## Contents

1 Links ................................................. 3

2 Documentation Contents ....................... 5
   2.1 About Vagga .................................. 5
   2.2 Installation .................................. 15
   2.3 Configuration ................................ 20
   2.4 Running ...................................... 76
   2.5 Network Testing .............................. 90
   2.6 Tips And Tricks .............................. 94
   2.7 Conventions ................................ 96
   2.8 Examples and Tutorials ..................... 98

3 Indices and tables ............................... 155

4 Credits ............................................ 157
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).

**Hint:** While vagga is perfect for development environments and to build containers, it should not be the tool of choice to run your software in production environments. For example, it does not offer features to automatically restart your services when those fail. For these purposes, you could build the containers with vagga and then transfer them into your production environment and start them with tools like: `docker`, `rocket`, `lxc`, `lxd`, `runc`, `systemd-nspawn`, `lithos` or even `chroot`.
Links

- Managing Dependencies with Vagga shows basic concepts of using vagga and what problems it solves
- The Higher Level Package Manager – discussion of vagga goals and future
- Evaluating Mesos discuss how to run network tolerance tests
- Container-only Linux Distribution
- Containerized PHP Development Environments with Vagga
- The story behind vagga (in russian)
2.1 About Vagga

Contents:

2.1.1 Entry Point

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).

Example

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

```yaml
containers:
  flask:
    setup:
      - !Ubuntu xenial
      - !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:

```bash
$ vagga py3
[ .. snipped container build log .. ]
Python 3.5.1+ (default, Mar 30 2016, 22:46:26)
[GCC 5.3.1 20160330] 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:

```yaml
containers:
  flask:
    setup:
    - !Ubuntu xenial
    - !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:

```bash
$ vagga py3
[ .. snipped container build log .. ]
Python 3.5.1+ (default, Mar 30 2016, 22:46:26)
[GCC 5.3.1 20160330] 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:

```yaml
containers:
  flask:
    setup:
```

(continues on next page)
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:

```bash
$ 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)

**More Reading**

- Managing Dependencies with Vagga shows basic concepts of using vagga and what problems it solves.
- The Higher Level Package Manager – discussion of vagga goals and future
- Evaluating Mesos discuss how to run network tolerance tests.

### 2.1.2 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

**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:

```bash
$ 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 javascript files and that we can run a server. We now just do:
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 xenial
    - !Install [make, nodejs, node-uglify]
  serve:
    setup:
    - !Ubuntu xenial
    - !UbuntuUniverse
    - !Install [python-django]
```

One for each command:

```
commands:
  build-js: !Command
```

(continues on next page)
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 xenial
      - !UbuntuUniverse
      - !Install [make, nodejs, node-uglify, python-django, python-nose]

commands:
  test:
    container: testing
    run: [nose
tests]

And your user never care how many containers are there. User only runs whatever commands 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)
extexample-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
```

(continues on next page)
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.

**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

(continues on next page)
Using Docker? Let’s see:

```bash
$ 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.

### 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:

```bash
#!/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 occasion.

Vagga’s way:

```yaml
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"
```

Now just run:

```bash
$ 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 (Yea, even with `kill -9`).

### 2.1.3 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 environments as easy as vagga.

**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 passwordless 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).
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/.

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)

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

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 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 really 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.

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 is at least as secure as running any unprivileged program in host system. 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.

**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.

### 2.1.4 Vagga vs Vagrant

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

**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.

**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.

**Linux-only**

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

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

**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.
2.2 Installation

Contents

- Binary Installation
- Runtime Dependencies
- Ubuntu
  - Ubuntu: Old Releases (precise, 12.04)
- Debian 8
- Arch Linux
- Building From Source
- Building From Source Using Docker
- OS X / Windows
- Raspbian Stretch (Debian 9)

2.2.1 Binary Installation

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

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

```
$ wget https://files.zerogw.com/vagga/vagga-0.8.1.tar.xz
$ tar -xJf vagga-0.8.1.tar.xz
$ cd vagga
$ sudo ./install.sh
```

Or you may try more obscure way:

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

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

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

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

2.2.2 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 privilege. 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 Arch Linux: ensure CONFIG_USER_NS=y enabled in kernel. Default kernel doesn’t contain it, you can check it with:

```bash
$ 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:

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

or you may get:

```bash
$ sysctl kernel.unprivileged_userns_clone
sysctl: cannot stat /proc/sys/kernel/unprivileged_userns_clone: No such file or...
```

Either one is a valid outcome.

In case you’ve got kernel.unprivileged_userns_clone = 0, use something along the lines of:

```bash
$ 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-unprivileged-users-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 set up automatically by useradd in new distributions. See man subuid if not. To check:

```bash
$ 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.

### 2.2.3 Ubuntu

To install from vagga’s repository just add the following to sources.list (see actual command below):

```bash
deb [arch=amd64 trusted=yes] https://ubuntu.zerogw.com vagga main
```

The process of installation looks like the following:

```bash
$ echo 'deb [arch=amd64 trusted=yes] https://ubuntu.zerogw.com vagga main' | sudo tee /etc/apt/sources.list.d/vagga.list
deb https://ubuntu.zerogw.com vagga main
$ sudo apt-get update
[.. snip ..]
Get: 10 https://ubuntu.zerogw.com vagga/main amd64 Packages [365 B]
[.. snip ..]
Fetching 9,215 kB in 17s (532 kB/s)
```

(continues on next page)
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 https://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 [arch=amd64 trusted=yes] https://ubuntu.zerogw.com vagga-testing main
```

It’s build right from git “master” branch and we are trying to keep “master” branch stable.

### 2.2.4 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 [arch=amd64 trusted=yes] https://ubuntu.zerogw.com vagga main" | sudo tee -i /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.
2.2.5 Debian 8

Install Vagga like in Ubuntu:

```
$ echo 'deb [arch=amd64 trusted=yes] https://ubuntu.zerogw.com vagga main' | sudo tee -i /etc/apt/sources.list.d/vagga.list
$ sudo apt-get update
$ sudo apt-get install vagga
```

Then fix runtime dependencies:

```
$ echo 'kernel.unprivileged_userns_clone = 1' | sudo tee --append /etc/sysctl.conf
$ sudo sysctl -p
```

Now your vagga is ready to go.

2.2.6 Arch Linux

Since 4.14.5-1 Arch Linux kernel has enabled CONFIG_USER_NS option, you can check it with:

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

The only thing you should to do with new kernel is to turn on sysctl flag:

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

To preserve the flag between reboots just execute:

```
echo kernel.unprivileged_userns_clone=1 | sudo tee -a /etc/sysctl.d/99-sysctl.conf
```

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
```

2.2.7 Building From Source

The recommended way to is to build with vagga. It’s as easy as installing vagga and running vagga make inside the clone of a vagga repository.

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.

You can also build vagga out-of-container by using rustup.rs. Make sure you have the musl target installed:

$ rustup target add x86_64-unknown-linux-musl

Also make sure you have musl-gcc in your path:

$ which musl-gcc
/usr/bin/musl-gcc

Then just build using cargo and the appropriate target:

$ cargo build --target x86_64-unknown-linux-musl

### 2.2.8 Building From Source Using Docker

Clone vagga repository:

$ git clone https://github.com/tailhook/vagga && cd vagga

Describe vagga version:

$ git describe

Result would be something like v0.8.1-19-g372bded

Compile vagga with dockerized rust:

$ docker run --rm --user "$(id -u)":"$(id -g)" -e VAGGA_VERSION=v0.8.1-19-g372bded -v ""$PWD"":/usr/src/vagga -w /usr/src/vagga rust cargo build --release

Compiled binary is moved to /usr/local/bin

### 2.2.9 OS X / Windows

We have two proof of concept wrappers around vagga:

- vagga-docker which leverages docker for mac to run vagga on OS X
- vagga-box a wrapper around VirtualBox (tested on OS X only so far)

If you’d like something more stable, try:

- vagrant-vagga (recommended)
- vagga-barge
- Or just your own vagrant config (but see this FAQ entry)
2.2.10 Raspbian Stretch (Debian 9)

Either compile on Raspberry Pi (be patient as it needs quite a while; take care not to run out of memory):

```bash
$ git clone https://github.com/tailhook/vagga.git
$ cd vagga
$ VAGGA_VERSION=$(git describe) CFLAGS=-I/usr/include/arm-linux-musleabihf cargo
  \→ build --target=arm-unknown-linux-musleabihf
```

Or cross compile (recommended):

```bash
$ git clone https://github.com/tailhook/vagga.git
$ cd vagga
$ vagga make-arm
$ scp target/arm-unknown-linux-musleabihf/debug/vagga <user@pi>:<path to vagga repo>
```

Installation needs to be run from inside cloned vagga repo on Raspberry Pi.

```bash
$ ./fetch_binaries.sh armhf
$ sudo ./install.sh
$ vagga -V
$ sudo apt install uidmap
```

Run container with Alpine should be fine on all Pi models whereas Ubuntu is only confirmed for “Pi 2 model B”

https://wiki.ubuntu.com/ARM/RaspberryPi

To run container with Ubuntu add ubuntu-mirror to your vagga settings file

```bash
$ echo 'ubuntu-mirror: http://ports.ubuntu.com/ubuntu-ports' >> ~/.vagga.yaml
$ # if you get error because of failed 'apt-get update' try different mirror, e.g.
```

In your vagga.yam select proper architecture:

```yaml
setup:
  - !UbuntuRelease { codename: xenial, arch: armhf }
```

2.3 Configuration

Main vagga configuration file is vagga.yaml. It’s usually in the root of the project dir.

Configuration format basics.

2.3.1 Overview

The vagga.yaml has two sections:

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

There is also two top-level options:

mixins
**This functionality is experimental**. Some details can change in future.

This is a list of vagga configs that will be “mixed in” into current config. This basically means that we import all the commands and containers from them literally.

When adding mixins, latter one overrides commands and containers in the former configs. And the ones in `vagga.yaml` override all the mixins.

There are a few use-cases for mixins:

1. Splitting config into several groups of things, while putting together containers and commands (latter contrasts to using includes).
2. Use a generated parts of configs. Because non-existing or invalid mixins are ignored (with a warning) you can generate or update mixins by vagga commands without risk of making defunct vagga config.
3. Use vagga config from a subproject (but be aware that paths resolve to original `vagga.yaml`, not the included one)
4. Override things from git-commited `vagga.yaml` to custom one (note the latter requires not to commit `vagga.yaml` itself, but only mixed in things)

New in version 0.7.1.

Changed in version 0.8.1: There are implicit mixins:

1. `vagga.local.yaml`
2. `.vagga.local.yaml` (hidden)
3. `.vagga/local.yaml` (in vagga dir)

They should be used to add or override local commands which shouldn’t be committed to a central repository. They work similarly as normal mixins and can contain additional mixins themselves.

**minimum-vagga**

(default is no limit) Defines minimum version to run the configuration file. If you put:

```yaml
minimum-vagga: v0.5.0
```

Into `vagga.yaml` other users will see the following error:

```
Please upgrade vagga to at least "v0.5.0"
```

This is definitely optional, but useful if you start using new features, and want to communicate the version number to a team. Versions from testing work as well. To see your current version use:

```
$ vagga --version
```

**Containers**

Example of one container defined:

```yaml
containers:
  sphinx:
    setup:
      - !Ubuntu xenial
      - !Install [python3-sphinx, make]
```

The YAML above defines a container named `sphinx`, which is built with two steps: download and unpack ubuntu xenial base image, and install packages name `python3-sphinx`, `make` inside the container.
Commands

Example of command defined:

```yaml
commands:
  build-docs:
    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.

2.3.2 Container Parameters

**setup**
List of steps that is executed to build container. See *Container Building Guide* and *Build Steps (The Reference)* 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 the container runs. User must have some ranges in `/etc/subuid` to run this container, and the total size of all allowed ranges must be larger or equal to the sum of sizes of all the ranges specified in `uids` parameter. Currently 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 a smaller number of uids available or run container in container.

**gids**
List of ranges of group ids that need to be mapped when the container runs. User must have some ranges in `/etc/subgid` to run this container, and the total size of all allowed ranges must be larger or equal to the sum of sizes of all the ranges specified in `gids` parameter. Currently 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 a 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 *Container Building Guide* 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 `/etc/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 the same considerations must be applied.

---

**auto-clean**

(experimental) Do not leave multiple versions of the container lying around. Removes the old container version after the new one 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.

---

**image-cache-url**

If there is no locally cached image and it is going to be built, first check for the cached image in the specified URL.

Example:

```
image-cache-url: http://example.org/${container_name}.${short_hash}.tar.xz
```

To find out how to upload an image see `push-image-cmd`.

---

Warning: The url must contain at least `{{short_hash}` substitution, or otherwise it will ruin the vagga’s container versioning.

---

Note: Similarly to `Tar` command we allow paths starting with . and `/volumes/` here. It’s of limited usage. And we still consider this experimental. This may be useful for keeping image cache on network file system, presumably on non-public projects.

---

**data-dirs**

List of directories and files that only should remain in the container. All other directories and files will be removed after container is build. Useful to create data containers that can be used as volumes inside normal containers.

Example:

---

2.3. Configuration
data-dirs: [/var/lib/postgres]

default-shell
(default [/bin/sh, -c, $cmdline, "--"]) Alternative shell to run commands with. This shell is only used to run commands which have \textit{run} argument as a string. If \textit{run} is a list then default shell is \textit{not} used.

You can use any command, not only sh-compatible, for example \texttt{python}, but that may be unexpected o user so use your best judgement. Also take closer look on the default invocation, in particular:

1. $cmdline is replaced with the contents of \textit{run}
2. \texttt{--} is a common convention to prevent user \textit{command’s arguments} conflicting with shell’s arguments (this is useful if \textit{accepts-arguments} is enabled)

### 2.3.3 Commands

Every command under \texttt{commands} in \texttt{vagga.yaml} is mapped with a tag that denotes the command type. The are two command types \texttt{!Command} and \texttt{!Supervise} illustrated by the following example:

```yaml
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
      i386: !Command
        container: ubuntu
```

Since 0.7.1 there is a third \textit{experimental} kind which is \texttt{!CapsuleCommand}.

### 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.
aliases

The list of names that might be used to refer to a command. For example:

```yaml
commands:
  test: !Command
    aliases: [check, run-tests]
  container: py3
  run: [pytest]
```

Note if there is a command with the same name it overrides an alias. When multiple commands have single alias the result is deemed `undefined` for now, we may turn it into an error/warning later or run both commands.

Overriding works also across the mixins. So command imported earlier overrides alias defined later. Aliases aren’t merged on encountering a mixin, but rather being overriden like other settings.

Another trick is that bare `vagga` and `vagga _list` (without `--hidden`) show command only if main name does not start with underscore. I.e. this is a way to hide non-underscored command by naming it _something and adding non-underscored alias.

**group**

(no default) Which group to print command under. It’s actual title of the group printed when running bare `vagga`. For example:

```yaml
commands:
  build: !Command # ...
  example-app1: !Command
    group: "Example applications"
    # ...
  example-app2: !Command
    group: "Example applications"
    # ...
```

Might be listed as:

```
$ vagga
Available commands:
  build Build the library

Example applications:
  example-app1 Run simple example application
  example-app2 Run a more comprehensive example applications
```

By default command is listed in the first section named “Available commands”, if all commands have a “group”, then aforementioned title is not printed.

**prerequisites**

The list of commands to run before the command, each time it is started.

Example:

```yaml
commands:
  make:
    container: build
    run: "make prog"
  run:
    container: build
    prerequisites: [make]
    run: "./prog"
```

The sequence of running of command with prerequisites is following:
1. Container is built if needed for each prerequisite
2. Container is built if needed for main command
3. Each prerequisite is run in sequence
4. Command is started

If any step fails, neither next step nor the command is run.

The prerequisites are recursive. If any of the prerequisite has prerequisites itself, they will be called. But each named command will be run only once. We use topology sort to ensure prerequisite commands are started before dependent commands. For cyclic dependencies, we ensure that command specified in the command line is run later, otherwise order of cyclic dependencies is unspecified.

The supervise command's --only and --except influences neither running prerequisites itself nor commands inside the prerequisite if the latter happens to be supervise command. But there is a global flag --no-prerequisites.

The prerequisites is not (yet) supported in the any of children of a !Supervise command, but you can write prerequisites for the whole command group.

expect-inotify-limit
Check the sysctl fs.inotify.max_user_watches and print a warning or set it automatically if auto-apply-sysctl is enabled. More info about max_user_watches

options
This is a docopt definition for the options that this command accepts. Example:

```
commands:
  test: !Supervise
  options: |
    Usage: vagga test [--redis-port=<n>] [options] [<tests>...]
    
    Options:
    -R, --redis-port <n>  Port to run redis on [default: 6379]
    <tests> ...  Name of the tests to run. By default all tests are run
  
  children:
  redis: !Command
    container: redis
    run: |
      redis-server --daemonize no --port "$VAGGAOPT_REDIS_PORT"
  
  first-line: !Command
    container: busybox
    run: |
      py.test --redis-port "$VAGGAOPT_REDIS_PORT" "$VAGGAOPT_TESTS"
```

As you might have noticed, options are passed in environment variables prefixed with VAGGAOPT_ and VAGGACL1_ (see below). Your scripts are free to use them however makes sense for your application.

**Note:**
- You should include [options] at least in one of the usage examples, to have -h, --help working as well as other built-in options (--only, --except in supervise commands)
- This setting overrides accepts-arguments

Every argument is translated into two variables:
- **VAGGAOPT_ARG** – has the raw value of the argument, for boolean flags it contains either `true` or nothing, for repeatable flags it contains a number of occurrences
- **VAGGACLI_ARG** – has a canonical representation of an argument, this includes option name and all needed escaping to represent multiple command line arguments

The **ARG** is usually a long name of the option if exists, or short name otherwise. For positional arguments it’s argument name. It’s always uppercased and has `-` replaced with `_`.

There are few shortcomings of both kinds:

1. **VAGGAOPT_** can’t represent list of arguments that can contain spaces. So it can’t be used for list of file names in the general case.
2. **VAGGACLI_** contains escaped versions of arguments so requires using `eval` to make proper argument list from it

Some shell patterns using **VAGGAOPT_**:

1. To propagate a flag, use either one:

   ```bash
   somecmd ${VAGGAOPT_FLAG:+--flag}
somecmd $VAGGACLI_FLAG
   ```

2. To optionally pass a value to a command, use either one (note the implications of `eval` in the second command):

   ```bash
   somecmd ${VAGGAOPT_VALUE:+--value} $VAGGAOPT_VALUE
eval somecmd $VAGGACLI_VALUE
   ```

To overcome limitations of `eval`, for example if you need to expand `$@hostname` in the command, you can use the following snippet:

```bash
eval printf "'%s\0'" $VAGGACLI_VALUE | xargs -0 somecmd -H${hostname}
```

3. To pass a list of commands each prefixed with a `--test=`, use either one:

   ```bash
   # any shell (but ugly)
eval printf "'%s\0'" $VAGGACLI_TESTS | sed -z 's/^/--test=/' | xargs -0
eval somecmd
   ```

   ```bash
   # bash only
eval "tests=($VAGGACLI_TESTS)"
somecmd "$@tests[@]/#/--test="
   ```

   (Note for some `sed` implementations you need to omit `-z` flag)

   This works if you have argument like `vagga test <tests>...`. However, if your vagga command-line is `vagga test --test=<name>...` use the following instead:

   ```bash
   eval somecmd $VAGGACLI_TEST
   ```

**symlink-name**

*This functionality is experimental*. Some details can change in future.

If parameter is set to some name, then **vagga _update_symlinks** will add a symlink to a **vagga** binary with this name. This should be used for commands that you want to run from your shell without **vagga** prefix. See **_update_symlinks** command for more info.

New in version 0.7.1.
**isolate-network**

Run command within isolated network namespace. Isolated network will have only a loopback device, so processes won’t have access to the internet. For example, it is possible to run several test suites each start service that binds the same port. Also you can run arbitrary command inside isolated network using `--isolate-network` commandline option.

New in version 0.7.0: only for `!Supervise`

New in version 0.8.1.

**Parameters of `!Command`**

**container**

The container to run command in.

**tags**

The list of tags for this command. Tags are used for processes filtering (with `--only` and `--exclude`) when running any `!Supervise` command.

Simple example:

```text
commands:
  run: !Supervise
    children:
      postgres: !Command
        tags: [service]
        run: ...
      redis: !Command
        tags: [service]
        run: ...
      app: !Command
        tags: [app]
        run: ...

$ vagga run --only service  # will start only postgres and redis processes
```

**run**

The command to run. It can be:

- either a string encompassing a shell command line (which is feeded to `/bin/sh -c`)
- or a list containing first the full path to the executable to run and then possibly static 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 a shell command line (if `run` is a string);
- `true` if command is an executable (if `run` is a list).

NB: If command is a shell command line - even if it’s composed of only one call to an executable -., arguments are given to its executing context, not appended to it.

**Note:** This setting is ignored when `options` is set.
environ

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

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.

The volumes defined here override volumes specified in the container definition (each volume name is con-
sidered separately).

Note: You must create a folder for each volume. See Container Building Guide for documentation.

pid1mode

This denotes what is run as pid 1 in container. It may be wait, wait-all-children or exec. The default
wait is okay 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)

group-id

The group id to run command as. Default is 0.

supplementary-gids

The list of group ids of the supplementary groups. By default it’s empty list.

pass-tcp-socket

Binds a TCP to the specified address and passes it to the application as a file descriptor #3.

Example:

```
nginx:
    container: nginx
```

(continues on next page)
run: nginx
pass-tcp-socket: 8080
environ:
  NGINX: "3;" # inform nginx not to listen on its own

You may specify what to listen to with the following formats:

- **8080** – just a port number – listens on 127.0.0.1
- ***:8080** – wildcard pattern for host – listens on every host
- **0.0.0.0:8080** – same as ***:8080**
- **192.0.2.1:8080** – listen on specified IPv4 host
- **[2001:db8::1]:8080** – listen on specified IPv6 host
- **localhost:8080** – resolve a name and listen that host (note: name must resolve to a single address)

This is better then listening by the application itself in the following cases:

1. If you want to test systemd socket activation
2. If you prepare your application to a powerful supervisor like lithos (lithos can run multiple processes on the same port using the feature)
3. To declare (document) that your application listens specified port (otherwise it may be hidden somewhere deeply in config)
4. To listen port in the **host** network namespace when applying network isolation (as an alternate to **public-ports**)

**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 exits (in **stop-on-failure** mode), vagga will send TERM signal to all the 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 signal unconditionally). If you don’t like this behavior set the parameter to some large value.

**Parameters of !CapsuleCommand**

This functionality is experimental. Some details can change in future.

It’s generally not recommended to use, unless you know what are you doing.

**uids**

**gids**

If you need to build container non-standard uid/gid mapping, you might want to configure these to match container’s settings.

Note: all containers built in this CapsuleCommand must match uids/gids with these. Defaults are usually okay.
**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)

**environ**

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

This is also a good way to pass some variables to a script that bootstraps a capsule.

**run**

The command to run. Rules are the same as in normal command, but usually this option either specifies a local script to run (committed in repository) or a single pattern like `vagga _capsule script https://some/location.sh`. See *Capsule Commands* for more info.

### 2.3.4 Container Building Guide

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

```yaml
containers:
    ubuntu:
        setup:
            - !Ubuntu xenial
            - !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.

