Vagga Documentation

Release 0.6.1

Paul Colomiets

June 13, 2016

Contents

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.

CHAPTER 1

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

CHAPTER 2

Documentation Contents

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`:

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

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

- – run command in container “flask”
- – the command-line is “python3”

To run command just run `vagga command_name`:

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

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

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

Compare that to vagrant:

```
$ vagrant up
# some machine(s) created
$ vagrant ssh
# now you are in new shell. What to do?
$ make
```

```
# ok probably something is built (if project uses make), what now?
$ less README
# long reading follows
```

Or compare that to docker:

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

Lazy Container Creation

There are few interesting cases where lazy containers help.

Application Requires Multiple Environments

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

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

```
containers:

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

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

One for each command:

```
commands:

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

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

Similarly might be defined test container and command:

```
containers:

  testing:
    setup:
    - !Ubuntu trusty
    - !UbuntuUniverse
```

```
    - !Install [make, nodejs, uglifyjs, python-django, nosetests]

commands:

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

And your user never care how many containers are there. User only runs whatever 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)
    example-plugin  example of plugin for markdown parser
$ vagga example-web
Now go to http://localhost:8000 to see the demo
```

How would you achieve the same with vagrant?

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

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

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

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

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

Docker case is very similar to Vagrant one.

Container Versioning and Rebuilding

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

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

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

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

How you do this with Vagrant:

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

Using Docker? Let's see:

```
$ git pull
$ docker run --rm -it me/somewebapp python manage.py runserver
ImportError
$ docker tag me/somewebapp:latest me/somewebapp:old
$ docker build -t me/somewebapp .
$ docker run --rm -it me/somewebapp python manage.py runserver
# Oh, that was simple
$ git checkout stable
```

```
$ docker run --rm -it me/somewebapp python manage.py runserver
# Oh, crap, I forgot to downgrade container
# We were smart to tag old one, so don't need to rebuild:
$ docker run --rm -it me/somewebapp:old python manage.py runserver
# Let's also rebuild dependencies
$ ./build.sh
Running: docker run --rm me/somewebapp_build python manage.py runserver
# Oh crap, we have hard-coded container name in build script?!?!
```

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

Running Multiple Processes

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

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

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

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

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

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

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

Vagga's way:

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

Now just run:

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

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

Let's see how it's helpful. In example above celery may crash (for example because of misconfiguration, or OOM, or whatever). Usually when running many services you have many-many messages on startup, so you may miss it. Or it may crash later. So you click on some task in web app, and wait when the task is done. After some time, you think that it *may* be too long, and start looking in logs here and there. And after some tinkering around you see that celery is just down. Now, you lost so much time just waiting. Wouldn't it be nice if everything is just crashed and you notice it immediately? Yes it's what `stop-on-failure` does.

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

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 paswordless `sudo`. This is because root user in docker container is same root that one on host. Also user that can start docker container can mount arbitrary folder in host filesystem into the container (So he can just mount `/etc` and change `/etc/passwd`).

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

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 realy needs to be root inside the container (e.g. it must spawn processes by different users) you just can't fix it.

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

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

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 sytem. It also uses chroot and linux namespaces for more isolation. Compare it to docker which doesn't consider running as root inside a container secure.

You can apply selinux or apparmor rules for both.

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 enviroments 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

2.2.1 Binary Installation

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

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

```
$ wget http://files.zerogw.com/vagga/vagga-0.6.1.tar.xz
$ tar -xJf vagga-0.6.1.tar.xz
$ cd vagga
$ sudo ./install.sh
```

Or you may try more obscure way:

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

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

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

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

2.2.2 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 privlege. This is increasingly available in modern distributions but may need to be enabled manually.

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

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

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

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

  or you may get:

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

  Either one is a valid outcome.

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

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

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

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

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

See instructions specific for your distribution below.

