database.

Restart Cyclid & Sidekiq

Restart both the Cyclid & Sidekiq processes.

Cyclid UI Server

Installation

These instructions cover installing & configuring your own Cyclid UI server. If you don't want to go through the trouble of doing that, you might want to try out the *Cyclid Vagrant instance* instead.

Prerequisites

1. A machine running Linux. Cyclid has been developed to run on Ubuntu 14.04, but other distributions and versions should work just as well.
2. Ruby 2.x. Cyclid has been developed against Ruby 2.3 but any 2.x version should work. If you use Ubuntu you might want to to consider installing Ruby 2.3 from the Brightbox RubyNG PPA.

Dependencies

The server for the Cyclid user interface has an optional (but highly recommended) dependency on Memcached.

Memcached

Memcached is an in-memory object cache. It is used by the Cyclid UI server to cache user details. You can run the Cyclid UI server without Memcached but this will result in more Cyclid API calls to retrieve user data.

Application server

You'll probably want to run the Cyclid UI behind a reverse proxy, through an application server. These instructions cover installing & configuring Unicorn and Nginx with the Cyclid UI.

Installing on Ubuntu

These instructions assume you are installing Cyclid on an Ubuntu 14.04 machine, with the Brightbox RubyNG PPA enabled, and that both the Cyclid API server & Cyclid UI server will run alongside each other on the same server.

Install dependencies

Install the dependencies for installing the Gems that Cyclid requires:

```
$ sudo apt-get install ruby2.3 ruby2.3-dev build-essential cmake memcached
$ sudo mkdir -p /var/lib/cyclid /var/run/cyclid
```

Install Cyclid

```
$ sudo gem install cyclid-ui
```

Rubygems will install the Ruby dependencies for Cyclid automatically, which may include building native extensions for some Gems.

Create the Cyclid configuration file

```
$ sudo mkdir /etc/cyclid
$ sudo vim /etc/cyclid/config
```

The configuration file specifies the API connection details and a few configuration options to use:

```
manage:
  api:
    server: http://localhost:8361
    client: http://<hostname>:8361
```

Replace the API server hostname with the name of the API server, which should be reachable by the client.

See the complete *documentation for the configuration file* for more information.

Configuring Unicorn & Nginx

Cyclid is a Ruby Sinatra application and can be run under any Rack application server. The following instructions cover configuring Cyclid to run with the Unicorn application server with Nginx as a reverse proxy.

Prerequisites

Install Nginx & Unicorn

```
$ sudo apt-get install nginx
$ sudo gem install unicorn
```

Create the application directory

```
$ sudo mkdir -p /var/lib/cyclid-ui
```

Create the log directory

```
$ sudo mkdir -p /var/log/cyclid-ui/
```

Configure Unicorn for the Cyclid UI server

Create a Rack configuration file for Unicorn

```
$ sudo vim /var/lib/cyclid-ui/config.ru
```

This is a standard Rack configuration file. The most basic configuration for Cyclid is:

```
require 'sinatra'
require 'cyclid_ui/app'

run Cyclid::UI::App
```

Configure Unicorn

Create the Unicorn configuration file:

```
$ sudo vim /var/lib/cyclid-ui/unicorn.rb

working_directory "/var/lib/cyclid-ui"
pid "/var/run/unicorn.cyclid-ui.pid"

stderr_path "/var/log/cyclid-ui/unicorn.cyclid-ui.log"
stdout_path "/var/log/cyclid-ui/unicorn.cyclid-ui.log"

listen "/var/run/unicorn.cyclid-ui.sock"

worker_processes 4
timeout 10
```

Start Unicorn

```
$ sudo unicorn -D -E production -c /var/lib/cyclid-ui/unicorn.rb
```

Make the static assets available

The Cyclid UI server includes some static files (Images, Javascript, CSS etc.) which Nginx must be able to find. The easiest way to do this is to create a symbolic link from the Cyclid UI application directory to the assets; Cyclid UI includes a command to help you find the location:

```
$ sudo ln -s $(cyclid-ui-assets) /var/lib/cyclid-ui/public
```

Configure Nginx

You must configure Nginx to act as a reverse proxy to Unicorn.

Cyclid UI Nginx site configuration

```
$ sudo vim /etc/nginx/sites-available/cyclid-ui

upstream cyclid-ui {
  server unix:/var/run/unicorn.cyclid-ui.sock fail_timeout=0;
}

server {
  listen 80;

  server_name cyclid.example.com;
  root /var/lib/cyclid-ui;

  location ~* \.(js|css|png|jpg|jpeg|gif|ico)$ {
    root /var/lib/cyclid-ui/public;
    expires max;
    add_header Cache-Control public;
    log_not_found off;
  }
  try_files $uri @cyclid-ui;

  location @cyclid-ui {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://cyclid-ui;
  }

  error_page 500 502 503 504 /500.html;
  client_max_body_size 4G;
  keepalive_timeout 10;
}
```

Replace the `server_name` with something more suitable for your installation. Restart Nginx Enable the Nginx configuration & restart Nginx:

```
$ sudo rm /etc/nginx/sites-enabled/default
$ sudo ln -s /etc/nginx/sites-available/cyclid-ui /etc/nginx/sites-enabled/cyclid-ui
$ sudo service nginx restart
```

You should now be able to configure your client to connect to the server, using the admin password & secret that were provided when you created the database.

Note: You may want to run Unicorn under a process supervisor, rather than starting it directly as a daemon. We prefer Runit for this but any process supervisor or init scheme should work.

Configuration file

Cyclid itself is very simple to configure: there's just a single configuration file, `/etc/cyclid/config`. The configuration file is in YAML format and contains a few simple configuration options. The Cyclid API server & Cyclid UI server use the same configuration file, with one section for each server:

```
server:
  database: "mysql://dbuser:dbpass@localhost/cyclid"
  log: /var/log/cyclid.log
  dispatcher: local
  builder: mist
manage:
  log: /var/log/cyclid-ui.log
  memcached: localhost:11211
  api: http://localhost:8361
```

Cyclid API server options

database	Provides the database connection details.
log	The path to the log file.
dis-patcher	Selects the Dispatcher plugin you want to use. This is almost always `local`.
builder	Selects the Builder plugin you want to use. In this example we're using the `mist` plugin, which manages LXC containers for Cyclid.

The database connection details in the Cyclid configuration file can be any valid ActiveRecord connection, and can either be a URL or a list of individual connection details. For example, if you wanted to connect to a local MySQL server via. a UNIX socket, you could configure Cyclid with

```
database:
  adapter: mysql
  host: localhost
  socket: /var/run/mysql.sock
  username: cyclid
  database: cyclid
```

Plugins

Plugins can also have their own server configuration. Plugin configurations are placed under the `plugin` namespace, with each plugin having its own configuration object.

For example, the SMTP Action plugin can be configured to use an SMTP relay to send emails. An example cofiguration may look something like:

```
server:
  ...
  plugins:
    email:
      server: smtp.example.com
      port: 25
```

You should refer to the documentation for each plugin for the full list of configuration options it supports.

Cyclid UI server options

log	The path to the log file.
memcached	The Memcached connection details.
api	Provides the Cyclid API connection details.

The API connection details are used both the Cyclid UI server and the client (web browser) to connect to the API server. Sometimes, the address that the server should use and the address that the client should use are different E.g. an internal and external address. If required, you can specify two different URLs:

```
api:
  server: http://internal.example.com:8361
  client: http://external.example.com:8361
```

Vagrant

If you just want to try Cyclid, you can download a pre-built Vagrant box which contains everything you need, already installed and configured. The box comes pre-configured with Cyclid and its dependencies, along with LXC and Mist to create and manage containers for your build hosts. You can start it up and begin using it straight away.

If you don't already have it, you should download and install Vagrant. You'll also need a virtual machine hypervisor; right now the Cyclid Vagrant box only support VirtualBox, so you'll also need to download and install it if you don't have it.

Creating a Cyclid Vagrant instance

Create a working directory somewhere and tell Vagrant to download & start the Cyclid box:

```
$ mkdir $HOME/Cyclid-Vagrant
$ vagrant init Liqwyd/Cyclid
$ vagrant up --provider virtualbox
```

When the Cyclid box starts it will write a Cyclid client configuration file for you into the config directory. You should copy this file to the client configuration directory and tell the Cyclid client to use it.

```
$ cp config/vagrant $HOME/.cyclid
$ cyclid organization use vagrant
```

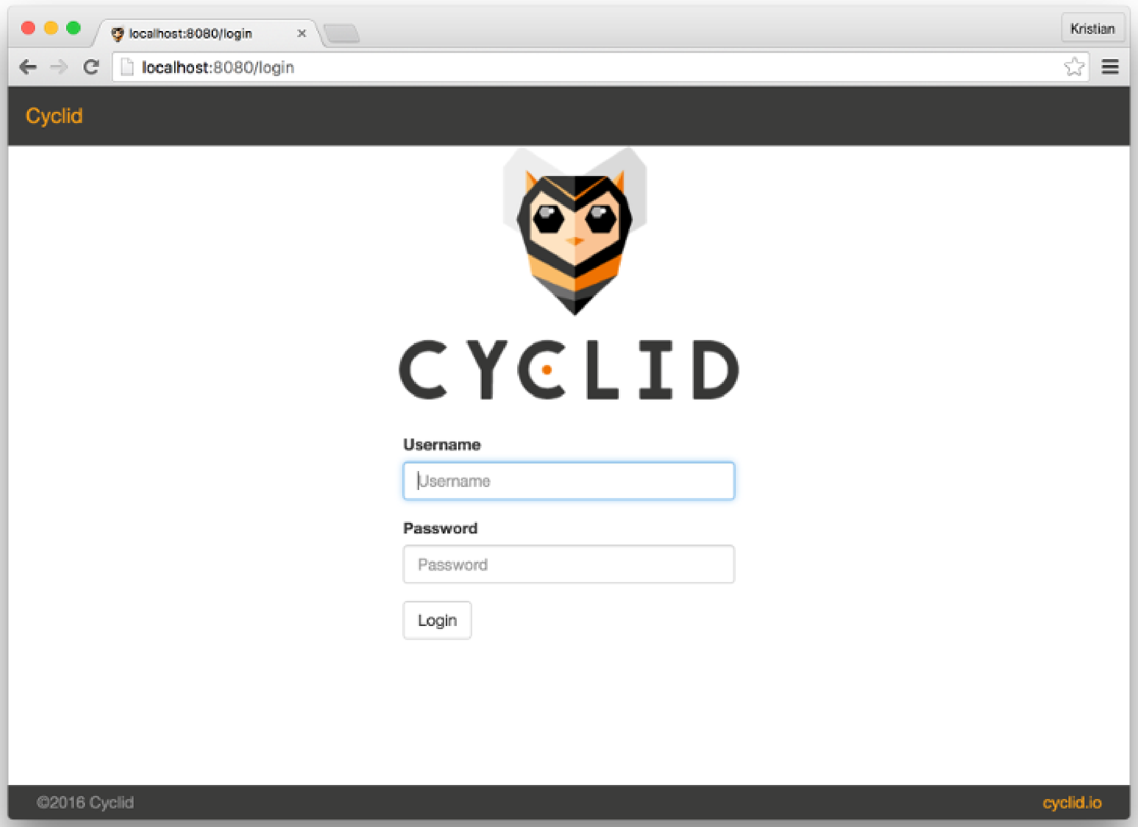
You'll probably want to set a password so that you can log in to the Cyclid User Interface.

```
$ cyclid user passwd
Password:
Confirm password:
```

See the *Cyclid client documentation* for more information on installing & configuring the command line client.

Using the Vagrant instance

The Vagrant instance is configured with the Cyclid API server running on port 8361, and the Cyclid User Interface on port 8080. Once you've changed your password, open a browser tab or window and connect to the Cyclid UI at http://localhost:8080/login. Log in using the username "admin" and the password you set.



Cyclid Client

Note: The client is still in development. Some features are missing entirely and the interface is liable to change at any moment.

Installation

```
$ gem install cyclid-client
```

Configuration

Configuration file format

The configuration file is a simple YAML file with the following options.

Option	Description
url	The URL of the Cyclid server.
server	The hostname of the Cyclid server.
port	The port to use for connections to the Cyclid server.
tls	Use TLS (HTTPS) for connections to the Cyclid server.
organization	The organization name.
username	The username that is associated with the organization.
secret	The HMAC signing secret for the user.

You can specify the server URL with *either* the `url` options or seperate `server`, `port` and `tls` options.

Examples

```
server: cyclid.example.com
port: 8361
tls: true
organization: my_organization
username: user
secret: b1fc42ef648b4407f30dc77f328dbb86b03121fb15aba256497ef97ec9a3cd02
```

```
url: https://cyclid.example.com:8361
organization: my_organization
username: user
secret: b1fc42ef648b4407f30dc77f328dbb86b03121fb15aba256497ef97ec9a3cd02
```

Authenticating with a server

You can use the `user authenticate` command to easily authenticate with a Cyclid server and automatically create your client configuration file(s).

```
$ cyclid user authenticate -s https://cyclid.example.com
Username: example
Password:
Authenticating example with https://cyclid.example.com:8361
Creating configuration file for organization example
```

One configuration file will be created for every organization that your user is a member of.