See *Build Steps (The Reference)* for additional details on specific commands. There is also an genindex

#### Generic Installers

To run arbitrary shell command use `!Sh`:

```yaml
setup:
    - !Ubuntu xenial
    - !Sh "apt-get update && apt-get install -y python"
```

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

```yaml
setup:
    - !Ubuntu xenial
    - !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 xenial
    - !Install [python]

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

  alpine:
    setup:
    - !Alpine v3.5
    - !Install [python]

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

setup:
  - !Ubuntu xenial
  - !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. If `url` starts with dot . it's treated as filename inside project directory. 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: https://static.rust-lang.org/dist/rust-1.10.0-x86_64-unknown-linux-gnu.tar.gz
    sha256: abcd1234...
    subdir: rust-1.10.0-x86_64-unknown-linux-gnu
    script: ./install.sh --prefix=/usr
```

Only the `url` is mandatory here too. Similarly, if `url` starts with dot . it's treated as filename inside project directory. 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" }

Multiple directories may be specified at once.

**Warning:** In this example, “apt-cache” is the name of the directory on your host. Unless changed in the Settings, the directory can be found in `.vagga/.cache/apt-cache`. It is shared both between all the containers and all the different builders (not only same versions of the single container). In case the user enabled shared-cache, the folder will also be shared between containers of different projects.

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

```yaml
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

---

**Ubuntu**

To install base ubuntu system use:

```yaml
setup:
  - !Ubuntu xenial
```

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

```yaml
setup:
  - !UbuntuRelease { codename: wily }
```

To install any ubuntu package use generic !Install command:

```yaml
setup:
  - !Ubuntu xenial
  - !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:

```yaml
setup:
  - !Ubuntu xenial
  - !UbuntuUniverse
  - !Install [checkinstall]
```

The !UbuntuRepo command adds additional repository. For example, to add marathon repository you may write:
setup:
- !Ubuntu xenial
- !UbuntuRepo
  url: http://repos.mesosphere.io/ubuntu
  suite: xenial
  components: [main]
- !Install [mesos, marathon]

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

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

### Alpine

To install base alpine system use:

setup:
- !Alpine v3.5

Potentially any alpine version instead of v3.4 should work.

To install any alpine package use generic !Install command:

setup:
- !Alpine v3.5
- !Install [python]

### Npm Installer

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

setup:
- !Ubuntu xenial
- !NpmInstall [babel]

All node packages are installed as --global which should be expected. If no distribution is specified before the !NpmInstall command, the implicit !Alpine v3.5 (in fact the latest version) will be executed.

setup:
- !NpmInstall [babel]

So above should just work as expected if you don’t need any special needs. E.g. it’s usually perfectly okay 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.

---

2.3. Configuration
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.

Python Installer

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

```bash
setup:
- !Ubuntu xenial
- !Py2Install [sphinx]
```

We always fetch latest pip for installing dependencies. 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-certificates 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:

```bash
setup:
- !Ubuntu xenial
- !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”:

```bash
setup:
- !Ubuntu xenial
- !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.

---

**PHP/Composer Installer**

Composer packages can be installed either explicitly or from `composer.json`. For example:

```
setup:
- !Ubuntu xenial
- !ComposerInstall [laravel/installer]
```

The packages will be installed using Composer's `global` `require` at `/usr/local/lib/composer/vendor`. This is only useful for installing packages that provide binaries used to bootstrap your project (like the Laravel installer, for instance):

```
setup:
- !Ubuntu xenial
- !ComposerInstall [laravel/installer]
- !Sh laravel new src
```

Alternatively, you can use Composer’s `crate-project` command:

```
setup:
- !Ubuntu xenial
- !ComposerInstall # just to have composer available
- !Sh composer create-project --prefer-dist laravel/laravel src
```

**Note:** In the examples above, it is used `src` (/work/src) instead of . (/work) because Composer only accepts creating a new project in an empty directory.

For your project dependencies, you should install packages from your `composer.json`. For example:

```
setup:
- !Ubuntu xenial
- !ComposerDependencies
```

This command will install packages (including dev) from `composer.json` into `/usr/local/lib/composer/vendor` using Composer's `install` command.

**Note:** The `/usr/local/lib/composer` directory will be automatically added to PHP’s `include_path`.

---

**Warning:** Most PHP frameworks expect to find the `vendor` directory at the same path as your project in order to require `autoload.php`, so you may need to fix your application entry point (in a Laravel 5 project, for example, you should edit `bootstrap/autoload.php` and change the line `require __DIR__.'/../vendor/autoload.php';` to require `'vendor/autoload.php';`. You can also specify some options available from Composer command line, for example:
If you want to use hhvm, you can disable the installation of the php runtime:

```
setup:
  - !Ubuntu xenial
  - !ComposerConfig
    install_runtime: false
    runtime_exe: /usr/bin/hhvm
```

**Note:** When setting the `runtime_exe` option, be sure to specify the full path of the binary (e.g. `/usr/bin/hhvm`).

**Note:** Vagga will try to create a symlink from `runtime_exe` into `/usr/bin/php`. If that location already exists, Vagga will **not** overwrite it.

Note that you will have to manually install hhvm and set the `include_path`:

```
setup:
  - !Ubuntu xenial
  - !Repo universe
  - !Install [hhvm]
    - !ComposerConfig
      install_runtime: false
      runtime_exe: /usr/bin/hhvm
    - !Sh echo 'include_path=./usr/local/lib/composer' >> /etc/hhvm/php.ini
  environ:
    HHVM_REPO_CENTRAL_PATH: /run/hhvm.hhbc
```

- `setup include_path` in hhvm config
- `tell hhvm` to store the build cache database in a writeable directory

Alpine v3.5 added support for php7 in their “community” repository while keeping php5 as the default runtime. In order to use php7, you have to specify all the packages required by composer (and any other php packages you may need):

```
setup:
  - !Alpine v3.5
  - !Repo community
    - !Install
      - php7
      - php7-openssl
      - php7-phar
      - php7-json
      - php7-pdo
      - php7-dom
      - php7-zip
    - !ComposerConfig
```

(continues on next page)
install_runtime: false
runtime_exe: /usr/bin/php7

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

Warning: PHP/Composer support was recently added to vagga, some things may change as we gain experience with the tool.

Ruby Installer

Ruby gems can be installed either by providing a list of gems or from a `Gemfile` using `bundler`. For example:

```bash
setup:
- !Ubuntu xenial
- !GemInstall [rake]
```

We will update `gem` to the latest version (unless specified not to) for installing gems. The `ruby-dev` headers are installed for the time of the build too and are removed when installation is finished.

The following `gem` 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)

```bash
setup:
- !Ubuntu xenial
- !Install [zlib1g]
- !BuildDeps [zlib1g-dev]
- !Env
  HOME: /tmp
- !GemInstall [rails]
- !Sh rails new --skip-bundle
```

Bundler is also available for installing gems from `Gemfile`. For example:

```bash
setup:
- !Ubuntu xenial
- !GemBundle
```

You can also specify some options to Bundler, for example:

```bash
setup:
- !Ubuntu xenial
- !GemBundle
gemfile: src/Gemfile # use this Gemfile
without: [development, test] # groups to exclude when installing gems
trust_policy: HighSecurity
```

It is possible to avoid installing ruby if you are providing it yourself:
Warning: Ruby/Gem support was recently added to vagga, some things may change as we gain experience with the tool.

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:

```
containers:
  base:
    setup:
      - !Ubuntu xenial
      - !Py3Install [django]
  test:
    setup:
      - !Container base
      - !Py3Install [nose]
```

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

```
containers:
  temporary:
    setup:
      - !Ubuntu xenial
      - !TarInstall
      url: http://download.zeromq.org/zeromq-4.1.4.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 xenial
    - !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`:

```yml
docker-parser:
  setup:
  - !Alpine v3.5
  - !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 `source` command:

```yml
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:

```yml
# vagga.yaml
containers:
  ubuntu:
    setup:
    - !Ubuntu xenial
    - !SubConfig
      path: packages.yaml
      container: packages
  alpine:
```

(continues on next page)
2.3.5 Build Steps (The Reference)

This is work in progress reference of build steps. See Container Building Guide for help until this document is done. There is also an alphabetic genindex.

All of the following build steps may be used as an item in setup setting.

Container Bootstrap

Command that can be used to bootstrap a container (i.e. may work on top of empty container):

- Alpine
- Ubuntu
- UbuntuRelease
- SubConfig
- Container
- Tar

Ubuntu Commands

Ubuntu

Simple and straightforward way to install Ubuntu release.

Example:

```yaml
containers:
  packages:
    setup:
      - !Ubuntu xenial
```

The value is single string having the codename of release xenial, trusty and precise known to work at the time of writing.

The Ubuntu images are updated on daily basis. But vagga downloads and caches the image. To update the image that was downloaded by vagga you need to clean the cache.

**Note:** This is shortcut install that enables all the default that are enabled in UbuntuRelease. You can switch to UbuntuRelease if you need fine-grained control of things.
UbuntuRelease
This is more exensible but more cumbersome way to setup ubuntu (comparing to Ubuntu). For example to install trusty you need:

- !UbuntuRelease { codename: trusty }

(note this works since vagga 0.6, previous versions required version field which is now deprecated).
You can also setup non-LTS release of different architecture:

- !UbuntuRelease { codename: vivid, arch: i386 }

All options:
codename Name of the ubuntu release. Like xenial or trusty. Either this field or url field must be specified. If both are specified url take precedence.
url Url to specific ubuntu image to download. May be any image, including server and desktop versions, but cloudimg is recommended. This must be filesystem image (i.e usually ending with root.tar.gz) not .iso image.
Example: http://cloud-images.ubuntu.com/xenial/current/xenial-server-cloudimg-amd64-root.tar.gz
arch The architecture to install. Defaults to amd64.
keep-chfn-command (default false) This may be set to true to enable /usr/bin/chfn command in the container. This often doesn’t work on different host systems (see #52 as an example). The command is very rarely useful, so the option here is for completeness only.
eatmydata (default true) Install and enable libeatmydata. This does not literally eat your data, but disables all fsync and fdatasync operations during container build. This works only on distributions where we have tested it: xenial, trusty, precise. On other distributions the option is ignored (but may be implemented in future).
The fsync system calls are used by ubuntu package management tools to secure installing each package, so that on subsequent power failure your system can boot. When building containers it’s both the risk is much smaller and build starts from scratch on any kind of failure anyway, so partially written files and directories do not matter.
I.e. don’t disable this flag unless you really want slow processing, or you have some issues with LD_PRELOAD’ing the library.

Note: On trusty and precise this also enables universe repository by default.

version The version of ubuntu to install. This must be digital YY.MM form, not a code name.

This field will also be removed at some point in future.

AptTrust
This command fetches keys with apt-key and adds them to trusted keychain for package signatures. The following trusts a key for fkrull/deadsnakes repository:

- !AptTrust keys: [5BB92C09DB82666C]

By default this uses hkps://keyserver.ubuntu.com:443, but you can specify alternative:
This is used to get rid of the error similar to the following:

```
WARNING: The following packages cannot be authenticated!
   libpython3.5-minimal python3.5-minimal libpython3.5-stdlib python3.5
E: There are problems and --y was used without --force-yes
```

Options:

- **server** (default `hkps://keyserver.ubuntu.com:443`) Server to fetch keys from. May be a hostname like `keyserver.ubuntu.com` or in `hkp://hostname:port` or `hkps://hostname:port` form.

- **keys** (default `[]`) List of keys to fetch and add to trusted keyring. Keys can include full fingerprint or suffix of the fingerprint. The most common is the 8 hex digits form.

**UbuntuRepo**

Adds arbitrary debian repo to ubuntu configuration. For example to add newer python:

```
- !UbuntuRepo
  url: http://ppa.launchpad.net/fkrull/deadsnakes/ubuntu
  suite: xenial
  components: [main]
- !Install [python3.5]
```

See **UbuntuPPA** for easier way for dealing specifically with PPAs.

Options:

- **url** Url to the repository. Default is the mirror url from the current ubuntu distribution.

- **suite** Suite of the repository. The common practice is that the suite is named just like the codename of the ubuntu release. For example `xenial`. Default is the codename of the current distribution.

- **components** List of the components to fetch packages from. Common practice to have a `main` component. So usually this setting contains just single element `components`: `[main]`. **Required.**

- **trusted** Marks repository as trusted. Usually useful for installing unsigned packages from local repository. Default is `false`.

**UbuntuPPA**

A shortcut to `UbuntuRepo` that adds named PPA. For example, the following:

```
- !Ubuntu xenial
- !AptTrust keys: [5BB92C09DB82666C]
- !UbuntuPPA fkrull/deadsnakes
- !Install [python3.5]
```

Is equivalent to:

```
- !Ubuntu xenial
- !UbuntuRepo
  url: http://ppa.launchpad.net/fkrull/deadsnakes/ubuntu
  suite: xenial
  components: [main]
- !Install [python3.5]
```
UbuntuUniverse
The singleton step. Just enables an “universe” repository:

- !Ubuntu xenial
- !UbuntuUniverse
- !Install [checkinstall]

Alpine Commands

Alpine

setup:
- !Alpine v3.5

AlpineRepo

Adds arbitrary alpine repository. For example to add testing repository:

- !AlpineRepo
  url: http://nl.alpinelinux.org/alpine/
  branch: edge
  repo: testing
  tag: testing
- !Install [app@testing]

Options:

url  Url to the repository. Default is the mirror url from the current alpine distribution.

branch  Branch of the repository. For example v3.4, edge. Default is the version of the current alpine distribution.

repo  Repository to fetch packages from. For example main, community, testing. Required.

tag  Tag for this repository. Alpine package manager will now by default only use the untagged repositories. Adding a tag to specific package will prefer the repository with that tag. To add a tag just put @tag after the package name. For example:

- !AlpineRepo
  branch: edge
  repo: testing
  tag: testing
- !Install [graphicsmagick@testing]

Distribution Commands

These commands work for any linux distributions as long as distribution is detected by vagga. Latter basically means you used Alpine, Ubuntu, UbuntuRelease in container config (or in parent config if you use SubConfig or Container)

Repo

Adds official repository to the supported linux distribution. For example:

setup:
- !Ubuntu xenial
- !Repo xenial/universe
- !Repo xenial-security/universe

(continues on next page)
- !Repo xenial-updates/universe

setup:
- !Ubuntu xenial
- !Repo universe # The same as "xenial/universe"

setup:
- !Alpine v3.5
- !Repo edge/testing

setup:
- !Alpine v3.5
- !Repo community # The same as "v3.5/community"

Install

setup:
- !Ubuntu xenial
- !Install [gcc, gdb] # On Ubuntu, equivalent to 'apt-get install gcc gdb -y'
- !Install [build-essential] # 'apt-get install build-essential -y'
# Note that 'apt-get install' is run 2 times in this example

BuildDeps

setup:
- !Ubuntu xenial
- !BuildDeps [wget]
- !Sh echo "We can use wget here, but no curl"
- !BuildDeps [curl]
- !Sh echo "We can use wget and curl here"
# Container built. Now, everything in BuildDeps (wget and curl) is removed from the container.

Generic Commands

Sh

Runs arbitrary shell command, for example:

- !Ubuntu xenial
- !Sh "apt-get install -y package"

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

setup:
- !Alpine v3.5
- !Sh |
  if [ ! -z "/bin/sh" ] && [ ! -z "/bin/lbu" ]; then
    echo "Alpine"
  fi
- !Sh echo "Finished building the Alpine container"

Warning: To run !Sh you need /bin/sh in the container. See Cmd for more generic command runner.
Note: 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.

**Cmd**

Runs arbitrary command in the container. The argument provided must be a YAML list. For example:

```
setup:
- !Ubuntu xenial
- !Cmd ["apt-get", "install", "-y", "python"]
```

You may use YAMLy features to get complex things. To run complex python code you may use:

```
setup:
- !Cmd
  - python
  - -c
  - |
    import socket
    print("Builder host", socket.gethostname())
```

Or to get behavior similar to `Sh` command, but with different shell:

```
setup:
- !Cmd
  - /bin/bash
  - -exc
  - |
    echo this is a bash script
```

**RunAs**

Runs arbitrary shell command as specified user (and/or group), for example:

```
setup:
- !Ubuntu xenial
- !RunAs
  user-id: 1
  script: |
    python -c "import os; print(os.getuid())"
```

Options:

- **script** (required) Shell command or script to run
- **user-id** (default 0) User ID to run command as. If the `external-user-id` is omitted this has same effect like using `sudo -u`.
- **external-user-id** (optional) See explanation of `external-user-id` for `!Command` as it does the same.
- **group-id** (default 0) Group ID to run command as.
- **supplementary-gids** (optional) The list of group ids of the supplementary groups. By default it’s an empty list.
- **work-dir** (default `/work`) Directory to run script in.
- **isolate-network** (default `false`) See explanation of `isolate-network` for `!Supervise` command type.

**Download**

Downloads file and puts it somewhere in the file system.

Example:
**Download**

```yaml
url: https://jdbc.postgresql.org/download/postgresql-9.4-1201.jdbc41.jar
path: /opt/spark/lib/postgresql-9.4-1201.jdbc41.jar
```

**Note:** This step does not require any download tool to be installed in the container. So may be used to put static binaries into container without a need to install the system.

Options:

- **url** (required) URL to download file from
- **path** (required) Path where to put file. Should include the file name (vagga doesn’t try to guess it for now). Path may be in /tmp to be used only during container build process.
- **mode** (default ‘0o644’) Mode (permissions) of the file. May be used to make executable bit enabled for downloaded script
- **sha256** (optional) Sha256 hashsum of the file. If real hashsum is different this step will fail.

New in version 0.8.0.

**Warning:** The download is cached similarly to other commands. Currently there is no way to control the caching. But it’s common practice to publish every new version of archive with different URL (i.e. include version number in the url itself)

**Tar**

Unpacks Tar archive into container’s filesystem.

Example:

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

Downloaded file is stored in the cache and reused indefinitely. It’s expected that the new version of archive will have a new url. But occasionally you may need to clean the cache to get the file fetched again.

- **url** Required. The url or a path of the archive to fetch. If the url startswith dot . it’s treated as a file name relative to the project directory. Otherwise it’s a url of the file to download.

**Note:** Since vagga 0.6 we allow to unpack local paths starting with /volumes/ as file on one of the volumes configured in settings (external-volumes). This is exprimental, and requires every user to update their setthings before building a container. Still may be useful for building company-internal things.

- **path** (default /). Target path where archive should be unpacked to. By default it’s a root of the filesystem.
- **subdir** (default .) Subdirectory inside the archive to extract . extracts the root of the archive.
- **sha256** (optional) Sha256 hashsum of the archive. If real hashsum is different this step will fail.

This command may be used to populate the container from scratch

**TarInstall**

Similar to **Tar** but unpacks archive into a temporary directory and runs installation script.
Example:

```yaml
setup:
- !TarInstall
  url: https://static.rust-lang.org/dist/rust-1.10.0-x86_64-unknown-linux-gnu.tar.gz
  script: ./install.sh --prefix=/usr
```

**url** Required. The url or a path of the archive to fetch. If the url startswith dot `.`, it's treated as a file name relative to the project directory. Otherwise it's a url of the file to download.

**subdir** (optional) Subdirectory which command is run in. May be . to run command inside the root of the archive.

The common case is having a single directory in the archive, and that directory is used as a working directory for script by default.

**sha256** (optional) Sha256 hashsum of the archive. If real hashsum is different this step will fail.

**script** The command to use for installation of the archive. Default is effectively a `.configure --prefix=/usr && make && make install`.

The script is run with `/bin/sh -exc`, to have better error handling and display. Also this means that dash/bash-compatible shell should be installed in the previous steps under path `/bin/sh`.

**Unzip**

Unpacks zip archive into container’s filesystem.

All options are the same as for **Tar** step.

Example:

```yaml
- !Unzip
  url: https://services.gradle.org/distributions/gradle-3.1-bin.zip
  path: /opt/gradle
  subdir: gradle-3.1
```

**Git**

Check out a git repository into a container. This command doesn’t require git to be installed in the container.

Example:

```yaml
setup:
- !Alpine v3.5
- !Install [python3]
- !Git
  url: git://github.com/tailhook/injections
  path: /usr/lib/python3.5/site-packages/injections
```

Options:

**url** (required) The git URL to use for cloning the repository

**revision** (optional) Revision to checkout from repository. Note if you don’t specify a revision, the latest one will be checked out on the first build and then cached indefinitely

**branch** (optional) A branch to check out. Usually only useful if revision is not specified

**path** (required) A path where to store the repository.

2.3. Configuration
GitInstall
Check out a git repository to a temporary directory and run script. This command doesn’t require git to be installed in the container.

Example:

```bash
setup:
- !Alpine v3.5
- !Install [python, py-setuptools]
- !GitInstall
  url: git://github.com/tailhook/injections
  script: python setup.py install
```

Options:
- **url** (required) The git URL to use for cloning the repository
- **revision** (optional) Revision to checkout from repository. Note if you don’t specify a revision, the latest one will be checked out on the first build and then cached indefinitely
- **branch** (optional) A branch to checkout. Usually only useful if revision is not specified
- **subdir** (default root of the repository) A subdirectory of the repository to run script in
- **script** (required) A script to run inside the repository. It’s expected that script does compile/install the software into the container. The script is run using `/bin/sh -exc`

GitDescribe
Finds the most recent tag for the given git repository, takes it into account when calculating container’s hash and writes it into a file if the last is specified.

Examples:

```bash
# Only put tag information into the container's hash
- !GitDescribe

# Also write the most recent tag into the file
- !GitDescribe /version.txt

# Specify another path to the git repository
- !GitDescribe
  repo: /work/subrepo
  output-file: /version.txt

# Find last tag that starts with "v", e.g. "v0.1.2"
- !GitDescribe
  match: v*
  output-file: /version.txt
```

Options:
- **repo** (default `/work`) Path to the git repository.
- **output-file** (optional) Writes the most recent tag into the file.
- **pattern** (optional) Takes into account only tags that match the glob pattern. This parameter corresponds to the `--match` option of the `git describe` command.
  
  New in version 0.8.1.

New in version 0.8.0.
Files and Directories

Text
Writes a number of text files into the container file system. Useful for writing short configuration files (use external files and file copy or symlinks for writing larger configs)

Example:

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

Copy
Copy file or directory into the container. Useful either to put build artifacts from temporary location into permanent one, or to copy files from the project directory into the container.

Example:

```yaml
setup:
- !Copy
  source: /work/config/nginx.conf
  path: /etc/nginx/nginx.conf
```

For directories you might also specify regular expression to ignore:

```yaml
setup:
- !Copy
  source: /work/mypkg
  path: /usr/lib/python3.4/site-packages/mypkg
  ignore-regex: "(-|\.py\[co\])$"
```

Symlinks are copied as-is. Path translation is done neither for relative nor for absolute symlinks. Hint: relative symlinks pointing inside the copied directory work well, as well as absolute symlinks that point to system locations.

**Note:** The command fails if any file name has non-utf-8 decodable names. This is intentional. If you really need bad filenames use traditional `cp` or `rsync` commands.

Options:
- **source** (required) Absolute to directory or file to copy. If path starts with `/work` files are checksummed to get the version of the container.
- **path** (required) Destination path
- **ignore-regex** (default `(^|/).\(git|hg|svn|vagga\)(\$|)\-\$|\.bak|\.orig|\^\#.*#$)`) Regular expression of paths to ignore. Default regexp ignores common revision control folders and editor backup files.
- **include-regex** (default `None`) Regular expression of paths to include. When path matches both ignore and include expressions it will be ignored. Also note that if include-regex matches only the folder, no contents will be included. For example `patches/.*\.sql$` will copy all `patches` directories with all `.sql` files inside them.
**owner-uid, owner-gid** (preserved by default) Override uid and gid of files and directories when copying. It’s expected that most useful case is owner-uid: 0 and owner-gid: 0 but we try to preserve the owner by default. Note that unmapped users (the ones that don’t belong to user’s subuid/subgid range), will be set to nobody (65535).

**preserve-permissions** (default false) Whether to preserve permissions of the copied files. If false umask option is taken into account.

**preserve-times** (default false) If true preserves modification times of the copied files. By default modification times set to the first second from the unix epoch. This behaviour is useful for reproducible builds. Before 0.8.0 modification times were set to the current time.

New in version 0.8.0.

**umask** (default 0o002) When preserve-permissions is disabled (set to false) umask determines how permissions are calculated. For directories and executable files it is 0o777 & !umask, for other files - 0o666 & !umask. Do not forget to use 0o prefix for octal notation.

**rules** Leverages glob patterns instead of regular expressions to match paths. This option conflicts with ignore-regex and include-regex options.

The rules are similar to those which is used in .gitignore file but meaning of include/exclude rules is opposite. Also there are several differences:

- Include patterns must be absolute so they have to start with a leading slash. This is done for performance reasons to exclude unknown directories from traversing. If you really want to match relative paths you can prepend pattern with a slash followed by two consecutive asterisks. Thus /*/*/*.py pattern will match test.py, dir/main.py, dir/subdir/test.py paths and so on.
- ! prefix negates the pattern. Negative patterns can be relative. Unlike .gitignore patterns it is possible to include a file when its parent directory was excluded. For instance rules ["!dir", "/dir/.*.py"] will match all python files inside the dir directory.
- If there is no matched files inside a directory the directory itself won’t be copied.
- If the pattern ends with a slash, it will match only with a directory and paths underneath it. Also /dir/ pattern will copy dir directory even if it is empty.
- Empty rules will match all underneath paths. But if you add only exclude rules no path will be matched. So you should explicitly add at least one include rule:

  [] will match all paths

  ["!dir/"] will match nothing

  ["!dir/", "/"] will match all paths except dir directories

By default there are some ignore rules that correspond ignore-regex expression: ["!.git/", ".hg/", ".svn/", ".vagga/", ".*bak", ".*orig", ".*~", ".#*#", ".#*" ] (Note: they apply to Copy but not Build)

New in version 0.8.0.

**no-default-rules** Disables default ignore rules which are given above. The option only works in pair with rules option.

New in version 0.8.0.
**Warning:** If the source directory starts with `/work` all the files are read and checksummed on each run of the application in the container. So copying large directories for this case may influence container startup time even if rebuild is not needed.

This command is useful for making deployment containers (i.e. to put application code to the container file system). For this case checksumming issue above doesn’t apply. It’s also useful to enable `auto-clean` for such containers.

**Remove**

Remove file or a directory from the container and **keep it clean on the end of container build**. Useful for removing cache directories.

This is also inherited by subcontainers. So if you know that some installer leaves temporary (or other unneeded files) after a build you may add this entry instead of using shell `rm` command. The `/tmp` directory is cleaned by default. But you may also add man pages which are not used in container.

Example:

```bash
setup:
- !Remove /var/cache/something
```

For directories consider use `EmptyDir` if you need to keep cleaned directory in the container.

**EnsureDir**

Setup:

```bash
#...
- !EnsureDir /var/cache/downloads
- !Sh if [ -d "/var/cache/downloads" ]; then echo "Directory created"; fi;
- !EnsureDir /creates/parent/directories
```

**EmptyDir**

Cleans up a directory. It’s similar to the `Remove` but keeps directory created.

**CacheDirs**

Adds build cache directories. Example:

```bash
- !CacheDirs
  /tmp/pip-cache/http: pip-cache-http
  /tmp/npm-cache: npm-cache
```

This maps `/tmp/pip-cache/http` into the cache directory of the vagga, by default it’s `~/.vagga/.cache/pip-cache-http`. This allows to reuse same download cache by multiple rebuilds of the container. And if shared cache is used also reuses the cache between multiple projects.

Be picky on the cache names, if file conflicts there may lead to unexpected build results.

**Note:** Vagga uses a lot of cache dirs for built-in commands. For example the ones described above are used whenever you use Py* and Npm* commands respectively. You don’t need to do anything special to use cache.

**Meta Data**

**Env**

Set environment variables for the build.

Example:
setup:
- !Env HOME: /root

Note: The variables are used only for following build steps, and are inherited on the Container directive. But they are not used when running the container.

Depends
Rebuild the container when a file changes. For example:

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

The example is not the best one, you could use Py3Requirements for the same task.

Only the hash of the contents of a file is used in versioning the container not an owner or permissions. Consider adding the auto-clean option if it’s temporary container that depends on some generated file (sometimes useful for tests).

Options:
- path (required) Relative path to directory or file inside /work directory. Matched files and directories will be checksummed to get the version of the container.
- ignore-regex See ignore-regex
- include-regex See include-regex
- rules See rules
- no-default-rules See no-default-rules

Sub-Containers

Container
Build a container based on another container:

```
container:
  base:
    setup:
      - !Ubuntu xenial
      - !Py3Install [django]
test:
  setup:
    - !Container base
    - !Py3Install [nose]
```

There two known use cases of functionality:

1. Build test/deploy containers on top of base container (example above)
2. Cache container build partially if you have to rebuild last commands of the container frequently

In theory, the container should behave identically as if the commands would be copy-pasted to the setup for dependent container, but sometimes things doesn’t work. Known things:

1. The packages in a BuildDeps are removed
2. *Remove* and *EmptyDir* will empty the directory

3. *Build* with *temporary-mount* is not mounted

If you have any other bugs with container nesting report in the bugtracker.

**Note:** *Container* step doesn’t influence *environ* and *volumes* as all other options of the container in any way. It only somewhat replicate *setup* sequence. We require whole environment be declared manually (you you can use YAMLy aliases)

### SubConfig

This feature allows to generate (parts of) *vagga.yaml* for the container. For example, here is how we use a *docker2vagga* script to transform *Dockerfile* into *vagga config*:

```yaml
docker-parser:
  setup:
    - !Alpine v3.5
    - !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:

```yaml
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 xenial
    - !SubConfig
      path: packages.yaml
      container: packages
  alpine:
    setup:
    - !Alpine v3.5
    - !SubConfig
      path: packages.yaml
      container: packages

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

Build
This command is used to build some parts of the container in another one. For example:

containers:
  webpack:
    setup:
    - !NpmInstall [webpack]
    - !NpmDependencies
  jsstatic:
    setup:
    - !Container webpack
    - !Copy
      source: /work/frontend
      path: /tmp/js
    - !Sh |
      cd /tmp/js
      webpack --output-path /var/javascripts
    auto-clean: true
  nginx:
    setup:
    - !Alpine v3.5
    - !Install [nginx]
    - !Build
      container: jsstatic
      source: /var/javascripts
      path: /srv/www

Note the following things:

- We use separate container for npm dependencies so we don’t have to rebuild it on each change of the sources
- We copy javascript sources into our temporary container. The important part of copying operation is that all the sources are hashed and versioned when copying. So container will be rebuild on source changes. Since we don’t need sources in the container we just put them in temporary folder.
- The temporary container is cleaned automatically (there is low chance that it will ever be reused)
Technically it works similar to !Container except it doesn’t apply configuration from the source container and allows to fetch only parts of the resulting container.

Since vagga v0.8.1 you can also use more fine-grained rules:

```yaml
nginx:
  setup:
  - !Alpine v3.5
  - !Install [nginx]
  - !Build
    container: jsstatic
    source: /var/assets
    path: /srv/www
    rules:
    - /js
    - /css
    - /img
```

See `Copy` command for better explanation of how rules work.

Another motivating example is building a package:

```yaml
containers:
  pkg:
    setup:
    - !Ubuntu xenial
    - !Install [build-essential]
    - !EnsureDir /packages
    - !Sh |
      checkinstall --pkgname=myapp --pakdir=/packages make
    auto-clean: true

nginx:
  setup:
  - !Ubuntu xenial
  - !Build
    container: pkg
    source: /packages
    temporary-mount: /tmp/packages
  - !Sh dpkg -i /tmp/packages/mypkg_0.1.deb
```

Normal versioning of the containers apply. This leads to the following consequences:

- Putting multiple `Build` steps with the same `container` will build container only once (this way you may extract multiple folders from the single container).
- Despite the name `Build` dependencies are not rebuilt.
- The `Build` command itself depends only on the container but on on the individual files. You need to ensure that the source container is versioned well (sometimes you need `Copy` or `Depends` for the task)

Options:

- `container` (required) Name of the container to build and to extract data from
- `source` (default `/`) Source directory (absolute path inside the source container) to copy files from
- `path` Target directory (absolute path inside the resulting container) to copy (either `path` or `temporary-mount` required)
- `rules` List of rules for copying. Similarly to `Copy` command, if this list is specified we copy only files matching rules from `source` to `path` instead of full directory.
New in version v0.8.1.

**temporary-mount**  A directory to mount source into. This is useful if you don’t want to copy files, but rather want to use files from there. The directory is created automatically if not exists, but not parent directories. It’s probably good idea to use a subdirectory of the temporary dir, like /tmp/package. The mount is read-only and persists until the end of the container build and is not propagated through Container step.

**content-hash** (default false) If true this container will depend on source container by the hash of the files copied.

We keep it false by default for two reasons:

1. true doesn’t work for temporary-mount
2. false means dependent container is always built, even if current container is up to date and doesn’t need to be rebuilt.

We still recommend to have it false for the reasons above as this is how everything else in vagga works. But here is the use case for true:

1. We build a data-only container with content-addressing (i.e. every file is accesses by a path derived from checksum of that file)
2. The list of checksums is imported into another container (i.e. it produces links to the files in first container)
3. The container in (1) is not binary reproducible (i.e. every build could possibly produce different checksums)
4. Container are then deployed independently

Note: the (3) in the list above is the main culprit of the feature. If you can force binary reproducibility you can ignore this discussion.

---

### Node.JS Commands

**NpmInstall**

Example:

```
setup:
- !NpmInstall [babel-loader@6.0, webpack]
```

Install a list of node.js packages. If no linux distributions were used yet !NpmInstall installs the latest Alpine distribution. Node is installed automatically and analog of the node-dev package is also added as a build dependency.

**Note:** Packages installed this way (as well as those installed by !NpmDependencies are located under /usr/lib/node_modules. In order for node.js to find them, one should set the environment variable NODE_PATH, making the example become

Example:

```
setup:
- !NpmInstall [babel-loader@6.0, webpack]
environ:
  NODE_PATH: /usr/lib/node_modules
```
**NpmDependencies**
Works similarly to *NpmInstall* but installs packages from *package.json*. For example:

```
- !NpmDependencies
```

This installs dependencies and *devDependencies* from *package.json* into a container (with `--global` flag).

You may also customize *package.json* and install other kinds of dependencies:

```
- !NpmDependencies
  file: frontend/package.json
  peer: true
  optional: true
  dev: false
```

**Note:** Since npm supports a whole lot of different versioning schemes and package sources, some features may not work or may not version properly. You may send a pull request for some unsupported scheme. But we are going to support only the popular ones. Generally, it’s safe to assume that we support a npmjs.org packages and git repositories with full url.

**Note:** We don’t use `npm install` to execute this command but rather use a command-line to specify every package there. It works better because `npm install --global` tries to install this specific package to the system, which is usually not what you want.

Options:

- **file** (default *package.json*) A file to get dependencies from
- **package** (default *true*) Whether to install package dependencies (i.e. the ones specified in *dependencies* key)
- **dev** (default *true*) Whether to install *devDependencies* (we assume that vagga is mostly used for development environments so dev dependencies should be on by default)
- **peer** (default *false*) Whether to install *peerDependencies*
- **bundled** (default *true*) Whether to install *bundledDependencies* (and *bundleDependencies* too)
- **optional** (default *false*) Whether to install *optionalDependencies*. By default npm tries to install them, but don’t fail if it can’t install. Vagga tries its best to guarantee that environment is the same, so dependencies should either install everywhere or not at all. Additionally because we don’t use “npm install package.json” as described earlier we can’t reproduce npm’s behavior exactly. But optional dependencies of dependencies will probably try to install.

**Warning:** This is a new command. We can change default flags used, if that will be more intuitive for most users.

**NpmConfig**
The directive configures various settings of npm commands above. For example, you may want to turn off automatic nodejs installation so you can use custom version of it:

```
- !NpmConfig
  install_node: false
```

(continues on next page)
### npm_exe

The npm command to use for installation of packages.

**install-node** (default true)

Whether to install nodejs and npm automatically. Setting the option to false is useful for setting up custom version of the node.js.

**install-yarn** (default true)

Whether to install yarn package automatically (even if install-node is false). Yarn only installed if !Yarn* commands are used.

New in version v0.8.0.

**yarn-exe** (default is /usr/bin/yarn)

The yarn command to use for installation of packages if !Yarn* commands are used.

The reason we have full path here is because vagga’s own search mechanisms for executables doesn’t resolve absolute symlinks well. (I.e. bare executable name will work here as long as it’s not an absolute symlink). We may fix this in the future releases.

New in version v0.8.0.

**yarn-version** (default is latest)

The yarn version to install for the !Yarn* commands. Only useful if install-yarn is enabled.

Example:

```
- !NpmConfig
  yarn-version: 1.0.1
```

New in version v0.8.0.

### YarnDependencies

Works similarly to NpmDependencies but installs packages using yarn. For example:

```
- !YarnDependencies
```

This installs dependencies and devDependencies from package.json into a container’s /usr/lib/node_modules. It also links all the binaries from node_modules/.bin directory installed by yarn into /usr/bin directory for easier access.

The settings in NpmConfig do affect this command.

Options:

**production** (default false)

If set to true does not install devDependencies (corresponds to --production flag for yarn)

**optional** (default false)

If set to true install optional dependencies

**dir** (default .)

Base directory where package.json is. Note: unlike in :setup:'NpmDependencies' we don’t specify the path to the file, but to the directory because we track both package.json and yarn.lock files in there.
New in version v0.8.0.

**Python Commands**

**PipConfig**

The directive configures various settings of pythonic commands below. The mostly used option is `dependencies`:

```
- !PipConfig
  dependencies: true
- !Py3Install [flask]
```

Most options directly correspond to the pip command line options so refer to [pip help](https://pip.pypa.io/en/stable/reference/) for more info.

**Note:** Every time `PipConfig` is specified, options are replaced rather than augmented. In other words, if you start a block of pythonic commands with `PipConfig`, all subsequent commands will be executed with the same options, no matter which `PipConfig` settings were before.

All options:

- **dependencies** (default `false`) allow to install dependencies. If the option is `false` (by default) pip is run with `pip --no-deps`
- **index-urls** (default `[]`) List of indexes to search for packages. This corresponds to `--index-url` (for the first element) and `--extra-index-url` (for all subsequent elements) options on the pip command-line.

  When the list is empty (default) the `pypi.python.org` is used.
- **find-links** (default `[]`) List of URLs to HTML files that need to be parsed for links that indicate the packages to be downloaded.
- **trusted-hosts** (default `[]`) List of hosts that are trusted to download packages from.
- **cache-wheels** (default `true`) Cache wheels between different rebuilds of the container. The downloads are always cached. Only binary wheels are toggled with the option. It’s useful to turn this off if you build many containers with different dependencies.

  Starting with vagga v0.4.1 cache is namespaced by linux distribution and version. It was single shared cache in vagga <= v0.4.0
- **install-python** (default `true`) Install python automatically. This will install either python2 or python3 with a default version of your selected linux distribution. You may set this parameter to `false` and install python yourself. This flag doesn’t disable automatic installation of pip itself and version control packages. Note that by default python-dev style packages are as build dependencies installed too.
- **allow_pre_releases** (default `false`) Allow install pre-release and development versions. By default, pip only finds stable versions.
- **python-exe** (default is either `python2` or `python3` depending on which command is called, e.g. `Py2Install` or `Py3Install`) This allows to change executable of python. It may be either just name of the specific python interpreter (`python3.5`) or full path. Note, when this is set, the command will be called both for `Py2*` commands and `Py3*` commands.

**Py2Install**

Installs python package for Python 2.7 using pip. Example:
We always fetch latest pip for installing dependencies. 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-certificates manually before)

All other forms may work but not supported. Specifying command-line arguments instead of package names is not supported.

See Py2Requirements for the form that is both more convenient and supports non-vagga installations better.

---

**Note:** If you configure python-exe in PipConfig there is no difference between Py2Install and Py3Install.

---

**Py2Requirements**

This command is similar to Py2Install but gets package names from the file. Example:

```
setup:
- !Ubuntu xenial
- !Py2Requirements "requirements.txt"
```

See Py2Install for more details on package installation and PipConfig for more configuration.

**Py3Install**

Same as Py2Install but installs for Python 3.x by default.

```
setup:
- !Alpine v3.5
- !Py3Install [sphinx]
```

See Py2Install for more details on package installation and PipConfig for more configuration.

**Py3Requirements**

This command is similar to Py3Install but gets package names from the file. Example:

```
setup:
- !Alpine v3.5
- !Py3Requirements "requirements.txt"
```

See Py2Install for more details on package installation and PipConfig for more configuration.
Note: PHP/Composer support was recently added to vagga, some things may change as we gain experience with the tool.

**ComposerInstall**

Example:

```plaintext
setup:
- !Alpine v3.5
- !ComposerInstall ["phpunit/phpunit:~5.2.0"]
```

Install a list of php packages using `composer global require --prefer-dist --update-no-dev`. Packages are installed in `/usr/local/lib/composer/vendor`.

Binaries are automatically installed to `/usr/local/bin` by Composer so they are available in your PATH.

Composer itself is located at `/usr/local/bin/composer` and available in your PATH as well. After container is built, the Composer executable is no longer available.

**ComposerDependencies**

Install packages from `composer.json` using `composer install`. For example:

```plaintext
- !ComposerDependencies
```

Similarly to **ComposerInstall**, packages are installed at `/usr/local/lib/composer/vendor`, including those listed at `require-dev`, as Composer default behavior.

Options correspond to the ones available to the `composer install` command line so refer to `composer cli docs` for detailed info.

Options:

- **working_dir** (default None) Use the given directory as working directory
- **dev** (default true) Whether to install `require-dev` (this is Composer default behavior).
- **prefer** (default None) Preferred way to download packages. Can be either `source` or `dist`. If no specified, will use Composer default behavior (use `dist` for stable).
- **ignore_platform_reqs** (default false) Ignore php, hhvm, lib-* and ext-* requirements.
- **no_autoloader** (default false) Skips autoloader generation.
- **no_scripts** (default false) Skips execution of scripts defined in `composer.json`.
- **no_plugins** (default false) Disables plugins.
- **optimize_autoloader** (default false) Convert PSR-0/4 autoloading to classmap to get a faster autoloader.
- **classmap_authoritative** (default false) Autoload classes from the classmap only. Implicitly enables `optimize_autoloader`.

**ComposerConfig**

The directive configures various settings of composer commands above. For example, you may want to use hhvm instead of php:

```plaintext
- !ComposerConfig
  install_runtime: false
  runtime_exe: /usr/bin/hhvm
- !ComposerInstall [phpunit/phpunit]
```
Note: Every time ComposerConfig is specified, options are replaced rather than augmented. In other words, if you start a block of composer commands with ComposerConfig, all subsequent commands will be executed with the same options, no matter which ComposerConfig settings were before.

All options:

**runtime** _**_exe_ (default /usr/bin/php) The command to use for running Composer. When setting this option, be sure to specify the full path for the binary. A symlink to the provided value will be created at /usr/bin/php if it not exists, otherwise, /usr/bin/php will remain the same.

**install** _**_runtime_ (default true) Whether to install the default runtime (php) automatically. Setting the option to false is useful when using hhvm, for example.

**install** _**_dev_ (default false) Whether to install development packages (php-dev). Defaults to false since it is rare for php projects to build modules and it may require manual configuration.

**include** _**_path_ (default .:/usr/local/lib/composer) Set include_path. This option overrides the default include_path instead of appending to it.

**keep** _**_composer_ (default false) If set to true, the composer binary will not be removed after build.

**vendor** _**_dir_ (default /usr/local/lib/composer/vendor) The directory where composer dependencies will be installed.

Note: Setting install_runtime to false still installs Composer.

Ruby Commands

Note: Ruby support is recently added to the vagga some things may change as we gain experience with the tool.

**GemInstall**

Example:

```
setup:
- ![Ubuntu xenial
- ![GemInstall [rake]
```

Install a list of ruby gems using gem install --bindir /usr/local/bin --no-document. The --bindir option instructs gem to install binaries in /usr/local/bin so they are available in your PATH.

**GemBundle**

Install gems from Gemfile using bundle install --system --binstubs /usr/local/bin. For example:

```
- ![GemBundle
```

Options correspond to the ones available to the bundle install command line, so refer to bundler documentation for detailed info.

Options:

**gemfile** (default Gemfile) Use the specified gemfile instead of Gemfile.
without (default []) Exclude gems that are part of the specified named group.

trust_policy (default None) Sets level of security when dealing with signed gems. Accepts LowSecurity, MediumSecurity and HighSecurity as values.

GemConfig

The directive configures various settings of ruby commands above:

- ![GemConfig]
  - install_ruby: true
  - gem_exe: gem
  - update_gem: true
- ![GemInstall] [rake]

Note: Every time GemConfig is specified, options are replaced rather than augmented. In other words, if you start a block of ruby commands with GemConfig, all subsequent commands will be executed with the same options, no matter which GemConfig settings were before.

All options:

install_ruby (default true) Whether to install ruby.

gem_exe (default /usr/bin/gem) The rubygems executable.

update_gem (default true) Whether to update rubygems itself.

Note: If you set install_ruby to false you will also have to provide rubygems if needed.

Note: If you set gem_exe, vagga will no try to update rubygems.

2.3.6 Volumes

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

```yaml
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 a tmpfs filesystem.
Options:

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  The mode (permission bits) of the root directory for a new filesystem.

subdirs  A mapping for subdirectories to create inside tmpfs, for example:

```yaml
volumes:
  /var: !Tmpfs
    mode: 0766
  subdirs:
    lib: # default mode is 0766
    lib/tmp: { mode: 0777 }
    lib/postgres: { mode: 0700 }
```

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

files  A mapping of a file to it’s contents to write into tmpfs filesystem. This is similar to Text build step. Directories for files are not created automatically. Use subdirs to create one.