2.2.3 Ubuntu

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

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

The process of installation looks like the following:

```
$ echo 'deb http://ubuntu.zerogw.com vagga main' | sudo tee /etc/apt/sources.list.d/vagga.list
deb http://ubuntu.zerogw.com vagga main
$ sudo apt-get update
[.. snip ..]
Get:10 http://ubuntu.zerogw.com vagga/main amd64 Packages [365 B]
[.. snip ..]
Fetched 9,215 kB in 17s (532 kB/s)
Reading package lists... Done
$ sudo apt-get install vagga
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  vagga
0 upgraded, 1 newly installed, 0 to remove and 113 not upgraded.
Need to get 873 kB of archives.
After this operation, 4,415 kB of additional disk space will be used.
WARNING: The following packages cannot be authenticated!
  vagga
Install these packages without verification? [y/N] y
Get:1 http://ubuntu.zerogw.com/ vagga/main vagga amd64 0.1.0-2-g8b8c454-1 [873 kB]
Fetched 873 kB in 2s (343 kB/s)
Selecting previously unselected package vagga.
(Reading database ... 60919 files and directories currently installed.)
Preparing to unpack .../vagga_0.1.0-2-g8b8c454-1_amd64.deb ...
Unpacking vagga (0.1.0-2-g8b8c454-1) ...
Setting up vagga (0.1.0-2-g8b8c454-1) ...
```

Now vagga is ready to go.

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

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

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

2.2.4 Ubuntu: Old Releases (precise, 12.04)

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

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

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

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

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

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

Or alternatively you may edit files by hand.

Now your vagga is ready to go.

2.2.5 Arch Linux

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

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

You may use binary package from authors of vagga, by adding the following to `/etc/pacman.conf`:

```
[linux-user-ns]
SigLevel = Never
Server = http://files.zerogw.com/arch-kernel/$arch
```

Note: alternatively you may use a package from AUR:

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

Package is based on `core/linux` package and differ only with `CONFIG_USER_NS` option. After it's compiled, update your bootloader config, for GRUB it's probably:

```
grub-mkconfig -o /boot/grub/grub.cfg
```

Warning: After installing a custom kernel you need to rebuild all the custom kernel modules. This is usually achieved by installing `*-dkms` variant of the package and `systemctl enable dkms`. More about DKMS is in Arch Linux wiki.

Then **reboot your machine** and choose `linux-user-ns-enabled` kernel at grub prompt. After boot, check it with `uname -a` (you should have text `linux-user-ns-enabled` in the output).

Note: TODO how to make it default boot option in grub?

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
```

2.2.6 Building From Source

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

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

Alternatively you can run `vagga cached-make` instead of `vagga make`. This downloads pre-built image that we use to run in Travis CI. This may be changed in future.

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

To install run:

```
$ make install
```

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

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

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

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

2.3 Configuration

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

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 additional top-level option:

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

```
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:

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

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

Commands

Example of command defined:

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

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

```
$ vagga build-docs html
```

Builds html docs using sphinx inside a container.

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

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 `/ets/hosts` from host. If the value is `null`, no file is copied. Default is `/etc/hosts`. The setting intention is very similar to *`resolv-conf-path`*, so 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 expreminental. This may be useful for keeping image cache on network file system, presumably on non-public projects.

2.3.3 Commands

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

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

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.

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

  Example:

```
    commands:
      make:
        container: build
        run: "make prog"
      run:
        container: build
```

```
    prerequisites: [make]
    run: "./prog"
```

The sequence of running of command with `prerequesites` 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.

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:

```
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.

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 considered 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 ok for most regular processes. See *What's Special With Pid 1?* for more info.

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

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

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

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

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

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

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
  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.

2.3.4 Container Building Guide

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

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

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

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`:

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

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

```
setup:
- !Ubuntu trusty
- !Sh |
   wget somepackage.tar.gz
   tar -xzf somepackage.tar.gz
   cd somepackage
   make && make install
```

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

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

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

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

```
containers:

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

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

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

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

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

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

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

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

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

Sometimes you want to rebuild container when some file changes. For example if you have used the file in the build. There is a `!Depends` command which does nothing per se, but add a dependency. The path must be relative to your project directory (the dir where `vagga.yaml` is). For example:

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

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

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

Only `url` field is mandatory. 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: http://static.rust-lang.org/dist/rust-0.12.0-x86_64-unknown-linux-gnu.tar.gz
  sha256: abcd1234...
```

```
  subdir: rust-0.12.0-x86_64-unknown-linux-gnu
  script: ./install.sh --prefix=/usr
```

Only the `url` is mandatory here too. 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" }
```

Mutliple 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:

```
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:

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

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

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

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

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

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

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

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

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

This effectively adds 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 v3.1
```

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

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

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

Npm Installer

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

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

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

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

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

The following `npm` features are supported:

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

Other forms may work, but are unsupported for now.

