Vagga Documentation

Release 0.4.0

Paul Colomiets

April 03, 2016

Contents

Contents:

CHAPTER 1

About Vagga

Contents:

1.1 What is Vagga?

Vagga is a tool to create development environments. In particular it is able to:

- Build container and run program with single command, right after "git pull"
- Automatically rebuild container if project dependencies change
- Run multiple processes (e.g. application and database) with single command
- Execute network tolerance tests

All this seamlessly works using linux namespaces (or containers).

1.1.1 Example

Let's make config for hello-world flask application. To start you need to put following in `vagga.yaml`:

```
containers:
  flask:
    setup:
    - !Ubuntu trusty
    - !UbuntuUniverse
    - !Install [python3-flask]
commands:
  py3: !Command
    container: flask
    run: python3
```

- – create a container "flask"
- – install base image of ubuntu
- – enable the universe repository in ubuntu
- – install flask from package (from ubuntu universe)
- – create a simple command "py3"
- – run command in container "flask"
- – the command-line is "python3"

To run command just run `vagga command_name`:

```
$ vagga py3
[ .. snipped container build log .. ]
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import flask
>>>
```

This is just a lazy example. Once your project starts to mature you want to use some specific version of flask and some other dependencies:

```
containers:
  flask:
    setup:
    - !Ubuntu trusty
    - !Py3Install
      - werkzeug==0.9.4
      - MarkupSafe==0.23
      - itsdangerous==0.22
      - jinja2==2.7.2
      - Flask==0.10.1
      - sqlalchemy==0.9.8
```

And if another developer does `git pull` and gets this config, running `vagga py3` next time will rebuild container and run command in the new environment without any additional effort:

```
$ vagga py3
[ .. snipped container build log .. ]
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import flask, sqlalchemy
>>>
```

Note: Container is rebuilt from scratch on each change. So *removing* package works well. Vagga also uses smart caching of packages to make rebuilds fast.

You probably want to move python dependencies into `requirements.txt`:

```
containers:
  flask:
    setup:
    - !Ubuntu trusty
    - !Py3Requirements "requirements.txt"
```

And vagga is smart enough to rebuild if `requirements.txt` change.

In case you've just cloned the project you might want to run bare `vagga` to see which commands are available. For example, here are some commands available in vagga project itself:

```
$ vagga
Available commands:
    make                Build vagga
    build-docs          Build vagga documentation
    test                Run self tests
```

(the descriptions on the right are added using `description` key in command)

1.1.2 More Reading

- Managing Dependencies with Vagga shows basic concepts of using vagga and what problems it solves.
- Evaluating Mesos discuss how to run network tolerance tests.

1.2 What Makes Vagga Different?

There are four prominent features of vagga:

- Command-centric workflow instead of container-centric
- Lazy creation of containers
- Containers are versioned and automatically rebuilt
- Running multiple processes without headache

Let's discuss them in details

1.2.1 Command-Centric Workflow

When you start working on project, you don't need to know anything about virtual machines, dependencies, paths whatever. You just need to know what you can do with it.

Consider we have an imaginary web application. Let's see what we can do:

```
> git clone git@git.git:somewebapp.git somewebapp
> cd somewebapp
> vagga
Available commands:
    build-js    build javascript files needed to run application
    serve       serve a program on a localhost
```

Ok, now we know that we probably expected to build javascipt files and that we can run a server. We now just do:

```
> vagga build-js
# container created, dependencies populated, javascripts are built
> vagga serve
Now you can go to http://localhost:8000 to see site in action
```

Compare that to vagrant:

```
> vagrant up
# some machine(s) created
> vagrant ssh
# now you are in new shell. What to do?
> make
# ok probably something is built (if project uses make), what now?
> less README
# long reading follows
```

Or compare that to docker:

```
> docker pull someuser/somewebapp
> docker run --rm --it someuser/somewebapp
# if you are lucky something is run, but how to build it?
# let's see the README
```

1.2.2 Lazy Container Creation

There are few interesting cases where lazy containers help.

Application Requires Multiple Environments

In our imaginary web application described above we might have very different environments to build javascript files, and to run the application. For example javascripts are usually built and compressed using Node.js. But if our server is written in python we don't need Node.js to run application. So it's often desirable to run application in a container without build dependencies, at least to be sure that you don't miss some dependency.

Let's declare that with vagga. Just define two containers:

```
containers:

  build:
    setup:
    - !Ubuntu trusty
    - !Install [make, nodejs, uglifyjs]

  serve:
    setup:
    - !Ubuntu trusty
    - !UbuntuUniverse
    - !Install [python-django]
```

One for each command:

```
commands:

  build-js: !Command
    container: build
    run: "make build-js"

  serve: !Command
    container: serve
    run: "python manage.py runserver"
```

Similarly might be defined test container and command:

```
containers:

  testing:
    setup:
    - !Ubuntu trusty
    - !UbuntuUniverse
    - !Install [make, nodejs, uglifyjs, python-django, nosetests]

commands:

  test:
```

```
    container: testing
    run: [nosetests]
```

And your user never care how many containers are there. User only runs whatever comands he needs.

How is it done in vagrant?

```
> vagrant up
# two containers are up at this point
> vagrant ssh build -- make
# built, now we don't want to waste memory for build virtual machine
> vagrant halt build
> vagrant ssh serve -- python manage.py runserver
```

Project With Examples

Many open-source projects and many proprietary libraries have some examples. Often samples have additional dependencies. If you developing a markdown parser library, you might have a tiny example web application using flask that converts markdown to html on the fly:

```
> vagga
Available commands:
    md2html         convert markdown to html without installation
    tests           run tests
    example-web     run live demo (flask app)
    example-plugin  example of plugin for markdown parser
> vagga example-web
Now go to http://localhost:8000 to see the demo
```

How would you achieve the same with vagrant?

```
> ls -R examples
examples/web:
Vagrantfile README flask-app.py

examples/plugin:
Vagrantfile README main.py plugin.py

> cd examples/web
> vagrant up && vagrant ssh -- python main.py --help
> vagrant ssh -- python main.py --port 8000
# ok got it, let's stop it
> vagrant halt && vagrant destroy
```

I.e. a `Vagrantfile` per example. Then user must keep track of what containers he have done `vagrant up` in, and do not forget to shutdown and destroy them.

Note: example with Vagrant is very imaginary, because unless you insert files in container on provision stage, your project root is inaccessible in container of `examples/web`. So you need some hacks to make it work.

Docker case is very similar to Vagrant one.

1.2.3 Container Versioning and Rebuilding

What if the project dependencies are changed by upstream? No problem:

```
> git pull
> vagga serve
# vagga notes that dependencies changed, and rebuilds container
> git checkout stable
# moving to stable branch, to fix some critical bug
> vagga serve
# vagga uses old container that is probably still around
```

Vagga hashes dependencies, and if the hash changed creates new container. Old ones are kept around for a while, just in case you revert to some older commit or switch to another branch.

Note: For all backends except `nix`, version hash is derived from parameters of a builder. For `nix` we use hash of nix derivations that is used to build container, so change in `.nix` file or its dependencies trigger rebuild too (unless it's non-significant change, like whitespace change or swapping lines).

How you do this with Vagrant:

```
> git pull
> vagrant ssh -- python manage.py runserver
ImportError
> vagrant reload
> vagrant ssh -- python manage.py runserver
ImportError
> vagrant reload --provision
#  If you are lucky and your provision script is good, dependency installed
> vagrant ssh -- python manage.py runserver
# Ok it works
> git checkout stable
> vagrant ssh -- python manage.py runserver
# Wow, we still running dependencies from "master", since we added
# a dependency it works for now, but may crash when deploying
> vagrant restart --provision
# We used ``pip install requirements.txt`` in provision
# and it doesn't delete dependencies
> vagrant halt
> vagrant destroy
> vagrant up
# let's wait ... it sooo long.
> vagrant ssh -- python manage.py runserver
# now we are safe
> git checkout master
# Oh no, need to rebuild container again?!?!
```

Using Docker? Let's see:

```
> git pull
> docker run --rm -it me/somewebapp python manage.py runserver
ImportError
> docker tag me/somewebapp:latest me/somewebapp:old
> docker build -t me/somewebapp .
> docker run --rm -it me/somewebapp python manage.py runserver
# Oh, that was simple
> git checkout stable
> docker run --rm -it me/somewebapp python manage.py runserver
# Oh, crap, I forgot to downgrade container
# We were smart to tag old one, so don't need to rebuild:
> docker run --rm -it me/somewebapp:old python manage.py runserver
```

```
# Let's also rebuild dependencies
> ./build.sh
Running: docker run --rm me/somewebapp_build python manage.py runserver
# Oh crap, we have hard-coded container name in build script?!?!
```

Well, docker is kinda easier because we can have multiple containers around, but still hard to get right.

1.2.4 Running Multiple Processes

Many projects require multiple processes around. E.g. when running web application on development machine there are at least two components: database and app itself. Usually developers run database as a system process and a process in a shell.

When running in production one usually need also a cache and a webserver. And developers are very lazy to run those components on development system, just because it's complex to manage. E.g. if you have a startup script like this:

```
#!/bin/sh
redis-server ./config/redis.conf &
python manage.py runserver
```

You are going to loose `redis-server` running in background when python process dead or interrupted. Running them in different tabs of your terminal works while there are two or three services. But today more and more projects adopt service-oriented architecture. Which means there are many services in your project (e.g. in our real-life example we had 11 services written by ourselves and we also run two mysql and two redis nodes to emulate clustering).

This means either production setup and development are too diverse, or we need better tools to manage processes.

How vagrant helps? Almost in no way. You can run some services as a system services inside a vagrant. And you can also have multiple virtual machines with services, but this doesn't solve core problem.

How docker helps? It only makes situation worse, because now you need to follow logs of many containers, and remember to `docker stop` and `docker rm` the processes on every occassion.