Switching between configurations

The client uses configuration files under `$HOME/.cyclid` You can have multiple configuration files and switch between the with the `organization use` command.

For example, of your user belongs to two organizations, you can have one configuration file for each organization E.g.

```
$HOME/.cyclid/organization_one
$HOME/.cyclid/organization_two
```

and then use the command `cyclid organization use organization_one` to select it as the current configuration.

To find the list of available configurations, use the `organization list` command.

Specifying a configuration file

You can use the `--config` or `-c` option to specify the path to a configuration file to use instead of the current configuration that has been set with the `organization use` command.

Commands

Cyclid commands are grouped under the following categories:

Group	Description
user	Manage your current user
organization	Manage your current organization
job	Manage and submit jobs
stage	Manage stage definitions
secret	Create secrets
admin	Administrator commands

User commands

user show

Display your current user details.

```
$ cyclid user show
Username: bob
Email: bob@example.com
Organizations
    example
```

user passwd

Change your current users password. The user password is only used for HTTP Basic authentication.

```
$ cyclid user passwd
Password: <enter new password>
Confirm password: <re-enter new password>
```

user modify

Change your current users email address, HMAC secret and/or password. You can pass the following options:

Option	Short option	Description
–email	-e	Change your email address
–secret	-s	Change your HMAC secret
–password	-p	Change your email address

Unlike the interactive `user passwd` command you can use `user modify` and pass your new password on the command line.

Your HMAC secret should ideally be a suitably long (at least 256 bit) and random string, which you should keep secure in your Cyclid configuration file. After changing your HMAC secret you will need to update your configuration file with the new secret before you can run any other Cyclid commands.

```
# Change your email
$ cyclid user modify --email robert@example.com
# Change your HMAC secret
$ cyclid user modify --secret␣
↪b072d8b51cec2755145c401b9249a60ebd89b4704eeebc5b6805ba682d7fac53
```

user authenticate

Authenticate with a Cyclid server, using your username & password, and create a configuration file in `$HOME/.cyclid` for each organization that your user belongs to.

You can pass the following options:

Option	Short option	Description
–server	-s	URL of the Cyclid server.
–username	-u	Your username.

If you do not pass a server URL the default of `https://api.cyclid.io` is assumed.

If you do not pass your username then you will be asked to enter it.

Organization commands

organization list

Lists all of the available organization configurations on your local machine.

```
$ cyclid org list
admins
    Server: http://example.com
    Organization: admins
    Username: admin
example
    Server: http://example.com
    Organization: example
    Username: bob
```

organization show

Display the details of your currently selected organization, including the list of organization members and its public key.

```
$ cyclid org show
Name: example
Owner Email: bob@example.com
Public Key: -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA8P8CMCfYLqMfAGq/pWyV
r92w8TMo3A5Irf1iZsFko42WGgIdOAnDuguODUFIzWmyrKm1WL0+V403j914gCRL
8Zi+To3qbQtLaD4etiP/p3Z6qEHt77rn67kRxKjpcyiHkwOtQxMO5VCXlYCvEnDz
0Rn2cq9VutrjrZcOjNCk7AkUtTZ3arkntYPaNBtPDpQz1x3dGdumSgVBUx1dcaqE
khLVc1SB1mqPNcIKoqIQF5oNGBdNWA6oBxk5CNj1GfpXayawixjgvq+tkJo3mDbu
F6UzJ4UGzbpC3EYqCkEByNOXv4J2aYaOjChFUiHn1XcSUVZHkrzFcb47Pif1wshi
lwIDAQAB
-----END PUBLIC KEY-----
```

```
Members:
    bob
    lucy
    dave
    leslie
```

organization use

Select an organization configuration to use by default. Pass a name of an organization from `organization list` to select it as your current configuration. If you do not pass a new organization name, the name of the currently selected organization is shown.

```
# Show the currently selected organization
$ cyclid organization use
example
# Select the 'admins' organization
$ cyclid organization use admins
```

organization modify

Modify the current organization. This command can only be used by organization admins.

You can pass the following options:

Option	Short option	Description
–email	-e	Change the owner email address

```
# Change the organization owner email address
$ cyclid organization modify --email lucy@example.com
```

organization member

The `organization member` command has a series of sub-commands which are used to manage users which belong to the organization.

organization member list

List all of the users who are members of the current organization.

```
$ cyclid organization member list
bob
lucy
dave
leslie
```

organization member show

Display the user details of an organization member, including the user permissions.

```
$ cyclid organization member show bob
Username: bob
Email: bob@example.com
Permissions
    Admin: false
    Write: true
    Read: true
```

organization member add

Add user(s) to the current organization. You must pass at least one username.

Users are added without any permissions set. You can use the `organization member permission` command to modify the user permissions after they have been added to the organization.

```
# Add a single user, 'bob', to the organization
$ cyclid organization member add bob
# Add multiple users, 'bob' and 'lucy', to the organization
$ cyclid organization member add bob lucy
```

organization member permission

Modify a users permissions for the organization. You must pass the username and the level of access you want the user to have. This can be one of:

- admin
- write
- read
- none

The 'admin' permission implies 'write', and the 'write' permission implies 'read'.

With 'none' the user remains an organization member but can not interact with it. See the `organization member remove` command if you want to actually remove a user from the organization.

```
# Give the user 'bob' read-only access to the organization
$ cyclid organization member permission bob read
# Give the user 'lucy' admin permissions for the organization
$ cyclid organization member permission lucy admin
```

organization member remove

Remove user(s) from the current organization. You must pass at least one username. By default the `organization member remove` command will ask you to confirm the removal first; you can over-ride this with the `--force/-f` option to force removal without confirmation.

Option	Short option	Description
–force	-f	Do not ask for confirmation before removing the user

```
# Remove the user 'bob' from the organization without asking for confirmation
$ cyclid organization member remove bob --force
```

organization config

The `organization config` command has a series of sub-commands which are used to get and set plugin configurations for your organization.

organization config show

Show the current organization specific configuration for a plugin. You must specify both the plugin type, and the plugin name.

```
# Show the current configuration for the Github API plugin
$ cyclid organization config show api github
Repository OAuth tokens
    None
Github HMAC signing secret: Not set
```

organization config edit

Modify the organization specific configuration for a plugin. You must specify both the plugin type, and the plugin name.

The `config edit` command expects the `$EDITOR` environment variable to be set to the path of a valid text editor that it can start.

```
$ cyclid organization config edit api github
# The Github plugin configuration is loaded in your text editor
```

Job commands

job stats

Show the total number of jobs.

```
$ cyclid job stats
Total jobs: 2
```

job list

List the details of all jobs.

```
$ cyclid job list
Name: test_job
  Job: 1
  Version: 1.0.0
Name: example_job
  Job: 2
  Version: 1.0.0
```

job show

Show the details of a job. You must pass a valid job ID.

```
$ cyclid job show 7
Job: 7
Name: test_job
Version: 1.0.0
Started: Thu Apr 21 16:40:57 2016
Ended: Thu Apr 21 16:41:04 2016
Status: Succeeded
```

job status

Show the status of a job. You must pass a valid job ID.

```
$ cyclid job status 7
Status: Succeeded
```

job log

Show the log from a job. You must pass a valid job ID.

```
$ cylid job log 7
2016-04-21 16:40:57 +0100 : Obtaining build host...
2016-04-21 16:41:47 +0100 : Preparing build host...
==============================================================================
2016-04-21 16:41:47 +0100 : Job started. Context: {"job_id"=>7, "job_name"=>"test_job
↪", "job_version"=>"1.0.0", "organization"=>"example", "os"=>"ubuntu_trusty", "name
↪"=>"mist-3c04c6134a3f776cbe8e91e396d4dace", "host"=>"192.168.1.247", "username"=>
↪"build", "workspace"=>"/home/build", "password"=>nil, "key"=>"~/.ssh/id_rsa_build",
↪"server"=>"build01", "distro"=>"ubuntu", "release"=>"trusty"}
------------------------------------------------------------------------------
2016-04-21 16:41:47 +0100 : Running stage example v1.0.0
...
```

job submit

Submit a Cyclid job file to be run. The `job submit` command expects to be passed a path to a valid Cyclid job file in either JSON or YAML format.

The `job submit` command will attempt to automatically detect the format of the job file. You can use the `--json/-j` or `--yaml/-y` options to over-ride the format detection.

The job ID for the job will be shown once the job has been submitted. You can then check the status of the job with the `job status`, `job show` and `job log` commands.

Option	Short option	Description
–json	-j	Parse the file as JSON
–yaml	-y	Parse the file as YAML

```
$ cyclid job submit job.yml
Job: 8
```

job lint

Lint (verify) a Cyclid job file and provide information about any Warning & Errors that are found. This can be used to ensure that your Cyclid job file is syntactically correct before you attempt to run it, allowing the user to catch potential problems quickly.

Option	Short option	Description
–json	-j	Parse the file as JSON
–yaml	-y	Parse the file as YAML

```
$ cyclid job lint job.yml
Error: The Job does not have a name.
Warning: No version is defined for the Job. The default of 1.0.0 will be used.
Warning: No environment is defined. Defaults will apply.
Warning: No version is given for the Stage 'hello-world'. The default of 1.0.0 will␣
↪be used.
Warning: A Stage in the Sequence does not specify a version for the Stage 'hello-world
↪'. The latest will always be used.

Errors: 1
Warnings: 4
```

job format

Reformat, or convert, a Cyclid job file into standardised JSON or YAML.

Option	Short option	Description
–json	-j	Parse the input file as JSON
–yaml	-y	Parse the input file as YAML
–jsonout	-J	Output JSON
–yamlout	-Y	Output YAML

```
$ cyclid job format job.yml --jsonout
{
  "name":"job",
  "environment":{},
  "stages":[
...
```

Stage commands

stage list

List all of the stages, and each version of each stage, that are defined for the organization.

```
$ cyclid stage list
example v0.0.1
example v0.0.2
example v0.1.0
success v1.0.0
success v1.0.1
failure v1.0.0
```

stage show

Show the details of a stage.

```
$ cyclid stage show example
Name: example
Version: 0.0.1
Steps
        Action: command
        Cmd: echo
        Args: ["'hello", "world'"]
Name: example
Version: 0.0.2
Steps
        Action: command
        Cmd: echo
        Args: ["'hello", "world'"]
Name: example
Version: 0.1.0
Steps
        Action: command
        Cmd: echo
        Args: ["'Hello", "universe'"]
```

stage create

Create a new stage, or a new version of a stage, from a stage definition in a file. The `stage create` command expects to be passed a path to a valid Cyclid stage definition file in either JSON or YAML format.

The `stage create` command will attempt to automatically detect the format of the stage file. You can use the `--json/-j` or `--yaml/-y` options to over-ride the format detection.

Option	Short option	Description
–json	-j	Parse the file as JSON
–yaml	-y	Parse the file as YAML

```
$ cyclid stage create stage.yml
```

stage edit

Edit a stage definition that exists on the server. Note that individual versions of a stage are immutable; once a version of a stage has been created it can not be deleted or modified. However, you can create a new version.

If you attempt to create a stage with the same name & version of an existing stage, the command will fail.

The `stage edit` command expects the `$EDITOR` environment variable to be set to the path of a valid text editor that it can start.

```
$ cyclid stage edit example
# The 'example' stage definition is loaded in your text editor
```

Secret commands

secret encrypt

Encrypts a string with the organizations public key. You can then add the encrypted secret to the `secrets` section of a Cyclid job definition.

```
$ cyclid secret encrypt
Secret: <Enter the secret to be encrypted>
Secret:␣
↪uzegcZfXPuj4KNo+EpP928cgPW37gMDhdKw9OoCE0YXKWWtJ+kJIHzLyOGrF7p6dDJ3cWNZhEDADINJqsYMoa$bSAdT5Gx+lAo
↪VnD9JN9OObwGTaycb1XryZWeU/Hfr45Y/
↪HObUnFhE+W+IHbAswMBO9bs3DogF672DFXkTtt+b0XW6ttyHGIqUqxoo8zFBEaDQlxa5oaW3iXSmcA+rrfolPO6gl9wI4PxH2k
↪3XzBQ==
```

Admin commands

Admin commands are used for server wide configuration, and are only available to server admins I.e. users who are members of the 'admins' group.

Admin commands are grouped under the following categories:

Group	Description
organization	Manage organizations
user	Manage users
job	Manage jobs

Admin organization commands

The `admin organization` command has a series of sub-commands which are used to manage organizations.

admin organization list

List all of the organizations on the server.

```
$ cyclid admin organization list
admins
example
initech
```

admin organization show

Show the details of an organization, including the owner email address, the list of organization members and its public key.

```
$ cyclid admin organization show example
Name: example
Owner Email: bob@example.com
Public Key: -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA8P8CMCfYLqMfAGq/pWyV
r92w8TMo3A5Irf1iZsFko42WGgIdOAnDuguODUFIzWmyrKm1WL0+V403j914gCRL
8Zi+To3qbQtLaD4etiP/p3Z6qEHt77rn67kRxKjpcyiHkwOtQxMO5VCXlYCvEnDz
0Rn2cq9VutrjrZcOjNCk7AkUtTZ3arkntYPaNBtPDpQz1x3dGdumSgVBUx1dcaqE
khLVc1SB1mqPNcIKoqIQF5oNGBdNWA6oBxk5CNj1GfpXayawixjgvq+tkJo3mDbu
```

```
F6UzJ4UGzbpC3EYqCkEByNOXv4J2aYaOjChFUiHn1XcSUVZHkrzFcb47Pif1wshi
lwIDAQAB
-----END PUBLIC KEY-----
Members:
    bob
    lucy
    dave
    leslie
```

admin organization create

Create a new organization. You must supply the name of the new organization, and the organization owners email. You may also optionally add a user as the initial organization admin using the `--admin/-a` option.