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

Python Installer

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

```
setup:
- !Ubuntu trusty
- !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-ceritificates` manually before)

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

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

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

---

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

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

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

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

The `Py2Requirements` command exists too.

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

PHP/Composer Installer

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

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

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

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

```
setup:
- !Ubuntu trusty
- !ComposerDependencies
  working_dir: src # run command inside src directory
  dev: false # do not install dev dependencies
  optimize_autoloader: true
```

If you want to use `hhvm`, you can disable the installation of the `php` runtime:

```
setup:
- !Ubuntu trusty
- !ComposerConfig
  install_runtime: false
  runtime_exe: hhvm
```

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

```
setup:
- !Ubuntu trusty
- !UbuntuUniverse
- !AptTrust keys: ["hhvm apt key here"]
- !UbuntuRepo
  url: http://dl.hhvm.com/ubuntu
  suite: trusty
  components: [main]
- !Install [hhvm]
- !ComposerConfig
  install_runtime: false
  runtime_exe: hhvm
- !Sh echo '.:/usr/local/lib/composer' >> /etc/hhvm/php.ini
```

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 is recently added to the 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:

```
setup:
- !Alpine v3.3
- !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)

```
setup:
- !Alpine v3.3
- !Install [libxml2, libxslt, zlib, sqlite-libs]
- !BuildDeps [libxml2-dev, libxslt-dev, zlib-dev, sqlite-dev]
- !Env
  NOKOGIRI_USE_SYSTEM_LIBRARIES: 1
  HOME: /tmp
- !GemInstall [rails]
- !Sh rails new . --skip-bundle
```

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

```
setup:
- !Alpine v3.3
- !GemBundle
```

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

```
setup:
- !Alpine v3.3
- !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:

```
setup:
- !Alpine v3.3
- !GemSettings
  install_ruby: false
  gem_exe: /usr/bin/gem
```

Warning: Ruby support is recently added to the 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:

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

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

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

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

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

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

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

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

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

somecontainer:
  setup:
  - !SubConfig
```

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

Few comments:

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

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

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

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

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

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

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

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:

```
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 shich 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 usuallly ending with *root.tar.gz*) not *.iso* image.

Example: `http://cloud-images.ubuntu.com/xenial/current/xenial-server-cloudimg-amd64-r`

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

Deprecated. Supported versions: `12.04`, `14.04`, `14.10`, `15.10`, `16.04`. Other version will not work. 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 `keyserver.ubuntu.com`, but you can specify alternative:

```
- !AptTrust
  server: hkp://pgp.mit.edu
  keys: 1572C52609D
```

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 `keyserver.ubuntu.com`) Server to fetch keys from. May be a hostname or `hkp://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: trusty
  components: [main]
- !Install [python3.5]
```

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

Options:

url Url to the repository. **Required**.

suite Suite of the repository. The common practice is that the suite is named just like the codename of the ubuntu release. For example `trusty`. **Required**.

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**.

UbuntuPPA

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

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

Is equivalent to:

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

UbuntuUniverse

The singleton step. Just enables an "universe" repository:

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

Alpine Commands

Alpine

```
setup:
- !Alpine v3.2
```

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*)

Install

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

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

```
setup:
- !Alpine v3.2
- !Sh |
   if [ ! -z "$(which apk)" ] && [ ! -z "$(which 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 trusty
- !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:

```
- !Ubuntu trusty
- !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.

Download
Downloads file and puts it somewhere in the file system.

Example:

```
- !Download
  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

> **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:

```
    - !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 Subdirectory inside the archive to extract. May be `.` to extract the root of the archive.