Vagga's way:

```
commands:
  run_full_app: !Supervise
    children:
      web: !Command
        container: python
        run: "python manage.py runserver"
      redis: !Command
        container: redis
        run: "redis-server ./config/redis.conf"
      celery: !Command
        container: python
        run: "python manage.py celery worker"
```

No just run:

```
> vagga run_full_app
# two python processes and a redis started here
```

It not only allows you to start processes in multiple containers, it also does meaningful monitoring of them. The `stop-on-failure` mode means if any process failed to start or terminated, terminate all processes. It's opposite to the usual meaning of supervising, but it's super-useful development tool.

Let's see how it's helpful. In example above celery may crash (for example because of misconfiguration, or OOM, or whatever). Usually when running many services you have many-many messages on startup, so you may miss it. Or it

may crash later. So you click on some task in web app, and wait when the task is done. After some time, you think that it *may* be too long, and start looking in logs here and there. And after some tinkering around you see that celery is just down. Now, you lost so much time just waiting. Wouldn't it be nice if everything is just crashed and you notice it immediately? Yes it's what `stop-on-failure` does.

Then if you want to stop it, you just press `Ctrl+C` and wait for it to shut down. If it hangs for some reason (may be you created a bug), you repeat or press `Ctrl+/` (which is `SIGQUIT`), or just do `kill -9` from another shell. In any case vagga will not exit until all processes are shut down and no hanging processes are left ever (Yes, even with `kill -9`).

1.3 Vagga vs Docker

Both products use linux namespaces (a/k/a linux containers) to the work. However, docker requires root privileges to run, and doesn't allow to make development evironments as easy as vagga.

1.3.1 User Namespaces

As you might noticed that adding user to `docker` group (if your docker socket is accessed by `docker` group), is just like giving him a paswordless `sudo`. This is because root user in docker container is same root that one on host. Also user that can start docker container can mount arbitrary folder in host filesystem into the container (So he can just mount `/etc` and change `/etc/passwd`).

Vagga is different as it uses a user namespaces and don't need any programs running as root or setuid programs or sudo (except systems' builtin `newuidmap`/`newgidmap` if you want more that one user inside a container, but `newuidmap` setuid binary is very small functionally and safe).

1.3.2 No Central Daemon

Vagga keeps your containers in `.vagga` dir inside your project. And runs them just like any other command from your shell. I.e. command run with vagga is child of your shell, and if that process is finished or killed, its just done. No need to delete container in some central daemon like docker has (i.e. docker doesn't always remove containers even when using `--rm`).

Docker also shares some daemon configuration between different containers even run by different users. There is no such sharing in vagga.

Also not having central daemon shared between users allows us to have a user-defined settings file in `$HOME/.config/vagga/`.

1.3.3 Children Processes

Running processes as children of current shell has following advantages:

- You can monitor process and restart when dead (needs polling in docker), in fact there a command type `supervise` that does it for you)
- File descriptors may be passed to process
- Processes/containers may be socket-activated (e.g. using `systemd --user`)
- Stdout and stderr streams are just inherited file descriptors, and they are separate (docker mixes the two; it also does expensive copying of the stream from the container to the client using HTTP api)

1.3.4 Filesystems

All files in vagga is kept in `.vagga/container_name/` so you can inspect all *persistent* filesystems easily, without finding cryptic names in some system location, and without sudo

1.3.5 Filesystem Permissions

Docker by default runs programs in container as root. And it's also a root on the host system. So usually in your development project you get files with root owner. While it's possible to specify your uid as a user for running a process in container, it's not possible to do it it portable. I.e. your uid in docker container should have `passwd` entry. And somebody else may have another uid so must have a different entry in `/etc/passwd`. Also if some process realy needs to be root inside the container (e.g. it must spawn processes by different users) you just can't fix it.

Note: In fact you can specify *uid* without adding a `passwd` entry, and that works most of the time. Up to the point some utility needs to lookup info about user.

With help of user namespaces Vagga runs programs as a root inside a container, but it looks like your user outside. So all your files in project dir are still owned by you.

1.3.6 Security

While docker has enterprise support, including security updates. Vagga doesn't have such (yet).

However, Vagga runs nothing with root privileges. So even running root process in guest system it's at least as secure as running any unprivileged program in host sytem. It also uses chroot and linux namespaces for more isolation. Compare it to docker which doesn't consider running as root inside a container secure.

You can apply selinux or apparmor rules for both.

1.3.7 Filesystem Redundancy

Vagga creates each container in `.vagga` as a separate directory. So theoretically it uses more space than layered containers in docker. But if you put that dir on `btrfs` filesystem you can use bedup to achieve much better redundancy than what docker provides.

1.4 Vagga vs Vagrant

Both products do development enviroments easy to setup. However, there is a big difference on how they do their work.

1.4.1 Containers

While vagrant emulates full virtual machine, vagga uses linux containers. So you don't need hardware virtualization and a supervisor. So usually vagga is more light on resources.

Also comparing to vagrant where you run project inside a virtual machine, vagga is suited to run commands inside a container, not a full virtual machine with SSH. In fact many vagga virtual machines don't have a shell and/or a package manager inside.

1.4.2 Commands

While vagrant is concentrated around `vagrant up` and VM boot process. Light containers allows you to test your project in multiple environments in fraction of second without waiting for boot or having many huge processes hanging around.

So instead of having `vagrant up` and `vagrant ssh` we have user-defined commands like `vagga build` or `vagga run` or `vagga build-a-release-tarball`.

1.4.3 Linux-only

While vagrant works everywhere, vagga only works on linux systems with recent kernel and userspace utulities.

If you use a mac, just run vagga inside a vagrant container, just like you used to run docker :)

1.4.4 Half-isolation

Being only a container allows vagga to share memory with host system, which is usually a good thing.

Memory and CPU usage limits can be enforced on vagga programs using cgroups, just like on any other process in linux. Vagga runs only on quite recent linux kernels, which has much more limit capabilities than previous ones.

Also while vagrant allows to forward selected network ports, vagga by default shares network interface with the host system. Isolating and forwarding ports will be implemented soon.

CHAPTER 2

Installation

2.1 Runtime Dependencies

Vagga is compiled as static binary, so it doesn't have many runtime dependencies. It does require user namespaces to be properly set up, which allows Vagga to create and administer containers without having root privlege. This is increasingly available in modern distributions but may need to be enabled manually.

- the `newuidmap`, `newgidmap` binaries are required (either from `shadow` or `uidmap` package)
- known exception for Archlinux: ensure `CONFIG_USER_NS=y` enabled in kernel. Default kernel doesn't contain it, you can check it with:

```
$ zgrep CONFIG_USER_NS /proc/config.gz
```

  See *Arch Linux*

- known exception for Debian and Fedora: some distributions disable unprivileged user namespaces by default. You can check with:

```
$ sysctl kernel.unprivileged_userns_clone
kernel.unprivileged_userns_clone = 1
```

  and to enable:

```
$ sudo sysctl -w kernel.unprivileged_userns_clone=1
kernel.unprivileged_userns_clone = 1
# make available on reboot
$ echo kernel.unprivileged_userns_clone=1 | \
    sudo tee /etc/sysctl.d/50-unprivleged-userns-clone.conf
kernel.unprivileged_userns_clone=1
```

- `/etc/subuid` and `/etc/subgid` should be set up. Usually you need at least 65536 subusers. This will be setup automatically by `useradd` in new distributions. See `man subuid` if not. To check:

```
$ grep -w $(whoami) /etc/sub[ug]id
/etc/subgid:<you>:689824:65536
/etc/subuid:<you>:689824:65536
```

The only other optional dependency is `iptables` in case you will be doing network tolerance testing.

See instructions specific for your distribution below.

2.2 Binary Installation

Note: If you're ubuntu user you should use package. See instructions below.

Visit http://files.zerogw.com/vagga/latest.html to find out latest tarball version. Then run the following:

```
$ wget http://files.zerogw.com/vagga/vagga-0.4.0.tar.xz
$ tar -xJf vagga-0.4.0.tar.xz
$ cd vagga
$ sudo ./install.sh
```

Or you may try more obscure way:

```
$ curl http://files.zerogw.com/vagga/vagga-install.sh | sh
```

Note: Similarly we have a *-testing* variant of both ways:

- http://files.zerogw.com/vagga/latest-testing.html

```
$ curl http://files.zerogw.com/vagga/vagga-install-testing.sh | sh
```

2.3 Ubuntu

To install from vagga's repository just add the following to *sources.list*:

```
deb http://ubuntu.zerogw.com vagga main
```

The process of installation looks like the following:

```
$ echo 'deb http://ubuntu.zerogw.com vagga main' | sudo tee /etc/apt/sources.list.d/vagga.list
deb http://ubuntu.zerogw.com vagga main
$ sudo apt-get update
[.. snip ..]
Get:10 http://ubuntu.zerogw.com vagga/main amd64 Packages [365 B]
[.. snip ..]
Fetched 9,215 kB in 17s (532 kB/s)
Reading package lists... Done
$ sudo apt-get install vagga
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  vagga
0 upgraded, 1 newly installed, 0 to remove and 113 not upgraded.
Need to get 873 kB of archives.
After this operation, 4,415 kB of additional disk space will be used.
WARNING: The following packages cannot be authenticated!
  vagga
Install these packages without verification? [y/N] y
Get:1 http://ubuntu.zerogw.com/ vagga/main vagga amd64 0.1.0-2-g8b8c454-1 [873 kB]
Fetched 873 kB in 2s (343 kB/s)
Selecting previously unselected package vagga.
(Reading database ... 60919 files and directories currently installed.)
```

```
Preparing to unpack .../vagga_0.1.0-2-g8b8c454-1_amd64.deb ...
Unpacking vagga (0.1.0-2-g8b8c454-1) ...
Setting up vagga (0.1.0-2-g8b8c454-1) ...
```

Now vagga is ready to go.

Note: If you are courageous enough, you may try to use `vagga-testing` repository to get new versions faster:

```
deb http://ubuntu.zerogw.com vagga-testing main
```

It's build right from git "master" branch and we are trying to keep "master" branch stable.