Example:

```yaml
volumes:
  /run: !Tmpfs
    size: 100Mi
  files:
    docker.sock: ""
    /run/docker.sock: !BindRO /volumes/docker.sock
```

Both subdirs and files are often used as mountpoints for some **BindRW** and **BindRO** directories.

**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 container 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:

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

There are currently two prefixes for **BindRW**:

- /work – which uses directory inside the project directory
- /volumes – which uses one of the volumes defined in settings (external-volumes)

The behavior of vagga when using any other prefix is undefined.

**BindRO**

Read-only bind mount of a folder inside a container to another folder. See **BindRW** for more info.

**CacheDir**

Mounts a directory cached by vagga. Useful if you need access to the cache during container run.
For example:

```
setup:
  # ...
- !CacheDirs
  /root/.m2/repository: maven-cache
  # ...
  volumes:
    /root/.m2/repository: !CacheDir maven-cache
```

**Empty**

Mounts an empty read-only directory. Technically mounts a new *Tmpfs* system with minimal size and makes it read-only. Useful if you want to hide some built-in directory or subdirectory of */work* from the container. For example:

```
volumes:
  /tmp: !Empty
```

Note, that hiding */work* itself is not supported. You may hide a subdirectory though:

```
volumes:
  /work/src: !Empty
```

**Snapshot**

Create a *Tmpfs* volume, copy contents of the original folder to the volume. And then mount the filesystem in place of the original directory.

This allows to pre-seed the volume at the container build time, but make it writeable and throwable.

Example:

```
volumes:
  /var/lib/mysql: !Snapshot
```

**Note:** Every start of the container will get it’s own copy. Even every process in !Supervise mode will get own copy. It’s advised to keep container having a snapshot volume only for single purpose (i.e. do not use same container both for postgresql and python), because otherwise excessive memory will be used.

Parameters:

- **size** *(default 100Mi)* Size of the allocated *Tmpfs* volume. Including the size of the original contents. This is the limit of how much data you can write on the volume.
- **container** Copy contents of the folder of another container. Path to the source folder is the same as the mountpoint.
- **owner-uid, owner-gid** *(default is to preserve)* The user id of the owner of the directory. If not specified the ownership will be copied from the original

Additional properties, like the source directory will be added to the later versions of vagga

**Container**

Mount a root file system of other container as a volume.

Example:

```
containers:
  app:
```

(continues on next page)
This may be useful to deploy the container without installing anything to the host file system. E.g. you can rsync the container’s file system to remote host. Or tar it (but better use _pack_image or _push_image for that). Or do other fancy things.

Unless you know what are you doing both containers should share same uids and gids.

**Note:** Nothing is mounted on top of container’s file system. I.e. /dev, /proc and /sys directories are empty. So you probably can’t chroot into the filesystem in any sensible way. But having that folders empty is actually what is useful for use cases like deploying.

**Persistent**

Makes a writable directory just for this container. It’s similar to BindRW but creates a volume inside `.vagga/volumes`

Example:

```plaintext
cmds:
  postgres: !Command
  volumes:
    /var/lib/postgres: !Persistent { name: "postgres" }
run: ...
```

Or the shorter form:

```plaintext
volumes:
  /var/lib/postgres: !Persistent "postgres"
```

There are a few reasons to use Persistent over BindRW:

1. User don’t need to create the directories
2. When running vagga in VM it’s a common practice to use more efficient (or more featureful, like supporting hardlinks) filesystem for .vagga
3. It may be a little bit clearer than throwing all that writable stuff into workdir (for example your .vagga is already in .gitignore)

Options:

- **name** *(required)* Name of the volume. Multiple containers using same name will mount same volume (same instance of volume). Multiple volumes in single container may reference same volume too. We currently don’t support mounting subvolumes but we may do in future.
- **owner-uid, owner-gid** User id and group id to set for the new volume. The are only set when volume is first created.
- **init-command** The name of the command that is used for initializing volume. Technically command is inserted into the prerequisites of every other command that uses this volume. The command must have this volume
either in container or in command definition. When command is first run, it has an empty directory at the mount point of the volume. If command fails we immediately stop running dependent commands, which effectively means no other command can run with the volume mounted.

It’s usually good idea to name the command starting with underscore, so it doesn’t show in the list of commands to use for daily work.

If volume is already initialized and command is run in a normal way (using `vagga xx` or as a prerequisite), it’s run as any other command. But if the volume is not initialized it will be run with the temporary directory at a volume mount point which will be committed to a volume on success. Basically this allows to debug the command easily.

If the same `init-command` is repeated in multiple volumes it is run only once (so you must initialize all the volumes that depend on it). Multiple volumes that have different `init-command` values can only be used in the command definition (not in container), because otherwise it’s impossible to establish an initialization order.

Currently `!Supervise` commands can’t be used to initialize a volume, but we may lift this limitation in the future.

To remove volumes that were created but had been removed since than run:

```
vagga _clean --unused-volumes
```

To remove all volumes (they will be created on the next run):

```
vagga _clean --volumes
```

### 2.3.7 Upgrading

#### Upgrading 0.8.0 -> 0.8.1

Patterns in rules of `!Copy` and `!Depends` previously allowed backslash to escape first `!` character (meaning negative rule). Now it can also be used to escape `*`, `?` and `\` character itself.

Also note that ubuntu precise is no longer supported as well as unsupported non-LTS releases, use on your own risk.

#### Upgrading 0.7.x -> 0.8.0

- `!Copy` now resets timestamps when copying, if you need timestamps add `preserve-times`. By default it sets timestamp to first second after epoch. This means timestamp is non-zero, but stable whenever you build container, meaning it improves reproducibility.

- The `/etc/resolv.conf` and `/etc/hosts` files are now cleaned after container build and mounted over by system files. This may slightly tweak behavior when running concurrent containers. Also, if you sync containers to remote hosts they will not contain `resolv.conf` from build machine (which is a good thing), but will be empty if not populated afterwards.

For most users upgrade should be seamless.

#### Upgrading 0.6.x -> 0.7.0

This release only introduces minor incompatibilities and also changes hashes of the containers (so all containers will be rebuild after `vagga upgrades`).

---

2.3. Configuration

69
Vagga Documentation, Release 0.8.1

• Py2/Py3Requirements now properly hashes files containing `-r` (basically includes). This means if you had previously `!Depends` commands for that files, you may now remove them. But it also means that included files should exist when running vagga (i.e. before containers are built).

• `vagga _run` now searches in the following precedence if no `PATH` was set in container `/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`. Previously the precedence was reversed. This may influence you if you have commands with the same names both in `/usr` and `/usr/local`.

• `Copy` and `Depends` do not respect file permissions. Most of the time this means that on machines with different `umask` you still have same container hash. But it also means that if you change permissions on the file container does not get rebuilt (executable bit is still versioned).

• `!Snapshot` respects the owner and permissions of the source directory rather than using defaults from `tmpfs` volume. We consider this a bugfix, but it may break some things if you relied on old behavior.

• `Environment variable precedence` changed to be more intuitive.

• `resolv.conf` and `hosts` files are replaced again after `Tar`, `Ubuntu`, `Container`, `SubConfig`. It’s a bugfix in most cases (i.e. some stalled files may be unpacked/copied in old vagga). But it may clobber your files if you expected old behavior.

• `eatmydata` is enabled for built-in commands only, if you relied on fast fsyncs earlier, your builds may be slow. You may use `!Env { LD_PRELOAD: "/usr/lib/x86_64-linux-gnu/libeatmydata.so" }` to restore old behavior (for xenial, for other distros path may be different).

• Previously we have ignored the error when we couldn’t remount root file system as read-only (e.g. on `tmpfs` or when otherwise some options like `nosuid` were enabled), this is no longer the case (we learned how to make those volumes readonly). In some scenarios it may mean that previously writable folders are now read-only.

• If you relied on a symlink to `/tmp/vagga/hosts`, we have removed it because it was rarely useful and sometimes imposed issues (for example when `/tmp` is readonly). We are working on a more long term solution. In the meantime you must either rely on hosts from the host system (by default) or create a file yourself (luckily IP addresses are static so it’s easy, although may be boring).

Upgrading 0.5.0 -> 0.6.0

This release doesn’t introduce any severe incompatibilities. The bump of version is motivated mostly by the change of container hashes because of refactoring internals.

Minor incompatibilities are:

• Vagga now uses images from partner-images.ubuntu.com rather than cdimage.ubuntu.com

• Vagga now uses single level of uid mappings and doesn’t use the actual mapping as part of container hash. This allows to use `mount` in container more easily and also means we have reproducible containers hashes across machines.

• `!Copy` command now uses paths inside the container as the `source`, previously was inside the capsule (because of a mistake), however using source outside of the `/work` has not been documented.

• Checksum checking in `!Tar` and `!TarInstall` now works (previously you could use an archive with wrong `sha256` parameter).

• Vagga now uses `tar-rs` library for unpacking archives instead of busybox, this may mean some features are new, and some archives could fail (please report if you find one).

• Vagga now runs `id -u -n` for finding out username, previously was using long names which aren’t supported by some distributions (alpine == busybox).

• Commands with name starting with underscore are not listed in `vagga` and `vagga _list` by default (like built-in ones).

70 Chapter 2. Documentation Contents
• Ubuntu commands now use libeatmydata by default, which makes installing packages about 3x faster
• We remove /var/spool/rsyslog in ubuntu, this is only folder that makes issues when rsyncing image because of permissions (it’s not useful in container anyway)
• Updated quire requires you need to write !*Unpack instead of !Unpack
• Remove change-dir option from SubConfig that never worked and was never documented

Upgrading 0.4.1 -> 0.5.0

This release doesn’t introduce any severe incompatibilities. Except in the networking support:
• Change gateway network from 172.18.0.0/16 to 172.23.0.0/16, hopefully this will have less collisions

The following are minor changes during the container build:
• The stdin redirected from /dev/null and stdout is redirected to stderr during the build. If you really need asking a user (which is an antipattern) you may open a /dev/tty.
• The .vagga/.mnt is now unmounted during build (fixes bugs with bad tools)
• !Depends doesn’t resolve symlinks but depends on the link itself
• !Remove removes files when encountered (previously removed only when container already built), also the command works with files (not only dirs)

The following are bugfixes in container runtime:
• The TERM and *_proxy env vars are now propagated for supervise commands in the same way as with normal commands (previously was absent)
• Pseudo-terminals in vagga containers now work
• Improved SIGINT handling, now Ctrl+C in interactive processes such as python (without arguments) works as expected
• The signal messages (“Received SIGINT…”) are now printed into stderr rather than stdout (for !Supervise type of commands)
• Killing vagga supervise with TERM mistakenly reported SIGINT on exit, fixed

And the following changes the hash of containers (this should not cause a headache, just will trigger a container rebuild):
• Add support for arch parameter in !UbuntuRelease this changes hash sum of all containers built using !UbuntuRelease

See Release Notes and Github for all changes.

Upgrading 0.4.0 -> 0.4.1

This is minor release so it doesn’t introduce any severe incompatibilities. The pip cache in this release is namespaced over distro and version. So old cache will be inactive now. And should be removed manually by cleaning .vagga/.cache/pip-cache directory. You may do that at any time

See Release Notes and Github for all changes.
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.

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).

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 containers 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 dir name 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 differentiation 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 or 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.

2.3.8 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 of 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.
Supervision Modes

There are two basic modes of operation:

- **stop-on-failure** – stops all processes as soon as any single one is dead (default)
- **wait-all-successful** – waits until all successful processes finish

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:

```plaintext
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 exit code of first process exited. And an 128+signal code when any other singal was sent to supervisor (and propagated to other processes).

Wait All Successful

In **wait-all-successful** mode `vagga` works same as in **stop-on-failure** mode, except processes that exit with exit code 0 (which is known as sucessful error code) do not trigger failure condition, so other processes continue to work. If any process exits on signal or with non-zero exit code “failure” mode is switched on and `vagga` exits the same as in **stop-on-failure** mode.

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.

In this mode `vagga` returns exit code zero if all processes exited successfully and exit code of the first failing process (or 128+signal if it was dead by signal) otherwise.
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.

2.3.9 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 the parent namespace it gets assigned other PID).

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

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

At a glance, first issue looks like the most annoying. But in practice the most inconvenient one is the last one. For development purposes it effectively means you can’t stop process by sending \texttt{SIGTERM} or \texttt{SIGINT}, if process have not installed a signal handler.

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

Outdated

The following text is outdated. Vagga doesn’t support any pid modes since version 0.2.0. This may be fixed in future. We consider this as mostly useless feature for development purposes. If you have a good use case please let us know.

In fact there are three modes of operation of PID 1 supported by vagga (set by \texttt{pid1mode}).

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

Note that in \texttt{wait} and \texttt{exec} modes, when you kill vagga itself with a signal, it will propagate the signal to the command itself. In \texttt{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.

2.3.10 Capsule Commands

This functionality is experimental. Some details can change in future.
New in version 0.7.1.

It’s generally not recommended to use CapsuleCommand, unless you know what are you doing.

This kind of command doesn’t require container to be built. It operates in intermediate container that we call capsule. Capsule is a container that provides same level of isolation as normal container but has neither config nor version, on the other hand it provides tools to build create and start other containers.

This feature is both: more powerful, as it provides a way to build/run different containers based on dynamic parameters and even change vagga.yaml. On the other hand it starts with bare shell, and it’s your job to bootstrap needed utilities and do all process supervision.

All the tools officially supported by vagga in capsule are prefixed with vagga_capsule, namely:

- `vagga_capsule build <container_name>` – builds container, similar to `vagga_build <container_name>`
- `vagga_capsule run <container> <cmd>` – runs command in a container. Container will be (re)built if required.
- `vagga_capsule script <url>` – fetches a script from the url, caches it and runs from cache
- `vagga_capsule download <url>` – fetches a file from the url and caches printing file path to stdout

There are few limitations of the capsule:

1. All containers must have same uid/gid maps (which is often the case)
2. `vagga_clean` doesn’t work in capsule yet
3. Volume init commands do not work
4. Supervise commands can’t be run in capsule (actually any commands configured in yaml can’t be run from the inside capsule, but most of them can be emulated with `vagga_capsule run`)

## 2.4 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
```

### 2.4.1 Command Line

When running vagga, it finds the vagga.yaml file in current working directory or any of its parents and uses that as a project root directory (alternative files are supported too, see below).

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

Full list of files that mark directory as vagga’s project:

1. `vagga.yaml` primary and preferred one
2. `.vagga/vagga.yaml` as an alternative to `vagga.yaml` (useful if you don’t want to commit it to a git)
3. `vagga.local.yaml` or `.vagga.local.yaml` or `.vagga/local.yaml` which contain additional *mixins* also mark project directory even if no `vagga.yaml` is present (since vagga 0.8.1)

**Multiple Commands**

Since vagga 0.6 there is a way to run multiple commands at once:

```
$ vagga -m cmd1 cmd2
```

This is similar to running:

```
$ vagga cmd1 && vagga cmd2
```

But there is one key difference: **containers needed to run all the commands are built beforehand.** This has two consequences:

1. When containers need to be rebuilt, they are rebuilt first, then you see the output of both commands in sequence (no container build log in-between)
2. If container for command 2 depends on side-effects of running command 1 (i.e. container contains a binary built by command 1), you will get wrong results. In that case you should rely on shell to do the work (for example in the repository of vagga itself `vagga -m make test` is *not* the right way, the right is `vagga make && vagga test`)

Obviously you can’t pass any arguments to either of commands when running `vagga -m`, this is also the biggest reason of why you can’t run built-in commands (those starting with underscore) using the option. But you can use global options, and they influence all the commands, for example:

```
$ vagga --environ DISPLAY:0 -m clean_profile run_firefox
```

**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.

More useful in the form:

```
$ vagga _build --force container_name
```
To rebuild a container that has previously been built.

**vagga_clean** Removes images and temporary files created by vagga.

The following command removes containers that are not used by current vagga config (considering the state of all files that `vagga.yaml` depends on):

```
$ vagga_clean --unused
```

Another for removes containers which were not uses for some time:

```
$ vagga_clean --unused --at-least 10days
```

This is faster as it only checks timestamps of the containers. Each time any command in a container is run, we update timestamp. This is generally more useful than bare `--unused`, because it allows to keep multiple versions of same container, which means you can switch between branches rapidly.

There an old and deprecated option for removing unused containers:

```
$ vagga_clean --old
```

This is different because it only looks at symlinks in `.vagga/*`. So may be wrong (if you changed `vagga.yaml` and did not run the command(s)). It’s faster because it doesn’t calculate the hashsums. But the difference in speed usually not larger than a few seconds (on large configs). The existence of the two commands should probably be treated as a historical accident and `--unused` variant preferred.

For other operations and parameters see `vagga_clean --help`

**vagga_list** List of commands (similar to running vagga without command)

**vagga_version_hash CONTAINER** Prints version hash for the container. In case the image has not been built (or config has been updated since) it should return new hash. But sometimes it’s not possible to determine the hash in advance. In this case command returns an error.

Might be used in some automation scripts.

**vagga_init_storage_dir** Deprecated. Use `storage-subdir-from-env-var` instead.

If you have configured a `storage-dir` in settings, say `/vagga-storage`, when you run `vagga_init_storage_dir abc` will create a `/vagga-storage/abc` and `.vagga` with `.vagga/.lnk` pointing to the directory. The command ensures that the storage dir is not used for any other folder (unless `--allow-multiple` is specified).

This is created for buildbots which tend to clean `.vagga` directory on every build (like gitlab-ci) or just very often.

Since vagga 0.6 there is `--allow-multiple` flag, that allows to keep shared subdirectory for multiple source directories. This is useful for CI systems which use different build directories for different builds.

**Warning:** While simultanenous builds of different source directories, with the same subdirectory should work most of the time, this functionality still considered exerimental and may have some edge cases.

**vagga_pack_image IMAGE_NAME** Pack image into the tar archive, optionally compressing and output it into stdout (use shell redirection > file.tar to store it into the file).

It’s very similar to `tar -cC .vagga/IMAGE_NAME/root` except it deals with file owners and permissions correctly. And similar to running `vagga_run IMAGE_NAME tar -c /` except it ignores mounted file systems.
vagga _push_image IMAGE_NAME  Push container image IMAGE_NAME into the image cache.

Actually it boils down to packing an image into `tar` (vagga _pack_image) and running `push-image-script`, see the documentation of the setting to find out how to configure image cache.

vagga _base_dir  Displays (writes to stdout) directory where active vagga.yaml is.

vagga _relative_work_dir  Displays (writes to stdout) current working directory relative to the base directory. Basically, this means that `${vagga _base_dir}/${vagga _relative_work_dir}` is current working directory.

When current working directory contains vagga.yaml this command returns empty string (output still contains a newline), not a single dot, as one may expect.

vagga _update_symlinks  This functionality is experimental. Some details can change in future.

Creates a set of symlinks in your home directory (~/.vagga/cmd) and in current vagga directory (.vagga/.cmd) which point to commands named in vagga. Symlinks are created to the current vagga binary (which is resolved using `readlink /proc/self/exe` not, `argv[0]`).

These directories can be added to PATH either in your shell or in your text editor, IDE, or any other kind of shell. Or you can pass them to scripts which allow customization (make RSYNC=/myproj/.vagga/.cmd/rsync).

Only commands which have symlink-name are linked with the name specified in the parameter. So you make create a hidden (underscored) name for some public command.

There are two directories, so basically two modes of operation:

1. User home directory ~/.vagga/cmd. It meant to use for utilities you’re going to use in multiple projects. When running such a command in some project dir, exact command from this project dir will be invoked. So if you run flake8 (a linter for python), correct version of linter for this project will be run. If you cd to another project, correct version of the tool with specific plugins and python interpreter will be picked there immediately.

2. Project directory proj/.vagga/.cmd. This directory may be used to specify utility directly or to point your IDE to in project settings. It’s not recommended to add this directory to your search PATH.

Note: for (1) it’s expected that single version of vagga is used for all of the projects, which is usually the case.

New in version 0.7.1.

vagga _verify <container-name>  Checks checksums of the container files against an index. This is only useful if index-all-images is set to true.

It’s useful to find out if some container was modified by a mistake (container contents are not protected in the host system).

New in version 0.8.1.

vagga _hardlink [--global]  Scans all containers for identical files (using some heuristics) and hardlink matching files.

See hard-link-identical-files for precautions.

The --global flag allows to scan all projects. This works only if storage-dir is set.

New in version 0.8.1.

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 `-`.

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

### 2.4.2 Environment

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

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

```bash
$ 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:

```bash
$ 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:

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

The order of precedence for environment variables from the highest priority to the lowest:

1. Options `-E/--environ` in the command-line
2. Options `-e/--use-env` in the command-line
3. `VAGGAENV_*` variables
4. Variables set inside one of the settings files: `environ` option
5. Variables set via `propagate-env` setting
6. Variables set in command: `environ` option
7. Variables set in container: `environ` option
8. Variables set in container: `environ-file` option
9. Variables that are auto-propagated: `_proxy`, `TERM`

### 2.4.3 Settings

#### 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 built 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.

storage-subdir-from-env-var
This option is designed specifically for Continuous Integration (CI) systems. When this option is set it identifies an environment variable that is used to specify the name of a subdirectory of the storage dir used for current project. It's only useful if storage-dir is set.

For example, for Gitlab you may want to set:

```text
storage-subdir-from-env-var: CI_PROJECT_NAME
```

Or or alternatively:

```text
storage-subdir-from-env-var: CI_PROJECT_NAMESPACE
```

Note: only dash, underscore, and alphanumeric chars are allowed in the name, all other characters will be replaced by dash (so technically clashes of names are possible). If this environment variable is empty, then vagga will fail.

Note 2: already set symlink in .vagga/.lnk overrides this setting. This also means that _init_storage_dir overrides the setting. Which means you may remove the link though, if your CI system does not do that by default.

New in version 0.7.2.

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
The mapping of project paths to settings for this specific project.

Example:

```text
site-settings:
/home/myuser/myproject:
cache-dir: /home/myuser/.cache/myproject
```

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.

propagate-environ
A list of variables and patterns that are propagated into the container by default. Example:

```text
propagate-environ:
- "GIT_BRANCH"
- "JENKINS_*"
- "CI_*"
```

This is intended to use on CI system where parameters of build job is safe to propagate.

While technically you can specify "*" it's very dangerous and error-prone option to enable.

external-volumes
A mapping of volume names to the directories inside the host file system.
Note: The directories must exist even if unused in any vagga.yaml.

For example, here is how you might export home:

```yaml
external-volumes:
  home: /home/user
```

Then in vagga.yaml you use it as follows (prepend with /volumes):

```yaml
volumes:
  /root: !BindRW /volumes/home
```

See Volumes for more info about defining mount points.

Warning:

1. Usage of volume is usually a subject for filesystem permissions. I.e. your user becomes root inside the container, and many system users are not mapped (not present) in container at all. This means that mounting /var/lib/mysql or something like that is useless, unless you chown the directory

2. Any vagga project may use the volume if it’s defined in global config. You may specify the volume in site-settings if you care about security (and you should).

push-image-script

A script to use for uploading a container image when you run vagga _push_image_.