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:

```
    setup:
    - !TarInstall
      url: http://static.rust-lang.org/dist/rust-1.4.0-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 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.

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 hadling and display. Also this means that dash/bash-compatible shell should be installed in the previous steps under path `/bin/sh`.

`Git`

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

Example:

```
    setup:
    - !Alpine v3.1
    - !Install [python]
    - !Git
      url: git://github.com/tailhook/injections
      path: /usr/lib/python3.5/site-packages/injections
```

(the example above is actually a bad idea, many python packages will work just from source dir, but you may get improvements at least by precompiling `*.pyc` files, see *GitInstall*)

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.

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:

```
setup:
- !Alpine v3.1
- !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 check out. 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*

Files and Directories

Text

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

Example:

```
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:

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

For directories you might also specify regular expression to ignore:

```
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.

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).

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:

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

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

`EnsureDir`

```
setup:
#...
- !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:

```
    - !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 conficts 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).

Sub-Containers

Container
Build a container based on another container:

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

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* fo 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:

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

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

Few comments:

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

•– name of the container used *inside* a "docker.yaml"

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

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

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

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

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

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

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 v3.3
    - !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.

Another motivating example is building a package:

```
containers:
  pkg:
    setup:
    - !Ubuntu trusty
    - !Install [build-essential]
    - !EnsureDir /packages
    - !Sh |
        checkinstall --pkgname=myapp --pakdir=/packages make
    auto-clean: true
  nginx:
    setup:
    - !Ubuntu trusty
    - !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 (absolue path inside the resulting container) to copy (either `path` or `temporary-mount` required)

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.

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 oversion of it:

```
- !NpmConfig
    install_node: false
    npm_exe: /usr/local/bin/npm
- !NpmInstall [webpack]
```

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

All options:

npm-exe (default is `npm`) 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.

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 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.

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:

```
setup:
- !Ubuntu trusty
- !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-ceritificates` 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 trusty
- !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 v3.3
- !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 v3.3
- !Py3Requirements "requirements.txt"
```

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

PHP/Composer Commands

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

ComposerInstall
Example:

```
setup:
- !Alpine v3.3
- !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:

```
- !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:

```
- !ComposerConfig
    install_runtime: false
    runtime_exe: 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 `php`) The command to use for running Composer.

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:
- !Alpine v3.3
- !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:

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

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

Available volume types:

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

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

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

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

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

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

  For example:

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

  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.

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.

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:
  - !Ubuntu trusty
  ...
  deploy-tools:
     setup:
     - !Alpine v3.3
     - !Install [rsync]
     volumes:
         /mnt: !Container app
```

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.

2.3.7 Upgrading

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 ouside 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)
- 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 contianers unless you have specific needs.

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

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

Commands

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

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

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

See *reference* for more info.

Missing Features

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

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

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 three basic modes of operation:

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

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

Stop on Failure

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

Let me show an example:

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

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

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

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

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

Wait

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

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

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

Restart

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

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

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

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 `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 `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 `SIGTERM` or `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 *`pid1mode`*).

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

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

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

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

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

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

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

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 rebuid 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 paremeters 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 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.

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:

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

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

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

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

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

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

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

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

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

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

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

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:

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

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

```
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:

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

To push image using *scp* utility (SFTP protocol):

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

The FTP(s) (for exxample, 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*.

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/`.

`alpine-mirror`
: Set to your preferred alpine mirror. By default it's the random one is picked from the list.

Note: Alpine package manager is used not only for building *`Alpine`* distribution, but also internally for fetching tools that are outside of the container filesystem (for example to fetch `git` for *`Git`* or *`GitInstall`* command(s))

`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.

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:

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

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

```
your_user_name:100000:65536
```

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

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

Can't find newuidmap or newgidmap

Full error usually looks like:

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

There might be two reasons for this:

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

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

The packages for Ubuntu >= 14.04:

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

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

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

Most distributions (known: Nix, Archlinux, Fedora) 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.
```

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.

2.4.5 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 quircks on variuos systems that are needed to enable this feature.

To check this run:

```
$ vagga _check_overayfs_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:

```
3.19.0-42-generic #48~14.04.1-Ubuntu SMP Fri Dec 18 10:24:49 UTC 2015
```