2.4 Ubuntu: Old Releases (precise, 12.04)

For old ubuntu you need *uidmap*. It has no dependencies. So if your ubuntu release doesn't have *uidmap* package (as 12.04 does), just fetch it from newer ubuntu release:

```
$ wget http://gr.archive.ubuntu.com/ubuntu/pool/main/s/shadow/uidmap_4.1.5.1-1ubuntu9_amd64.deb
$ sudo dpkg -i uidmap_4.1.5.1-1ubuntu9_amd64.deb
```

Then run same sequence of commands, you run for more recent releases:

```
$ echo 'deb http://ubuntu.zerogw.com vagga main' | sudo tee /etc/apt/sources.list.d/vagga.list
$ sudo apt-get update
$ sudo apt-get install vagga
```

If your ubuntu is older, or you upgraded it without recreating a user, you need to fill in `/etc/subuid` and `/etc/subgid`. Command should be similar to the following:

```
$ echo "$(id -un):100000:65536" | sudo tee /etc/subuid
$ echo "$(id -un):100000:65536" | sudo tee /etc/subgid
```

Or alternatively you may edit files by hand.

Now your vagga is ready to go.

2.5 Arch Linux

Default Arch Linux kernel doesn't contain `CONFIG_USER_NS=y` in configuration, you can check it with:

```
$ zgrep CONFIG_USER_NS /proc/config.gz
```

To install kernel with this config enabled you can use linux-user-ns-enabled AUR package:

```
$ yaourt -S linux-user-ns-enabled
```

It's based on `core/linux` package and differs only with this option. After it's compiled, update your bootloader config (for GRUB it's `grub-mkconfig`).

Installing vagga from binary archive using AUR package (please note that vagga-bin located in new AUR4 repository so it should be activated in your system):

```
$ yaourt -S vagga-bin
```

If your `shadow` package is older than `4.1.5`, or you upgraded it without recreating a user, after installation you may need to fill in `/etc/subuid` and `/etc/subgid`. You can check if you need it with:

```
$ grep $(id -un) /etc/sub[ug]id
```

If output is empty, you have to modify these files. Command should be similar to the following:

```
$ echo "$(id -un):100000:65536" | sudo tee -a /etc/subuid
$ echo "$(id -un):100000:65536" | sudo tee -a /etc/subgid
```

2.6 Building From Source

The only supported way to build from source is to build with vagga. It's as easy as installing vagga and running `vagga make` inside the the clone of a vagga repository.

Note: First build of vagga is **very slow** because it needs to build rust with musl standard library. When I say slow, I mean it takes about 1 (on fast i7) to 4 hours and more on a laptop. Subsequent builds are much faster (less than minute on my laptop).

There is also a `vagga build-packages` command which builds ubuntu and binary package and puts them into `dist/`.

To install run:

```
$ make install
```

or just (in case you don't have `make` in host system):

```
$ ./install.sh
```

Both support `PREFIX` and `DESTDIR` environment variables.

Note: We stopped supporting out-of-container build because rust with musl is just too hard to build. In case you are brave enough, just look at `vagga.yaml` in the repository. It's pretty easy to follow and there is everything needed to build rust-musl with dependencies.

CHAPTER 3

Configuration

Main vagga configration file is `vagga.yaml` it's usually in the root of the project dir. It can also be in `.vagga/vagga.yaml` (but it's not recommended).

The `vagga.yaml` has two sections:

- `containers` – description of the containers
- `commands` – a set of commands defined for the project

3.1 Container Parameters

`setup`
: List of steps that is executed to build container. See *Build Steps* for more info.

`environ-file`
: The file with environment definitions. Path inside the container. The file consists of line per value, where key and value delimited by equals = sign. (Its similar to `/etc/environment` in ubuntu or `EnvironmentFile` in systemd, but doesn't support commands quoting and line wrapping yet)

`environ`
: The mapping, that constitutes environment variables set in container. This overrides `environ-file` on value by value basis.

`uids`
: List of ranges of user ids that need to be mapped when container runs. User must have some ranges in `/etc/subuid` to run this contiainer, and total size of all allowed ranges must be larger or equal to the sum of sizes of all ranges specified in `uids` parameter. Currenlty vagga applies ranges found in `/etc/subuid` one by one until all ranges are satisfied. It's not always optimal or desirable, we will allow to customize mapping in later versions.

  Default value is `[0-65535]` which is usually good enough. Unless you have smaller number of uids available or run container in container.

`gids`
: List of ranges of group ids that need to be mapped when container runs. User must have some ranges in `/etc/subgid` to run this contiainer, and total size of all allowed ranges must be larger or equal to the sum of sizes of all ranges specified in `gids` parameter. Currenlty vagga applies ranges found in `/etc/subgid` one by one until all ranges are satisfied. It's not always optimal or desirable, we will allow to customize mapping in later versions.

  Default value is `[0-65535]` which is usually good enough. Unless you have smaller number of gids available or run container in container.

volumes
The mapping of mount points to the definition of volume. Allows to mount some additional filesystems inside the container. See *Volumes* for more info. Default is:

```
volumes:
    /tmp: !Tmpfs { size: 100Mi, mode: 0o1777 }
```

Note: You must create a folder for each volume. See *Build Steps* for documentation.

resolv-conf-path
The path in container where to copy `resolv.conf` from host. If the value is `null`, no file is copied. Default is `/etc/resolv.conf`. Its useful if you symlink `/etc/resolv.conf` to some tmpfs directory in `setup` and point `resolv-conf-path` to the directory.

Note: The default behavior for vagga is to overwrite `/etc/resolv.conf` inside the container at the start. It's violation of read-only nature of container images (and visible for all containers). But as we are doing only single-machine development environments, it's bearable. We are seeking for a better way without too much hassle for the user. But you can use the symlink if it bothers you.

hosts-file-path
The path in container where to copy `/ets/hosts` from host. If the value is `null`, no file is copied. Default is `/etc/hosts`. The setting intention is very similar to *`resolv-conf-path`*, so same condiderations applied.

auto-clean
(experimental) Do not leave multiple versions of container lying around. Removes old container version after new is successfully build. This is mostly useful for containers which depend on binaries locally built (i.e. the ones that are never reproduced in future because of timestamp). For most containers it's a bad idea because it doesn't allow to switch between branches using source-control quickly. Better use `vagga _clean --old` if possible.

3.2 Commands

Every commands in `vagga.yaml` is mapping with a tag that denotes command type. The are two command types `!Command` and `!Supervise` illustrated by the following example:

```
containers: {ubuntu: ... }
commands:
  bash: !Command
    description: Run bash shell inside the container
    container: ubuntu
    run: /bin/bash
  download: !Supervise
    description: Download two files simultaneously
    children:
      amd64: !Command
        container: ubuntu
        run: wget http://cdimage.ubuntu.com/ubuntu-core/trusty/daily/current/trusty-core-amd64.tar.gz
      i386: !Command
        container: ubuntu
        run: wget http://cdimage.ubuntu.com/ubuntu-core/trusty/daily/current/trusty-core-i386.tar.gz
```

3.2.1 Common Parameters

These parameters work for both kinds of commands:

description
: Description that is printed in when vagga is run without arguments

banner
: The message that is printed before running process(es). Useful for documenting command behavior.

banner-delay
: The seconds to sleep before printing banner. For example if commands run a web service, banner may provide a URL for accessing the service. The delay is used so that banner is printed after service startup messages not before. Note that currently vagga sleeps this amount of seconds even if service is failed immediately.

epilog
: The message printed after command is run. It's printed only if command returned zero exit status. Useful to print further instructions, e.g. to display names of build artifacts produced by command.

3.2.2 Parameters of *!Command*

container
: The container to run command in

run
: The command to run. It's either a string (which is passed to `/bin/sh -c`) or a list of command and arguments.

work-dir
: The working directory to run in. Path relative to project root. By default command is run in the same directory where vagga started (sans the it's mounted as `/work` so the output of `pwd` would seem to be different)

accepts-arguments
: Denotes whether command accepts additional arguments. Defaults to `false` for shell commands, and `true` for regular commands.

environ
: The mapping of environment to pass to command. This overrides environment specified in container on value by value basis.

pid1mode
: This denotes what is run as pid 1 in container. It may be `wait`, `wait-all-children` or `exec`. The default `wait` is ok for most regular processes. See *What's Special With Pid 1?* for more info.

write-mode
: The parameter specifies how container's base file system is used. By default container is immutable (corresponds to the `read-only` value of the parameter), which means you can only write to the `/tmp` or to the `/work` (which is your project directory).

  Another option is `transient-hard-link-copy`, which means that whenever command is run, create a copy of the container, consisting of hard-links to the original files, and remove the container after running command. Should be used with care as hard-linking doesn't prevent original files to be modified. Still very useful to try package installation in the system. Use `vagga _build --force container_name` to fix base container if that was modified.

user-id
: The user id to run command as. If the `external-user-id` is omitted this has same effect like using `sudo -u` inside container (except it's user id instead of user name)

external-user-id
: **(experimental)** This option allows to map the `user-id` as seen by command itself to some other user id inside container namespace (the namespace which is used to build container). To make things a little less confusing, the following two configuration lines:

```
user-id: 1
external-user-id: 0
```

Will make your command run as user id 1 visible inside the container (which is "daemon" or "bin" depending on distribution). But outside the container it will be visible as your user (i.e. user running vagga). Which effectively means you can create/modify files in project directory without permission errors, but `tar` and other commands which have different behaviour when running as root would think they are not root (but has user id 1)

3.2.3 Parameters of *!Supervise*

mode
: The set of processes to supervise and mode. See *Supervision* for more info

children
: A mapping of name to child definition of children to run. All children are started simultaneously.

kill-unresponsive-after
: (default *2* seconds) If some process have exited (in `stop-on-failure` mode) vagga will send TERM signal to all other processes. If they don't finish in the specified number of seconds, vagga will kill them with KILL signal (so they finish without being able to intercept singal unconditionally). If you don't like this behavior set the parameter to some large value.

3.3 Build Steps

Build commands are tagged values in your container definition. For example:

```
containers:
  ubuntu:
    setup:
    - !Ubuntu trusty
    - !Install [python]
```

This contains two build commands `!Ubuntu` and `!Install`. They mostly run sequentially, but some of them are interesting, for example `!BuildDeps` installs package right now, but also removes package at the end of the build to keep container smaller and cleaner.