Option	Short option	Description
–admin	-a	Username of the initial organization admin

```
# Create the 'example' organization with no initial admin
$ cyclid admin organization create example bob@example.com
# Create the 'initech' organization with the user 'lucy' as the initial admin
$ cyclid admin organization create initech lucy@example.com --admin lucy
```

admin organization modify

Change an organizations owner email address or organization membership. You can pass the following options:

Option	Short option	Description
–email	-e	Change the organization owner email address
–members	-m	Set a list of organization members

Note: The `--members/-m` option will *overwrite* the complete list of members for an organization. Organization admins can use the `organization member` collection of commands to add & remove individual members in an organization.

```
# Change the owner email for the 'example' organization
$ cyclid admin organization modify example --email robert@example.com
```

admin organization delete

Delete an organization. By default the `organization delete` command will ask you to confirm the deletion first; you can over-ride this with the `--force/-f` option to force deletion without confirmation.

Note: Deleting organizations is not currently supported by the API and this command will always fail.

Option	Short option	Description
–force	-f	Do not ask for confirmation before deleting the organization

```
# Delete the 'initech' organization
$ cyclid admin organization delete initech
```

Admin user commands

The `admin user` command has a series of sub-commands which are used to manage users.

admin user list

List all of the users on the server.

```
$ cyclid admin user list
admin
bob
lucy
dave
leslie
```

admin user show

Show the details of a user, including their email address and the list organizations they belong to.

```
$ cyclid admin user show bob
Username: bob
Email: bob@example.com
Organizations:
    example
```

admin user create

Create a new user. You must supply the username of the new user, and the users email address.

You may also optionally set the users HTTP Basic password with the `--password/-p` option, or set their HMAC secret with the `--secret/-s` option. You must at least set their password *or* their HMAC secret for the user to be able to log in to the server.

The users HMAC secret should ideally be a suitably long (at least 256 bit) and random string, which the user should keep secure in their Cyclid configuration file.

Option	Short option	Description
–password	-p	The new users initial HTTP Basic password
–secret	-s	The new users HMAC signing secret

```
# Create the user 'bob' with an initial HMAC secret
$ cyclid admin user create bob bob@example.com -s␣
↪b072d8b51cec2755145c401b9249a60ebd89b4704eeebc5b6805ba682d7fac53
```

admin user passwd

Change a users password. The user password is only used for HTTP Basic authentication.

```
# Change the password for the user 'bob'
$ cyclid admin user passwd bob
Password: <enter new password>
Confirm password: <re-enter new password>
```

admin user modify

Change a users email address, HMAC secret and/or password. You can pass the following options:

Option	Short option	Description
–email	-e	Change the users email address
–secret	-s	Change the users HMAC secret
–password	-p	Change the users email address

Unlike the interactive `user passwd` command you can use `user modify` and pass the users new password on the command line.

Your HMAC secret should ideally be a suitably long (at least 256 bit) and random string, which the user should keep secure in their Cyclid configuration file. After changing a users HMAC secret they will need to update their configuration file with the new secret before they can run any other Cyclid commands.

```
# Change the email address for the user 'bob'
$ cyclid admin user modify bob --email robert@example.com
# Change the HMAC secret for the user 'lucy'
$ cyclid admin user modify lucy --secret␣
↪b072d8b51cec2755145c401b9249a60ebd89b4704eeebc5b6805ba682d7fac53
```

admin user delete

Delete a user. By default the `user delete` command will ask you to confirm the deletion first; you can over-ride this with the `--force/-f` option to force deletion without confirmation.

Option	Short option	Description
–force	-f	Do not ask for confirmation before deleting the user

```
# Delete the user 'bob' without asking for confirmation
$ cyclid admin user delete bob --force
```

Admin job commands

The `admin job` command has a series of sub-commands which are used to manage jobs.

admin job stats

Show the total number of jobs.

```
$ cyclid job stats example
Total jobs: 2
```

admin job list

List the details of all jobs.

```
$ cyclid job list example
Name: test_job
  Job: 1
  Version: 1.0.0
```

```
Name: example_job
  Job: 2
  Version: 1.0.0
```

Frequently Asked Questions

Contents

Cyclid

What is Cyclid?

Cyclid is an Open Source Continuous Integration/Continuous Delivery (CI/CD) system. There are three main components: the continuous integration server (*Cyclid*), the user interface (*Cyclid UI*) and a command line client (*Cyclid Client*)

How can I try Cyclid?

There have four options for running Cyclid:

- Let us run it for you: just sign up for hosted Cyclid.
- Run it on Docker.
- Install it yourself on your own cloud instance(s) or servers.
- Test Cyclid with a pre-built Vagrant virtual machine.

What Open Source license(s) does Cyclid use?

The vast majority of the code is released under the Apache 2.0 license.

The Cyclid logo is licensed for use with Cyclid under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

A small amount of third party code is licensed under the MIT license.

What language is Cyclid written in?

Cyclid, Cyclid UI & Cyclid Client are all written in Ruby, and Cyclid UI includes client side Javascript. The Cyclid REST API uses JSON for data interchange.

Cyclid job definitions can be written in either JSON or YAML.

What cloud providers/virtualization platforms/container platforms does Cyclid support?

By design Cyclid is entirely agnostic about which platforms it supports. Almost all the functionality is provided by plugins, which means in theory Cyclid can support any cloud provider, virtualization platform or container platform you would like to use.

Cyclid itself currently supports Google Cloud Platform, Docker containers and provides official plugins for Digitalocean cloud and LXD based containers. Support for other platforms is planned, and you can of course also write your own plugins to support specific cloud providers or platforms.

Why did you build Cyclid?

We feel that Continuous Integration is not a solved problem, and we want something that works for modern DevOps tools and workflows, not just traditional software development.

Github

Why does Cyclid need such broad access to my Github account?

This is a known issue with Github OAuth permissions.

Because Cyclid supports both Public & Private Github repositories, even for Free users, it requires the appropriate permissions with Github for both. Sadly the Github OAuth permissions are not particularly granular, especially those for Private repositories, and so it may appear that Cyclid is asking for overly broad access to your Github account.

Although Cyclid must request these permissions from Github, in reality the only operation the Github plugin itself performs is to update your Pull Request status.

Continuous Integration

What is Continuous Integration/Continuous Delivery?

At it's most simple, Continuous Integration/Continuous Delivery (CI/CD) is a method of automating your software build & release process. Cyclid can build, test & deploy changes to your software every time a developer commits their changes to source control.

The Wikipedia article is a good place to find more information.

I'm an Operations/DevOps/Site Reliability/Network/Database Engineer, should I care about CI/CD?

Yes! Thanks to concepts such as Infrastructure as Code you probably deal with software every day, and Cyclid can help you to automate the process of integrating, testing & deploying your changes.

Overview

Jobs are defined in a single file. They can be written in either JSON or YAML format.

A Cyclid job defines:

1. Some information about the *Job* itself, including the namei & version.

2. The *Environment* to use to run the job. This includes the operating system type and version, and any extra packages to be installed.
3. The *Stages* to run, and the *Sequence* to run them in.
4. Any additional *Stages* that are specific to this job.
5. Additional information such as *Sources* to be checked out or encrypted *Secrets* that can be used when running the job.

Every time a job is run Cyclid provides a "context", which is information about the running job. The *job context* data can be used in the job definition as variables.

Normally the job file is placed in the root of your project, with the name `.cyclid.json` for JSON formatted jobs, or `.cyclid.yml` (`cyclid.yaml`) for YAML formatted jobs. If you use a webhook to automatically trigger Cyclid jobs, it will look for one of these files in your project to run. If you don't use webhooks, or want to define additional jobs that you'll run using the *job submit* command, you can use any name for the file: although it's easier if you still use the `.json` or `.yml` file extensions.

Examples

Minimal job

The absolutely most minimal job that could be run is:

```
{
  "name" : "Minimal example",
  "sequence" : [
    {
      "stage" : "predefined"
    }
  ]
}
```

```
---
name: Minimal example
sequence:
- stage: predefined
```

This job:

- Uses the default operating system (whatever is defined by the server).
- Does not install any additional packages.
- Defines no additional *Stages*.
- Defines no *Secrets* or *Sources* to use.
- Runs the single *Stages* `predefined`, which must be created seperately on the server.

Representative job

In reality your Cyclid jobs are more than likely going to look something more like the following:

```
{
  "name" : "Representative example",
  "environment" : {
    "os" : "ubuntu_14_04",
    "packages": [
      "build-essential",
      "cmake"
    ]
  },
  "secrets": {
    "github_token": "NST5WwAL7b0JFjO94C9q3mv5F3jus69dBr6o9gwGUG177kk7Y/
↪5spP5P+um+VyhrPwJ44WCuwhr7wCR/
↪UiuBZvoq89tsYXg7uEtCJ9uJP18WHHCW9iguTLyXUxDSfWihP2fPHvEka+8K8A2r1Z0FOvZjXSw4+E/
↪COdUdWLQp4GQTHeRvZGV8FS/onoz5V/SYvozHkH6+tw+ZH0k4mkMKVGBl+VPH+RV4PbL9UfhY4/
↪8ZMoaiSLiWg469a49W80qcimnfR3AP+v6vronoHg+d5mqWH+i0LpUeavMzoQnocQmD7axBs+lfVOKbKa2dPwPxLBdaxs2LPhL+I
↪"
  },
  "sources": [
    {
      "type": "git",
      "url": "https://github.com/example/Project.git",
      "token": "%{github_token}"
    }
  ],
  "stages": [
    {
      "name": "prepare",
      "steps": [
        {
          "action": "command",
          "cmd": "make deps"
        }
      ]
    }
  ],
  "sequence" : [
    {
      "stage" : "prepare",
      "on_failure" : "failure",
      "on_success" : "build"
    },
    {
      "stage": "build",
      "on_failure": "failure",
      "on_success": "success"
    }
  ]
}
```

```
---
name: Representative example
environment:
  os: ubuntu_14_04
  packages:
  - build-essential
  - cmake
secrets:
  github_token: NST5WwAL7b0JFjO94C9q3mv5F3jus69dBr6o9gwGUG177kk7Y/
↪5spP5P+um+VyhrPwJ44WCuwhr7wCR/
↪UiuBZvoq89tsYXg7uEtCJ9uJP18WHHCW9iguTLyXUxDSfWihP2fPHvEka+8K8A2r1Z0FOvZjXSw4+E/
↪COdUdWLQp4GQTHeRvZGV8FS/onoz5V/SYvozHkH6+tw+ZH0k4mkMKVGBl+VPH+RV4PbL9UfhY4/
↪8ZMoaiSLiWg469a49W80qcimnfR3AP+v6vronoHg+d5mqWH+i0LpUeavMzoQnocQmD7axBs+lfVOKbKa2dPwPxLBdaxs2LPhL+I
```

```
sources:
- type: git
  url: https://github.com/example/Project.git
  token: '%{github_token}'
stages:
- name: prepare
  steps:
  - action: command
    cmd: make deps
sequence:
- stage: prepare
  on_failure: failure
  on_success: build
- stage: build
  on_failure: failure
  on_success: success
```

This job:

- Uses the Ubuntu 14.04 (Trusty) operating system.
- Installs the additional packages `build-essential` and `cmake` before the job is run.
- Defines the additional *Stage* `prepare`
- Defines the *Source* from https://github.com/example/Project.git...
- ...along with the *Secret* `github_token`, the encrypted token to use when cloning the Source.
- Runs the *Stages* `prepare` followed by `build`. If either Stage fails, the Stage `failure` will be run. When the `build` Stage succeeds, it will run the Stage `success`. The `failure` and `success` Stages must be created seperately on the server.

Example projects

We provide complete example projects that show how to use Cyclid with different languages and types of project. The come complete with a working Cyclid job file that you submit to Cyclid. They will work with a *Vagrant* instance if you just want to experiment.

- Example Ruby project

Sections

Job information

Every job must have a name, and can optionally have a version. If you don't specify a version, Cyclid will use "1.0.0" as the default version.

Example

```
{
  "name" : "An example job",
  "version" : "0.1.2",
  ...
}
```

Stages

Stages are the key piece of any Cyclid job. Jobs are composed of one or more Stages, and each Stage defines one or more Steps which perform actions.

Stages can be defined in the job itself, or can be defined on the Cyclid server, where they can be referenced from and used by any job.

See *Stages, Steps & Actions* to see how they relate to each other and how to use them to build your Cyclid job.

Sequence

The Sequence defines your Cyclid job pipeline. It lists the Stages to run, and the order to run them in. It can also tell Cyclid what to do when a Stage succeeds, or fails.

See *Sequences* for more information on how to define the job pipeline in your Cyclid job.

Environment

The Environment specifies how the build image should be configured for the running job. You can specify the operating system and version to use, additional software repositories to configure and additional packages to install before the job is run.