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`:

```
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.

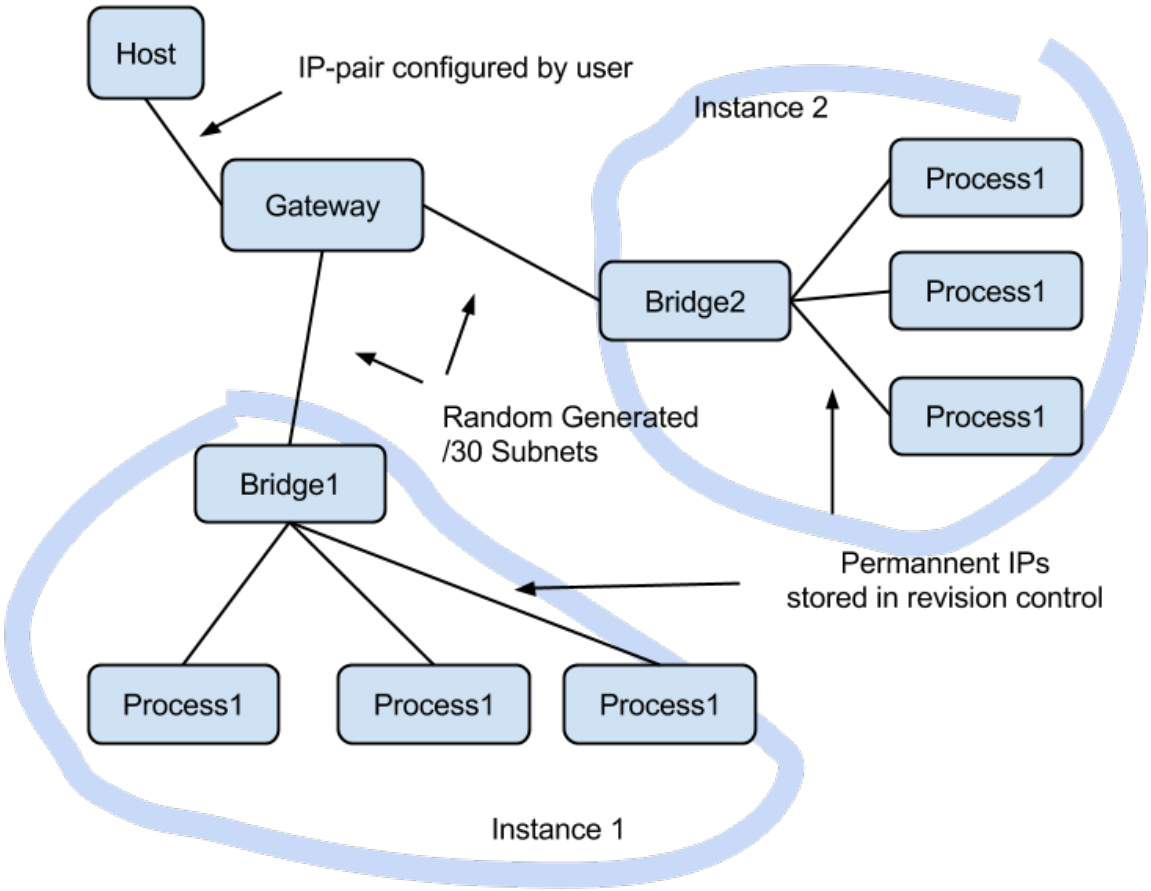
Warning: This documentation is awfully incomplete. There is a good article in the meantime. Except `vagga_network` command is replaced by `vagga _network` subcommand (note the space after `vagga`)

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:
    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/userns'

The following iptables rules will be established:
    ["-I", "INPUT", "-i", "vagga", "-d", "127.0.0.1", "-j", "ACCEPT"]
    ["-t", "nat", "-I", "PREROUTING", "-p", "tcp", "-i", "vagga", "-d", "172.23.255.1", "--dport", "5
    ["-t", "nat", "-I", "PREROUTING", "-p", "udp", "-i", "vagga", "-d", "172.23.255.1", "--dport", "5
    ["-t", "nat", "-A", "POSTROUTING", "-s", "172.23.255.0/30", "-j", "MASQUERADE"]
```

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

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

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

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

To destroy the created network you can run:

```
$ vagga _destroy_netns
```

This uses `sudo` too

Warning: if you have `172.23.0.0/16` 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:

```
$ 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
```

```
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     6.07ms    1.05ms  20.21ms   94.69%
    Req/Sec   827.65     78.83     0.92k    86.00%
  Latency Distribution
     50%    5.82ms
     75%    6.11ms
     90%    6.54ms
     99%   11.62ms
  16485 requests in 10.00s, 2.86MB read
Requests/sec:   1647.73
Transfer/sec:    292.78KB
```

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
- 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 How to Debug Slow Build?