3.3.1 Generic Installers

To run arbitrary shell command use `!Sh`:

```
setup:
- !Ubuntu trusty
- !Sh "apt-get install -y python"
```

If you have more than one-liner you may use YAMLy *literal* syntax for it:

```
setup:
- !Ubuntu trusty
- !Sh |
   wget somepackage.tar.gz
```

```
  tar -xzf somepackage.tar.gz
  cd somepackage
  make && make install
```

Warning: The `!Sh` command is run by `/bin/sh -exc`. With the flags meaning `-e` – exit if any command fails, `-x` – print command before executing, `-c` – execute command. You may undo `-ex` by inserting `set +ex` at the start of the script. But it's not recommended.

To run `!Sh` you need `/bin/sh`. If you don't have shell in container you may use `!Cmd` that runs command directly:

```
setup:
# ...
- !Cmd [/usr/bin/python, '-c', 'print "hello from build"']
```

To install a package of any (supported) linux distribution just use `!Install` command:

```
containers:

  ubuntu:
    setup:
    - !Ubuntu trusty
    - !Install [python]

  ubuntu-precise:
    setup:
    - !Ubuntu precise
    - !Install [python]

  alpine:
    setup:
    - !Alpine v3.1
    - !Install [python]
```

Occasionally you need some additional packages to use for container building, but not on final machine. Use `!BuildDeps` for them:

```
setup:
- !Ubuntu trusty
- !Install [python]
- !BuildDeps [python-dev, gcc]
- !Sh "make && make install"
```

The `python-dev` and `gcc` packages from above will be removed after building whole container.

To add some environment arguments to subsequent build commands use `!Env`:

```
setup:
# ...
- !Env
  VAR1: value1
  VAR2: value2
- !Sh "echo $VAR1 / $VAR2"
```

Note: The `!Env` command doesn't add environment variables for processes run after build. Use `environ` setting for that.

Sometimes you want to rebuild container when some file changes. For example if you have used the file in the build.

There is a `!Depends` command which does nothing per se, but add a dependency. The path must be relative to your project directory (the dir where `vagga.yaml` is). For example:

```
setup:
# ...
- !Depends requirements.txt
- !Sh "pip install -r requirements.txt"
```

To download and unpack tar archive use `!Tar` command:

```
setup:
- !Tar
  url: http://something.example.com/some-project-1.0.tar.gz
  sha256: acd1234...
  path: /
  subdir: some-project-1.0
```

Only `url` field is mandatory. The `path` is target path to unpack into, and `subdir` is a dir inside tar file. By default `path` is root of new filesystem. The `subdir` is a dir inside the tar file, if omitted whole tar archive will be unpacked. You *can* use `!Tar` command to download and unpack the root filesystem from scratch.

There is a shortcut to download tar file and build and install from there, which is `!TarInstall`:

```
setup:
- !TarInstall
  url: http://static.rust-lang.org/dist/rust-0.12.0-x86_64-unknown-linux-gnu.tar.gz
  sha256: abcd1234...
  subdir: rust-0.12.0-x86_64-unknown-linux-gnu
  script: ./install.sh --prefix=/usr
```

Only the `url` is mandatory here too. The `script` is by default `./configure --prefix=/usr; make; make install`. It's run in `subdir` of unpacked archive. If `subdir` is omitted it's run in the *only* subdirectory of the archive. If archive contains more than one directory and `subdir` is empty, it's an error, however you may use `.` as `subdir`.

To remove some data from the image after building use `!Remove` command:

```
setup:
# ...
- !Remove /var/cache/something
```

To clean directory but ensure that directory exists use `!EmptyDir` command:

```
setup:
# ...
- !EmptyDir /tmp
```

Note: The `/tmp` directory is declared as `!EmptyDir` implicitly for all containers.

To ensure that directory exists use `!EnsureDir` command. It's very often used for future mount points:

```
setup:
# ...
- !EnsureDir /sys
- !EnsureDir /dev
- !EnsureDir /proc
```

Note: The `/sys`, `/dev` and `/proc` directories are created automatically for all containers.

Sometimes you want to keep some cache between builds of container or similar containers. Use `!CacheDirs` for that:

```
setup
# ...
- !CacheDirs { "/var/cache/apt": "apt-cache" }
```

Mutliple directories may be specified at once.

Warning: The "apt-cache" name is a name of the directory like `.vagga/.cache/apt-cache`. So the directory is shared both between all the containers and all the different builders (not only same versions of the single container). In case user enabled `shared-cache` the folder will be also shared between containers of different projects.

Sometimes you just want to write a file in target system:

```
setup:
# ...
- !Text
  /etc/locale.conf: |
     LANG=en_US.UTF-8
     LC_TIME=uk_UA.UTF-8
```

Note: You can use any YAML'y syntax for file body just the "literal" one which starts with a pipe `|` character is the most handy one

3.3.2 Ubuntu

To install base ubuntu system use:

```
setup:
- !Ubuntu trusty
```

Potentially any ubuntu long term support release instead of `trusty` should work. To install a non LTS release, use:

```
setup:
- !UbuntuRelease { version: 14.10 }
```

To install any ubuntu package use generic `!Install` command:

```
setup:
- !Ubuntu trusty
- !Install python
```

Many interesting ubuntu packages are in the "universe" repository, you may add it by series of `!UbuntuRepo` commands (see below), but there is shortcut `!UbuntuUniverse`:

```
setup:
- !Ubuntu trusty
- !UbuntuUniverse
- !Install [checkinstall]
```

The `!UbuntuRepo` command adds additional repository. For example, to add marathon repository you may write:

```
setup:
- !Ubuntu trusty
```

```
- !UbuntuRepo
  url: http://repos.mesosphere.io/ubuntu
  suite: trusty
  components: [main]
- !Install [mesos, marathon]
```

This effectively adds repository and installs `mesos` and `marathon` packages.

Note: Probably the key for repository should be added to be able to install packages.

3.3.3 Alpine

To install base alpine system use:

```
setup:
- !Alpine v3.1
```

Potentially any alpine version instead of `v3.1` should work.

To install any alpine package use generic `!Install` command:

```
setup:
- !Alpine v3.1
- !Install [python]
```

3.3.4 Npm Installer

You can build somewhat default nodejs environment using `!NpmInstall` command. For example:

```
setup:
- !Ubuntu trusty
- !NpmInstall [react-tools]
```

All node packages are installed as `--global` which should be expected. If no distribution is specified before the `!NpmInstall` command, the implicit `!Alpine v3.1` (in fact the latest version) will be executed.

```
setup:
- !NpmInstall [react-tools]
```

So above should just work as expected if you don't need any special needs. E.g. it's usually perfectly ok if you only use node to build static scripts.

The following `npm` features are supported:

- Specify `package@version` to install specific version **(recommended)**
- Use `git://` url for the package. In this case git will be installed for the duration of the build automatically
- Bare `package_name` (should be used only for one-off environments)

Other forms may work, but are unsupported for now.

Note: The `npm` and additional utilities (like `build-essential` and `git`) will be removed after end of container building. You must `!Install` them explicitly if you rely on them later.

3.3.5 Python Installer

There are two separate commands for installing packages for python2 and python3. Here is a brief example:

```
setup:
- !Ubuntu trusty
- !Py2Install [sphinx]
```

Currently packages are installed by system pip. We consider this an implementation detail and will use latest pip in future. The `python-dev` headers are installed for the time of the build too. Both `python-dev` and `pip` are removed when installation is finished.

The following `pip` package specification formats are supported:

- The `package_name==version` to install specific version **(recommended)**
- Bare `package_name` (should be used only for one-off environments)
- The `git+` and `hg+` links (the git and mercurial are installed as build dependency automatically), since vagga 0.4 `git+https` and `hg+https` are supported too (required installing `ca-ceritificates` manually before)

All other forms may work but not supported. Specifying command-line arguments instead of package names is not supported. To configure pip use `!PipConfig` directive. In the example there are full list of parameters:

```
setup:
- !Ubuntu trusty
- !PipConfig
  index-urls: ["http://internal.pypi.local"]
  find-links: ["http://internal.additional-packages.local"]
  dependencies: true
- !Py2Install [sphinx]
```

They should be self-descriptive. Note unlike in pip command line we use single list both for primary and "extra" indexes. See pip documentation for more info about options

---

Note: By default `dependencies` is false. Which means pip is run with `--no-deps` option. Which is recommended way for setting up isolated environments anyway. Even `setuptools` are not installed by default. To see list of dependencies and their versions you may use `pip freeze` command.

Better way to specify python dependencies is to use "requirements.txt":

```
setup:
- !Ubuntu trusty
- !Py3Requirements "requirements.txt"
```

This works the same as `Py3Install` including auto-installing of version control packages and changes tracking. I.e. It will rebuild container when "requirements.txt" change. So ideally in python projects you may use two lines above and that's it.

The `Py2Requirements` command exists too.

Note: The "requirements.txt" is checked semantically. I.e. empty lines and comments are ignored. In current implementation the order of items is significant but we might remove this restriction in the future.

3.3.6 Dependent Containers

Sometimes you want to build on top of another container. For example, container for running tests might be based on production container, but it might add some test utils. Use `!Container` command for that:

```
container:
  base:
    setup:
    - !Ubuntu trusty
    - !Py3Install [django]
  test:
    setup:
    - !Container base
    - !Py3Install [nosetests]
```

It's also sometimes useful to freeze some part of container and test next build steps on top of of it. For example:

```
container:
  temporary:
    setup:
    - !Ubuntu trusty
    - !TarInstall
      url: http://download.zeromq.org/zeromq-4.1.0-rc1.tar.gz
  web:
    setup:
    - !Container temporary
    - !Py3Install [pyzmq]
```

In this case when you try multiple different versions of pyzmq, the zeromq itself will not be rebuilt. When you're done, you can append build steps and remove the `temporary` container.