See *Environments* for details on configuring the job environment.

Sources

Sources define any additional source code, software or tools that your job requires that must be cloned or checked out into the build image before the job is run.

When Cyclid builds a job from a webhook it can automatically determine the location of the source code for the project, but if you use the *job submit* command or your project requires other additional projects, you'll need to specify them under Sources.

For example Cyclid itself can be cloned from https://github.com/Cyclid/Cyclid. It depends on both the Cyclid-core & Mist-client Gems. As Cyclid is being built from source, we also want to clone Cyclid-core & Mist from Github, too. So the Sources for the Cyclid job are:

```
"sources" :
[
  {
    "type" : "git",
    "url" : "https://github.com/Cyclid/Cyclid-core.git"
  },
  {
    "type" : "git",
    "url" : "https://github.com/Cyclid/Mist.git"
  }
]
```

See *Sources* for details on configuring sources.

Secrets

Sometimes you may need to store sensitive data, such as a password or token, in a job file. Each organization has an RSA keypair which can be used to securely encrypt data which can then be decrypted by the server when the job is run.

You can view the organizations public key with the *organization show* command, and you can encrypt data with the *secret encrypt* command. Once you have encrypted the secret, you can add it to the Secrets definition in your job.

See *Secrets* for details on creating and using secrets.

Job contexts

Every time a job is run, Cyclid provides a context which contains various pieces of information that can be inserted into the job using variables, as the job runs.

Some of the context information is generated automatically, but will also include any decrypted *Secrets* that are defined by the job.

An example job context might look something like the following:

```
{
  "job_id"=>309,
  "job_name"=>"Cyclid",
  "job_version"=>"1.0.0",
  "organization"=>"admins",
  "os"=>"ubuntu_trusty",
  "distro"=>"ubuntu",
  "release"=>"trusty",
  "repos"=>[
    {"url"=>"ppa:brightbox/ruby-ng"}
  ],
  "packages"=>[
    "ruby2.3",
    "ruby2.3-dev",
    "build-essential",
    "git",
    "zlib1g-dev",
    "libsqlite3-dev",
    "mysql-client",
    "libmysqlclient-dev"
  ],
  "server"=>"builder01",
  "name"=>"mist-cd2cc1f51e353aa6ddd36946689b679c",
  "host"=>"192.168.1.66",
  "username"=>"build",
  "password"=>nil,
  "key"=>"~/.ssh/id_rsa_build",
  "workspace"=>"/home/build"
}
```

Some information may be more immediately useful to the running job than others, and other information may be available depending on which plugins are in use. The job context may also change as the job runs; hence the name!

Data from the job context can be inserted into the job as variables inside of strings using the `%{ }` operator. For example `"The build image is called %{name} and it is running %{distro} %{release}"` would produce the canonical string `"The build image is called mist-cd2cc1f51e353aa6ddd36946689b679c and it is running ubuntu trusty"`

Context variables can be used pretty much anywhere where you would use a string with Sources, Stages or Steps (it doesn't make sense to use them in Environments or Secrets!)

See Contexts for a complete list of the available job context variables. Plugins may also add additional job context variables; you should check the plugin documentation for information on the variables they might add.

Stages, Steps & Actions

Stages are the key piece of any Cyclid job. Jobs are composed of one or more Stages, and each Stage defines one or more Steps which perform Actions. By building Stages with Steps of Actions, and then connecting those Stages together, you can create complex and powerful job pipelines.

Once you've defined your Stages & Steps, see *the section on defining a sequence* for more information on how to use them to create your job pipeline.

Steps

Steps are the simple building blocks of your pipeline. Each step is a single atomic action that either succeeds or fails.

Each Step defines the type of Action to take, and the additional data to perform that action. Some actions may run on the build image, directly affecting the build process, or some may run on the server to perform an ancillary action such as sending a notification email. Each Action is actually a plugin, so new Actions can be defined by adding plugins.

Example

```
{
  "action": "command",
  "cmd": "make deps"
}
```

action

Name	Description
action	The action to take for the step

See the documentation for *Action plugins* for more information on which Actions are available and what options are supported for each action.

Stages

Stages collect together one or more Steps into a single unit. Each Step in a Stage is run in the order that they are defined. If the Step succeeds, the next Step in the Stage is run. If a Step fails, then execution of any further Steps stops, and the Stage fails.

For example, given the following (but rather contrived) example:

```
{
  "name" : "always-fail",
  "steps" : [
    {
      "action" : "command",
      "cmd" : "echo 'At the begining'"
    },
    {
      "action" : "command",
      "cmd" : "/bin/false"
    },
    {
      "action" : "command",
      "cmd" : "echo 'We got to the end'"
    }
  ]
}
```

The first Step will run successfully, echoing `At the begining`. Because the first Step succeeded, the next Step will run. This Step will always fail (because `/bin/false` will always fail): when this happens, the Stage will stop and the third Step will never run.

Example

A simple Stage, with a single Step that runs a command:

```
{
  "name" : "prepare",
  "steps" : [
    {
      "action": "command",
      "cmd": "make deps"
    }
  ]
}
```

A more complex Stage, with multiple Steps:

```
{
  "name" : "bundle-install",
  "steps" : [
    {
      "action" : "command",
      "cmd" : "sudo gem install bundler --no-ri --no-doc"
    },
    {
      "action" : "command",
      "cmd": "bundle install --path vendor/bundle",
      "path" : "%{workspace}/Example"
    }
  ]
}
```

name

Name	Description
name	The name of the stage

Every Stage must have a name. This name should be unique. Names should not contain spaces and should ideally be descriptive and human readable.

Note: If you define a Stage in your job definition with the same name as a Stage that has been defined on the server, the Stage in your job will take precedence. You should NOT rely on this behaviour: it may change at any time in the future.

steps

Name	Description
steps	A list of Steps to be run by this Stage

A Stage must define at least one Step to run. There is no limit on the number of Steps you can define in a single Stage, but you should probably try to keep the number low and split large lists of Steps up into multiple Stages with fewer Steps in each Stage.

See the section on *Steps* for more information on how to define Steps.

Sequence

The Sequence defines how Cyclid runs your job pipeline, by chaining together different Stages and telling Cyclid how to handle the success or failure of each Stage.

Example

Run the `prepare` Stage, followed by the `test` Stage, and finally the `build` Stage. If the `build` Stage succeeds, run the `success` Stage. If any of the Stages fail, run the `failure` Stage.

```
"sequence" : [
  {
    "stage" : "prepare",
    "on_failure" : "failure"
  },
  {
    "stage" : "test",
    "on_failure" : "failure"
  },
  {
    "stage": "build",
    "on_success": "success",
    "on_failure": "failure"
  }
]
```

stage

Name	Description
stage	The name of a Stage to run

The name of a Stage to run. Each Step in the Stage will be run until either one fails, or they all succeed.

The named Stage must either be defined in the job itself, or defined on the server where the job is running.

on_success

Name	Description
on_success	The name of a Stage to run

The name of a Stage to run if the Stage succeeds. If no `on_success` Stage is defined, Cyclid will run the next Stage in the Sequence; if no more Stages are defined in the Sequence, the job will stop.

on_failure

Name	Description
on_failure	The name of a Stage to run

The name of a Stage to run if the Stage fails. If no `on_failure` Stage is defined, Cyclid will run the next Stage in the Sequence; if no more Stages are defined in the Sequence, the job will stop.

Modifiers

Overview

The `only_if` and `not_if` modifiers can be used to decide if a Stage should be run as part of your Sequence. The modifers offer a complete range of operators that you can use for comparisons, and you can compare strings, integers, floating point numbers and even percentages

If the result of the comparision is `true` (`only_if`) or `false` (`not_if`) the Stage will be run as part of the Sequence. If the Stage is not run it is skipped, the stage is considered successful, and the next Stage or the Stage defined in the `on_success` handler is run.

In addition, the `fail_if` modifier can be used to decide if a Stage should fail, regardless of the status of any Steps within the Stage.

Example

Run the `prepare` Stage, followed by the `test` Stage. Fail at the `test` Stage if test coverage if less than 90%. Only run the `build` Stage if the Job context variable `github_branch` is `master`:

```
"sequence" : [
  {
    "stage" : "prepare",
    "on_failure" : "failure"
  },
  {
    "stage" : "test",
    "fail_if" : "%{cobertura_line_rate} < 90%"
    "on_failure" : "failure"
  },
  {
    "stage": "build",
    "only_if" : "%{github_branch} === 'master'"
    "on_success": "success",
    "on_failure": "failure"
  }
]
```

Statements

only_if

`only_if` will cause a Stage to be run if the comparision evaluates to `true` E.g. `1 eq 1`, `'a' != 'b'`.

not_if

`not_if` will cause a Stage to be run if the comparision evaluates to `false` E.g. `8 eq 9`, `'x' != 'y'`.

fail_if

`fail_if` will cause a Stage to fail if the comparision evaluates to `true` E.g. `1 eq 1`, `'a' != 'b'`.

Operators

Operator	Description
==	Case insensitive string equals
===, eq	Case sensitive string equals, number equals
!=, ne	Not equal
<, lt	Less than
>, gt	Greater than
<=, le	Less than or equal to
>=, ge	Greater than or equal to

Strings must be enclosed within single quotes (`'`).

You can compare strings, integers, floating point numbers and even percentages, so all of the following are valid comparisons:

```
'this' != 'that'
1 lt 2
99% > 80%
3.14 gt 2.71
```

Job context variables can be used in statements:

```
'%{github_branch}' == 'master'
'%{release}' != 'trusty'
```

Environments

The Environment specifies how the build image should be configured for the running job. You can specify the operating system and version to use, additional software repositories to configure and additional packages to install before the job is run. Different Builders & Provisioners may support additional options, such as the virtual machine size, or package repository keys.

Example

```
"environment" : {
  "os" : "ubuntu_14_04",
  "size" : "small",
  "repos" : [
    {
      "url" : "ppa:brightbox/ruby-ng"
    }
  ],
  "packages" : [
    "ruby2.3",
    "ruby2.3-dev",
    "build-essential",
    "cmake"
  ]
}
```

Operating system

Name	Description
os	Name & version of the operating system to use

The `os` option defines which operating system, and which version of the operating system, to use when creating the build image, for example `ubuntu_14_04`, `fedora_24`

The Cyclid server will map the requested operating system to an image. The actual image used will depend on which Builder plugin is configured on the server. In addition, the server must have a *Provisioner* plugin that supports the requested operating system.

For example, if your Cyclid server is configured to use Amazon Web Services to provide build images and request an `ubuntu_14_04` image, the AWS Builder may create an instance using the `ami-1b0d920c` AMI; but a different Cyclid server configured to use LXD for its build images map download the pre-built Ubuntu Trusty LXD image.

Build host size

Name	Description
size	Desired build host size

The `size` options selects on of five generic build host instance sizes: `micro`, `mini`, `small`, `medium` and `large`.

The Builder plugin can map these generic sizes to the actual virtual machine type. The actual build host configuration may differ between Builders, so you should consult the documentation for your *Builder* for information on what each generic size means.

This option largely only makes sense for virtual machine Build hosts, and is generally ignored for container type Build hosts.

Additional data

Additional environment options can be provided. Exactly which options you can use depends on which Provisioner is used I.e. they are operating system dependent.

For example, in *the example above* the Ubuntu provisioner allows you to define a list of additional Ubuntu PPA repositories, and a list of packages to install into the build image.

See the documentation for *Provisioner plugins* for more information on which options are supported for each provisioner.

Sources

Sources define any additional source code, software or tools that your job requires that must be cloned or checked out into the build image before the job is run.

Example

```
"sources" : [
  {
     "type": "git",
     "url": "https://github.com/example/Project.git"
  }
]
```

Type

Name	Description
type	The source SCM system type

The `type` option defines the SCM system to use when cloning or checking out sources, for example `git` or `svn`.

Additional data

Additional options can be provided. Exactly which options you can use depends on which Source plugin is used I.e. they are SCM system dependent.

For example, in *the example above* the Git plugin allows you to define a repository URL, and an optional branch and API token.

See the documentation for *Source plugins* for more information on which options are supported for each source plugin.

Secrets

Sometimes you may need to store sensitive data, such as a password or token, in a job file. Each organization has an RSA keypair which can be used to securely encrypt data which can then be decrypted by the server when the job is run.

Each organization on your Cyclid server has a unique RSA keypair. The public key can be used to encrypt information which can only be decrypted with the private key. You can view your organizations public key using the *organization show* command, but the private key is never visible and never leaves the server it was created on.

Secrets are decrypted by the Cyclid server using your organizations private key. The decrypted secrets are added to the *job context* and you can then use context variables to insert them where they are required in the job definition.

You can encrypt new secrets with the *secret encrypt* command, which will encrypt a given string with your organizations public RSA key, encode it with Base64 and print it. You can then copy the encrypted secret into your job definition.