To push image using webdav:

```bash
push-image-script: "curl -T $image_path \n    http://example.org/${container_name}.${short_hash}.tar.xz"
```

To push image using scp utility (SFTP protocol):

```bash
push-image-script: "scp $image_path/ \n    user@example.org:/target/path/${container_name}.${short_hash}.tar.xz"
```

The FTP(s) (for example, using _lftp_ utility) or S3 (using _s3cmd_) are also valid choices.

Note: This is that rare case where command is run by vagga in your host filesystem. This allows you to use your credentials in home directory, and ssh-agent’s socket. But also this means that utility to upload images must be installed in host system.

Variables:

- **container_name** The name of the container as declared in vagga.yaml
- **short_hash** The short hash of container setup. This is the same hash that is used to detect whether container configuration changed and is needed to be rebuilt. And the same hash used in directory name .vagga/roots.

**auto-apply-sysctl**

Set sysctls required by command. We do our best to only apply “safe” sysctls by vagga automatically. Still it may exhaust resources of your system, so use this option on your own risk.

We apply settings with _sudo -k_ which means it will prompt for password each time setting is tuned (probably only after system reboot).

2.4. Running
Settings currently exists:

<table>
<thead>
<tr>
<th>Key in vagga.yaml</th>
<th>Sysctl Name</th>
<th>Hardcoded Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>expect-inotify-limit</td>
<td>fs.inotify.max_user_watches</td>
<td>524288</td>
</tr>
</tbody>
</table>

All project-local settings are also allowed here.

**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. Default is currently a special url `mirror://mirrors.ubuntu.com/mirrors.txt` which choses local mirror for you. But it sometimes fails. Therefore we reserve an option to change the default later.

The best value for this settings is probably `http://<COUNTRY_CODE>.archive.ubuntu.com/ubuntu/`.

**ubuntu-skip-locking**

Enables `-o Debug::NoLocking=yes`. This is *super-experimental*, but allows to build multiple ubuntu images in parallel even when they use the same cache (i.e. in the same project, or when using `cache-dir`).

**alpine-mirror**

Set to your preferred alpine mirror. By default it’s the random one is picked from the list.

**build-lock-wait**

By default (`build-lock-wait: false`) vagga stops current command and prints a message when some other process have already started to build the image. When this flag is set to `true` vagga will wait instead. This is mostly useful for CI systems.

**environ**

The mapping, that overrides environment variables set in container or command.

**run-symlinks-as-commands**

(default `true`) If the setting is true, when there is a symlink named `yyy` that points to a vagga, and vagga is run by calling the name of that symlink vagga finds a command with `symlink-name` which equals to this command and runs it directly, passing all the arguments to that command (i.e. vagga doesn’t try to parse command-line itself).

New in version 0.7.1.
index-all-images
(default false) When the setting is true then vagga will hash all the files inside the containers and will create a special signature file.

New in version 0.7.1.

hard-link-identical-files
This functionality is experimental. Use at your own risk.

(default false) This setting is paired with index-all-images. When both settings are true vagga will search identical files inside other containers and will replace the same files with hard links. This can significantly reduce a disk space occupied by the containers.

There are two precautions about this setting:

1. Date modified, date created and most other metadata is ignored
2. If you edit file directly in .vagga/<container-name> you may edit files in multiple containers at the same time (this is similar to transient-hard-link-copy in a some way)

New in version 0.7.2.

New in version 0.8.1: You can now use vagga _verify <container> to check that contents of the container match checksums in the index and are not mistakenly modified.

hard-link-between-projects
Similar to hard-link-identical-files but also makes hardlinks to containers of different projects. Similar precautions apply.

This option works only if storage-dir is specified.

This option optimizes disk space right after container is built. Use vagga _hardlink --global to optimize older containers.

New in version 0.8.1.

disable-auto-clean
Disables auto-clean option in all containers. This is useful on CI systems where multiple parallel builds should work.

versioned-build-dir
(default false) When building container, say mycontainer by default we use .tmp.mycontainer dir for building. This settings enables naming dir .tmp.mycontainer.1a2b3c4 where 1a2b3c4 is a container version being built.

Note: for some containers where we can’t determine version before building a container this setting does nothing.

It’s useful to turn this setting on on CI systems with configured storage_dir, when multiple versions of a single container could be being built simultaneously. It makes little sense to enable it on a workstation.

2.4.4 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.

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

The full error might look like:
The first message above means there is no `/etc/subuid` file. It probably means you need to create one. The second option means there is a `/etc/subuid` file but your user is absent in the file.

The recommended contents of `/etc/subuid` are following:

```
your_user_name:100000:65536
```

You should also check `/etc/subgid`, add presumably the same contents to `/etc/subgid` (In subgid file the first field still contains your user name not a group name).

You may get another similar error:

```
ERROR:vagga::container::uidmap: Error reading uidmap: /etc/subuid:2: Bad syntax: "user:100000:1000"
WARN:vagga::container::uidmap: Could not read /etc/subuid or /etc/subgid (see http://bit.ly/err_subuid)
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”).

**Can’t find newuidmap or newgidmap**

Full error usually looks like:

```
WARN:vagga::process_util: Can't find `newuidmap` or `newgidmap` (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):

```bash
$ 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, Arch Linux, Fedora) have binaries as part of “shadow” package, so have them installed on every system.

### 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. You may not have needed permissions (see http://bit.ly/err_root)
```

Both show that you don’t run vagga with the user that owns the project. The legitimate 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 --ignore-owner-check`. If you have good case where this works, please file an issue and we might make the check better.

### Don’t run vagga on shared folders

If you are testing or using vagga using an OS X machine, you might be tempted to set up a Linux virtual machine and share your definitions into the machine. Sadly, this leads to the shared folder implementation lying to you: it pretends all files are owned by the user you use on the guest system, but this is not fully true. This manifests in errors such as this:
and subsequent errors, including vagga not being allowed to clean up after itself.

Don’t use shared folders as your cache dir or storage dir, see Settings for ways to change them.

**Out of Shared Memory**

We use 100MiB for shared memory by default, to increase it add the following to the volumes of your container (or command):

```
containers:
  some-container:
    volumes:
      /run: !Tmpfs
        size: 1Gi # your new size of shared memory
    subdirs:
      shm:  # create shm directory
```

**Known scenarios**

Facebook’s flow requires a lot of shared memory. The error that can be seen in the log is as follows:

```
[2016-08-11 06:59:40] We've run out of filesystems to use for shared memory
SharedMem.Out_of_shared_memory
```

The amount of memory needed probably depends on an application that is being compiled by flow.

### 2.4.5 System Settings

Vagga sometimes hints and if permitted can tune few options on a host system. This is the reference of the options that vagga can fix.

See auto-apply-sysctl for a table of options and limits.

**Sysctl fs.inotify.max_user_watches**

The inotify is used to notify user processes that some file or directory is changed by another process. It’s tweaked by expect-inotify-limit.

It’s very useful for the following things:

1. Run processes with automatic restart on reload
2. Run build system and automatically rebuild on file change
3. Start unit tests on each file change
Unfortunately on some systems (namely ubuntu xenial, docker on mac) it’s very common to have a limit of 8192 inotify watches. Which is too slow on some systems.

The error is manifested as:

- inotify watch limit reached
- ENOSPC/No space left on device (yes, this is not a typo)
- Internal watch failed: watch ENOSPC
- Some programs just crash (see #291)

Tuning it is usually harmless unless the value is too large. Each user watch takes up to 1080 bytes. So values up to 512K are fine on most current systems.

To tune it (temporarily) you need to run:

```bash
sudo sysctl fs.inotify.max_user_watches=524288
```

To store for the next reboot you may try to add `-w`:

```bash
sudo sysctl -w fs.inotify.max_user_watches=524288
```

But it doesn’t work for some linux distributions (hello, NixOS)

Alternatively, you may set `auto-apply-sysctl`. This tells vagga to automatically run `sudo -k sysctl ...` on your behalf (probably asking for a password).

### 2.4.6 OverlayFS

This page documents overlayfs support for vagga. This is currently a work in progress.

Since `unprivileged` overlayfs is unsupported in mainline kernel, you may need some setup. Anyway, `ubuntu`’s stock kernel has the patch applied.

**The Plan**

1. Make of use of overlayfs in `Snapshot` volume. This will be enabled by a volume-level setting initially. In perspective the setting will be default on systems that support it.

2. Use overlayfs for `_run --writable` and transient copies

3. Use overlayfs for `Container` step. This will be enabled by a container-level setting. Which, presumably, will always be disabled by default.

4. Add `vagga _build container --cache-each-step` to ease debugging of container builds (actually to be able to continue failing build from any failed step)

Smaller things:

- `vagga _check_overlayfs_support`

We need a little bit more explanation about why we would keep overlayfs disabled by default. The first thing to know, is that while we will mount overlays for filesystems inside the container, we can’t mount overlays outside of the container.

So we want to have first class IDE support by default (so you can point to one folder for project dependencies, not variable list of layered folders)

For `--cache-each-step` the main reason is performance. From experience with Docker we know that snapshotting each step is not zero-cost.
Setup

This section describes quirks on various systems that are needed to enable this feature.

To check this run:

```
$ vagga _check_overlayfs_support
supported
$ uname -r -v
4.5.0 #1-NixOS SMP Mon Mar 14 04:28:54 UTC 2016
```

If first command reports supported please report your value of `uname -rv` so we can add it to the lists below.

The original patch made by Canonical’s employee is just one line, and has pretty extensive documentation about why it’s safe enough.

Ubuntu

It works by default on Ubuntu trusty 14.04. It’s reported successfully on the following systems:

```
```

Arch Linux

Since you already use custom kernel, you just need another patch. If you use the package recommended in installation page<archlinux_> your kernel already supports overlayfs too.

The AUR package has he feature enabled too, this is were you can find the PKGBUILD to build the kernel yourself.

NixOS

On NixOS you need to add a patch and rebuild the kernel. Since the patch is already in the nixos source tree, you need just the following in your `/etc/nixos/configuration.nix`:

```nix
nixpkgs.config.packageOverrides = pkgs: {
  linux_4_5 = pkgs.linux_4_5.override { kernelPatches = [ pkgs.kernelPatches.ubuntu_unprivileged_overlayfs ]; }; 
};
```

Adjust kernel version as needed.

2.5 Network Testing

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

---

**Warning:** This documentation is awfully incomplete. There is a good article in the meantime. Except vagga_network command is replaced by vagga _network subcommand (note the space after vagga)
2.5.1 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.

2.5.2 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:

```bash
sudo 'ip' 'link' 'add' 'vagga_guest' 'type' 'veth' 'peer' 'name' 'vagga'
sudo 'ip' 'link' 'set' 'vagga_guest' 'netns' '16508'
sudo 'ip' 'addr' 'add' '172.23.255.1/30' '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/users'
```

The following iptables rules will be established:

```bash
["-I", "INPUT", "-i", "vagga", "-d", "127.0.0.1", "-j", "ACCEPT"]
["-t", "nat", "-A", "POSTROUTING", "-s", "172.23.255.0/30", "-j", "MASQUERADE"]
```

Then immediately 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 172.23.0.0/16 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.

### 2.5.3 Containers

Here is a quick example of how to run network tests: `vagga.yaml`

The configuration runs flask application with nginx and periodically stops network between processes. For example here is test for normal connection:

```bash
$ vagga run-normal &
$ vagga wrk http://172.23.255.2:8000 --latency
Running 10s test @ http://172.23.255.2:8000
2 threads and 10 connections
```

(continues on next page)
Here is the same test with bad network connection:

```
$ vagga run-flaky &
$ vagga wrk http://172.23.255.2:8000 --latency
Running 10s test @ http://172.23.255.2:8000
  2 threads and 10 connections
Thread Stats  Avg  Stdev  Max  +/- Stdev
  Latency     241.69ms 407.98ms 1.41s  81.67%
  Req/Sec     631.83  299.12  1.14k  71.05%
Latency Distribution
  50%       7.27ms
  75%      355.09ms
  90%     991.64ms
  99%      1.37s
5032 requests in 10.01s, 0.87MB read
Requests/sec: 502.64
Transfer/sec:  89.32KB
```

The run-flaky works as follows:

- Stop networking packets going between nginx and flask (`iptables .. -j DROP`)
- Sleep for a second
- Restore network
- Sleep for a second
- Repeat

The respective part of the configuration looks like:

```
interrupt: !BridgeCommand
container: test
run: |
  set -x
  while true; do
    vagga _network isolate flask
    sleep 1
    vagga _network fullmesh
    sleep 1
  done
```

As you can see in the test there are interesting differences:

- average latency is 241ms vs 5ms
- median latency is about the same

2.5. Network Testing
• 99 percentile of latency is 1.37s vs 11.62ms (i.e. 100x bigger)
• request rate 502 vs 1647

The absolute scale doesn’t matter. But intuitively we could think that if network doesn’t work 50% of the time it should be 3x slower. But it isn’t. Different metrics are influenced in very different way.

### 2.5.4 Partitioning

# TBD

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

### 2.6 Tips And Tricks

#### 2.6.1 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.

#### 2.6.2 Multiple Build Attempts

Despite of all the caching vagga does, it’s usually to slow to rebuild a big container when trying to install even a single package. You might try something like this:

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

Note that the flag --writeable or shorter -W doesn’t write into the container itself, but creates a (hard-linked) copy, which is destructed on exit. 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 the container is built. This is done to keep the image smaller. So, if you need for example to run apt-get install you would always need to run apt-get update first.

Another technique is to use PHP/Composer Installer.

#### 2.6.3 Debug Logging

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

```
$ RUST_LOG=debug vagga _build container
```
2.6.4 I’m Getting “permission denied” Errors

When starting vagga, if you see the following error:

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

Then you might not have the appropriate 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 that 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 the Installation page, please report an issue with the information of your distribution (at least `uname` and `/etc/os-release`), so I can add instructions.

2.6.5 Fix “insufficient permissions” for USB device

To allow access on a USB device from inside the container, the device permissions need to be set properly on the host system.

Either you use `chown` to set the owner of the device under `/dev/bus/usb/...` to `<your_host_user>` or you define a udev rule on your host system to grant access to the USB device.

A simple rule which grants access to all users for all devices of a vendor may look like this:

```
ATTRS{idVendor}=="04b8", ATTRS{idProduct}=="*", MODE="0777"
```

`MODE="0777"` in your udev rule will allow access for every user, while `OWNER="your_host_user"` will only grant access to your user.

To list your device attributes, use e.g.:

```
$ udevadm info -a -n /dev/bus/usb/001/003
```

2.6.6 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
```

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:
2.6.7 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 container 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 including 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:

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

Or specific version can be looked:

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

The latter form is useful to compare with older versions of the same container.

2.7 Conventions

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

2.7.1 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

2.7.2 Command Naming

`run`

To run a project you should just start:

```bash
$ 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).
2.8 Examples and Tutorials

2.8.1 Tutorials

Building a Django project

This example will show how to create a simple Django project using vagga.

- Creating the project structure
- Freezing dependencies
- Let’s add a dependency
- Adding some code
- Trying out memcached
- Why not Postgres?

Creating the project structure

In order to create the initial project structure, we will need a container with Django installed. First, let’s create a directory for our project:

```sh
```

Now create the `vagga.yaml` file and add the following to it:

```yaml
containers:
  django:
    setup:
    - !Alpine v3.5
    - !Py3Install ['Django >=1.10,<1.11']
```

and then run:

```sh
$ vagga _run django django-admin startproject MyProject .
```

This will create a project named `MyProject` in the current directory. It will look like:

```
~/projects/vagga-django-tutorial
  MyProject
    __init__.py
    settings.py
    urls.py
    wsgi.py
    manage.py
    vagga.yaml
```

Notice that we used 'Django >=1.10,<1.11' instead of just Django. It is a good practice to always specify the major and minor versions of a dependency. This prevents an update to an incompatible version of a library breaking your project. You can change the Django version if there is a newer version available ('Django >=1.11,<1.12' for instance).
Freezing dependencies

It is a common practice for python projects to have a requirements.txt file that will hold the exact versions of the project dependencies. This way, any developer working on the project will have the same dependencies.

In order to generate the requirements.txt file, we will create another container called app-freezer, which will list our project’s dependencies and output the requirements file.

```yaml
collectors:
  app-freezer:
    setup:
      - !Alpine v3.5
      - !Py3Install
      - 'Django >=1.10,<1.11'
      - !Sh python3 -m pip freeze > requirements.txt
  django:
    setup:
      - !Alpine v3.5
      - !Py3Requirements requirements.txt
```

- our new container
- call pip trough python3 -m to generate the requirements file
- just reference the requirements file from django container

Every time we add a new dependency, we need to rebuild the app-freezer container to generate the updated requirements.txt.

Now, build the app-freezer container:

```
$ vagga _build app-freezer
```

You will notice the new requirements.txt file holding a content similar to:

```
Django==1.10.5
```

And now let’s run our project. Edit vagga.yaml to add the run command:

```yaml
collectors:
  # same as before
commands:
  run: !Command
description: Start the django development server
  container: django
  run: python3 manage.py runserver
```

and then run:

```
$ vagga run
```

If everything went right, visiting localhost:8000 will display Django’s welcome page saying ‘It worked!’.

Let’s add a dependency

By default, Django is configured to use sqlite as its database, but we want to use a database url from an environment variable, since it’s more flexible. However, Django does not understand database urls, so we need django-environ to parse configuration urls into the format Django understands.
Add `django-environ` to our `app-freezer` container:

```yaml
containers:
  app-freezer:
    setup:
      - !Alpine v3.5
      - !PipConfig dependencies: true
      - !Py3Install
      - 'Django >=1.10,<1.11'
      - 'django-environ >=0.4,<0.5'
      - !Sh python3 -m pip freeze > requirements.txt

• – `django-environ` have a dependency on the package `six` which would not be installed unless we tell pip to do so

Rebuild the `app-freezer` container to update `requirements.txt`:

```
$ vagga _build app-freezer
```

Set the environment variable:

```yaml
containers:
  #...
  django:
    environ:
      DATABASE_URL: sqlite:///db.sqlite3
    setup:
      - !Alpine v3.5
      - !Py3Requirements requirements.txt

• – will point to `/work/db.sqlite3`

Now let’s change our project’s settings by editing `MyProject/settings.py`:

```python
# MyProject/settings.py
import os
import environ
env = environ.Env()

# other settings
DATABASES = {
    'default': env.db()
}
```

Let’s add a shortcut command for `manage.py`:

```yaml
commands:
  # ...
  manage.py: !Command
    description: Shortcut to manage.py
    container: django
    run: [python3, manage.py]
```

**Note:** This command accept arguments by default, so instead of writing it long `vagga _run django python3`
manage.py runserver we will be able to shorten it to vagga manage.py runserver

To see if it worked, let’s run the migrations from the default Django apps and create a superuser:

```bash
$ vagga manage.py migrate
$ vagga manage.py createsuperuser
```

After creating the superuser, run our project:

```bash
$ vagga run
```

visit localhost:8000/admin and log into the Django admin.

### Adding some code

Before going any further, let’s add a simple app to our project.

First, start an app called ‘blog’:

```bash
$ vagga manage.py startapp blog
```

Add it to INSTALLED_APPS:

```python
# MyProject/settings.py
INSTALLED_APPS = [
    ...,
    'blog',
]
```

Create a model:

```python
# blog/models.py
from django.db import models

class Article(models.Model):
    title = models.CharField(max_length=100)
    body = models.TextField()
```

Create the admin for our model:

```python
# blog/admin.py
from django.contrib import admin
from .models import Article

@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ('title',)
```

Create and run the migration:

```bash
$ vagga manage.py makemigrations
$ vagga manage.py migrate
```

Run our project:

2.8. Examples and Tutorials
$ vagga run

And visit localhost:8000/admin to see our new model in action.

Now create a couple views:

# blog/views.py

```python
from django.views import generic
from .models import Article

class ArticleList(generic.ListView):
    model = Article
    paginate_by = 10

class ArticleDetail(generic.DetailView):
    model = Article
```

Create the templates:

```html
<!-- blog/templates/blog/article_list.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Article List</title>
</head>
<body>
    <h1>Article List</h1>
    <ul>
      {% for article in article_list %}
        <li><a href="{% url 'blog:article_detail' article.id %}">
          {{ article.title }}
        </a></li>
      {% endfor %}
    </ul>
</body>
</html>

<!-- blog/templates/blog/article_detail.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Article List</title>
</head>
<body>
    <h1>{{ article.title }}</h1>
    <p>{{ article.body }}</p>
</body>
</html>
```

Set the urls:

# blog/urls.py

```python
from django.conf.urls import url
from . import views
```

(continues on next page)
urlpatterns = [
    url(r'^$', views.ArticleList.as_view(), name='article_list'),
    url(r'^(?P<pk>\d+?)$', views.ArticleDetail.as_view(), name='article_detail'),
]

# MyProject/urls.py
from django.conf.urls import url, include
from django.contrib import admin
urlpatterns = [
    url(r'^$', include('blog.urls', namespace='blog')),
    url(r'^admin/', admin.site.urls),
]

Note:  Remember to import include at the first line

Now run our project:

$ vagga run

and visit localhost:8000. Try adding some articles through the admin to see the result.

Trying out memcached

Many projects use memcached to speed up things, so let's try it out.