Sometimes you need to generate (part of) `vagga.yaml` itself. For some things you may just use shell scripting. For example:

```
container:
  setup:
  - !Ubuntu trusty
  - !Env { VERSION: 0.1.0 }
  - !Sh "apt-get install somepackage==$VERSION"
```

Note: Environment of user building container is always ignored during build process (but may be used when running command).

In more complex scenarios you may want to generate real `vagga.yaml`. You may use that with ancillary container and `!SubConfig` command. For example, here is how we use a docker2vagga script to transform `Dockerfile` to vagga config:

```
docker-parser:
  setup:
  - !Alpine v3.1
  - !Install [python]
  - !Depends Dockerfile
  - !Depends docker2vagga.py
  - !Sh 'python ./docker2vagga.py > /docker.yaml'

somecontainer:
  setup:
  - !SubConfig
```

```
    source: !Container docker-parser
    path: docker.yaml
    container: docker-smart
```

Few comments:

- – container used for build, it's rebuilt automatically as a dependency for "somecontainer"
- – normal dependency rules apply, so you must add external files that are used to generate the container and vagga file in it
- – put generated vagga file inside a container
- – the "path" is relative to the source if the latter is set
- – name of the container used *inside* a "docker.yaml"

Warning: The functionality of `!SubConfig` is experimental and is a subject to change in future. In particular currently the `/work` mount point and current directory used to build container are those of initial `vagga.yaml` file. It may change in future.

The `!SubConfig` command may be used to include some commands from another file without building container. Just omit `generator` command:

```
subdir:
  setup:
  - !SubConfig
    path: subdir/vagga.yaml
    container: containername
```

The YAML file used may be a partial container, i.e. it may contain just few commands, installing needed packages. The other things (including the name of the base distribution) can be set by original container:

```
# vagga.yaml
containers:
  ubuntu:
    setup:
    - !Ubuntu trusty
    - !SubConfig
      path: packages.yaml
      container: packages
  alpine:
    setup:
    - !Alpine v3.1
    - !SubConfig
      path: packages.yaml
      container: packages

# packages.yaml
containers:
  packages:
    setup:
    - !Install [redis, bash, make]
```

3.4 Volumes

Volumes define some additional filesystems to mount inside container. The default configuration is similar to the following:

```
volumes:
  /tmp: !Tmpfs
    size: 100Mi
    mode: 0o1777
  /run: !Tmpfs
    size: 100Mi
    mode: 0o766
    subdirs:
      shm: { mode: 0o1777 }
```

Warning: Volumes are **not** mounted during container build, only when some command is run.

Available volume types:

Tmpfs
: Mounts `tmpfs` filesystem. There are two parameters for this kind of volume:

  - `size` – limit for filesystem size in bytes. You may use suffixes `k`, `M`, `G`, `ki`, `Mi`, `Gi` for bigger units. The ones with `i` are for power of two units, the other ones are for power of ten;
  - `mode` – filesystem mode.
  - `subdirs` – a mapping for subdirectories to create inside tmpfs, for example:

```
volumes:
   /var: !Tmpfs
       mode: 0o766
       subdirs:
           lib: # default mode is 0o766
           lib/tmp: { mode: 0o1777 }
           lib/postgres: { mode: 0o700 }
```

  The only property currently supported on a directory is `mode`.

VaggaBin
: Mounts vagga binary directory inside the container (usually it's contained in `/usr/lib/vagga` in host system). This may be needed for *Network Testing* or may be for vagga in vagga (i.e. container in container) use cases.

BindRW
: Binds some folder inside a countainer to another folder. Essentially it's bind mount (the `RW` part means read-writeable). The path must be absolute (inside the container). This directive can't be used to expose some directories not already visible. This is often used to put some temporary directory in development into well-defined production location. For example:

```
volumes:
  /var/lib/mysql: !BindRW /work/tmp/mysql
```

3.5 Upgrading

3.5.1 Upgrading 0.3.x -> 0.4.x

The release is focused on migrating from small amount of C code to "unshare" crate and many usability fixes, including ones which have small changes in semantics of configuration. The most important changes:

- The `!Sh` command now runs shell with `-ex` this allows better error reporting (but may change semantics of script for some obscure cases)
- There is now *`kill-unresponsive-after`* setting for `!Supervise` commands with default value of `2`. This means that processes will shut down unconditionally two seconds after `Ctrl+C`.

See Release Notes and Github for all changes.

3.5.2 Upgrading 0.2.x -> 0.3.x

This upgrade should be seamless. The release is focused on migrating code from pre-1.0 Rust to... well... rust 1.2.0.

Other aspect of code migration is that it uses `musl` libc. So building vagga from sources is more complex now. (However it's as easy as previous version if you build with vagga itself, except you need to wait until rust builds for the first time).

3.5.3 Upgrading 0.1.x -> 0.2.x

There are basically two things changed:

1. The way how containers (images) are built
2. Differentiation of commands

Building Images

Previously images was build by two parts: `builder` and `provision`:

```
rust:
  builder: ubuntu
  parameters:
    repos: universe
    packages: make checkinstall wget git uidmap
  provision: |
    wget https://static.rust-lang.org/dist/rust-0.12.0-x86_64-unknown-linux-gnu.tar.gz
    tar -xf rust-0.12.0-x86_64-unknown-linux-gnu.tar.gz
    cd rust-0.12.0-x86_64-unknown-linux-gnu
    ./install.sh --prefix=/usr
```

Now we have a sequence of steps which perform work as a `setup` setting:

```
rust:
  setup:
  - !Ubuntu trusty
  - !UbuntuUniverse ~
  - !TarInstall
    url: http://static.rust-lang.org/dist/rust-1.0.0-alpha-x86_64-unknown-linux-gnu.tar.gz
    script: "./install.sh --prefix=/usr"
```

```
  - !Install [make, checkinstall, git, uidmap]
  - !Sh "echo Done"
```

Note the following things:

- Downloading and unpacking base os is just a step. Usually the first one.
- Steps are executed sequentially
- The amount of work at each step is different as well as different level of abstractions
- The `provision` thing may be split into several `!Sh` steps in new vagga

The description of each step is in *Reference*.

By default `uids` and `gids` are set to `[0-65535]`. This default should be used for all contianers unless you have specific needs.

The `tmpfs-volumes` key changed for the generic `volumes` key, see *Volumes* for more info.

The `ensure-dirs` feature is now achieved as `- !EnsureDir dirname` build step.

Commands

Previously type of *command* was differentiated by existence of `supervise` and `command`/`run` key.

Now first kind of command is marked by `!Command` yaml tag. The `command` and `run` differentation is removed. When `run` is a list it's treated as a command with arguments, if `run` is a string then it's run by shell.

The `!Supervise` command contains the processes to run in `children` key.

See *reference* for more info.

Missing Features

The following features of vagga 0.1 are missing in vagga 0.2. We expect that they were used rarely of at all.

- Building images by host package manager (builders: debian-debootstrap, debian-simple, arch-simple). The feature is considered too hard to use and depends on the host system too much.
- Arch and Nix builders. Will be added later. We are not sure if we'll keep a way to use host-system nix to build nix container.
- Docker builder. It was simplistic and just PoC. The builder will be added later.
- Building images without `uidmap` and properly set `/etc/subuid` and `/etc/subgid`. We believe that all systems having `CONFIG_USER_NS` enabled have subuids either already set up or easy to do.
- The `mutable-dirs` settings. Will be replaced by better mechanism.

3.6 Supervision

Vagga may supervise multiple processes with single command. This is very useful for running multiple-component and/or networking systems.

By supervision we mean running multiple processes and watching until all them exit. Each process is run in it's own container. Even if two processes share the key named "container", which means they share same root filesystem, they run in different namespaces, so they don't share `/tmp`, `/proc` and so on.

3.6.1 Supervision Modes

There are three basic modes of operation:

- `stop-on-failure` – stops all processes as soon as any single one is dead (default)
- `wait-all` – wait for all processes to finish
- `restart` – always restart dead processes

In any mode of operation supervisor itself never exits until all the children are dead. Even when you kill supervisor with `kill -9` or `kill -KILL` all children will be killed with `-KILL` signal too. I.e. with the help of namespaces and good old `PR_SET_PDEATHSIG` we ensure that no process left when supervisor killed, no one is reparented to `init`, all traces of running containers are cleared. Seriously. It's very often a problem with many other ways to run things on development machine.

Stop on Failure

It's not coincidence that `stop-on-failure` mode is default. It's very useful mode of operation for running on development machine.

Let me show an example:

```
commands:
  run_full_app: !Supervise
    mode: stop-on-failure
    children:
      web: !Command
        container: python
        run: "python manage.py runserver"
      celery: !Command
        container: python
        run: "python manage.py celery worker"
```

Imagine this is a web application written in python (`web` process), with a work queue (`celery`), which runs some long-running tasks in background.

When you start both processes `vagga run_full_app`, often many log messages with various levels of severity appear, so it's easy to miss something. Imagine you missed that celery is not started (or dead shortly after start). You go to the web app do some testing, start some background task, and wait for it to finish. After waiting for a while, you start suspect that something is wrong. But celery is dead long ago, so skimming over recent logs doesn't show up anything. Then you look at processes: "Oh, crap, there is no celery". This is time-wasting.

With `stop-on-failure` you'll notice that some service is down immediately.

In this mode vagga returns `1` if some process is dead before vagga received `SIGINT` or `SIGTERM` signal. Exit code is `0` if one of the two received by vagga. And an `128+signal` code when any other singal was sent to supervisor (and propagated to other processes).

Wait

In `wait` mode vagga waits that all processes are exited before shutting down. If any is dead, it's ok, all other will continue as usual.

This mode is intended for running some batch processing of multiple commands in multiple containers. All processes are run in parallel, like with other modes.

Note: Depending on `pid1mode` of each proccess in each container vagga will wait either only for process spawned by vagga (`pid1mode: wait` or `pidmode: exec`), or for all (including daemonized) processes spawned by that command (`pid1mode: wait-all-children`). See *What's Special With Pid 1?* for details.

Restart

This is a supervision mode that most other supervisors obey. If one of the processes is dead, it will be restarted without messing with other processes.

It's not recommended mode for workstations but may be useful for staging server (Currenly, we do not recommend running vagga in production at all).