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

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

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

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

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

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

2.6.6 How to Find Out Versions of Installed Packages?

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

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

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

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

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

Or specific version can be looked:

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

The latter form is useful to compare with 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:

```
$ 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

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:

```
$ mkdir -p ~/projects/vagga-django-tutorial && cd ~/projects/vagga-django-tutorial
```

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

```
containers:
  django:
    setup:
    - !Alpine v3.3
    - !Py3Install ['Django >=1.9,<1.10']
```

and then run:

```
$ 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
-- manage.py
-- MyProject
|   -- __init__.py
|   -- settings.py
|   -- urls.py
|   -- wsgi.py
-- vagga.yaml
```

Notice that we used `'Django >=1.9,<1.10'` 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 you project. You can change the Django version if there is a newer version available (`'Django >=1.10,<1.11'` 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.

```
containers:
  app-freezer:
    setup:
    - !Alpine v3.3
    - !Py3Install
      - pip
      - 'Django >=1.9,<1.10'
    - !Sh pip freeze > requirements.txt
  django:
    setup:
    - !Alpine v3.3
    - !Py3Requirements requirements.txt
```

- – our new container
- – we need pip available to freeze dependencies
- – 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==1.9.2
```

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

```
containers:
  # 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 `dj-database-url` to convert the database url into what Django understand.

Add `dj-database-url` to our `app-freezer` container:

```
containers:
  app-freezer:
    setup:
    - !Alpine v3.3
    - !Py3Install
      - pip
      - 'Django >=1.9,<1.10'
      - 'dj-database-url >=0.4,<0.5'
    - !Sh pip freeze > requirements.txt
```

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

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

Set the environment variable:

```
containers:
  #...
  django:
    environ:
      DATABASE_URL: sqlite:///db.sqlite3
    setup:
    - !Alpine v3.3
    - !Py3Requirements requirements.txt
```

- – will point to /work/db.sqlite3

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

```
# MyProject/settings.py
import os
import dj_database_url

# other settings

DATABASES = {
    # will read DATABASE_URL from environment
    'default': dj_database_url.config()
}
```

Let's another shortcut command for `manage.py`:

```
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:

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

After creating the superuser, run our project:

```
$ 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':

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

Add it to `INSTALLED_APPS`:

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

Create a model:

```
# 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:

```
# 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:

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

Run our project:

```
$ vagga run
```

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

Now create a couple views:

```
# blog/views.py
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:

```
{# 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
from django.conf.urls import url
from . import views

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),
]
```

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` and `django-cache-url` to our `app-freezer`, as well as the build dependencies of `pylibmc`:

```
containers:
  app-freezer:
    setup:
    - !Alpine v3.3
    - !BuildDeps
      - libmemcached-dev
      - zlib-dev
    - !Py3Install
      - pip
      - 'Django >=1.9,<1.10'
      - 'dj-database-url >=0.4,<0.5'
      - 'pylibmc >=1.5,<1.6'
```

```
      - 'django-cache-url >=1.0,<1.1'
    - !Sh pip freeze > requirements.txt
```

- – libraries needed to build pylibmc
- – used to configure the cache through an url

And rebuild the container:

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

Add the `pylibmc` runtime dependencies to our `django` container:

```
containers:
  # ...
  django:
    setup:
    - !Alpine v3.3
    - !Install
      - libmemcached
      - zlib
      - libsasl
    - !Py3Requirements requirements.txt
    environ:
      DATABASE_URL: sqlite:///db.sqlite3
```

- – libraries needed by pylibmc at runtime

Crate a new container called `memcached`:

```
containers:
  # ...
  memcached:
    setup:
    - !Alpine v3.3
    - !Install [memcached]
```