Add pylibmc to our app-freezer, as well as its build dependencies:

containers:
  app-freezer:
    setup:
      - !Alpine v3.5
      &build_deps !BuildDeps
      - libmemcached-dev
      - zlib-dev
      - cyrus-sasl-dev
      - !PipConfig
dependencies: true
      - !Py3Install
        - 'Django >=1.10,<1.11'
        - 'django-environ >=0.4,<0.5'
        - 'pylibmc >=1.5,<1.6'
      - !Sh python3 -m pip freeze > requirements.txt
  django:
    environ:
      DATABASE_URL: sqlite:///db.sqlite3
    setup:
      - !Alpine v3.5
      - *build_deps
      - !Py3Requirements requirements.txt

  • – we used an YAML anchor (&build_deps) to avoid repetition of the build dependencies
• – libraries needed to build pylibmc

• – the YAML alias *build_deps references the anchor declared in the app-freezer container, so we don’t need to repeat the build dependencies on both containers

And rebuild the container:

$ vagga _build app-freezer

Add the pylibmc runtime dependencies to our django container:

```
containers:
  # ...
  django:
    setup:
    - !Alpine v3.5
    - *build_deps
    - !Install
      - libmemcached
      - zlib
      - libsasl
    - !Py3Requirements requirements.txt
    environ:
      DATABASE_URL: sqlite:///db.sqlite3
```

• – libraries needed by pylibmc at runtime

Create a new container called memcached:

```
containers:
  # ...
  memcached:
    setup:
      - !Alpine v3.5
      - !Install [memcached]
```

Create the command to run with caching:

```
commands:
  # ...
  run-cached: !Supervise
  description: Start the django development server alongside memcached
  children:
    cache:
      container: memcached
      run: memcached -u memcached -vv
    app:
      container: django
      environ:
        CACHE_URL: pymemcache://127.0.0.1:11211
      run: python3 manage.py runserver
```

• – run memcached as verbose so we see can see the cache working

• – set the cache url

Change MyProject/settings.py to use our memcached container:

```
import os
import environ

(continues on next page)```
env = environ.Env()

# other settings
CACHES = {
    # will read CACHE_URL from environment
    # defaults to memory cache if environment is not set
    'default': env.cache(default='locmemcache://')
}

Configure our view to cache its response:

```python
# blog/urls.py
from django.conf.urls import url
from django.views.decorators.cache import cache_page
from . import views

cache_15m = cache_page(60 * 15)

urlpatterns = [
    url(r'^$', views.ArticleList.as_view(), name='article_list'),
    url(r'^(?P<pk>[\d+]+)$', cache_15m(views.ArticleDetail.as_view()), name='article_detail'),
]
```

Now, run our project with memcached:

```
$ vagga run-cached
```

And visit any article detail page, hit Ctrl+r to avoid browser cache and watch the memcached output on the terminal.

**Why not Postgres?**

We can test our project against a Postgres database, which is probably what we will use in production.

First add psycopg2 and its build dependencies to app-freezer:

```yaml
containers:
  app-freezer:
    setup:
      - !Alpine v3.5
      - !BuildDeps
        - libmemcached-dev
        - zlib-dev
        - cyrus-sasl-dev
        - postgresql-dev
      - !PipConfig
        dependencies: true
      - !Py3Install
        - 'Django >=1.10,<1.11'
        - 'django-environ >=0.4,<0.5'
        - 'pylibmc >=1.5,<1.6'
        - 'psycopg2 >=2.6,<2.7'
      - !Sh python3 -m pip freeze > requirements.txt

• – library needed to build psycopg2
```

2.8. Examples and Tutorials 105
• – psycopg2 dependency

Rebuild the container:

$ vagga _build app-freezer

Add the runtime dependencies of psycopg2:

containers:
  django:
    setup:
    - !Alpine v3.5
    - *build_deps
    - !Install
    - libmemcached
    - zlib
    - libtasl
    - libpq
    - !Py3Requirements requirements.txt
    environ:
      DATABASE_URL: sqlite:///db.sqlite3

• – library needed by psycopg2 at runtime

Before running our project, we need a way to automatically create our superuser. We can create a migration to do this. First, create an app called common:

$ vagga manage.py startapp common

Add it to INSTALLED_APPS:

INSTALLED_APPS = [
    # ...
    'common',
    'blog',
]

Create the migration for adding the admin user:

$ vagga manage.py makemigrations -n create_superuser --empty common

Change the migration to add our admin user:

# common/migrations/0001_create_superuser.py
from django.db import migrations
from django.contrib.auth.hashers import make_password

def create_superuser(apps, schema_editor):
    User = apps.get_model("auth", "User")
    User.objects.create(username='admin',
        email='admin@example.com',
        password=make_password('change_me'),
        is_superuser=True,
        is_staff=True,
        is_active=True)
Create the database container:

```python
class Migration(migrations.Migration):
    dependencies = [
        ('auth', '__latest__')
    ]

    operations = [
        migrations.RunPython(create_superuser)
    ]
```

Now add the command to initialize the database:

```bash
commands:
    # ...
    _pg-init: !Command
        description: Init postgres database
        container: postgres
        user-id: 200
```

Note: The database will be persisted in `.vagga/.volumes/postgres`.
group-id: 200
run: |
  set -ex
  ls -la /data
  $PG_BIN/pg_ctl initdb
  $PG_BIN/pg_ctl -w -o '-F --port=$PG_PORT -k /tmp' start
  $PG_BIN/createuser -h 127.0.0.1 -p $PG_PORT $PG_USER
  $PG_BIN/createdb -h 127.0.0.1 -p $PG_PORT $PG_DB -O $PG_USER
  $PG_BIN/psql -h 127.0.0.1 -p $PG_PORT -c "ALTER ROLE $PG_USER WITH ENCRYPTED PASSWORD '$PG_PASSWORD';"
  $PG_BIN/pg_ctl stop

And then add the command to run with Postgres:

commands:
  # ...
  run-postgres: !Supervise
description: Start the django development server using Postgres database
children:
  app: !Command
container: django
environ:
  DATABASE_URL: postgresql://vagga:vagga@127.0.0.1:5433/test
run: |
  python3 manage.py migrate
  python3 manage.py runserver
  
  db: !Command
container: postgres
user-id: 200
group-id: 200
run: exec $PG_BIN/postgres -F --port=$PG_PORT

Now run:

$ vagga run-postgres

Visit localhost:8000/admin and try to log in with the user and password we defined in the migration.

Building a Laravel project

This tutorial will show how to create a simple Laravel project on Alpine using PHP7, MySQL and Redis.

• Create the project
• Setup the database
• Inspecting the database
• Adding some code
• Setup Redis

Create the project

In order to create the initial project structure, we will need a container with the Laravel installer. First, let’s create a directory for our project:
Create the `vagga.yaml` file and add the following to it:

```yaml
containers:
  app:
    setup:
      - !Alpine v3.5
      - !Repo community
      - !Install
        - ca-certificates
        - php7
        - php7-openssl
        - php7-mbstring
        - php7-phar
        - php7-json
      - !ComposerConfig
        install-runtime: false
        runtime-exe: /usr/bin/php7
        keep-composer: true
      - !ComposerInstall
    environ:
      HOME: /tmp
```

Here we are building a container from Alpine v3.5 and telling it to install PHP7 and everything needed to run Composer. Now let’s create our new project:

```bash
$ vagga _run app composer create-project \
  --prefer-dist --no-install --no-scripts \
  laravel/laravel src 5.4.*
$ mv src/* src/.* .
$ rmdir src
```

The first command is quite big! It tells composer to create a new project from `laravel/laravel` version 5.4 and place it into the `src` directory. The three flags tell composer to:

- `--prefer-dist` install packages from distribution source when available;
- `--no-install` do not run `composer install` after creating the project;
- `--no-scripts` do not run scripts defined in the root package.

We want our project’s files to be in the current directory (the one containing `vagga.yaml`) but Composer only accepts an empty directory, so we tell it to create the project into `src`, move its contents into the current directory and remove `src`.

Now that we have our project created, change our container as follows:

```
containers:
  app-base:
    setup:
      - !Alpine v3.5
      - !Repo community
      - !Install
        - ca-certificates
        - php7
        - php7-openssl
        - php7-mbstring
        - php7-pdo_mysql
        - php7-mbstring
```

(continues on next page)
- php7-xml
- php7-session
- php7-dom
- php7-phar
- php7-json
- php7-posix
- php7-ctype
- !Sh ln -s /usr/bin/php7 /usr/bin/php

app:
environ: &env
    APP_ENV: development
    APP_DEBUG: true
    APP_KEY: YourRandomGeneratedEncryptionKey
setup:
    - !Container app-base
    - !Env { <<: *env }
    - !ComposerConfig
        install-runtime: false
        runtime-exe: /usr/bin/php7
        keep-composer: true
    - !EnsureDir /work/vendor
    - !EnsureDir /usr/local/lib/composer/vendor
    - !Sh mount --bind,ro /usr/local/lib/composer/vendor /work/vendor
    - !ComposerDependencies
    - !Sh umount /work/vendor
    volumes:
        /work/vendor: !BindRO /vagga/root/usr/local/lib/composer/vendor

This might look complex, but let's break it down:

app-base:
    setup:
        - !Alpine v3.5
        - !Repo community
        - !Install
            ca-certificates
            php7
            php7-openssl
            php7-pdo_mysql
            php7-mbstring
            php7-xml
            php7-session
            php7-dom
            php7-phar
            php7-json
            php7-posix
            php7-ctype
        - !Sh ln -s /usr/bin/php7 /usr/bin/php

The container for our application is based on Alpine linux v3.5 and we will use PHP7, so we need to enable the “community” repository from Alpine and install php7 and the modules needed for both Laravel and Composer.

We also link the php7 executable into /usr/bin/php to make it available as just php.

This container will be used as the base for the app container in order to speed up builds.
Here we are configuring our application. Laravel comes out of the box with its configuration done through environment variables, so we are setting these to what we need to a development environment. The default project template uses `dotenv` to load configuration into environment automatically from a `.env` file, but we won’t use that. Instead, we tell vagga to set the environment for us.

We are also setting and yaml anchor (`&env`) so we can reference it later.

```
setup:
- !Container app-base
- !Env { <<: *env }
```

We are extending the `app-base` container and referencing the yaml anchor we defined earlier to make the environment available during build.

```
- !ComposerConfig
  install-runtime: false
  runtime-exe: /usr/bin/php7
  keep-composer: true
```

Since we installed php by ourselves, we tell vagga to use version we installed instead of the default version from Alpine.

```
- !EnsureDir /work/vendor
- !EnsureDir /usr/local/lib/composer/vendor
- !Sh mount --bind,ro /usr/local/lib/composer/vendor /work/vendor
- !ComposerDependencies
- !Sh umount /work/vendor
```

The `composer.json` created by Laravel has some post install hooks that expect the `vendor` directory to be available at the project root, but vagga installs composer dependencies under `/usr/local/lib/composer`. To make it available to our application during the build, we mount that directory into `/work/vendor` and `umount` afterwards.

To test if everything is ok, let’s add a command to run our project:

```
containers:
  # ...
commands:
  run: !Command
    container: app
    description: run the laravel development server
    run: |
      php artisan cache:clear
      php artisan config:clear
      php artisan serve
```

• – clear application cache to prevent previous runs from interfering on subsequent runs.

Now run our project:

```
$ vagga run
```

And visit `localhost:8000`. If everything is OK, you will see Laravel default page saying “Laravel 5”.

2.8. Examples and Tutorials
Setup the database

Every PHP project needs a database, and ours is not different, so let’s create a container for our database:

```yaml
containers:
  # ...
  mysql:
    setup:
      - !Ubuntu xenial
      - !UbuntuUniverse
      - !Sh |
        addgroup --system --gid 200 mysql
        adduser --uid 200 --system --home /data --no-create-home \
          --shell /bin/bash --group --gecos "MySQL user" \n        mysql
      - !Install
        - mysql-server-5.7
        - mysql-client-5.7
      - !Remove /var/lib/mysql
      - !EnsureDir /data
    environ: &db_config
      DB_DATABASE: vagga
      DB_USERNAME: vagga
      DB_PASSWORD: vagga
      DB_HOST: 127.0.0.1
      DB_PORT: 3307
      DB_DATA_DIR: /data
    volumes:
      /data: !Persistent
        name: mysql
        owner-uid: 200
        owner-gid: 200
        init-command: _mysql-init
      /run: !Tmpfs
        subdirs:
          mysqld: { mode: 0o777 }
```

- Use fixed user id and group id for mysql
- Put an anchor at the database environment so we can reference it later
- Vagga command to initialize the volume

**Note:** The database will be persisted in .vagga/.volumes/mysql.

Add the command to initialize the database:

```yaml
commands:
  # ...
  _mysql-init: !Command
    description: Init MySQL data volume
    container: mysql
    user-id: 200
    group-id: 200
    run: |
      set -ex
```

(continues on next page)
mysqld --initialize-insecure --datadir=$DB_DATA_DIR \ --log-error=log

mysqld --datadir=$DB_DATA_DIR --skip-networking --log-error=log &

while [ ! -S /run/mysqld/mysqld.sock ]; do sleep 0.2; done

mysqladmin -u root create $DB_DATABASE

mysql -u root -e "CREATE USER '$DB_USERNAME'@'localhost' IDENTIFIED BY '$DB_PASSWORD';"

mysql -u root -e "GRANT ALL PRIVILEGES ON $DB_DATABASE.* TO '$DB_USERNAME'@'localhost';"

mysql -u root -e "FLUSH PRIVILEGES;"

mysqladmin -u root shutdown

Add the php mysql module to our container:

containers:
  app-base:
    - !Alpine v3.5
    - !Repo community
    - !Install
      - ca-certificates
      - php7
        # ...
      - php7-pdo_mysql # mysql module
        # ...

Now change our run command to start the database alongside our project:

commands:
  run: !Supervise
    description: run the laravel development server
    children:
      app: !Command
        container: app
        environ: *db_config
        run: |
          php artisan cache:clear
          php artisan config:clear
          php artisan serve
      db: !Command
        container: mysql
        user-id: 200
        group-id: 200
        run: |
          exec mysqld --datadir=$DB_DATA_DIR \ --bind-address=$DB_HOST --port=$DB_PORT \ --log-error=log --gdb

  • – Reference the database environment

And run our project:

$ vagga run
Inspecting the database

Now that we have a working database, we can inspect it using a small php utility called adminer. Let’s create a container for it:

```
containers:
  # ...
  adminer:
    setup:
      - !Alpine v3.5
      - !Repo community
      - !Install
        - php7
        - php7-pdo_mysql
        - php7-session
      - !EnsureDir /opt/adminer
      - !EnsureDir /opt/adminer/plugins
      - !Download
        url: https://www.adminer.org/static/download/4.2.5/adminer-4.2.5-mysql.php
        path: /opt/adminer/adminer.php
      - !Download
        url: https://raw.github.com/vrana/adminer/master/designs/nette/adminer.css
        path: /opt/adminer/adminer.css
      - !Download
        path: /opt/adminer/plugins/plugin.php
      - !Download
        path: /opt/adminer/plugins/login-servers.php
      - !Text
        /opt/adminer/index.php: |
        ```php
        function adminer_object() {
          include_once './plugins/plugin.php';
          foreach (glob('./plugins/*.php') as $filename) { include_once './'.$filename; }
          $plugins = [new AdminerLoginServers(['127.0.0.1:3307' => 'Dev DB'])];
          return new AdminerPlugin($plugins);
        }
        ```
    ```

- -- download the adminer script.
- -- use a better style (optional).
- -- adminer plugin support
- -- login-servers plugin to avoid typing server address and port
- -- setup adminer

The container above will install PHP7 along with the mysql and session modules, then it will download adminer itself, the optional style, the plugin support and the “login-servers” plugin. This plugin will allow us to select the database we are connecting to from a list instead of filling in the host and port. The last part of the container setup configures adminer with our database.

Now change our run command to start the adminer container:
commands:
  run: !Supervise
  description: run the laravel development server
  children:
    app: !Command
      # ...
    db: !Command
      # ...
    adminer: !Command
      container: adminer
      run: php7 -S 127.0.0.1:8001 -t /opt/adminer

This command will start the php embedded server with its root pointing to the directory we setup for Adminer.
To access adminer, visit localhost:8001 and fill the username and password fields with “vagga”.

Adding some code

Now that we have our project working and our database is ready, let’s add some.
Let’s add a shortcut command for running artisan

commands:
  # ...
  artisan: !Command
    description: Shortcut for running artisan cli
    container: app
    run: [php, artisan]

Now, we need a layout. Fortunately, Laravel can give us one, we just have to scaffold authentication:

$ vagga artisan make:auth

This will give us a nice layout at resources/views/layouts/app.blade.php.

Now create a model:

$ vagga artisan make:model --migration Article

This will create a new model at app/Article.php and its respective migration at database/migrations/2016_03_24_172211_create_articles_table.php (yours will have a slightly different name).

Open the migration file and tell it to add two fields, title and body, to the database table for our Article model:

```php
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateArticlesTable extends Migration
{
    public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->increments('id');
            $table->string('title', 100);
            $table->text('body');
        });
    }
}
```

(continues on next page)
$table->timestamps();

public function down()
{
    Schema::drop('articles');
}

Open routes/web.php and setup routing:

```php
<?php
Route::get('/', 'ArticleController@index');
Route::resource('/article', 'ArticleController');
Auth::routes();
Route::get('/home', 'HomeController@index');
```

Create our controller:

```bash
$vagga artisan make:controller --resource ArticleController
```

This will create a controller at app/Http/Controllers/ArticleController.php populated with some CRUD method stubs.

Now change the controller to actually do something:

```php
namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Article;

class ArticleController extends Controller
{
    public function index()
    {
        $articles = Article::orderBy('created_at', 'asc')->get();
        return view('article.index', [
            'articles' => $articles
        ]);  
    }

    public function create()
    {
        return view('article.create');
    }

    public function store(Request $request)
    {
        $this->validate($request, [
            'title' => 'required|max:100',
            'body' => 'required'
```
$article = new Article;
$article->title = $request->title;
$article->body = $request->body;
$article->save();

return redirect('/');
}

class ArticleController extends Controller
{
    public function create(Request $request)
    {
        $article = new Article;
        $article->title = $request->title;
        $article->body = $request->body;
        $article->save();

        return redirect('/');
    }

    public function show($id)
    {
        $article = Article::find($id);
        return view('article.show', [
            'article' => $article
        ]);    }

    public function edit(Article $article)
    {
        return view('article.edit', [
            'article' => $article
        ]);    }

    public function update(Request $request, Article $article)
    {
        $article->title = $request->title;
        $article->body = $request->body;
        $article->save();

        return redirect('/');
    }

    public function destroy(Article $article)
    {
        $article->delete();
        return redirect('/');
    }
}

Create the views for our controller:

<!-- resources/views/article/show.blade.php -->
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <h2>{{ $article->title }}</h2>
            <p>{{ $article->body }}</p>
        </div>
    </div>
</div>
@endsection
<!-- resources/views/article/index.blade.php -->
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <h2>Article List</h2>
            @if (count($articles) > 0)
            <table class="table table-bordered table-striped">
                <thead>
                    <th>id</th>
                    <th>title</th>
                    <th>actions</th>
                </thead>
                <tbody>
                    @foreach($articles as $article)
                    <tr>
                        <td>{{ $article->id }}</td>
                        <td>{{ $article->title }}</td>
                        <td>
                            <a href="{{ url('article/'.$article->id) }}" class="btn btn-success"><i class="fa fa-btn fa-eye"></i>View</a>
                            <a href="{{ url('article/'.$article->id.'/edit') }}" class="btn btn-primary"><i class="fa fa-btn fa-pencil"></i>Edit</a>
                            <form action="{{ url('article/'.$article->id) }}" method="post" style="display: inline-block">{!! csrf_field() !!} {!! method_field('DELETE') !!}
                                <button type="submit" class="btn btn-danger" onclick="if (!window.confirm('Are you sure?')) { return false; }"><i class="fa fa-btn fa-trash"></i>Delete</button>
                            </form>
                        </td>
                    </tr>
                    @endforeach
                </tbody>
            </table>
            @endif
        </div>
    </div>
</div>
@endsection

<!-- resources/views/article/create.blade.php -->
@extends('layouts.app')
@section('content')
(continues on next page)
2.8. Examples and Tutorials
@if (count($errors) > 0)
<div class="alert alert-danger">
<ul>
@foreach ($errors->all() as $error)
<li>{{ $error }}</li>
@endforeach
</ul>
</div>
@endif

Create a seeder to prepopulate our database:

$ vagga artisan make:seeder ArticleSeeder

This will create a seeder class at `database/seeds/ArticleSeeder.php`. Open it and change it as follows:

```php
<?php
use Illuminate\Database\Seeder;
use App\Article;

class ArticleSeeder extends Seeder
{
    private $articles = [
        ['title' => 'Article 1', 'body' => 'Lorem ipsum dolor sit amet'],
        ['title' => 'Article 2', 'body' => 'Lorem ipsum dolor sit amet'],
        ['title' => 'Article 3', 'body' => 'Lorem ipsum dolor sit amet'],
        ['title' => 'Article 4', 'body' => 'Lorem ipsum dolor sit amet'],
        ['title' => 'Article 5', 'body' => 'Lorem ipsum dolor sit amet'],
    ];

    public function run()
    {
        if (Article::all()->count() > 0) {
            return;
        }

        foreach ($this->articles as $article) {
            $new = new Article;
            $new->title = $article['title'];
            $new->body = $article['body'];
            $new->save();
        }
    }
}
```

Change `database/seeds/DatabaseSeeder.php` to include `ArticleSeeder`:

```php
<?php
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    public function run()
    {
        $this->call(ArticleSeeder::class);
    }
}
```
Change the `run` command to execute the migrations and seed our database:

```
commands:
  run: Supervise
    description: run the laravel development server
    children:
      app:
        container: laravel
        environ: *
        run:
          # wait for database to be ready before starting
          dsn="mysql:host=$DB_HOST;port=$DB_PORT"
          while ! php -r "new PDO('$dsn', '$DB_USERNAME', '$DB_PASSWORD');" 2> /dev/null; do
            echo 'Waiting for database'
            sleep 2
          done
          php artisan cache:clear
          php artisan config:clear
          php artisan migrate
          php artisan db:seed
          php artisan serve
      db: Command
        # ...
      adminer: Command
        # ...
```

If you run our project, you will see the articles we defined in the seeder class. Try adding some articles, then access adminer at `localhost:8001` to inspect the database.

**Setup Redis**

Laravel can make use of `redis` to perform tasks like queues and events. In our project, we will use it to cache data from the database. First, let’s create a command to call composer:

```
commands:
  # ...
  composer: Command
    container: app
    description: run compose cli
    environ:
      COMPOSER_HOME: /usr/local/lib/composer
      COMPOSER_VENDOR_DIR: /usr/local/lib/composer/vendor
      COMPOSER_CACHE_DIR: /tmp
      COMPOSER_ALLOW_SUPERUSER: 1
    volumes:
      /usr/local/lib/composer/vendor: !Tmpfs
      /tmp: !CacheDir composer-cache
    run: [/usr/local/bin/composer]
```

- -- setup composer home, vendor dir, cache dir and allow running as root
• – mount directory as Tmpfs to make it writeable
• – mount composer cache directory

This command setup the environment needed by composer to run properly and mount the composer cache volume to
avoid downloading cached packages. The directory /usr/local/lib/composer/vendor needs to be writeable
(composer will will put packages there) so we mount it as Tmpfs.

Now let’s install predis/predis:

```
$ vagga composer require predis/predis
```

With predis installed, we can proceed to create a container for Redis:

```
containers:
redis:
  setup:
  - !Alpine v3.5
  - !Install [redis]
```

Add some yaml anchors on the run command so we can avoid repetition:

```
cmds:
  run: !Supervise
  description: run the laravel development server
  children:
    app: !Command
      container: app
      environ: *db_config
      run: &app_cmd | #
        # ...
    db: &db_cmd !Command
      # ...
    adminer: &adminer_cmd !Command
      # ...

  - set an anchor at the app child command
  - set an anchor at the db child command
  - set an anchor at the adminer child command
```

Create the command to run with caching:

```
cmds:
  # ...
  run-cached: !Supervise
  description: Start the laravel development server alongside redis
  children:
    cache: !Command
      container: redis
      run: redis-server --daemonize no --port 6380 --loglevel verbose
    app: !Command
      container: app
      environ:
        <<: *db_config
        CACHE_DRIVER: redis
        REDIS_HOST: 127.0.0.1
        REDIS_PORT: 6380
      run: *app_cmd
```

(continues on next page)
Now let’s change our controller to use caching:

```php
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Article;
use Cache;

class ArticleController extends Controller {
    public function index() {
        $articles = Cache::rememberForever('article:all', function() {
            return Article::orderBy('created_at', 'asc')->get();
        });
        return view('article.index', [
            'articles' => $articles
        ]);}
    }

    public function create() {
        return view('article.create');
    }

    public function store(Request $request) {
        $this->validate($request, [
            'title' => 'required|max:100',
            'body' => 'required'
        ]);        $article = new Article;
        $article->title = $request->title;
        $article->body = $request->body;
        $article->save();
        Cache::forget('article:all');
        return redirect('/');
    }

    public function show($id) {
        $article = Cache::rememberForever('article:'.$id, function() use ($id) {
            return Article::find($id);
        });
        return view('article.show', [
    }
}}
```
public function edit(Article $article) {
    return view('article.edit', [
    'article' => $article
    ]);}
}

public function update(Request $request, Article $article) {
    $article->title = $request->title;
    $article->body = $request->body;
    $article->save();

    Cache::forget('article:'.$article->id);
    Cache::forget('article:all');

    return redirect('/');}

public function destroy(Article $article) {
    $article->delete();
    Cache::forget('article:'.$article->id);
    Cache::forget('article:all');

    return redirect('/');}
}

Now run our project with caching:

$ vagga run-cached

Keep an eye on the console to see Laravel talking to redis, you will see something like:

3:M 15 Mar 15:20:06.418 - DB 0: 5 keys (0 volatile) in 8 slots HT.