Example

Define two different secrets `github_token` and `encfs_passwd`

```
"secrets" : {
  "github_token" : "NST5WwAL7b0JFjO94C9q3mv5F3jus69dBr6o9gwGUG177kk7Y/
↪5spP5P+um+VyhrPwJ44WCuwhr7wCR/
↪UiuBZvoq89tsYXg7uEtCJ9uJP18WHHCW9iguTLyXUxDSfWihP2fPHvEka+8K8A2r1Z0FOvZjXSw4+E/
↪COdUdWLQp4GQTHeRvZGV8FS/onoz5V/SYvozHkH6+tw+ZH0k4mkMKVGBl+VPH+RV4PbL9UfhY4/
↪8ZMoaiSLiWg469a49W80qcimnfR3AP+v6vronoHg+d5mqWH+i0LpUeavMzoQnocQmD7axBs+lfVOKbKa2dPwPxLBdaxs2LPhL+
↪",
  "encfs_passwd" :
↪"F6By3LcsONzQ2VJ3OqvImXymwnBCbUpv4JxgmjO52UbMFyErSbSmhP+6PzHmb4LFi7zXel3ujmEk9T5VCibjNkHoTCdKPZ2Pi
↪ECbVqmPFscd3fjRlqkk2oMss/ZfpLl8NdKMCp3KaxB8w7dHfZq0ZmO3kgSNiD3JL+UsWoWy/
↪K7+r9RiBHggZs7rcdwtGmmC55V6R4AJJeVW5HHg8uq+Crjh6HfYSplgaGgFc3Zhskn/
↪OK9SiuhDIpz4jBtt4rZmicXV4OW/yO81e4sUkFTzlPDSj2EkSyqpz3mBB5Zg1iGp2hqFEn2BTA4Kh0/
↪M00AAHFNg7gtqdHbxUXITXA=="
}
```

Actions

Contents

Command

The Command plugin runs a command on the build image. It supports the following options.

cmd

Name	Description
cmd	Command string to run on the build image.

The `cmd` option specifies the command to run. This can either be a complete command complete with arguments, or just the name of the name of the command (where the arguments will be passed with the `arguments` option)

The command must be in $PATH, or you must specify the absolute path to the command binary.

A return code of 0 is considered success; any other return code is considered a failure.

Example

Run the command `bundle` from the default working directory:

```
{
  "action" : "command",
  "cmd" : "bundle install --path vendor/bundle"
}
```

args

Name	Description
args	Command arguments.

An optional list of individual arguments to pass to the command.

Example

Run the command `bundle` from the default working directory:

```
{
  "action" : "command",
  "cmd" : "bundle",
  "args": [
     "install",
     "--path",
     "vendor/bundle"
  ]
}
```

env

Name	Description
env	Environment variables to set before running the command

Defines any environment variables that must be set when the command is run. Each environment variable should specify the name and the value to set.

Example

Set the $GEM_HOME environment variable when the `bundle install` command is run:

```
{
  "action" : "command",
  "cmd" : "bundle install --path vendor/bundle",
  "env" : [
    { "GEM_HOME" : "/var/lib/gems" }
  ]
}
```

path

Name	Description
path	Path to a directory on the build image to run the command from

Specify the path to the working directory on the build image where the command should be run.

If no path is specified the default is to run the command in the root directory of the workspace.

Example

Run the `bundle install` command from the `Project` directory in the workspace:

```
{
  "action" : "command",
  "cmd" : "bundle install --path vendor/bundle",
  "path" : "%{workspace}/Project"
}
```

sudo

Name	Description
sudo	Run the command using `sudo`

Run the command using `sudo` (if the remote user is not already `root`).

Example

Run `gem install` to install a Gem system-wide, using `sudo`:

```
{
  "action" : "command",
  "cmd" : "gem install bundler",
  "sudo" : true
}
```

Script

The Script plugin defines a script and runs it on the build image. It supports the following options.

script

Name	Description
script	Define the script to run.

Defines the script. Smaller scripts can be defined as a single string, seperated by literal `\n` characters, but scripts can also be defined as an array of lines.

Example

A simple script defined as a single string:

```
{
  "action" : "script",
  "script" : "#!/bin/sh\necho 'hello from a simple script'\necho 'I am %{username} of
↪%{organization}'"
}
```

A simple script defined as an array:

```
{
  "action": "script",
  "script": [
    "#!/bin/bash",
    "echo 'Hello from a multi-line script'",
    "echo 'I am %{username} of %{organization}'"
  ]
}
```

env

Name	Description
env	Environment variables to set before running the script

Defines any environment variables that must be set when the script is run. Each environment variable should specify the name and the value to set.

Example

Set the $EXAMPLE environment variable when the script is run:

```
{
  "action": "script",
  "env" : [
    { "EXAMPLE" : "Hello from a script" }
  ],
  "script": [
    "#!/bin/bash",
    "echo $EXAMPLE"
  ]
}
```

path

Name	Description
path	Path to a directory on the build image to run the script from

Specify the path to the working directory on the build image where the script should be run.

If no path is specified the default is to run the script in the root directory of the workspace.

Example

Run the script from the `Project` directory in the workspace:

```
{
  "action": "script",
  "path" : "%{workspace}/Project"
  "script": [
    "#!/bin/bash",
    "echo 'Hello from a multi-line script'",
    "echo 'I am %{username} of %{organization}'"
  ]
}
```

sudo

Name	Description
sudo	Run the script using `sudo`

Run the script using `sudo` (if the remote user is not already `root`).

Example

A simple script defined as a single string, running as `root` using `sudo`:

```
{
  "action" : "script",
  "script" : "#!/bin/sh\necho 'hello from a simple script'\necho 'I am %{username} of
↪%{organization}'",
  "sudo" : true
}
```

Log

The Log plugin writes a message to the build log. It supports the following options.

message

Name	Description
message	Log message

Specify the log message.

Example

```
{
  "action" : "log",
  "message" : "Hello from Cyclid"
}
```

Email

The Email plugin send an email notification. It supports the following options.

message

Name	Description
message	Email message body

Specify the email message body.

to

Name	Description
to	Email recipiant address

The email address to send the message to.

Example

```
{
  "action" : "email",
  "message" : "This is an email from Cyclid",
  "to" : "user@example.com"
}
```

subject

Name	Description
subject	Email message subject

An optional subject. If no subject is specified the default of `Cyclid notification` is used.

Example

```
{
  "action" : "email",
  "subject" : "Example message",
  "message" : "This is an email from Cyclid",
  "to" : "user@example.com"
}
```

color

Name	Description
color	Email body highlight color

Email messages sent by Cyclid highlight the message subject; you can use the `color` option to set this color for different classes of emails E.g. a failure message could set the color to red.

Example

```
{
  "action" : "email",
  "color" : "red",
  "message" : "This is an email from Cyclid",
  "to" : "user@example.com"
}
```

Configuration

The email plugin supports the following configuration options.

See the *Configuration file* documentation for more information on configuring plugins.

Name	Required?	Default	Description
server	No	localhost	The SMTP relay server.
port	No	587	SMTP server port.
from	No	cyclid@cyclid.io	"From" address of the sender.
username	No		SMTP server username.
password	No		SMTP server password.

Slack

The Slack plugin send a Slack message notification. It supports the following options.

subject

Name	Description
subject	Slack message subject

The subject of the Slack message.

message

Name	Description
message	Slack message body

The message body text of the Slack message.

color

Name	Description
color	Slack message highlight color

You can use the `color` option to select the highlight color of the Slack message E.g. a failure notification can set the color to `danger`. If no color is specified the default of `good` is used.

Example

Send a failure notification to the default Slack channel, with the color set to `danger`:

```
{
  "action": "slack",
  "subject": "%{job_name} failed",
  "message": "Job %{organization}/%{job_name} (job #%{job_id}) failed.",
  "color": "danger"
}
```

url

The Slack API URL. By default the Slack API URL is configured organization-wide, and this URL will be used when no URL is specified. However if you need to send a notification to a different Slack group, you can over-ride the default with the `url` option.

Example

```
{
  "action": "slack",
  "url" : "https://hooks.slack.com/services/T00000000/B00000000/
↪XXXXXXXXXXXXXXXXXXXXXXXX",
  "subject": "%{job_name} succeeded",
  "message": "Job %{organization}/%{job_name} (job #%{job_id}) completed successfully.
↪",
}
```

Simplecov

The Simplecov plugin reads a Simplecov test coverage report. It supports the following options.

path

Name	Description
path	Path to the coverage report.

The `path` option gives a fully qualified path to the generated Simplecov coverage report, in JSON format.

The `covered_percent` metric from the report is added to the Job context in the `simplecov_coverage` variable, as a percentage to 2 decimal places E.g. if the generated report contains `{"covered_percent":86.9795918367347}` then this will be rounded to `86.98%` by the plugin.

Your project should install & configure the Simplecov-JSON Gem to generate a JSON coverage report.

Example

```
{
  "action" : "simplecov",
  "path" : "%{workspace}/project/coverage.json"
}
```

Cobertura

The Cobertura plugin reads a Cobertura compatable test coverage report. It supports the following options.

path

Name	Description
path	Path to the coverage report.

The `path` option gives a fully qualified path to the generated coverage report, in XML format.

The `line-rate` metric from the report is added to the Job context in the `cobertura_line_rate` variable, and the `branch-rate` metric is added as `cobertura_branch_rate`. Both metrics are provided as a percentage to 2 decimal places E.g. if the generated report contains `<coverage line-rate="0.9" branch-rate="0.75">` then `cobertura_line_rate` will be `90%` and `cobertura_branch_rate` will be `75%`

The Cobertura plugin can be used to read any Cobertura compatible coverage report.

Example

```
{
  "action" : "cobertura",
  "path" : "%{workspace}/project/coverage.xml"
}
```

Sources

Contents

Git

The Git plugin can clone repositories from a remote Git server. It supports the following options.

url

Name	Description
url	Location of a repository on remote Git server

The `url` option provides the location to a Git repository which should be cloned into the build image before the job starts. Supported URLs are git://, http:// and https://

Repositories are cloned into the working directory of the build image, with the default repository name E.g. if you clone the Cyclid repository from `https://github.com/Cyclid/Cyclid` the source will be located in `%{workspace}/Cyclid`

Example

Clone the Cyclid repository into the working directory:

```
{
  "type" : "git",
  "url" : "https://github.com/Cyclid/Cyclid"
}
```

branch

Name	Description
branch	Remote branch to clone

By default the Git plugin will clone the repository with the master branch checked out. The `branch` option can be used to specify that a different upstream branch should be checked out.

Example

Clone the Cyclid repository into the working directory and switch to the “example” branch:

```
{
  "type" : "git",
  "url" : "https://github.com/Cyclid/Cyclid",
  "branch" : "example"
}
```

token

Name	Description
token	Github API token

The `token` option can be used to supply an authentication token which is used to clone a private Github repository.

Warning: You should encrypt any such tokens and place them in the *Secrets* section of your job, and use the decrypted token from the job context variable, as shown.

Example

Clone a private repository into the working directory using an API key that has been provided as a secret named “github_token”:

```
{
  "type" : "git",
  "url" : "https://github.com/Cyclid/Cyclid",
  "token" : "%{github_token}"
}
```

API extensions

Contents

Github

The Github plugin provides API endpoints which are used for Github event callbacks.

The Github plugin implements the Github OAuth2 authentication process which allows users to obtain an OAuth2 token for their organization which can in turn be used to authenticate Cyclid with Github, including cloneing private repositories.

Job context

The Github plugin provides the following job context information.

Name	Comment
github_event	Github event that triggered the job; `pull_request` or `push`.
github_user	User that created the Pull Request or Push
github_ref	Name of the branch that the event refers to.
github_comment	Title of the pull request or push message.

Configuration

The Github plugin supports the following configuration options.

See the *Configuration file* documentation for more information on configuring plugins.

Name	Required?	Default	Description
api_url	Yes		Public URL of the Cyclid API server.
ui_url	Yes		Public URL of the Cyclid UI server.
client_id	Yes		Your registered Github client ID.

You must register your Cyclid server with Github to obtain a unique client_id which is used to identify your Cyclid server when it performs API calls to Github.

The API URL is used to construct linkback URLs which are passed to Github; in turn Github will call API endpoints via. this URL.

When you register your application you must provide the “Authorization callback URL”; this URL **must** match the value you provide in api_url. For example, if your Cyclid server is accessable from the URL http://cyclid.example.com:8361 you must set the “Authorization callback URL” to `http://cyclid.example.com:8361`

The UI URL is used to construct the “details” link to the Cyclid UI for each job.

Builders

Contents

Google

The Google Builder plugin creates creates Google Compute Engine (GCE) virtual machines build hosts using Google Cloud.

Instance Sizes

The Google builder maps the generic instance size types to the following Google Compute machine types.

Generic Size	Google Machine Type
micro	f1-micro
mini	g1-small
small	n1-standard-1
medium	n1-standard-2
large	n1-standard-4

The default instance size (E.g. if none, or `default` is given) is the machine type set in the `machine_type` configuration option (below).

Configuration

The Google builder supports the following plugin configuration options.

See the *Configuration file* documentation for more information on configuring plugins.

Name	Required?	Default	Description
project	Yes		The Google Cloud project name.
client_email	Yes		The Google Cloud service account email address.
json_key_location	Yes		Path to the JSON key file for the service account.
zone	No	us-central-1a	Google Compute zone to create instances in.
machine_type	No	f1-micro	Instance machine type.
network	No	default	Network name.
username	No	build	Username to bootstrap and run builds with.
ssh_private_key	No	`/etc/cyclid/id_rsa_build`	Path to the build users private key.
ssh_public_key	No	`/etc/cyclid/id_rsa_build.pub`	Path to the build users public key.
instance_name	No	cyclid-build	Cyclid build host name prefix.