Note: The whole container is restarted on process failure, so `/tmp` is clean, all daemonized processes are killed, etc. See also *What's Special With Pid 1?*.

3.6.2 Tips

Restarting a Subset Of Processes

Sometimes you may work only on one component, and don't want to restart the whole bunch of processes to test just one thing. You may run two supervisors, in different tabs of a terminal. E.g:

```
# run everything, except the web process we are debugging
$ vagga run_full_app --exclude web
# then in another tab
$ vagga run_full_app --only web
```

Then you can restart `web` many times, without restarting everything.

3.7 What's Special With Pid 1?

The first process started by the linux kernel gets PID 1. Similarly when new PID namespace is created first process started in that namespace gets PID 1 (the PID as seen by the processes in that namespace, in parent namespace it gets assigned other PID).

The process with PID 1 differ from other processes in the following ways

- When the process with pid 1 die for any reason, all other processes are killed with `KILL` signal
- When any process having children dies for any reason, its children are reparented to process with PID 1
- Many signals which have default action of `Term` do not have one for PID 1.

I may look like the most disruping one is first. But in practice most inconvenient one for development purposes is the last one, because, effectively you can't stop process by sending `SIGTERM` or `SIGINT`, if process have not installed a singal handler.

At the end of the day all above means most processes that where not explicitly designed to run as PID 1 (which are all applicactions except supervisors), do not run well. Vagga fixes that by not running process as PID 1.

In fact there are three modes of operation of PID 1 supported by vagga (set by `pid1mode` parameter in *command configuration*):

- `wait` – (default) run command (usually it gets PID 2) and wait until it exits
- `wait-all-children` – run command, then wait all processes in namespace to finish
- `exec` – run the command as PID 1, useful only if command itself is process supervisor like upstart, systemd or supervisord

Note that in `wait` and `exec` modes, when you kill vagga itself with a signal, it will propagate the signal to the command itself. In `wait-all-children` mode, signal will be propagated to all processes in the container (even if it's some supplementary command run as a child of some intermediary process). This is rarely the problem.

3.8 Containers

Example of one container defined:

```
containers:
  sphinx:
    setup:
    - !Ubuntu trusty
    - !Install [python-sphinx, make]
```

The YAML above defines a container named `sphinx`, which is built with two steps: download and unpack ubuntu `trusty` base image, and install install packages name `python-sphinx, make` inside the container.

3.9 Commands

Example of command defined:

```
commands:
  build-docs: !Command
    description: Build vagga documentation using sphinx
    container: sphinx
    work-dir: docs
    run: make
```

The YAML above defines a command named `build-docs`, which is run in container named `sphinx`, that is run in `docs/` sub dir of project, and will run command `make` in container. So running:

```
> vagga build-docs html
```

Builds html docs using sphinx inside a container.

See *commands* for comprehensive description of how to define commands.

CHAPTER 4

Running

Usually running vagga is as simple as:

```
$ vagga run
```

To find out commands you may run bare `vagga`:

```
$ vagga
Available commands:
    run                 Run mysample project
    build-docs          Build documentation using sphinx
```

4.1 Command Line

When runnin `vagga`, it finds the `vagga.yaml` or `.vagga/vagga.yaml` file in current working directory or any of its parents and uses that as a project root directory.

When running `vagga` without arguments it displays a short summary of which commands are defined by `vagga.yaml`, like this:

```
$ vagga
Available commands:
    run                 Run mysample project
    build-docs          Build documentation using sphinx
```

Refer to *Commands* for more information of how to define commands for vagga.

There are also builtin commands. All builtin commands start with underscore _ character to be clearly distinguished from user-defined commands.

4.1.1 Builtin Commands

All commands have `--help`, so we don't duplicate all command-line flags here

- `vagga _run CONTAINER CMD ARG...` – run arbitrary command in container defined in vagga.yaml
- `vagga _build CONTAINER` – builds container without running a command
- `vagga _clean` – removes images and temporary files created by vagga. To fully remove `.vagga` directory you can run `vagga _clean --everything`. For other operations see `vagga _clean --help`
- `vagga _list` – list of commands (including builtin ones when using `--builtin` flag)

- `vagga _version_hash` – prints version hash for the container, might be used in some automation scripts

4.1.2 Normal Commands

If *command* declared as `!Command` you get a command with the following usage:

```
Usage:
    vagga [OPTIONS] some_command [ARGS ...]

Runs a command in container, optionally builds container if that does not
exists or outdated. Run `vagga` without arguments to see the list of
commands.

positional arguments:
  some_command          Your defined command
  args                  Arguments for the command

optional arguments:
  -h,--help             show this help message and exit
  -E,--env,--environ NAME=VALUE
                        Set environment variable for running command
  -e,--use-env VAR      Propagate variable VAR into command environment
  --no-build            Do not build container even if it is out of date.
                        Return error code 29 if it's out of date.
  --no-version-check    Do not run versioning code, just pick whatever
                        container version with the name was run last (or
                        actually whatever is symlinked under
                        `.vagga/container_name`). Implies `--no-build`
```

All the `ARGS` that follow command are passed to the command even if they start with dash `-`.

4.1.3 Supervise Commands

If *command* declared as `!Supervise` you get a command with the following usage:

```
Usage:
    vagga run [OPTIONS]

Run full server stack

optional arguments:
  -h,--help             show this help message and exit
  --only PROCESS_NAME [...]
                        Only run specified processes
  --exclude PROCESS_NAME [...]
                        Don't run specified processes
  --no-build            Do not build container even if it is out of date.
                        Return error code 29 if it's out of date.
  --no-version-check    Do not run versioning code, just pick whatever
                        container version with the name was run last (or
                        actually whatever is symlinked under
                        `.vagga/container_name`). Implies `--no-build`
```

Currently there is no way to provide additional arguments to commands declared with `!Supervise`.

The `--only` and `--exclude` arguments are useful for isolating some single app to a separate console. For example, if you have `vagga run` that runs full application stack including a database, cache, web-server and your little django

application, you might do the following:

```
$ vagga run --exclude django
```

Then in another console:

```
$ vagga run --only django
```

Now you have just a django app that you can observe logs from and restart independently of other applications.

4.2 Environment

There are a few ways to pass environment variables from the runner's environment into a container.

Firstly, any enviroment variable that starts with `VAGGAENV_` will have it's prefix stripped, and exposed in the container's environment:

```
$ VAGGAENV_FOO=BAR vagga _run container printenv FOO
BAR
```

The `-e` or `--use-env` command line option can be used to mark environment variables from the runner's environment that should be passed to container:

```
$ FOO=BAR vagga --use-env=FOO _run container printenv FOO
BAR
```

And finally the `-E`, `--env` or `--environ` command line option can be used to assign an environment variable that will be passed to the container:

```
$ vagga --environ FOO=BAR _run container printenv FOO
BAR
```

4.3 Settings

4.3.1 Global Settings

Settings are searched for in one of the following files:

- `$HOME/.config/vagga/settings.yaml`
- `$HOME/.vagga/settings.yaml`
- `$HOME/.vagga.yaml`

Supported settings:

`storage-dir`
: Directory where to put images build by vagga. Usually they are stored in `.vagga` subdirectory of the project dir. It's mostly useful when the `storage-dir` points to a directory on a separate partition. Path may start with `~/` which means path is inside the user's home directory.

`cache-dir`
: Directory where to put cache files during the build. This is used to speed up the build process. By default cache is put into `.vagga/.cache` in project directory but this setting allows to have cache directory shared between multiple projects. Path may start with `~/` which means path is inside the user's home directory.

site-settings
: (experimental) The mapping of project paths to settings for this specific project.

proxy-env-vars
: Enable forwarding for proxy environment variables. Default `true`. Environment variables currently that this setting influence currently: `http_proxy`, `https_proxy`, `ftp_proxy`, `all_proxy`, `no_proxy`.

All project-local settings are also allowed here.

4.3.2 Project-Local Settings

Project-local settings may be in the project dir in:

```
* ``.vagga.settings.yaml``
* ``.vagga/settings.yaml``
```

All project-local settings are also allowed in global config.

While settings can potentially be checked-in to version control it's advised not to do so.

version-check
: If set to `true` (default) vagga will check if the container that is already built is up to date with config. If set to `false` vagga will use any container with same name already built. It's only useful for scripts for performance reasons or if you don't have internet and containers are not too outdated.

ubuntu-mirror
: Set to your preferred ubuntu mirror. By default it's `mirror://mirrors.ubuntu.com/mirrors.txt` which means mirror will be determined automatically. Note that it's different from default in ubuntu itself where `http://archive.ubuntu.com/ubuntu/` is the default.

4.4 Errors

The document describes errors when running vagga on various systems. The manual only includes errors which need more detailed explanation and troubleshooting. Most errors should be self-descriptive.

4.4.1 Could not read /etc/subuid or /etc/subgid

The full error might look like:

```
ERROR:vagga::container::uidmap: Error reading uidmap: Can't open /etc/subuid: No such file or directo
WARN:vagga::container::uidmap: Could not read /etc/subuid or /etc/subgid (see http://bit.ly/err_subui
error setting uid/gid mappings: Operation not permitted (os error 1)
```

This means there is no `/etc/subuid` file. It probably means you need to create one. The recommended contents are following:

```
your_user_name:100000:65536
```

You may get another similar error:

```
ERROR:vagga::container::uidmap: Error reading uidmap: /etc/subuid:2: Bad syntax: "user:100000:100O"
WARN:vagga::container::uidmap: Could not read /etc/subuid or /etc/subgid (see http://bit.ly/err_subui
error setting uid/gid mappings: Operation not permitted (os error 1)
```

This means somebody has edited `/etc/subuid` and made an error. Just open the file (note it's owned by root) and fix the issue (in the example the last character should be zero, but it's a letter "O").

4.4.2 Can't find newuidmap or newgidmap

Full error usually looks like:

```
WARN:vagga::process_util: Can't find `newuidmap` or `newuidmap` (see http://bit.ly/err_idmap)
error setting uid/gid mappings: No such file or directory (os error 2)
```

There might be two reasons for this:

1. The binaries are not installed (see below)
2. The commands are not in `PATH`

In the latter case you should fix your `PATH`.