Building a Rails project

This example will show how to create a simple Rails project using vagga.

- Creating the project structure
- Configuring the database from environment
- Adding some code
- Caching with memcached
- We should try Postgres too
Creating the project structure

First, let’s create a directory for our new project:

```bash
$ mkdir -p ~/projects/vagga-rails-tutorial && cd ~/projects/vagga-rails-tutorial
```

Now we need to create our project’s structure, so let’s create a new container and tell it to do so.

Create the `vagga.yaml` file and add the following to it:

```yaml
containers:
  rails:
    setup:
    - !Ubuntu xenial
    - !Install
    - zlib1g
    - !BuildDeps
    - zlib1g-dev
    - !GemInstall [rails:5.0]
    environ:
      HOME: /tmp
```

- `rails` depends on nokogiri, which depends on zlib.
- `--GemInstall` tells gem to install rails.
- `rails new` command, which we are going to use shortly, will complain if we do not have a `$HOME`. After our project is created, we won’t need it anymore.

We explicitly installed rails version 5.0. You can change to a newer version if it is available (5.1, for example) but your project may be slightly different.

And now run:

```bash
$ vagga _run rails rails new . --skip-bundle
```

This will create a new rails project in the current directory. The `--skip-bundle` flag tells `rails new` to not run `bundle install`, but don’t worry, vagga will take care of it for us.

Now that we have our rails project, let’s change our container fetch dependencies from `Gemfile`:

```yaml
containers:
  base:
    setup:
    - !Ubuntu xenial
    - !UbuntuUniverse
    - !Install
    - zlib1g
    - libsqlite3-0
    - nodejs
    - !BuildDeps
    - zlib1g-dev
    - libsqlite3-dev
    - !GemInstall
    - ffi
    - nokogiri
    - sqlite3
  rails:
    setup:
```

(continues on next page)
We are using two containers here, base and rails, for a good reason: some gems require building modules that can take some time to compile, so building them on the base container will avoid having to build them every time we need to rebuild our main container.

To test if everything is Ok, let’s create a command to run our project:

```
commands:
  run: !Command
    container: rails
    description: start rails development server
    run: rails server
```

Run the project:

```
$ vagga run
```

Now visit localhost:3000 to see rails default page.

**Note:** You may need to remove “tmp/pids/server.pid” in subsequent runs, otherwise, rails will complain that the server is already running.

### Configuring the database from environment

By default, the rails new command will setup sqlite as the project database and store the configuration in config/database.yml. However, we will use an environment variable to tell rails where to find our database.

To do so, delete the rails database file:

```
$ rm config/database.yml
```

And then set the enviroment variable in our vagga.yaml:

```
containers:
  rails:
    setup:
      # ...
    environ:
      DATABASE_URL: sqlite3:db/development.sqlite3
```

This will tell rails to use the same file that was configured in database.yml.

Now if we run our project, everything should be the same.

### Adding some code

Before going any further, let’s add some code to our project:
Rails scaffolding will generate everything we need, we just have to run the migrations:

```bash
$ vagga _run rails rails g scaffold article title:string:index body:text
```

Now we need to tell rails to use our articles index page as the root of our project. Change `config/routes.rb` as follows:

```ruby
# config/routes.rb
Rails.application.routes.draw do
  root 'articles#index'
  resources :articles
  # ...
end
```

Run the project now:

```bash
$ vagga run
```

You should see the articles list page rails generated for us.

### Caching with memcached

Many projects use [memcached](https://memcached.org/) to speed up things, so let’s try it out.

First, add `dalli`, a pure ruby memcached client, to our Gemfile:

```ruby
gem 'dalli'
```

Then, open `config/environments/development.rb`, find the line that says `config.cache_store = :memory_store` and change it as follows:

```ruby
# config/environments/production.rb
# ...
# config.cache_store = :memory_store
if ENV['MEMCACHED_URL']
  config.cache_store = :mem_cache_store, ENV['MEMCACHED_URL']
else
  config.cache_store = :memory_store
end
# ...
```

Create a container for memcached:

```yaml
containers:
  # ...
  memcached:
    setup:
      - !Alpine v3.5
      - !Install memcached
```

Create the command to run with caching:

```bash
$ vagga _run rails rails g scaffold article title:string:index body:text
```
commands:
# ...
run-cached: !Supervise
  description: Start the rails development server alongside memcached
  children:
    cache: !Command
      container: memcached
      run: memcached -u memcached -vv
    app: !Command
      container: rails
      environ:
        MEMCACHED_URL: memcached://127.0.0.1:11211
      run: |
        if [ ! -f 'tmp/caching-dev.txt' ]; then
          touch tmp/caching-dev.txt
        fi
        rails server

• – run memcached as verbose so we see can see the cache working
• – set the cache url
• – creating this file will tell rails to activate cache in development

Now let’s change some of our views to use caching:

<!-- app/views/articles/show.html.erb -->
<%# ... %>
<% cache @article do %>
  <p>
    <strong>Title:</strong><%= @article.title %>
  </p>
  <p>
    <strong>Body:</strong><%= @article.body %>
  </p>
<% end %>
<%# ... %>

<!-- app/views/articles/index.html.erb -->
<%# ... %>
<table>
  <tbody>
    <%= @articles.each do |article| %>
    <%= cache article do %>
      <tr>
        <td><%= article.title %></td>
        <td><%= article.body %></td>
        <td><%= link_to 'Show', article %></td>
        <td><%= link_to 'Edit', edit_article_path(article) %></td>
        <td><%= link_to 'Destroy', article, method: :delete, data: { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
<% end %>

(continues on next page)
Run the project with caching:

```bash
$ vagga run-cached
```

Try adding some records. Keep an eye on the console to see rails talking to memcached.

**We should try Postgres too**

We can test our project against a Postgres database, which is probably what we will use in production.

First, add gem `pg` to our Gemfile

```erb
gem 'pg'
```

Then add the system dependencies for gem `pg`

```yaml
containers:
  base:
    setup:
      - !Ubuntu xenial
      - !UbuntuUniverse
      - !Install
        - zlib1g
        - libsqlite3-0
        - nodejs
        - libpq5
      - !BuildDeps
        - zlib1g-dev
        - libsqlite3-dev
        - libpq-dev
      - !GemInstall
        - ffi
        - nokogiri
        - sqlite3
        - pg
  rails:
    setup:
      - !Container base
      - !GemBundle
    environ:
      DATABASE_URL: sqlite3:db/development.sqlite3
```

- runtime dependency
- build dependency

Create the database container

```yaml
containers:
  # ...
  postgres:
    setup:
```

(continues on next page)
- **Ubuntu** xenial
- **EnsureDir** /data
- **Sh** |
  
  addgroup --system --gid 200 postgres
  adduser --uid 200 --system --home /data --no-create-home \
  --shell /bin/bash --group --gecos "PostgreSQL administrator" \
  postgres
- **Install** [postgresql-9.5]

  environ:
  PGDATA: /data
  PG_PORT: 5433
  PG_DB: test
  PG_USER: vagga
  PG_PASSWORD: vagga
  PG_BIN: /usr/lib/postgresql/9.5/bin

  volumes:
  /data: **Persistent**
   name: postgres
   owner-uid: 200
   owner-gid: 200
   init-command: _pg-init
  /run: **Tmpfs**
   subdirs:
     postgresql: { mode: 0o777 }

  • Use fixed user id and group id for postgres
  • Vagga command to initialize the volume

**Note:** The database will be persisted in .vagga/.volumes/postgres.

Now add the command to initialize the database:

```bash
commands:
  # ...
  _pg-init: **Command**
   description: Init postgres database
   container: postgres
   user-id: 200
   group-id: 200
   run: |
     set -ex
     ls -la /data
     $PG_BIN/pg_ctl initdb
     $PG_BIN/pg_ctl -w -o '-F --port=$PG_PORT -k /tmp' start
     $PG_BIN/createuser -h 127.0.0.1 -p $PG_PORT $PG_USER
     $PG_BIN/createdb -h 127.0.0.1 -p $PG_PORT $PG_DB -u $PG_USER
     $PG_BIN/psql -h 127.0.0.1 -p $PG_PORT -c "ALTER ROLE $PG_USER WITH ENCRYPTED PASSWORD '$PG_PASSWORD';"
     $PG_BIN/pg_ctl stop
```

And then add the command to run with Postgres:

```bash
commands:
  # ...
```

(continues on next page)
run-postgres: **Supervise**

**description:** Start the rails development server using Postgres database

**children:**

**app:**

**Command**

- container: rails
- environ:
  - DATABASE_URL: postgresql://vagga:vagga@127.0.0.1:5433/test
  - run: |
    rake db:migrate
    rails server

**db:**

**Command**

- container: postgres
- user-id: 200
- group-id: 200
- run: exec $PG_BIN/postgres -F --port=$PG_PORT

Now run:

```
$ vagga run-postgres
```

We can also add some default records to the database, so we don’t start with an empty database. To do so, add the following to `db/seeds.rb`:

```ruby
# db/seeds.rb
if Article.count == 0
  Article.create(
    [ { title: 'Article 1', body: 'Lorem ipsum dolor sit amet' },
      { title: 'Article 2', body: 'Lorem ipsum dolor sit amet' },
      { title: 'Article 3', body: 'Lorem ipsum dolor sit amet' }
    ]
  )
end
```

Now change the `run-postgres` command to seed the database:

```ruby
commands:
# ...
run-postgres: **Supervise**
**description:** Start the rails development server using Postgres database

**children:**

**app:**

**Command**

- container: rails
- environ:
  - DATABASE_URL: postgresql://vagga:vagga@127.0.0.1:5433/test
  - run: |
    rake db:migrate
    rake db:seed
    rails server

**db:**

**Command**

- # ...

  • – populate the database.

Now, we run `run-postgres`, we will already have our database populated.
Running on Alpine Linux

Alpine is a distribution focused on containers, being able to produce smaller smaller images than other distributions.

To run our project on Alpine, we just need two more containers:

```text
containers:
  # ...
  base-alpine:
    setup:
      - !Alpine v3.5
      - !Install
        - zlib
        - sqlite-libs
        - nodejs
        - libpq
        - tzdata
        - ruby-bigdecimal
        - ruby-json
      - !BuildDeps
        - zlib-dev
        - sqlite-dev
        - postgresql-dev
        - libffi-dev
      - !GemInstall
        - ffi
        - nokogiri
        - sqlite3
        - pg

  rails-alpine:
    setup:
      - !Container base-alpine
      - !GemBundle
    environ:
      DATABASE_URL: sqlite3:db/development.sqlite3
```

- Rails needs these packages to work properly on Alpine. The packages `ruby-bigdecimal` and `ruby-json` could be added in Gemfile as well.

With our containers set up, we just need a command:

```text
commands:
  # ...
  run-alpine: !Command
    container: rails-alpine
    description: Start the rails development server on Alpine container
    run: rails server
```

The command is almost identical to the first `run`, with the only difference being the container used.

### 2.8.2 Examples 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`. 
Databases

PostgreSQL

Here is one example of running postgres.

```bash
# Sample Vagga configuration for running PostgreSQL server
#
containers:
  ubuntu:
    setup:
    - !Ubuntu xenial
      # Use fixed user id and group id for postgres, because in some cases
      # we may need to upgrade (rebuild) a postgres container, but keep the data
      # on a `!Persistent` volume still usable. User ids in ubuntu packages are
      # not guaranteed to be same on every installation.
      #
      # The command-lines are from the postgres-common package except
      # added --uid 200 --gid 200
      - !Sh |
        addgroup --system --gid 200 postgres
        adduser --uid 200 --system --home /data --no-create-home \
          --shell /bin/bash --group --gecos "PostgreSQL administrator" \
          postgres
    - !Install [postgresql-9.5]
    - !EnsureDir /data
    environ:
      PG_PORT: 5433  # Port of host to use
      PG_DB: vagga
      PGDATA: /data
      PG_BIN: /usr/lib/postgresql/9.5/bin
    volumes:
      /data: !Persistent
        name: postgres
        owner-uid: 200
        owner-gid: 200
        init-command: _pg-init
    /run: !Tmpfs
      subdirs:
        postgresql: { mode: 0o777 }  # until we have user, group options
    commands:
      _pg-init: !Command
      description: Init postgres database
      container: ubuntu
      user-id: 200
      group-id: 200
      run: |
        set -ex
        ls -la /data
        $PG_BIN/pg_ctl initdb
        $PG_BIN/pg_ctl -w -o '-F --port=$PG_PORT -k /tmp' start
        $PG_BIN/createuser -h 127.0.0.1 -p $PG_PORT vagga
        $PG_BIN/createdb -h 127.0.0.1 -p $PG_PORT $PG_DB -O vagga
```

(continues on next page)
# init schema usually schema shouldn't be inline here, but contained
# in a separate file

```sql
psql postgres://vagga:vagga@127.0.0.1:$PG_PORT/$PG_DB <<ENDSQL
CREATE TABLE random_stuff (x TEXT);
ENDSQL

$PG_BIN/pg_ctl stop
```

**postgres**  
*Command*  
description: Run postgres database  
container: ubuntu  
user-id: 200  
group-id: 200  
run: |
  trap "$PG_BIN/pg_ctl -w stop; trap - INT; kill -INT $$" INT
  $PG_BIN/pg_ctl -w -o '-F --port=$PG_PORT -k /tmp' start
  sleep infinity

**psql**  
*Command*  
description: Run postgres shell  
container: ubuntu  
run: |
  psql -U vagga postgres://$PG_USER:$PG_PASSWORD@127.0.0.1:$PG_PORT/$PG_DB

run:  
*Supervise*  
description: Run both postgres and shell  
children:  
  postgres:  
  psql: 

There is a more complicated example of postgres with alembic migrations

**Redis**

Simplest container with redis looks like this:

```yaml
containers:
  redis:
    setup:
    - !Alpine v3.5
    - !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:
containers:
  ubuntu:
    setup:
      - !Ubuntu xenial
      - !UbuntuUniverse
      - !Sh apt-key adv --keyserver keyserver.ubuntu.com --recv-keys C7917B12
      - !UbuntuRepo
        url: http://ppa.launchpad.net/chris-lea/redis-server/ubuntu
        suite: xenial
        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 \n      --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 \n      --appendonly no

meet-nodes: !Command

(continues on next page)
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;

Consul

containers:

  ubuntu-consul:
    setup:
    - !Ubuntu xenial
    - !Install [unzip, wget, ca-certificates]
    - !Sh |
      cd /tmp
      wget https://releases.hashicorp.com/consul/0.6.4/consul_0.6.4_linux_amd64.zip
      unzip consul_0.6.4_linux_amd64.zip
      cp consul /usr/bin/consul

commands:

  consul-server: !Command
description: Start consul in server mode
container: ubuntu-consul
run: |
  /usr/bin/consul agent -server -bootstrap-expect=1 \
  -data-dir=/tmp/consul -log-level=debug \
  -advertise=127.0.0.1

Elasticsearch

The elasticsearch example uses Persistent volume so works only on development version of vagga:

containers:

  elastic:
    setup:
    - !Ubuntu xenial
    - !UbuntuUniverse
    # elastic PGP & Repo
    - !AptTrust
      server: pgp.mit.edu
      keys: [D88E42B4]
    - !UbuntuRepo
      url: http://packages.elastic.co/elasticsearch/2.x/debian
      suite: stable
      components: [main]
    - !Install
      - ca-certificates
      - ca-certificates-java
      - openjdk-8-jre-headless
      - elasticsearch=2.3.3
- !EnsureDir /var/elastic
  volumes:
  /var/elastic: !Persistent { name: elastic }

commands:
  elastic: !Command
description: Run elasticsearch
container: elastic
user-id: 1
external-user-id: 0
run:
  - /usr/share/elasticsearch/bin/elasticsearch
  - -Des.path.conf=/etc/elasticsearch
  - -Des.path.logs=/tmp
  - -Des.path.work=/tmp
  - -Des.path.data=/var/elastic/data

Influx DB

Influx db is described as a scalable datastore for metrics, events, and real-time analytics

Home / Github

Example config:

containers:
  influxdb:
    setup:
    - !Ubuntu xenial
    - !Download
      url: https://dl.influxdata.com/influxdb/releases/influxdb_0.13.0_amd64.deb
      path: /tmp/influxdb.deb
    - !Sh dpkg -i /tmp/influxdb.deb && rm /tmp/influxdb.deb
    - !EnsureDir /var/lib/influxdb
    volumes:
      /var/lib/influxdb: !Persistent { name: influxdb }

commands:
  influx: !Command
description: Run influxdb
container: influxdb
user-id: 1
external-user-id: 0
run:
  - influxd

RethinkDB

RethinkDB is described as:

RethinkDB is the open-source, scalable database that makes building realtime apps dramatically easier.

Because RethinkDB has an Ubuntu package, it’s easy to setup:
containers:
  ubuntu:
    setup:
    - !Ubuntu xenial
    - !UbuntuRepo
      url: http://download.rethinkdb.com/apt
      suite: xenial
      components: [main]
    - !Download
      url: https://download.rethinkdb.com/apt/pubkey.gpg
      path: /tmp/pubkey.gpg
    - !Sh apt-key add /tmp/pubkey.gpg
    - !Install [rethinkdb]
    - !EnsureDir /data
      volumes:
        /data: !Persistent data
    - !NpmInstall [git://github.com/rethinkdb/rethinkdb-example-nodejs-chat.git]

commands:
  rethink: &rethink !Command
    description: Run rethink database

We also have a configured example chat application in the repository, that you may run with alongside with the database itself as follows:

vagga example-chat

**CockroachDB**

Here is one example of running single node CockroachDB:

```bash
# Sample Vagga configuration for running CockroachDB server
#
containers:
  ubuntu:
    setup:
    - !Ubuntu xenial
    - !Tar
      url: https://binaries.cockroachdb.com/cockroach-v1.0.4.linux-amd64.tgz
      subdir: cockroach-v1.0.4.linux-amd64
      path: /usr/bin
    - !EnsureDir /data
      volumes:
        /data: !Persistent
          name: cockroach

commands:
  server: &server !Command
    description: Run cockroach database
    container: ubuntu
```

(continues on next page)
run: [cockroach, start, --insecure, --store=path=/data]

client: &client !Command
description: Run cockroach shell
container: ubuntu
accepts-arguments: true
run: |
  sleep 1
  exec cockroach sql --insecure "$@

run: !Supervise
description: Run both cockroach server and shell
children:
  server: *server
  client: *client

ClickHouse

ClickHouse is an open-source column-oriented database management system (DBMS) for online analytical processing (OLAP).

Home / Github

containers:
  ubuntu:
    setup:
      - !Ubuntu xenial
      - !AptTrust keys: [E0C56BD4]
      - !UbuntuRepo
        url: https://repo.yandex.ru/clickhouse/xenial
        suite: stable
        components: [main]
      - !Install [tzdata, clickhouse-server-common, clickhouse-client]
      - !EnsureDir /var/lib/clickhouse
        volumes:
          /var/lib/clickhouse: !Persistent
            name: clickhouse-data

commands:
  server: &server !Command
description: Run clickhouse server
container: ubuntu
work-dir: /etc/clickhouse-server
run: clickhouse server

client: &client !Command
description: Run clickhouse client
container: ubuntu
run: sleep 3 && clickhouse-client

run: !Supervise
description: Run clickhouse server and client
children:
  server: *server

(continues on next page)
Kafka

containers:
  kafka:
    volumes:
      /tmp: !Tmpfs
        size: 1G
        subdirs:
          kafka-logs: {}
          zookeeper: {}
    setup:
      - !Ubuntu xenial
      - !UbuntuUniverse
      - !Install
        - openjdk-8-jre-headless
      - !Tar
        url: http://apache.volia.net/kafka/0.11.0.1/kafka_2.12-0.11.0.1.tgz
        sha256: c776f2dbb7f4e1af6b61e32d482b513a1788bf5f39602746b005ab92601a46f2
        path: /opt/kafka
        subdir: kafka_2.12-0.11.0.1
      - !EnsureDir /var/log/kafka
      - !EnsureDir /var/log/zookeeper

kafka-manager:

  # First build can take a long time
  setup:
    - !Ubuntu xenial
    - !Install
      - openjdk-8-jre-headless
    - !AptTrust
      keys: [2EE0EA64E40A89B84B2DF73499E82A75642AC823]
    - !UbuntuRepo
      url: https://dl.bintray.com/sbt/debian
      suite: ""
      components: [ ]
    - !BuildDeps
      - openjdk-8-jdk
      - sbt
      - unzip
    - !Env
      VERSION: 1.3.3.14
    - !Tar
      url: https://github.com/yahoo/kafka-manager/archive/1.3.3.14.tar.gz
      path: /tmp
    - !CacheDirs
      /root/.ivy2/cache: ivy2
    - !Sh |
      cd /tmp/kafka-manager-{$VERSION}
      sbt dist
      unzip /tmp/kafka-manager-{$VERSION}/target/universal/kafka-manager-{$VERSION}.
      → zip -d /opt
      mv /opt/kafka-manager-{$VERSION} /opt/kafka-manager
zk-shell:
  setup:
    - !Alpine v3.6
    - !PipConfig {dependencies: true}
    - !Py2Install [zk-shell]
  environ:
    HOME: /work
    LANG: en_US.UTF-8
    PYTHONIOENCODING: UTF-8

commands:
  kafka: !Supervise
    description: Run kafka with zookeeper
    kill-unresponsive-after: 6
  children:
    zookeeper: !Command
      container: kafka
      volumes:
        /tmp/zookeeper: !Persistent zookeeper
        /var/log/zookeeper: !Tmpfs
      run: |
        export LOG_DIR=/var/log/zookeeper
        cd /opt/kafka
        bin/zookeeper-server-start.sh config/zookeeper.properties &
        PID=$!
        handler() {
          # give kafka some time to gracefully shutdown
          sleep 5
          kill $PID
        }
        trap handler INT
        wait $PID
        echo "Zookeeper stopped"

  kafka: !Command
    container: kafka
    volumes:
      /tmp/kafka-logs: !Persistent kafka
      /var/log/kafka: !Tmpfs
    run: |
      export LOG_DIR=/var/log/kafka
      cd /opt/kafka
      bin/kafka-server-start.sh config/server.properties

  kafka-topics:
    description: Run kafka-topics script
    container: kafka
    run: [/opt/kafka/bin/kafka-topics.sh, --zookeeper, localhost:2181]

  kafka-manager:
    description: Run kafka manager tool
    container: kafka-manager
    environ:
      ZK_HOSTS: localhost:2181
    run:
      - /opt/kafka-manager/bin/kafka-manager
- -Dpidfile.path=/var/run/kafka-manager.pid

zk-shell: !Command
description: Run zk-shell against localhost zookeeper
container: zk-shell
run: [zk-shell, localhost:2181]

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:

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

The vagga configuration for the command:

```yaml
containers:
  travis:
    setup:
      - !Ubuntu xenial
      - !UbuntuUniverse
      - !BuildDeps [autoconf, automake, autotools-dev, autogen, libtool, libltdl-dev]
      - !GemInstall [travis]
    commands:
      travis: !Command
      container: travis
      run: [travis]
      environ: { HOME: /tmp }
```

Selenium Tests

Running selenium with vagga is as easy as anything else.