The client_email & key file found at json_key_location are used to authenticate with Google for your specified project. See https://cloud.google.com/storage/docs/authentication#service_accounts for more information on how to create your service account & associated JSON key file.

SSH keys

By default the keypair */etc/cyclid/id_rsa_build* (private) and */etc/cyclid/id_rsa_build.pub* (public) will be used to log into instances. You can set paths to a different keypair using the ssh_private_key and ssh_public_key options.

Example

Create f1-micro sized instances in the 'cyclid' project in the europe-west1-b region:

```
server:
  ...
  builder: google
  ...
  plugins:
    google:
      project: cyclid
      client_email: 123456789099-compute@developer.gserviceaccount.com
      json_key_location: /var/lib/google/gcp.json
      zone: europe-west1-b
```

Docker

The Docker Builder plugin creates creates container build hosts using Docker.

Configuration

The Docker builder supports the following plugin configuration options.

See the *Configuration file* documentation for more information on configuring plugins.

Name	Required?	Default	Description
api	No	unix:///var/run/docker.sock	Your Docker daemon socket.
instance_name	No	cyclid-build	Cyclid build host name prefix.

Example

Create Docker instances on a remote server:

```
server:
  ...
  builder: docker
  ...
  plugins:
    docker:
      api: tcp://example.com:5422
```

Provisoners

Contents

Ubuntu

The Ubuntu provisioner supports the following environment options.

Repositories

Name	Description
repos	A list of additional software (package) repositories

The `repos` option defines a list of additional software repositories which are configured on the build image. Each repository definition is defined with the following options:

Name	Required?	Description
url	Yes	An Ubuntu software repository. Supported URLs are ppa://, http:// & https://
components	*Note 1*	The components to use for Debian style repositories E.g. "main universe".
key_id	*Note 1*	The GPG key ID to use for Debian style repositories E.g. F5DA5F09C3173AA6

Notes

1. Only required for Debian style repositories.

Example

Add the BrightBox Ruby-NG PPA and install the Ruby 2.3 package

```
"environment" : {
  "os" : "ubuntu_14_04",
  "repos" : [
    {
      "url" : "ppa:brightbox/ruby-ng"
    }
  ],
  "packages" : [
    "ruby2.3"
  ]
}
```

Debian

The Debian provisioner supports the following environment options.

Repositories

Name	Description
repos	A list of additional software (package) repositories

The `repos` option defines a list of additional software repositories which are configured on the build image. Each repository definition is defined with the following options:

Name	Required?	Description
url	Yes	A Debian software repository. Supported URLs are http:// & https://
components	Yes	The components to use E.g. "main universe"
key_id	Yes	The GPG key ID to use E.g. F5DA5F09C3173AA

Example

```
"environment" : {
  "os" : "debian_jessie",
  "repos" : [
    {
      "url" : "http://www.deb-multimedia.org",
      "components" : "stable main non-free",
      "key_id" : "5C808C2B65558117"
    }
  ],
  "packages" : [
    "ffmpeg"
  ]
}
```

Fedora

The Fedora provisioner supports both YUM & DNF package managers. DNF will be used for Fedora 22 and newer, and YUM will be used for Fedora 21 or older.

The Fedora provisioner supports the following environment options.

Repositories

Name	Description
repos	A list of additional software (package) repositories

The `repos` option defines a list of additional software repositories which are configured on the build image. Each repository definition is defined with the following options:

Name	Required?	Description
url	Yes	A Fedora software repository file or RPM that configures a software repository. Supported URLs are http:// & https://
key_url	No	URL of the matching repository signing key

Example

Add the JPackage repository from a repository definition and install the `ecj` package:

```
"environment" : {
  "os" : "fedora_25",
  "repos" : [
    {
      "url": "http://www.jpackage.org/jpackage50.repo"
    }
  ],
  "packages" : [
    "ecj"
  ]
}
```

Add the RPM Fusion repository from an RPM and signing key, and install the `x265-devel` package:

```
"environment" : {
  "os" : "fedora_25",
  "repos" : [
    {
      "url": "http://download1.rpmfusion.org/free/fedora/rpmfusion-free-release-25.
↪noarch.rpm",
      "key_url": "http://ccrma.stanford.edu/planetccrma/apt/configuration/all/RPM-GPG-
↪KEY.planetccrma.txt",
    }
  ],
  "packages" : [
    "x265-devel"
  ]
}
```

CentOS

The RHEL provisioner will normally use YUM, but will fall back to using RPM directly for some operations on RHEL 5 and older.

The CentOS provisioner supports the following environment options.

Repositories

Name	Description
repos	A list of additional software (package) repositories

The `repos` option defines a list of additional software repositories which are configured on the build image. Each repository definition is defined with the following options:

Name	Required?	Description
url	Yes	A CentOS software repository file or RPM that configures a software repository. Supported URLs are http:// & https://
key_url	No	URL of the matching repository signing key

Example

Add the JPackage repository from a repository definition and install the `ecj` package:

```
"environment" : {
  "os": "centos_7",
  "repos" : [
    {
      "url": "http://www.jpackage.org/jpackage50.repo"
    }
  ],
  "packages" : [
    "ecj"
  ]
}
```

RHEL (Redhat Enterprise Linux)

The RHEL provisioner will normally use YUM, but will fall back to using RPM directly for some operations on RHEL 5 and older.

The RHEL provisioner supports the following environment options.

Repositories

Name	Description
repos	A list of additional software (package) repositories

The `repos` option defines a list of additional software repositories which are configured on the build image. Each repository definition is defined with the following options:

Name	Required?	Description
url	Yes	A RHEL software repository file or RPM that configures a software repository. Supported URLs are http:// & https://
key_url	No	URL of the matching repository signing key

Example

Add the JPackage repository from a repository definition and install the `ecj` package:

```
"environment" : {
  "os": "rhel_7",
  "repos" : [
    {
      "url": "http://www.jpackage.org/jpackage50.repo"
    }
  ],
  "packages" : [
    "ecj"
  ]
}
```

Release Notes

- *Cyclid API*
- *Cyclid UI*
- *Cyclid Client*

Cyclid 0.4

Contents

Upgrading

Refer to the standard instructions for *upgrading* from a previous release of Cyclid.

v0.4.0

Highlights

Improved error handling of invalid job definitions, complete with on-the-fly linting and error reporting, and generally better internal error handling.

Use of `sudo` has generally been cleaned up, and Transports will now only use `sudo` when explitely requested. A new `sudo` flag has been added to Action plugins where it is useful.

Plugins can now provide metadata about themselves and a new `/plugins` API endpoint allows clients to retrieve information about the available plugins.

Changes

- Add the `#metadata`, `#version`, `#author`, `#license` & `#homepage` methods to the Plugin base class, and implement an appropriate `#metadata` method for all of the core plugins.
- Add the `/plugins` API endpoint to return metadata about all of the available plugins.
- Modify the Transport#exec method to take additional arguments as a hash and modify calling code to pass the `path` and `sudo` arguments in the hash, as required.
- **In both the Docker & Github Transports only prepend commands with `sudo` if**
  1. The user is not already `root`
  2. The `sudo` flag has been specifically passed

- Use the Linter (In Cyclid-core) to lint every job definition before creating a new job, and reject any jobs which produce one or more errors (warnings are ignored)
- Significantly improve exception handling around job dispatch, and ensure that we handle exceptions & return appropriate HTTP error messages or Github statuses as required.
- Fix a serious bug in the Google Builder where it would accidentally destroy the first instance returned by the GCP API, rather than the intended named instance.
- Configure the Local Dispatcher & associated Sidekiq worker to use the `cyclid_jobs` named Redis queue, rather than the `default` queue.
- Add the current Thread ID to every log message, to make it easier to trace/debug things like Sidekiq workers.

Cyclid 0.3

Contents

Upgrading

Refer to the standard instructions for *upgrading* from a previous release of Cyclid.

v0.3.3

Highlights

Extended logic with the addition of the fail_if modifier, which will cause a stage to fail if the condition is true.

This release also offers a few bug fixes and enhancements to the Github API plugin.

Changes

- Add the fail_if modifier. If the test evaulates to “true” then the stage will be considered failed regardless of the status of the actual Steps & Stages.
- Add github_owner, github_repository and github_number to the context when a job is triggered via. a Github Pull Request event.
- Fix an issue where the Github API plugin will fail to find the Cyclid job file if the job file has a .yaml extension.

Issues closed

#114 - Match .yaml extensions in Github plugin

v0.3.2

Highlights

A small release, mostly intended to add some features required by the hosted version of Cyclid.

Changes

- Allow the job to specificy an instance size (E.g. ‘small’) and map this to a virtual machine type where it makes sense. Currently this is only supported by the Google Builder.
- Fix a database connection leak with the Local (Sidekiq based) Dispatcher; it will now ensure all database connections are “checked in” to the pool once the job has finished.

v0.3.1

Highlights

This release adds support for Docker build hosts, allowing you to run your Cyclid jobs in lightweight containers.

Also new is support for “Redhat-ish” build hosts, with support for Fedora, CentOS & RHEL, including support for variations across different versions (I.e yum, dnf & rpm).

Last but not least there are a pair of new Action plugins. The Simplecov plugin reads a JSON coverage report generated by the Simplecov coverage tool, and the Cobertura plugin reads an XML coverage report generated by the Cobertura coverage tool, or any coverage tool that produces a Cobertura compatible XML report.

Changes

- Add the *Docker Builder* & DockerApi Transport plugins.
- Add Redhat-ish Provisioner plugins:
  - *Fedora*, with support for both yum (Fedora 21 and older) and dnf (Fedora 22 and newer)
  - *CentOS*, with support for CentOS <5, 6 & 7+

  - *RHEL* (Redhat Enterprise Linux), with support for RHEL <5, 6, 7+
- Add *Cobertura* & *Simplecov* Action plugins that can read test coverage reports.

v0.3.0

Highlights

This is the first release with support for logic modifiers. The implementation of Sequences has been fixed to work as documented, so you no longer need to declare `on_success` handlers for every Stage. The Github API, Debian & Ubuntu Provisioner plugins have all seen updates & bug fixes.

Changes

- Add the `only_if` and `not_if` *modifiers*.
- Sequences now follow the documented behaviour: if no explicit `on_success` handler is defined, the next stage in the sequence will run automatically (if the Stage succeeds)
- Pull Request events from forked Github repositories now work, and the logic for Github Pull Requests has been generally improved.
- Quietened & cleaned up the Debian & Ubuntu Provisioner output when running `apt` commands.
- All context variable interpolation is now "safe" and will not raise an error if the variable being interpolated does not exist.

Issues closed

#65 - Bash modulo (%) operator causes job to hang

Cyclid 0.2

Contents

Upgrading

Refer to the standard instructions for *upgrading* from a previous release of Cyclid.

v0.2.5

Highlights

Minor bug-fixes and improvments.

Changes

- Remove the `logout` method from the SSH transport so that we don't needlessly run `exit` at the end of a build.
- Always run `apt-get update` first in Ubuntu & Debian provisioners so that the cache is up to date before attempting to install anything.
- Fix a bug in the Github plugin so that it pulls from the correct upstream repository when a Pull Request is opened from a forked repository.
- Let the user (or a script) pass the initial admin user secret & password to `cyclid-db-init`

Issues closed

- #94 : Github PR events fail to clone if the PR comes from a fork

v0.2.4

Highlights

This release reorganizes most of the Builder plugins, and enables support for the new LXD & *Google* Builder plugins in addition to the previous Digitalocean plugin.

Thanks to the new LXD plugin, the dependency on the Mist LXC driver has been removed.

Changes

- Remove the **mist** plugin
  - Move the **mist** Builder to it's own plugin Gem.
  - Remove any dependencies on Mist
- Add the *google Builder plugin*
  - Creates build hosts as Google Compute (GCE) instances.
- Add the *log Action plugin* that writes a message to the Job log.
- Some improvements and fixes for the LXD Builder plguin
  - *Ubuntu* & *Debian* plugins
    - Don't prefix commands with `sudo`
  - **ssh** plugin
    - Prepend `sudo -E` to commands if the user is *not* `root`
  - Pass the Job context to Transport plugins.
  - Make sure the Job record `log` database column is at least 16MB (I.e. a column type of `MEDIUMTEXT` on MySQL)
  - Split out some of the RSpec test framework so that it can be used by plugins to run their own tests against a full Cyclid server.
- Proper support for database upgrades
  - Add the `cyclid-db-migrate` command and all of the ActiveRecord migrations to the Gem. Database migrations will be run "standalone" from the `cyclid-db-migrate` command.

v0.2.3

Highlights

This is a maintenance release to fix a dependency conflict with the new Digitalocean Builder plugin, with no functional changes from the v0.2.2 release.

v0.2.2

Highlights

A complete overhaul of the API plugin framework so that plugins can now create their own arbitary API endpoints.

The re-written *Github* plugin use this new functionality to provide support for Github OAuth: Cyclid can now obtain a Github organization OAuth token to authenticate with Github. The plugin also now uses the official Octokit Gem to access the Github API, and adds support for `push` events.