The packages for Ubuntu >= 14.04:

```
$ sudo apt-get install uidmap
```

The Ubuntu 12.04 does not have the package. But you may use the package from newer release (the following version works fine on 12.04):

```
$ wget http://gr.archive.ubuntu.com/ubuntu/pool/main/s/shadow/uidmap_4.1.5.1-1ubuntu9_amd64.deb
$ sudo dpkg -i uidmap_4.1.5.1-1ubuntu9_amd64.deb
```

Most distributions (known: Nix, Archlinux, Fedora), does have binaries as part of "shadow" package, so have them installed on every system.

4.4.3 You should not run vagga as root

Well, sometimes users get some permission denied errors and try to run vagga with sudo. Running as root is **never** an answer.

Here is a quick check list on permission checks:

- Check owner (and permission bits) of `.vagga` subdirectory if it exists, otherwise the directory where `vagga.yaml` is (project dir). In case you have already run vagga as root just do `sudo rm -rf .vagga`
- *Could not read /etc/subuid or /etc/subgid*
- *Can't find newuidmap or newgidmap*
- Check `uname -r` to have version of `3.9` or greater
- Check `sysctl kernel.unprivileged_userns_clone` the setting must either *not exist* at all or have value of `1`
- Check `zgrep CONFIG_USER_NS /proc/config.gz` or `grep CONFIG_USER_NS "/boot/config-`uname -r`"` (ubuntu) the setting should equal to `y`

The error message might look like:

```
You should not run vagga as root (see http://bit.ly/err_root)
```

Or it might look like a warning:

```
WARN:vagga::launcher: You are running vagga as a user different from the owner of project directory.
```

Both show that you don't run vagga by user that owns project. The ligitimate reasons to run vagga as root are:

- If you run vagga in container (i.e. in vagga itself) and the root is not a real root
- If your project dir is owned by root (for whatever crazy reason)

Both cases should inhibit the warning automatically, but as a last resort you may try `vagga --no-owner-check`. If you have good case where this works, please file an issue and we might make the check better.

CHAPTER 5

Network Testing

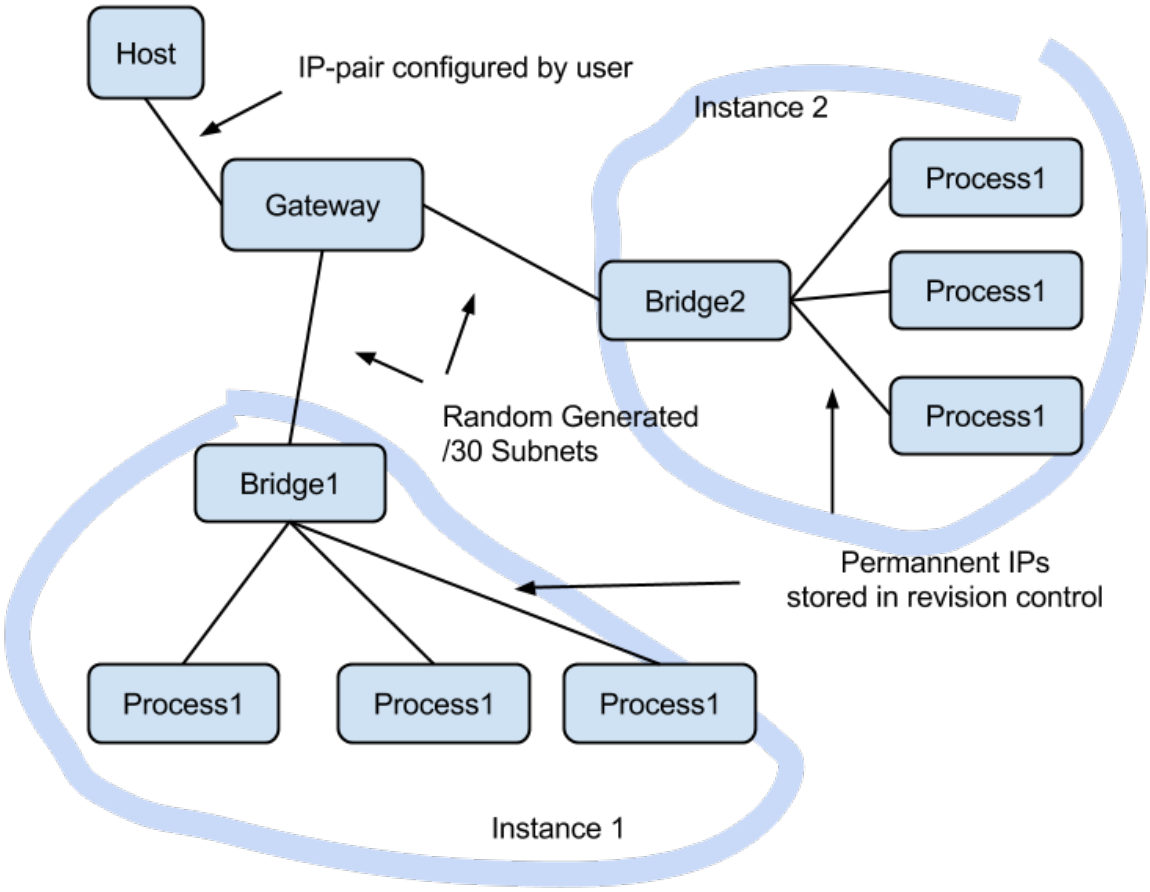
Usually vagga runs processes in host network namespace. But there is a mode for network testing.

5.1 Overview

For testing complex networks we leverage `!Supervise` type of commands to run multiple nodes. But we also need a way to setup network. What we need in particular:

1. The IPs should be hard-coded (i.e. checked in into version control)
2. Multiple different projects running simultaneously (and multiple instances of same project as a special case of it)
3. Containers should be able to access internet if needed

So we use "double-bridging" to get this working, as illustrated below:



The *Setup* section describes how to setup a gateway in the host system, and *Containers* section describes how to configure containers in `vagga.yaml`. And *Partitioning* section describes how to implement tests which break network and create network partitions of various kinds.

5.2 Setup

Unfortunately we can't setup network in fully non-privileged way. So you need to do some preliminary setup. To setup a bridge run:

```
vagga _create_netns
```

Running this will show what commands are going to run:

```
We will run network setup commands with sudo.
You may need to enter your password.

The following commands will be run:
    sudo 'ip' 'link' 'add' 'vagga_guest' 'type' 'veth' 'peer' 'name' 'vagga'
    sudo 'ip' 'link' 'set' 'vagga_guest' 'netns' '16508'
    sudo 'ip' 'addr' 'add' '172.18.255.1/30' 'dev' 'vagga'
    sudo 'sysctl' 'net.ipv4.conf.vagga.route_localnet=1'
```

```
    sudo 'mount' '--bind' '/proc/16508/ns/net' '/run/user/1000/vagga/netns'
    sudo 'mount' '--bind' '/proc/16508/ns/user' '/run/user/1000/vagga/userns'

The following iptables rules will be established:
    ["-I", "INPUT", "-i", "vagga", "-d", "127.0.0.1", "-j", "ACCEPT"]
    ["-t", "nat", "-I", "PREROUTING", "-p", "tcp", "-i", "vagga", "-d", "172.18.255.1", "--dport", "5
    ["-t", "nat", "-I", "PREROUTING", "-p", "udp", "-i", "vagga", "-d", "172.18.255.1", "--dport", "5
    ["-t", "nat", "-A", "POSTROUTING", "-s", "172.18.255.0/30", "-j", "MASQUERADE"]
```

Then immediatelly the commands are run, this will probably request your password by sudo command. The `iptables` commands may depend on DNS server settings in your `resolv.conf`.

Note: you can't just copy these commands and run (or push exact these commands to `/etc/sudoers`), merely because the pid of the process in mount commands is different each time.

You may see the commands that will be run without running them with `--dry-run` option:

```
vagga _create_netns --dry-run
```

To destroy the created network you can run:

```
vagga _destroy_netns
```

This uses `sudo` too

Warning: if you have `172.18.0.0/16` network attached to your machine, the `_create_netns` and `_destroy_netns` may break that network. We will allow to customize the network in future versions of vagga.

5.3 Containers

TBD

5.4 Partitioning

TBD

There is an article on how the network interface was designed and why.

CHAPTER 6

Tips And Tricks

6.1 Faster Builds

There are *Settings* which allow to set common directory for cache for all projects that use vagga. I.e. you might add the following to `$HOME/.config/vagga/settings.yaml`:

```
cache-dir: ~/.cache/vagga/cache
```

Currently you must create directory by hand.

6.2 Multiple Build Attempts

Despite of all caching vagga does it's usually to slow to rebuild big container for trying to install single package. You might try something like this:

```
$ vagga _run --writeable container_name pip install pyzmq
```

Note the flag `--writeable` or shorter `-W` doesn't write into container itself, but creates a (hard-linked) copy, which is destructed on exit. So to run multiple commands you might use bash:

```
host-shell$ vagga _run -W container bash
root@localhost:/work# apt-get update
root@localhost:/work# apt-get install -y something
```

Note: We delete package indexes of ubuntu after container is built. It's done to keep image smaller. So you always need `apt-get update` step.

Another technique is to use *Dependent Containers*.

6.3 Debug Logging

You can enable additional debug logging by setting the environment variable `RUST_LOG=debug`. For example:

```
$ RUST_LOG=debug vagga _build container
```

6.4 I'm Getting "permission denied" Errors

If when starting vagga you're getting:

```
ERROR:container::monitor: Can't run container wrapper: Error executing: permission denied
```

Then you're probably don't have apropriate kernel option enabled. You may try:

```
$ sysctl -w kernel.unprivileged_userns_clone=1
```

If that works, you should add it to your system startup. If it doesn't unfortunately it may mean you need to recompile the kernel. It's not that complex nowadays, but still disturbing.

Anyway if you didn't find specific instructions for your system on *Installation* page please report an issue with information of your distribution (at least `uname` and `/etc/os-release`) so I can add instructions.