Setting up the GUI may take some effort because you need a display, but starting PhantomJS as a driver looks like the following:

```yaml
containers:
  selenium:
    setup:
      - !Ubuntu xenial
      - !UbuntuUniverse
      - !Install [libfontconfig1]
      - !Py3Install [selenium, py, pytest]
      # The phantomjs from Ubuntu repository seems to have problems with headless environments, so we fetch the binary provided by the developers
      - !TarInstall
        url: https://bitbucket.org/ariya/phantomjs/downloads/phantomjs-2.1.1-linux-x86_64.tar.bz2
        script: cp bin/phantomjs /usr/local/bin/phantomjs
```

(continues on next page)
commands:
  test: !Command
    description: Run selenium test
    container: selenium
    run: [py.test, test.py]

And the test may look like the following:

```python
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

def test_example():
    driver = webdriver.PhantomJS()
    driver.get("http://vagga.readthedocs.org/")
    assert "Welcome to Vagga" in driver.title
    driver.close()

if __name__ == '__main__':
    test_example()
```

To run the test just type:

```
> vagga test
```

Firebox Browser

To run firefox or any other GUI application there are some extra steps involved to setup a display.

The `/tmp/.X11-unix/` directory should be mounted in the container. This can be accomplished by making it available to vagga under the name X11 by writing the following lines in your `global configuration ~/.vagga.yaml`:

```yaml
external-volumes:
  X11: /tmp/.X11-unix/
```

Next, you can use the following `vagga.yaml` file to setup the actual configuration (we redefine the variable `HOME` because firefox needs to write profile information).

```yaml
containers:
  browser:
    setup:
      - !Ubuntu xenial
      - !UbuntuUniverse
      - !Install [firefox]
    volumes:
      /tmp: !Tmpfs
        size: 100Mi
        mode: 0o1777
        subdirs:
          .X11-unix:
            /tmp/.X11-unix: !BindRW /volumes/X11

commands:
```

(continues on next page)
Note: If Firefox is already running on your host system, it will connect to it to avoid creating another instance and it will use the resources of your host system instead of the container’s.

We pass `--no-remote` to tell it to create a new instance inside the container, to avoid exposing the host file system.

When calling vagga, remember to export the DISPLAY environment variable:

```
vagga -eDISPLAY firefox
```

To prevent DBUS-related errors also export the DBUS_SESSION_BUS_ADDRESS environmental variable:

```
vagga -eDISPLAY -eDBUS_SESSION_BUS_ADDRESS firefox
```

### WebGL Support

To enable WebGL support further steps are necessary to install the drivers inside the container, that depends on your video card model.

To setup the proprietary nvidia drivers, download the driver from the NVIDIA website in the your working directory and use the following `vagga.yaml`:

```
containers:
  browser:
    setup:
      - !Ubuntu xenial
      - !UbuntuUniverse
      - !Install [binutils, pkg-config, mesa-utils]
      - !Sh sh /work/NVIDIA-Linux-x86_64-331.67.run -a -N --ui=none --no-kernel-module
      - !Sh nvidia-xconfig -a --use-display-device=None --enable-all-gpus --virtual=1280x1024
      - !Install [firefox]
    volumes:
      /tmp: !Tmpfs
      size: 100Mi
      mode: 0o1777
      subdirs:
        .X11-unix:
          /tmp/.X11-unix: !BindRW /volumes/X11
    commands:
      firefox: !Command
      container: browser
      environ: { HOME: /tmp }
      run: [firefox, --no-remote]
```

For intel video cards use the following `vagga.yaml` (this includes also chromium and java plugin):

---

144 Chapter 2. Documentation Contents
Adding a Custom Certificate

This is useful if you have self-signed certificates that you use on local or staging or corporate resources.

In Ubuntu it looks like this:

```yaml
containers:
  some-container:
    setup:
    - !Ubuntu xenial
    - !UbuntuUniverse
    - !Install [chromium-browser, firefox, icedtea-plugin, xserver-xorg-video-intel, mesa-utils, libgl1-mesa-dri]
    volumes:
      /tmp: !Tmpfs
      size: 100Mi
      mode: 0o1777
      subdirs:
        .X11-unix:
        /tmp/.X11-unix: !BindRW /volumes/X11

commands:
  firefox: !Command
  container: browser
  environ: { HOME: /tmp }
  run: [firefox, --no-remote]
```

Important thing here is that http://example.com/your_company_root.crt should be either on a HTTP (not encrypted) host or have a certificate signed by a well-known authority (included in ubuntu ca-certificates package).

Network Tolerance Testing (and Nginx)

Somewhat tiny example of the network tolerance testing code is contained in the following example:

```yaml
containers:
  flask:
    setup:
    - !Ubuntu xenial
    - !PipConfig { dependencies: true }
    - !Py3Install [flask]

nginx:
```

(continues on next page)
setup:
  - !Ubuntu xenial
  - !Install [nginx]

test:
  setup:
    - !Alpine v3.5
    - !Install [iptables]
    - !EnsureDir /vagga
  volumes:
    /vagga: !VaggaBin
  environ:
    PATH: /bin:/vagga:/sbin

bench:
  setup:
    - !Alpine v3.5
    - !Repo edge/testing
    - !Install [wrk]

commands:
  run-normal: !Supervise
  description: Just run flask behind an nginx (http://172.23.255.2:8000)
  children:
    nginx: !Command
      container: nginx
      network:
        ip: 172.23.0.1
        ports: {8000: 8000}
      run: [nginx, -c, "/work/nginx.conf"]
    flask: !Command
      container: flask
      network:
        ip: 172.23.0.2
        ports: {5000: 5000}
      run: "python3 app.py"

  run-flaky: !Supervise
  description: |
      Just run flask behind nginx with network that doesn't work 50% of the time (http://172.23.255.2:8000)
  children:
    nginx: !Command
      container: nginx
      network:
        ip: 172.23.0.1
        ports: {8000: 8000}
      run: [nginx, -c, "/work/nginx.conf"]
    flask: !Command
      container: flask
      network:
        ip: 172.23.0.2
        ports: {5000: 5000}
      run: "python3 app.py"

interrupt: !BridgeCommand
  container: test
run: |
set -x
while true; do
  vagga _network isolate flask
  sleep 1
  vagga _network fullmesh
  sleep 1
done

wrk: !Command
  description: Run wrk (should try against running server)
  container: bench
  run: [wrk]

This example also includes almost a smallest possible nginx configuration:

daemon off;
master_process off;
worker_processes 1;
user root;

error_log stderr;
#pid /tmp/nginx.pid;

events {
  worker_connections 1024;
}

http {
  include /etc/nginx/mime.types;
  default_type application/octet-stream;
  access_log off;

  client_body_temp_path /tmp 1 2;
  proxy_temp_path /tmp 1 2;
  fastcgi_temp_path /tmp 1 2;
  uwsgi_temp_path /tmp 1 2;
  scgi_temp_path /tmp 1 2;

  sendfile on;

  keepalive_timeout 65;

  server {
    listen 8000;
    large_client_header_buffers 4 64k;

    charset utf-8;

    location / {
      proxy_pass http://172.18.0.2:5000;
    }
  }
}
Note: The nginx spits the following message just after start:

```bash
nginx: [alert] could not open error log file: open() "/var/log/nginx/error.log" failed (30: Read-only file system)
```

It’s fine, we can’t change this directory as it’s hardcoded into the source. While we can mount $Tmpfs volume into /var/log/nginx we don’t have to, as all other messages are actually logged into the stderr as configured. So this is just annoying and useless warning that is safe to ignore.

**Graphite**

```yaml
_defaults:
  carbon-volumes: &carbon-volumes
    /var/lib/carbon: !Persistent
      name: carbon
      init-command: _init-carbon-storage
      /var/log/carbon: !Tmpfs
        subdirs:
          webapp: {}
  grafana-volumes: &grafana-volumes
    /var/lib/grafana: !Persistent
      grafana

containers:
  graphite:
    environ:
      PYTHONPATH: "/opt/graphite/lib/:/opt/graphite/webapp/"
      GRAPHITE_STORAGE_DIR: /var/lib/carbon
    setup:
      - !Ubuntu xenial
        # Install carbon & graphite
      - !Install
        - libcairo2-dev
        - libffi-dev
        - libpython-dev
      - !PipConfig { dependencies: true }
      - !Py2Install
        - setuptools
      - !Env
        PIP_NO_BINARY: "':all:'"
      - !Py2Install
        - carbon
        - whisper
        - graphite-web
        - uwsgi
      - !Sh
        cd /opt/graphite/conf
        cp carbon.conf.example carbon.conf
        cp storage-schemas.conf.example storage-schemas.conf
        cp storage-aggregation.conf.example storage-aggregation.conf
      - !Text
        /opt/graphite/webapp/grafana/local_settings.py: |
        LOG_DIR = '/var/log/carbon/webapp'
      - !EnsureDir /var/lib/carbon
      - !EnsureDir /var/log/carbon
```

(continues on next page)
# Install grafana

- !AptTrust
  keys: [418A7F2FB0E1E6E7EABF6FE8C2E73424D59097AB]
- !UbuntuRepo
  url: https://packagecloud.io/grafana/stable/debian
  suite: jessie
  components: [main]
- !Install [grafana]

commands:

__init-carbon-storage__: !Command
  container: graphite
  volumes: *carbon-volumes
  run: |
    django-admin.py migrate --run-syncdb --settings=graphite.settings

grafana: !Supervise
  description: Run carbon, graphite and grafana
  options: |
    Usage: vagga grafana [options] [--graphite-port=<graphite_port>] [--grafana-port=<grafana_port>]

      Options:
    --graphite-port=<graphite_port> Graphite port to listen on
    --grafana-port=<grafana_port> Grafana port to listen on

children:

carbon: &carbon !Command
  container: graphite
  volumes: *carbon-volumes
  run: |
    /opt/graphite/bin/carbon-cache.py 
      --nodaemon 
      --logdir=/var/log/carbon 
      --pidfile=/var/run/carbon.pid 
      start

graphite: !Command
  container: graphite
  volumes: *carbon-volumes
  run: |
    uwsgi --http localhost:${VAGGAOPT_GRAPHITE_PORT:-8080} 
      --wsgi-file /opt/graphite/webapp/graphite/wsgi.py 
      --static-map /static=/opt/graphite/webapp/content

grafana: !Command
  container: graphite
  volumes: *grafana-volumes
  run: |
    grafana-server 
      --homepath /usr/share/grafana 
      --config /etc/grafana/grafana.ini 
      --pidfile /var/run/grafana.pid 
    cfg:default.paths.data=/var/lib/grafana 
    cfg:default.paths.logs=/var/log/grafana 
    cfg:default.server.http_port=${VAGGAOPT_GRAFANA_PORT:-3000}
Build tools

Gradle

```
_build: 
gradle-home: &gradle-home /work/.gradle 
 cmd-opts: &cmd-opts 
  container: jdk 
  # Some java applications don't allow to run from root e.g. Elasticsearch. 
  # Uncomment next rows to circumvent such software 
  # user-id: 1 
  # external-user-id: 0 
  prerequisites: [_mk-gradle-cache] 
  volumes: 
  /work/.gradle/caches: !CacheDir gradle-cache 

containers: 
  jdk: 
    environ: 
    JAVA_HOME: /usr/lib/jvm/java-8-openjdk-amd64 
    GRADLE_USER_HOME: *gradle-home 
    setup: 
    - !Ubuntu xenial 
    - !Install 
      - openjdk-8-jdk-headless 
    - !EnsureDir /opt/gradle 
    - !Unzip 
      url: https://services.gradle.org/distributions/gradle-4.2.1-bin.zip 
      subdir: gradle-4.2.1 
      path: /opt/gradle 
    - !Sh ln -s /opt/gradle/bin/gradle /usr/local/bin/gradle 
    # Uncomment if you need kotlin compiler 
    # - !EnsureDir /opt/kotlinc 
    # - !Unzip 
    # url: https://github.com/JetBrains/kotlin/releases/download/v1.1.51/kotlin-compiler-1.1.51.zip 
    # subdir: kotlinc 
    # path: /opt/kotlinc 
    # - !Sh ln -s /opt/kotlinc/bin/kotlinc /usr/local/bin/kotlinc

commands: 
  _mk-gradle-cache: !Command 
  container: jdk 
  run: mkdir -p $GRADLE_USER_HOME/caches

gradle: !Command 
  <<: *cmd-opts 
  description: Runs arbitrary gradle task 
  run: [gradle]

daemon: !Command 
  <<: *cmd-opts 
  description: Runs gradle daemon 
  run: 
    # --foreground option runs a daemon with incompatible jvm options 
    # so another gradle process rejects to connect to the daemon 
    gradle --dry-run --daemon
```

(continues on next page)
GRADLE_PID=$(pidof java)
echo "Gradle daemon successfully started [PID: $GRADLE_PID]."
while [ -e /proc/$GRADLE_PID ]; do sleep 1; done

run: !Command
  <<: *cmd-opts
  description: Runs this project as a JVM application
  run: [gradle, run]

assemble: !Command
  <<: *cmd-opts
  description: Assembles the outputs of this project
  run: [gradle, assemble]

build: !Command
  <<: *cmd-opts
  description: Assembles and tests this project
  run: [gradle, assemble]

test: !Command
  <<: *cmd-opts
  description: Runs the unit tests
  run: [gradle, test]

check: !Command
  <<: *cmd-opts
  description: Runs all checks
  run: [gradle, test]

compile: !Command
  <<: *cmd-opts
  description: Compiles all source sets
  run: [gradle, classes, -x, processResources]

compile-test: !Command
  <<: *cmd-opts
  description: Compiles all test source sets
  run: [gradle, testClasses, -x, processTestResources]

doc: !Command
  <<: *cmd-opts
  description: Generates Javadoc API documentation for the main source code
  run: [gradle, javadoc]

clean: !Command
  <<: *cmd-opts
  description: Deletes the build directory
  run: [gradle, clean]

Documentation

Sphinx Documentation

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

doc:
  setup:
  - !Alpine v3.6
  - !Install [alpine-base, py-sphinx, make]
  # If you require additional packages to build docs uncomment this
  # - !Py3Requirements doc/requirements.txt

commands:

doc: !Command
description: Build documentation
container: doc
run: [make, html]
work-dir: doc
epilog: |
  --------------------------------------------------------
  Documentation is built under doc/_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 doc sphinx-quickstart ./doc

And add it to the git repository:

echo "/_build" >> doc/.gitignore
git add doc

External Links

- A collection of examples from Andrea Ferretti. Includes nim, ocaml, scala and more.

2.8.3 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

• **xRandom** a web project described as “Site that allows you see adult movie free without advertisements”. Vagga config features:
  – Installation of elasticsearch (which is also an example to setup DB)
  – The full web server stack run with single command (nginx + nodejs)
  – The hard way of setting up the same thing for comparison
CHAPTER 3

Indices and tables

- genindex
Credits

Logo created by NineArt.
Symbols

**This functionality is experimental. Some details can change in future.**

- Option, 20

A

- accepts-arguments
  - Option, 28
- aliases
  - Option, 24
- Alpine
  - Build Step, 45
- alpine-mirror
  - Option, 84
- AlpineRepo
  - Build Step, 45
- AptTrust
  - Build Step, 43
- auto-apply-sysctl
  - Option, 83
- auto-clean
  - Option, 23

B

- banner
  - Option, 24
- banner-delay
  - Option, 24
- BindRO
  - Volume Type, 66
- BindRW
  - Volume Type, 66
- Build
  - Build Step, 56
- Build Step
  - Alpine, 45
  - AlpineRepo, 45
  - AptTrust, 43
  - Build, 56
  - BuildDeps, 46

- CacheDirs, 53
- Cmd, 47
- ComposerConfig, 63
- ComposerDependencies, 63
- ComposerInstall, 63
- Container, 54
- Copy, 51
- Depends, 54
- Download, 47
- EmptyDir, 53
- EnsureDir, 53
- Env, 53
- GemBundle, 64
- GemConfig, 65
- GemInstall, 64
- Git, 49
- GitDescribe, 50
- GitInstall, 49
- Install, 46
- NpmConfig, 59
- NpmDependencies, 58
- NpmInstall, 58
- PipConfig, 61
- Py2Install, 61
- Py2Requirements, 62
- Py3Install, 62
- Py3Requirements, 62
- Remove, 53
- Repo, 45
- RunAs, 47
- Sh, 46
- SubConfig, 55
- Tar, 48
- TarInstall, 48
- Text, 51
- Ubuntu, 42
- UbuntuPPA, 44
- UbuntuRelease, 42
- UbuntuRepo, 44
- UbuntuUniverse, 44
Unzip, 49
YarnDependencies, 60
build-lock-wait
Option, 84
BuildDeps
Build Step, 46

C

cache-dir
Option, 82
CacheDir
Volume Type, 66
CacheDirs
Build Step, 53
children
Option, 30
Cmd
Build Step, 47
Command
doc, 97
run, 96
test, 97
test-whatever, 97
ComposerConfig
Build Step, 63
ComposerDependencies
Build Step, 63
ComposerInstall
Build Step, 63
Container
Build Step, 54
Volume Type, 67
container
Option, 28
Copy
Build Step, 51

D
data-dirs
Option, 23
default-shell
Option, 24
Depends
Build Step, 54
description
Option, 24
disable-auto-clean
Option, 85
doc
Command, 97
Download
Build Step, 47

E

Empty
Volume Type, 67
EmptyDir
Build Step, 53
EnsureDir
Build Step, 53
Env
Build Step, 53
environ
Option, 22, 28, 31, 84
environ-file
Option, 22
epilog
Option, 24
expect-inotify-limit
Option, 26
external-user-id
Option, 29
external-volumes
Option, 82

G

GemBundle
Build Step, 64
GemConfig
Build Step, 65
GemInstall
Build Step, 64
gids
Option, 22, 30
Git
Build Step, 49
GitDescribe
Build Step, 50
GitInstall
Build Step, 49
group
Option, 25
group-id
Option, 29

H

hard-link-between-projects
Option, 85
hard-link-identical-files
Option, 85
hosts-file-path
Option, 23

I

image-cache-url
Option, 23
index-all-images
  Option, 84
Install
  Build Step, 46
isolate-network
  Option, 27
K
kill-unresponsive-after
  Option, 30
M
minimum-vagga
  Option, 21
mixins
  Option, 20
mode
  Option, 30
N
NpmConfig
  Build Step, 59
NpmDependencies
  Build Step, 58
NpmInstall
  Build Step, 58
O
Option
 **This functionality is experimental. Somedetails can change infuture.**, 20
accepts-arguments, 28
aliases, 24
alpine-mirror, 84
auto-apply-sysctl, 83
auto-clean, 23
banner, 24
banner-delay, 24
build-lock-wait, 84
cache-dir, 82
children, 30
container, 28
data-dirs, 23
default-shell, 24
description, 24
disable-auto-clean, 85
eviron, 22, 28, 31, 84
eviron-file, 22
epilog, 24
expect-inotify-limit, 26
external-user-id, 29
external-volumes, 82
gids, 22, 30
group, 25
group-id, 29
hard-link-between-projects, 85
hard-link-identical-files, 85
hosts-file-path, 23
image-cache-url, 23
index-all-images, 84
isolate-network, 27
kill-unresponsive-after, 30
minimum-vagga, 21
mixins, 20
mode, 30
options, 26
pass-tcp-socket, 29
pid1mode, 29
prerequisites, 25
propagate-environ, 82
proxy-env-vars, 82
push-image-script, 83
resolv-conf-path, 23
run, 28, 31
run-symlinks-as-commands, 84
setup, 22
site-settings, 82
storage-dir, 81
storage-subdir-from-env-var, 82
symlink-name, 27
tags, 28
ubuntu-mirror, 84
ubuntu-skip-locking, 84
uids, 22, 30
user-id, 29
version-check, 84
versioned-build-dir, 85
volumes, 22, 29
work-dir, 28, 30
write-mode, 29
options
  Option, 26
P
pass-tcp-socket
  Option, 29
Persistent
  Volume Type, 68
pid1mode
  Option, 29
PipConfig
  Build Step, 61
prerequisites
  Option, 25
propagate-environ
  Option, 82
proxy-env-vars
  Option, 82
push-image-script
  Option, 83
Py2Install
  Build Step, 61
Py2Requirements
  Build Step, 62
Py3Install
  Build Step, 62
Py3Requirements
  Build Step, 62
R
Remove
  Build Step, 53
Repo
  Build Step, 45
resolv-conf-path
  Option, 23
run
  Command, 96
  Option, 28, 31
run-symlinks-as-commands
  Option, 84
RunAs
  Build Step, 47
S
setup
  Option, 22
Sh
  Build Step, 46
site-settings
  Option, 82
Snapshot
  Volume Type, 67
storage-dir
  Option, 81
storage-subdir-from-env-var
  Option, 82
SubConfig
  Build Step, 55
supplementary-gids
  Option, 29
symlink-name
  Option, 27
T
tags
  Option, 28
Tar
  Build Step, 48
TarInstall
  Build Step, 48
test
  Command, 97
test-whatever
  Command, 97
Text
  Build Step, 51
Tmpfs
  Volume Type, 65
U
Ubuntu
  Build Step, 42
ubuntu-mirror
  Option, 84
ubuntu-skip-locking
  Option, 84
UbuntuPPA
  Build Step, 44
UbuntuRelease
  Build Step, 42
UbuntuRepo
  Build Step, 44
UbuntuUniverse
  Build Step, 44
uids
  Option, 22, 30
Unzip
  Build Step, 49
user-id
  Option, 29
V
VaggaBin
  Volume Type, 66
version-check
  Option, 84
versioned-build-dir
  Option, 85
Volume Type
  BindRO, 66
  BindRW, 66
  CacheDir, 66
  Container, 67
  Empty, 67
  Persistent, 68
  Snapshot, 67
  Tmpfs, 65
  VaggaBin, 66
volumes
  Option, 22, 29
W
work-dir
Option, 28, 30
write-mode
  Option, 29

Y
YarnDependencies
  Build Step, 60