There's also additional functionality that the Cyclid UI makes use of for managing plugin configurations.

Changes

- Cyclid UI integration
  - Add the `GET /organizations/:organization/configs` endpoint to return a list of plugins which have configurations.
  - Add the `Plugins::Base.config?` method to indicate if a plugin supports configuration.
  - Add the "password" data type and use it where a configuration item is a password.
- Support for real plugins
  - Load any Cyclid plugins found in other Gems (E.g. anything under `cyclid/plugins/`
- Overhaul the API plugin framework
  - Still provide the default routes (`GET PUT POST DELETE /organizations/:organization/confis/:type/:plugin`) but also allow plugins to add *additional* endpoints underneath
  - Clean up processesing for the default routes.
  - Provide the helper methods `organization_name` to obtain the current name and `retrieve_organization` to retrieve the Organization object inside of an API plugin.
- Rewrite the *github API plugin*
  - Replace hand-rolled API processing with the Octokit Gem.
  - Add Github OAuth support
    * Add the additional API endpoints to support the Github Web OAuth process flow.
    * Set an Organization OAuth token if a user completes the OAuth flow with Github.
  - Add support for the `push` event.
  - Re-factor everything
    * Remove the unused `hmac_secret` configuration item.
- Speed up package installation
  - Pass the entrie list of packages to the Provisioner plugin instead of iterating over them and calling the Provisioner multiple times, once per. package.
  - The Ubuntu & Debian Provisioners pass the list of packages to `apt` as a single list.

v0.2.1

Highlights

Source URL deduplication. This solves the issue where a Github Pull Request event would provide a URL to the branch to be built which conflicts with the URL provided in the Job file; Cyclid will now make an effort to remove duplicates, while maintaining relevent information such as which branch to build.

Changes

- Improve "Source" processing
  - Group sources from the job by "type" and pass them each group to the approrpiate plugin as a single list.
  - *Git* plugin

    * Attempt to deduplicate the list of repositories by normalizing each URL and then comparing the normalized URLs to find duplicates.

Issues closed

- #53 : Add a test/monitoring endpoint
- #67 : De-dup "source" locations

v0.2.0

Highlights

Major support for the new Cyclid UI, including CORS support to allow access to the Cyclid API from a web browser AJAX request, and JWT authentication.

Changes

- Cyclid UI integration
  - Enable CORS across the API so that the UI can perform AJAX requests.
  - Re-write the placeholder "token" authentication method to support proper JWT token authentication.
  - Add a `POST /token/:username` endpoint to retrieve a JWT token.
  - Move the Cyclid API configuration under the `server` declarationin the configuration file so that Cyclid UI can use the same configuration file with its own namespace.
- Add the `GET /:organization/jobs` endpoint that returns a list of jobs
  - The request can include search parameters to filter results. Supported parameters are:
    * s_name - Return jobs matching the name
    * s_status - Return jobs matching the status
    * s_from - Return jobs started on or after the date
    * s_to - Return jobs started on or before the date
    * limit - Maximum number of jobs to return
    * offset - First job record to return
    * stats_only - Don't return any job details
- Add a real name field to users
- Add a healthcheck framework
  - Add the `GET /health/status` and `GET /health/info` endpoints. `status` returns either a 200 (OK) or 503 (ERROR) response, and can be used for a load balancer healthcheck. `info` always returns a 200 response, with a JSON body with healthcheck information, and can be used by a server status dashboard.
  - Plugins can implement the `#status` method if they have external dependencies; the healthcheck framework will call the `#status` method of every plugin and collates the results, so an individual plugin can indicate an error or warning.

- Re-factor the source
  - Rename `lib` to `app`
  - Provide a proper initialization file under `lib` that middleware can `require` from the Gem

Issues closed

- #5 : Stage success/failure handlers are not automatically included & serialized
- #61 : Job "Ended" time not being set on success
- #59 : Usage of SHA-1 for HMAC

Cyclid UI 0.2

Contents

Upgrading

Refer to the standard instructions for *upgrading* from a previous release of Cyclid.

v0.2.5

Highlights

A small release, mostly intended to add some features required by the hosted version of Cyclid.

Changes

- Add an API that flushes the caches User object, which can be used to force a refresh whenever the user changes E.g. adding or removing an Organization.
- Redirect to / on API authentication failure, rather than directly to /login
- Add a “Catch all” route that redirects unknown locations to /
- Fix the generated Cyclid Client configuration files.
- Slight modifications to the session cookies.

v0.2.4

Highlights

Support for login redirects, as well as some minor bug fixes & improvements.

Changes

- Attempt to redirect the user to the original target after login.
- If signup mode is enabled display an Introduction page if the user does not belong to any organizations.
- Insert the users HMAC secret when they download their client configuration file.

v0.2.3

Highlights

Minor bug-fixes and improvments.

Changes

- Allow the user to set SESSION_SECRET and COOKIE_SECRET as seperate environment variables and set COOKIE_SECRET to a default value if one isn’t set in the environemnt to offer a basic level of protection to the JWT cookie.

v0.2.2

Highlights

Some UI & authentication improvements. Both the server & client side Javascript now handle invalid or expired authentication tokens far more gracefully, forcing re-authentication when necessary.

Changes

- Improve invalid/revoked/expired JWT token handling
  - Add exception handling around the `Users` model.
  - Fail authentication if the user can not be retrieved.
  - Delete the JWT cookie when the user is unauthenticated.
  - Catch common authentication issues client-side and force re-authentication if an AJAX call fails or returns 401 (Unauthorized)
- Don't display the "Jobs" and "Configuration" links in the navbar if no Organization is set (I.e. the User doesn't belong to any Organizations)

Issues closed

- #21 : Improve invalid token handling

v0.2.1

Highlights

Support for the updated hosted Cyclid signups process, and CORS support for the health API endpoints.

Changes

- CORS support
  - Enable CORS for the `/health/status` & `/health/info` endpoints so that a server can be monitored from a dashboard.
- Further hosted Cyclid integration
  - Display "Create new organization" links in the Organizations menu and on the users profile if the `signups` URL is configured.
- Configure development instances to run on port 9393 by default.

v0.2.0

Highlights

Provide a UI for Cyclid plugin configuration data. Every plugin can define it's own schema, so forms are generated dynamically, client side, from the schema information provided from the plugin API.

Changes

- Add organization configs
  - Retrieve the current plugin configurations from the server and provide a a form to display & edit the configuration data. Forms are generated dynamically from the configuration schema that the plugin itself provides.
  - Currently supported data types are:
    - string
    - password
    - integer
    - hash-list
    - link-relative
- Integrate with hosted Cyclid signups
  - Added the `signup` configuration option.
  - Display a "Sign up here" link on the login page if the `signup` configuration option is set.

Issues closed

- #13 : Page title is not set on login page

Cyclid Client 0.4

Contents

v0.4.1

Changes

- Added the `job lint` and `job format` commands, which lint (verify) a job and (re)format a job file, respectively.
- Provide a helpful message if the user has not selected an organization with `org use` but are attempting to run commands.

v0.4.0

Changes

- Added the `job stats` and `job list` commands to show the number of jobs, and list the details of all jobs.
- Added the `user authenticate` command to log in to a server with a username & password and automatically download & create configuration files for every organization the user is a member of. Requires a server running at least Cyclid 0.3.0 or newer.
- Full support for HTTPS for all API calls; the `url` and `use_tls` configurations options have been added.
- Plugable output formatters and better support for output to things that are not terminals E.g. piping output to `less` will no longer produce XTerm control characters in the output.

New methods

- `Job::job_stats` to retrieve job statistics for an organization.
- `Job::job_list` to retrieve a list of jobs.

Cyclid Client 0.3

Contents

v0.3.3

Changes

- The `User::user_add` and `User::user_modify` methods now accept a previously-hashed Bcrypt2 password string in addition to a plaintext password that it will then hash itself.

v0.3.2

Changes

- Support the `password` configuration data type when the user displays a plugin configuration with the `config show` command. Passwords are always displayed as a series of `*` characters.

v0.3.1

Changes

- Added client support for all 3 authentication methods supported by Cyclid (Password, HMAC & JWT)

New methods

- `Auth::token_get` to retreive a JWT token from a Cyclid server.
- `Health::health_ping` to retrieve the current health status from a Cyclid server.

Authentication

Available authentication methods

There are three authentication schemes currently supported:

1. HTTP Basic
2. HMAC request signing
3. API token

HTTP Basic

HTTP Basic is self explanitory and works with any standard HTTP client.

HMAC

The HMAC scheme uses a shared secret (The `User.secret` data) to generate an HMAC from the request, with an optional nonce. The header format is:

```
Authorization: HMAC [user]:[hmac]
X-HMAC-Nonce: [nonce]
```

API Token

The Cyclid server can generate a JWT token for use by simple non-sensitive clients which don't or can't support HMAC signing. The generated token should be considered opaque. The header format is:

```
Authorization: Token [user]:[token]
```

Plugins

Plugin types

The majority of the functionality in Cyclid is implemented in a series of Plugins. There are currently 7 different types of Plugin:

- Action
- API
- Builder
- Dispatcher
- Provisioner
- Source
- Transport

Action

Each action within a Step maps to an Action plugin. For example, the Step:

```
{
  "action" : "command",
  "cmd" : "bundle install"
}
```

defines an action that will use the "command" plugin to run a command on the build host. A different Action plugin may send a notification message to a chat system, for example.

API

API plugins can extend the Cyclid API to provide additional functionality E.g. a callback endpoint for Github, or a chatbot integration. *API plugins are some of the most complex* Cyclid plugins.

Builder

Builders create build hosts that can be used to run jobs. A cloud Builder may call an external cloud API to create a new virtual instance, the details of which it can then pass to the Job runner to perform the build. For example, there could an Amazon Web Services (AWS) Builder which calls the AWS API, or an OpenStack Builder which calls an OpenStack native API, or a Builder which creates LXC containers on physical hardware.

Dispatcher

Dispatchers queue jobs to be run, and provide a method for the job runner to return information to Cyclid as it runs. The most simple Dispatcher is the 'local' plugin, which creates an asynchronous process which runs on the same host as the API.

Dispatcher plugins also create Notifiers *that are used by the Job Runner to update the database* as the Job runs.

Provisioner

Provisioners work alongside Builders to create & prepare build hosts. A Provisioner plugin knows about an operating system or distribution, and is called to perform any additional configuration required after the Builder has created a build host. For example, the Ubuntu Provisioner can add additional apt repositories and install additional packages on an Ubuntu based host. A Redhat Provisioner would likewise be able to install additional yum repositories and install RPM packages on a Redhat based build host.

Source

Source plugins can checkout/clone sources from a Source Code Management (SCM) system. For example, a 'git' plugin can clone a Git repository, an 'svn' plugin can clone a Subversion repository etc.

Transport

Transports are used to connect to build hosts and run commands. The most obvious example is an SSH transport, but E.g. a Windows build host may require a WinRM transport plugin.

API Plugins

API plugins can extend the core Cyclid API within their own namespace, and perform arbitrary actions. This can be used to create an API which can be used by external applications as a callback, or to extend the core functionality of Cyclid itself.

API plugins themselves are easy to write, but the mechanism that makes them work is more complex. The main aim is to abstract the framework (Sinatra) and internal details (ActiveRecord models etc.) from the plugin.

Controller

The API plugin base provides a Controller which defines Sinatra API endpoints for GET, POST, PUT & DELETE actions. The controller methods provide the internal boilerplate, for example validating that the organization exists, parsing the body (for POST & PUT methods) and retrieving the plugin configuration data. The method then invokes a callback in an external Module that implements the logic to handle the HTTP event.

API plugins do not and *should* not create their own subclass of the Controller. Instead, they should provide a Module which implements the callback methods the plugin wishes to handle.

Methods

There are four callback methods, which map directly onto the HTTP verbs that the API plugin controller supports:

```
module Methods
  def get(headers, config)
    ....
  end

  def post(data, headers, config)
    ...
  end

  def put(data, headers, config)
    ...
  end

  def delete(headers, config)
    ...
  end
end
```

The API plugin base provides a set of methods which simply return an HTTP 405 error (“Not implemented”) to the caller. A plugin can then implement only the callbacks it needs:

```
def ExampleMethods
  include Api::Methods

  def post(data, headers, config)
    ...
  end
end
```

Here, the Example plugin only wishes to handle HTTP POST events, so only provides it’s own implementation for the post callback. By including the Api::Methods ‘base’ module, the default get, put & delete callbacks will be used (which simply return a 405 error).

Helpers

The API plugin base provides some helper methods. These mostly wrap (and therefore abstract) internal methods. These helpers are:

authorize

- method : **Required** : Authenticate the user and check that they are authorized to perform the operation. *method* is one of ‘get’, ‘post’, ‘put’ or ‘delete’, which map to a READ, WRITE or ADMIN permission.

By default API plugin endpoints are not authenticated & authorized in the same way as the rest of the Cyclid API is. If a plugin wishes to apply authorization to an endpoint (I.e. a user must exist in Cyclid and be a member of the organization) it can call the authorize helper.

return_failure

- code : **Required** : HTTP return code.
- message : **Required*** : Message to be returned as part of the response body.

Causes the HTTP request to fail with the given HTTP response code. The message body is a JSON object which contains a code and the message.

http_headers

An internal method which extracts & formats the raw HTTP request headers, which are then passed to the callback method (See below).