6.5 How to Debug Slow Build?

There is a log with timings for each step, in container's metadata folder. The easiest way to view it:

```
$ cat .vagga/<container_name>/../timings.log
  0.000   0.000   Start 1425502860.147834
  0.000   0.000   Prepare
  0.375   0.374   Step: Alpine("v3.1")
  1.199   0.824   Step: Install(["alpine-base", "py-sphinx", "make"])
  1.358   0.159   Finish
```

Note: Note the `/../` part. It works because `.vagga/<container_name>` is a symlink. Real path is something like `.vagga/.roots/<container_name>.<hash>/timings.log`

First column displays time in seconds since container started building. Second column is a time of this specific step.

You should also run build at least twice to see the impact of package caching. To rebuild container run:

```
$ vagga _build --force <container_name>
```

6.6 How to Find Out Versions of Installed Packages?

You can use typical `dpkg -l` or similar command. But since we usually deinstall `npm` and `pip` after setting up container for space efficiency we put package list in contianer metadata. In particular there are following lists:

- `alpine-packages.txt` – list of packages for Alpine linux
- `debian-packages.txt` – list of packages for Ubuntu/Debian linux
- `pip2-freeze.txt`/`pip3-freeze.txt` – list of python packages, in a format directly usable for `requirements.txt`
- `npm-list.txt` – a tree of npm packages

The files contain list of all packages incuding ones installed implicitly or as a dependency. All packages have version. Unfortunately format of files differ.

The files are at parent directory of the container's filesystem, so can be looked like this:

```
$ cat .vagga/<container_name>/../pip3-freeze.txt
```

Or specific version can be looked:

```
$ cat .vagga/.roots/<container_name>.<hash>/pip3-freeze.txt
```

The latter form is useful to compare with old version of the same container.

CHAPTER 7

Conventions

This document describes the conventions for writing vagga files. You are free to use only ones that makes sense for your project.

7.1 Motivation

Establishing conventions for vagga file have the following benefits:

- Easy to get into your project for new developers
- Avoid common mistakes when creating vagga file

7.2 Command Naming

`run`
: To run a project you should just start:

```
$ vagga run
```

This should obey following rules:

1. Run all the dependencies: i.e. database, memcache, queues, whatever
2. Run in host network namespace, so user can access database from host without any issues
3. You shouldn't need to configure anything before running the app, all defaults should be out of the box

`test`
: To run all automated tests you should start:

```
$ vagga test
```

The rules for the command:

1. Run all the test suites that may be run locally
2. Should not include tests that require external resources
3. If that's possible, should include ability to run individual tests and *–help*
4. Should run all needed dependencies (databases, caches,..), presumably on different ports from ones used for `vagga run`

It's expected that exact parameters depend on the underlying project. I.e. for python project this would be a thin wrapper around *nosetests*

test-whatever

Runs individual test suite. Named `whatever`. This may be used for two purposes:

1. Test suite requires some external dependencies, say a huge database with real-life products for an e-commerce site.
2. There are multiple test suites with different runners, for example you have a *nosetests* runner and *cunit* runner that require different command-line to choose individual test to run

Otherwise it's similar to *run* and may contain part of that test suite

doc

Builds documentation:

```
$ vagga doc
[.. snip ..]
------------------------------------------------------------
Documentation is built under docs/_build/html/index.html
```

The important points about the command:

1. Build HTML documentation
2. Use *epilog* to show where the documentation is after build
3. Use *work-dir* if your documentation build runs in a subdirectory

If you don't have HTML documentation at all, just ignore rule #1 and put whatever documentation format that makes sense for your project.

Additional documentation builders (different formats) may be provided by other commands. But main `vagga doc` command should be enough to validate all the docs written before the commit.

The documentation may be built by the same container that application runs or different one, or even just inherit from application's one (useful when some of the documentation is extracted from the code).

CHAPTER 8

Examples

8.1 By Category

Bellow is a list of sample configs from vagga/examples. To run any of them just jump to the folder and run `vagga`.

8.1.1 Databases

PostgreSQL

```
#
# Sample Vagga configuration for running PostgreSQL server
#

containers:
  ubuntu:
    setup:
    - !Ubuntu trusty
    - !Install
      - postgresql-9.3
    - !EnsureDir /data
    environ:
      PG_PORT: 5433   # Port of host to use
      PG_DB: vagga-test
      PG_USER: vagga
      PG_PASSWORD: vagga
      PGDATA: /data
      PG_BIN: /usr/lib/postgresql/9.3/bin
    volumes:
      /data: !Tmpfs
        size: 100M
        mode: 0o700

commands:
  psql: !Command
    description: Run postgres shell
    container: ubuntu
    # This long script initialized new empty postgres database each time
    # container is run
    run: |
      chown postgres:postgres $PGDATA;
      su postgres -c "$PG_BIN/pg_ctl initdb";
```

```
      su postgres -c "$PG_BIN/pg_ctl -w -o '-F --port=$PG_PORT -k /tmp' start";
      su postgres -c "$PG_BIN/psql -h 127.0.0.1 -p $PG_PORT -c \"CREATE USER $PG_USER WITH PASSWORD
      su postgres -c "$PG_BIN/createdb -h 127.0.0.1 -p $PG_PORT $PG_DB -O $PG_USER";
      psql postgres://$PG_USER:$PG_PASSWORD@127.0.0.1/$PG_DB
```

Redis

Simplest container with redis looks like this:

```
containers:
  redis:
    setup:
    - !Alpine v3.2
    - !Install [redis]

commands:

  server: !Command
    container: redis
    run: "redis-server --daemonize no"

  cli: !Command
    container: redis
    run: [redis-cli]
```

Here is more comprehensive example of redis installed on ubuntu and has two instances started in parallel:

```
#
# Sample Vagga config for installing and running Redis Server v3.0
# in Ubuntu 15.04 box.
#

containers:
  ubuntu:
    setup:
    - !UbuntuRelease {version: 15.04}
    - !UbuntuUniverse
    - !Sh apt-key adv --keyserver keyserver.ubuntu.com --recv-keys C7917B12
    - !UbuntuRepo
      url: http://ppa.launchpad.net/chris-lea/redis-server/ubuntu
      suite: vivid
      components: [main]
    - !Install
      - redis-server
    environ:
      REDIS_PORT1: 6380
      REDIS_PORT2: 6381

commands:

  redis-server: !Command
    description: Run instance of Redis server
    container: ubuntu
    run: |
      redis-server --daemonize no --port $REDIS_PORT1 --logfile "" --loglevel debug

  cluster: !Supervise
```

```
    description: Run 2 instances of redis in cluster mode and provide redis-cli
    mode: stop-on-failure
    kill-unresponsive-after: 1
    children:
      redis1: !Command
        container: ubuntu
        run: |
          redis-server --daemonize no \
            --port $REDIS_PORT1 \
            --cluster-enabled yes \
            --cluster-config-file /tmp/cluster.conf \
            --logfile /work/redis-node-1.log \
            --dir /tmp \
            --appendonly no

      redis2: !Command
        container: ubuntu
        run: |
          redis-server --daemonize no \
            --port $REDIS_PORT2 \
            --cluster-enabled yes \
            --cluster-config-file /tmp/cluster.conf \
            --logfile /work/redis-node-2.log \
            --dir /tmp \
            --appendonly no

      meet-nodes: !Command
        container: ubuntu
        run: |
          until [ "$(redis-cli -p $REDIS_PORT1 ping 2>/dev/null)" ]; do sleep 1; done;
          until [ "$(redis-cli -p $REDIS_PORT2 ping 2>/dev/null)" ]; do sleep 1; done;
          redis-cli -p $REDIS_PORT1 CLUSTER MEET 127.0.0.1 $REDIS_PORT2;
          redis-cli -p $REDIS_PORT1;
```

8.1.2 Miscellaneous

Travis Gem

The following snippet installs travis gem (into container). For example to provide github token to Travis CI (so that it can push to github), you can run the following:

```
$ vagga travis encrypt --repo xxx/yyy --org GH_TOKEN=zzz
```

The vagga configuration for the command:

```
containers:
  travis:
    setup:
    - !Ubuntu trusty
    - !UbuntuUniverse
    - !Install [build-essential, ruby, automake, autoconf, libtool, make, ruby-dev]
    - !Sh "gem install travis --no-rdoc --no-ri"

commands:

  travis: !Command
    container: travis
```

```
    run: [travis]
    environ: { HOME: /tmp }
```

8.1.3 Documentation

Sphinx Documentation

The simplest way to generate sphinx documentation is to use `py-sphinx` package from Alpine linux:

```
containers:

  docs:
    setup:
    - !Alpine v3.2
    - !Install [alpine-base, py-sphinx, make]
    # If you require additional packages to build docs uncomment this
    # - !Py2Requirements docs/requirements.txt

commands:

  doc: !Command
    description: Build documentation
    container: docs
    run: [make, html]
    work-dir: docs
    epilog: |
      ------------------------------------------------------------
      Documentation is built under docs/_build/html/index.html
```

To start documentation from scratch (if you had no sphinx docs before), run the following once (and answer the questions):

```
vagga _run docs sphinx-quickstart target_doc_directory
```

8.2 Real World Examples

This section contains real-world examples of possibly complex vagga files. They are represented as external symlinks (github) with a description. Send a pull request to add your example here.

First Time User Hint

All the examples run in containers and install dependencies in `.vagga` subfolder of project dir. So all that possibly scary dependencies are installed automatically and **never touch your host system**. That makes it easy to experiment with vagga.

- Vagga itself – fairly complex config, includes:
  - *Building* Rust with musl libc support
  - Docs using sphinx and additional dependencies
  - Running vagga in vagga for tests
- Presentation config for simple impress.js presentation generated from restructured text (`.rst`) files. Includes:

- Installing hovercraft by Pip (Python 3), which generates the HTML files
- The simple `serve` command to serve the presentation on HTTP
- The `pdf` command which generates PDF files using wkhtmltopdf and some complex bash magic

CHAPTER 9

Indices and tables

- genindex
- modindex
- search