HTTP headers

The API Controller processes the raw HTTP request headers and extracts any `HTTP_` header. The header is “cleaned up” and added to the list of headers which is then passed to the callback method. For example, given the following raw HTTP headers:

```
"HTTP_CONTENT_TYPE" => "application/json"
"HTTP_X_EXAMPLE_ACTION" => "event"
```

would become the list of headers:

```
"Content-Type" => "application/json"
"X-Example-Action" => "event"
```

callback methods can then use the headers, if required.

This, again, abstracts the plugin from the raw implementation details and allows Cyclid to safely pass HTTP headers to a plugin without leaking any potentially sensitive information E.g. session signing secret.

Joining Methods to a Controller

The API plugin class itself provides a single method that simply returns a new Api::Controller instance. It passes in the Module that implements its callback methods, and the two are joined together when the Controller instance is registered in Sinatra.

```ruby
class Example < Api
  def self.controller
    return ApiExtension::Controller.new(ApiExtension::ExampleMethods)
  end
end
```

Methods, not Classes

You may have noticed that the use of Modules and that the bulk of the plugin itself is actually implemented on a Module, rather than a subclass of the Api plugin class, as most of the other plugins are implemented.

This complication stems from the fact each Sinatra Controller is a Module, which you then `register` into the Sinatra application. Sinatra then calls a `registered` callback method on the Module, passing it the Sinatra Application instance. The callback can then define the API endpoints against the application instance.

Because these are plugins, that requires us to dynamically create a new Module *instance* that can in turn be registered by Sinatra; the Controller class is actually a subclass of Module, which allows the plugin to create the Module on the fly. Due to the fact that it's quite difficult to create a "submodule" which can overload methods (as you would with a subclass which can overload a method), the Controller itself & the Methods are split in two; this then allows the plugin create it's own module in the normal manner, without having to resort to complex Ruby metaprogramming.

Plugin configurations

Plugin configs provide a mechanism for Plugins to create & store their own configuration data without having to extend the core database schema, or have any knowledge at all of the underlying schema. Plugin configuration is specific to each organization, and each organization can have its own configuration for any given plugin.

The Plugin configuration also provides an API that can be used to get & set the configuration for each plugin.

API

The plugin config API provides two endpoints: one to GET the plugin configuration, and one to PUT the plugin configuration. For example:

```
$ curl -u admin http://api.dev.cyclid.io/organizations/test/configs/api/github
{"id":1,"plugin":"github","version":"1.0.0","config":{"repository_tokens":[{"url":
↪"http://github.com/example/Project","token":"abcdefg"},{"url":"http://github.com/
↪example/Other","token":"uvwxyz"}],"hmac_secret":null},"organization_id":2,"schema":[
↪{"name":"repository_tokens","type":"hash-list","description":"Repository OAuth␣
↪tokens","default":[]},{"name":"hmac_secret","type":"string","description":"Github␣
↪HMAC signing secret","default":null}]}
```

The data includes some metadata about the plugin, the configuration data itself and a schema.

If no config has yet been created for the plugin when an API call is made, a default configuration will be created and stored in the database.

Configuration data schema

Because each plugin has its own configuration data, the data is stored as JSON in a single database TEXT column; this means the configuration data itself is effectively schemaless. In order to allow clients to understand the configuration data presented by the plugin, the plugin configuration also includes a schema which describes it. This data can be used by the client E.g. a web UI can use the schema data to generate a form that the user can modify.

```
"schema" : [
   {
      "type" : "hash-list",
      "default" : [],
      "description" : "Repository OAuth tokens",
      "name" : "repository_tokens"
   },
   {
      "type" : "string",
      "default" : null,
      "name" : "hmac_secret",
      "description" : "Github HMAC signing secret"
   }
]
```

Each item in the schema data describes an entry in the configuration data:

- name : **Required** : The JSON key name.
- type : **Required** : The data type. This is not a direct mapping to the JSON data type (see below).
- description : **Required** : A human readable description of the configuration data.

Currently defined data types are:

- string : A single or multi-line string.
- integer : A whole number.
- boolean : A `true` or `false` value.
- list : A simple list (array) of items.
- hash-list : A list (array) of objects.

Other types may be added over time.

Implementation

The plugin Base class implements the following methods:

- get_config
- set_config
- update_config
- default_config

- config_schema

get_config

- org : **Required** : The organization to retrieve the configuration for.

Retrieves the plugin config for the given organization from the database. If no configuration exists yet, creates a new entry in the database. The new database entry contains the data returned by default_config.

set_config

- new_config : **Required** : Updated configuration data.
- org : **Required** : The organization to store the configuration for.

Merges the new configuration data with the current configuration and stores it in the database. The configuration is merged by calling update_config with a copy of the current and new configurations.

update_config

- current : **Required** : A copy of the current plugin configuration data.
- new : **Required** : The new configuration to be applied.

Validates the new configuration data and merges it with the current data. The method can implement logic for each configuration item; I.e a single item which can only have a single entry will be over-written, but a list may have new entries appended to it.

The merged configuration data is returned to set_config to be stored in the database.

default_config

Returns a sensible default configuration, as a hash. The absolute minimum default is an empty hash.

config_schema

Returns the schema definition for the configuration (See above), as a hash.

Implementing configuration in a plugin

Every plugin has an API endpoint for configuration data. If the plugin does not implement any callbacks, the plugin will simply have an empty configuration and any changes will be ignored.

A plugin which wishes to implement a configuration must provide implementations of the `update_config`, `default_config` and `config_schema` methods.

```
class Example < Api
  class << self
    def update_config(current, new)
      current.merge(new)
    end

    def default_config
      {wax: true}
    end

    def config_schema
      schema = []
      schema << { name: 'wax',
                  type: 'boolean',
                  default: true,
                  description: 'Whether to apply wax (true) or buff the wax off␣
↪(false)' }
      schema
    end
  end
end
```

Jobs, JobRecords, LogBuffers, Notifiers & Callbacks

Internal classes

Jobs

Jobs are passed in as JSON or YAML documents and describe various items which are needed to successfully run a job; they do not exist, as is, in the database.

Jobs are run by a job Runner, which contains all of the logic necessary to run jobs consistently. The actual job Runner instance may be local (within the context of the API application) or remote (running as a standalone process on a different host).

JobRecords

JobRecords are just what they sound like: a record in the database of a job that was submitted, and the metadata related to that job:

- started : The time & date of when the job was submitted.
- ended : The time & date of when the job finished, successfully or otherwise.
- status : The *current* status of the job; for jobs that have finished, this is either succeeded or failed. Jobs that are in progress may have other states, such as waiting.
- log : The log output from the job.

Every time a job is submitted, a new JobRecord is created.

LogBuffers

During the process of running a job, data will be generated from various different sources. A LogBuffer collates all of that data into a single place, and also reflect data to various other sources. For example, every time new log output it written to a LogBuffer, the LogBuffer also updates the log in the JobRecord.

Notifiers

Job runners may run remotely, in which case they may have no access to the database and the related ActiveRecord models. Because of this a job Runner may not be able to simply write directly to a LogBuffer or update the status of the JobRecord.

Notifiers abstract the JobRecord & LogBuffer objects from the Runner; the Notifier knows how to do this on behalf of the runner. In the case of a local job Runner, the Notifier can simply update the JobRecord & LogBuffer itself directly, but with a remote job Runner the Notifier may instead place a message onto a queue, which would be consumed by a background process running in the API application context which would then update the JobRecord & LogBuffer.

Callbacks

Because Jobs run asynchronously, it can be useful to connect arbitrary code to Notifier events which can perform some self-defined action. For example, during a Job run, the Github plugin can update the Pull Request status via. the Github API.

Plugins can create their own Callback instance, which contains hooks which will be called by the Notifier when various events (status changes, something is written to the JobRecord log, completion) occur. The Callback instance can then run whatever arbitrary code it wishes.

Relationship

The only object which knows if a Runner is local or remote is the Dispatcher. The Dispatcher plugin also defines a suitable Notifier (and Worker, if required). The process that creates the Runner will also create a matching Notifier, ensuring that the Runner can always communicate & update the database via. a suitable Notifier. Callbacks, if used, are created externally and passed to the Dispatcher.

Stages, Steps, Actions & Jobs

The user's view

The user's viewpoint is quite simple. A job is defined with either JSON or YAML:

```
---
name: Test project
version: 1.0.0
environment:
- os: Ubuntu 14.04
  repos:
  - ppa:brightbox/ruby-ng
  packages:
  - build-essential
  - cmake
  - ruby2.2
  - ruby2.2-dev
sources:
- type: git
  url: https://github.com/example/Project.git
stages:
- name: 'Custom stage #1'
  steps:
  - action: command
```

```
    cmd: bundle install -p vendor/bundle
- name: failure notification
  steps:
  - action: slack_notifier
    message: It's all gone Pete Tong
  - action: command
    cmd: push_notification.pl -p '1234' com.example.derp.tong
    path: /usr/bin
    env:
    - a_variable: 'thing'
      an_other_var: 'other thing'
- name: success notification
  steps:
  - action: slack_notifier
    message: Yay, it worked!
job:
- stage: 'Custom stage #1'
  version: 1.0.0
  on_failure:
  - stage: failure notification
  on_success:
  - stage: foo
    version: 1.1
  # The 'foo' stage would be a pre-defined job and we can just over-ride attributes
  # like on_success
- stage: foo
  version: 1.1
  on_success:
  - stage: success notification
```

The Job has our main parts to it:

1. An Environment, which defines how the host which is used to run the job is configured.
2. The Sources, which will be checked out onto the build host.
3. A series of Stages, which in turn have Steps, which define a list of Actions to be run.
4. The Job itself, which is a series of Stages to be run and what to do next when each stage succeeds or fails.

Stages can either be previously defined Stages from the database, or defined "ad-hoc" in the Job. "Ad-hoc" stages and are not stored in the database. We'll skip Environments for now.

The internal view

The internal relationship is more complex.

database diagram

Organizations have one or more Stages, Stages have one or more Steps and each Step defines one or more Actions. Jobs are more complicated: Jobs are composable objects I.e. the Stages used in a Job may either be defined in the database already, or defined ad-hoc in the Job itself. In addition, Jobs can over-ride the on_success & on_failure handlers of pre-defined jobs. Jobs may or may not be run local (I.e. on the same host) to the API application. Jobs are asynchronous I.e. the API application controller will return once a job is successfully *created* and leave it to run in the background, and the client must poll the API for the current job status.

The process for creating a Job, and all of it's moving parts, is:

1. A JSON or YAML document is submitted to the API. The API parses this into a Hash, creates a new JobView and passes it the Hash representing the Job definition.
2. For each ad-hoc Stage defined in the Job, the JobView creates a StageView object which is added to a list.
3. For each Action defined in the ad-hoc Stage, an Action object is created. The Action object is serialised into the StageView.
4. For each Stage defined in the job section:

1. Check if the matching StageView has already been created from an ad-hoc job.
2. If the Stage is not ad-hoc, find the matching Stage in the database and create a StageView from it.

5. Once the list of Stages is complete, serialise the StageViews.
6. The API controller serializes the JobView (The serialised StageViews, plus the Environment, plus some additional data about the Job) and passes it to the Dispatcher.
7. The Dispatcher dispatches the Job; what this means depends on the Dispatcher E.g. create a SideKiq worker to run the Job, or placing it on a queue to be consumed by a remote worker etc.
8. Create a JobRecord which records details about the job E.g. current status, when it was started, the user that requested it etc.
9. The Dispatcher returns the Job ID to the API controller, which in turn returns it to the client.

From this point on the job is now asynchronous and is processed in the background. The client can poll the API to find out the latest job status, retrieve the log data etc. We'll assume the job is running locally under SideKiq:

1. Request a BuildHost from the Builder, passing in the Environment that was defined as part of the Job.
2. Create a Transport which can be used to communicate with the BuildHost.
3. Prepare the BuildHost (calling prepare() will do things like install the list of packages etc. required for the host, as defined by the Job)
4. Create a LogBuffer. This will store the combined log output from the job run.
5. Un-serialize each StageView in the Job.
6. For each StageView, un-serialize each Action: this yields an Action plugin object of some form which can be performed. Each plugin is passed the Transport & LogBuffer objects; plugins may or may not use either E.g. the 'command' plugin will use the Transport to execute a command on the BuildHost, and the output will be logged to the LogBuffer, but a notification plugin will run locally and may not log anything into the LogBuffer.
7. Once the Stages have all run (either successfully or otherwise), the worker is finished.

The worker will periodically update the JobRecord for the Job to indicate the current state. The client may (or may not be) requesting the latest job status.

The process for a remote job worker is not too different:

1. The Job is composed & serialized as above.
2. The Dispatcher creates a local worker & returns the Job ID to the API controller.
3. The local worker places the serialized job onto a queue and then waits for a message to tell it a remote worker has pulled it from the queue.
4. The remote worker sends messages via. the queue to the local worker, which updates the JobRecord status & appends log data into a LogBuffer.
5. The API client can poll the API for the Job status as before.

The local worker spends most of its time waiting for messages to arrive and thus has much lower overhead. In addition the remote workers can be mapped to API application *n:m* E.g. there can be more workers than there are API applications.