Create the command to run with caching:

```
commands:
  # ...
  run-cached: !Supervise
    description: Start the django development server alongside memcached
    children:
      cache: !Command
        container: memcached
        run: memcached -u memcached -vv
      app: !Command
        container: django
        environ:
          CACHE_URL: memcached://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 dj_database_url
```

```
import django_cache_url
# ...
CACHES = {
    # will read CACHE_URL from environment
    'default': django_cache_url.config()
}
```

Configure our view to cache its response:

```
# 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`:

```
containers:
  app-freezer:
    setup:
    - !Alpine v3.3
    - !BuildDeps
      - libmemcached-dev
      - zlib-dev
      - postgresql-dev
    - !Py3Install
      - pip
      - 'Django >=1.9,<1.10'
      - 'dj-database-url >=0.4,<0.5'
      - 'pylibmc >=1.5,<1.6'
      - 'django-cache-url >=1.0,<1.1'
      - 'psycopg2 >=2.6,<2.7'
    - !Sh pip freeze > requirements.txt
```

- – library needed to build psycopg2
- – psycopg2 dependency

Rebuild the container:

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

Add the runtime dependencies of `psycopg2`:

```
containers:
  django:
    setup:
    - !Alpine v3.3
    - !Install
      - libmemcached
      - zlib
      - libsasl
      - 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 crate 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)


class Migration(migrations.Migration):

    dependencies = [
        ('auth', '__latest__')
    ]

    operations = [
        migrations.RunPython(create_superuser)
    ]
```

Create the database container:

```
containers:
  #..
  postgres:
    setup:
    - !Ubuntu trusty
    - !Install [postgresql]
    - !EnsureDir /data
    environ:
      PGDATA: /data
      PG_PORT: 5433
      PG_DB: test
      PG_USER: vagga
      PG_PASSWORD: vagga
      PG_BIN: /usr/lib/postgresql/9.3/bin
    volumes:
      /data: !Tmpfs
        size: 100M
        mode: 0o700
```

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: |
            touch /work/.dbcreation # Create lock file
            while [ -f /work/.dbcreation ]; do sleep 0.2; done # Acquire lock
            python3 manage.py migrate
            python3 manage.py runserver
      db: !Command
        container: postgres
        run: |
            chown postgres:postgres $PGDATA;
            su postgres -c "$PG_BIN/pg_ctl initdb";
            su postgres -c "echo 'host all all all trust' >> $PGDATA/pg_hba.conf"
            su postgres -c "$PG_BIN/pg_ctl -w -o '-F --port=$PG_PORT -k /tmp' start";
            su postgres -c "$PG_BIN/psql -h 127.0.0.1 -p $PG_PORT -c \"CREATE USER $PG_USER WITH PASS
            su postgres -c "$PG_BIN/createdb -h 127.0.0.1 -p $PG_PORT $PG_DB -O $PG_USER";
            rm /work/.dbcreation # Release lock
            sleep infinity
```

Now run:

```
$ vagga run-postgres
```

Visit `localhost:8000/admin` and try to log in with the user and password we defined in the migration.

Making Postgres data persistent It is possible to make the data stored in Postgres persist between runs. To do so, change our `postgres` container as follows:

```
containers:
  postgres:
    setup:
```

```
    - !Ubuntu trusty
    - !Install [postgresql]
    - !EnsureDir /data
    - !EnsureDir /work/.db/data
    environ:
      PGDATA: /data
      PG_PORT: 5433
      PG_DB: test
      PG_USER: vagga
      PG_PASSWORD: vagga
      PG_BIN: /usr/lib/postgresql/9.3/bin
    volumes:
      /data: !BindRW /work/.db/data
```

- – we will persist postgres data in `.db/data`, so ensure it exists
- – bind `/data` to our persistent directory instead of "!Tmpfs"

And also change the `run-postgres` command:

```
commands:
  run-postgres: !Supervise
  description: Start the django development server using Postgres database
  children:
    # ...
    db: !Command
      container: postgres
      run: |
          chown postgres:postgres $PGDATA;
          if [ -z $(ls -A $PGDATA) ]; then
            su postgres -c "$PG_BIN/pg_ctl initdb";
            su postgres -c "echo 'host all all all trust' >> $PGDATA/pg_hba.conf"
            su postgres -c "$PG_BIN/pg_ctl -w -o '-F --port=$PG_PORT -k /tmp' start";
            su postgres -c "$PG_BIN/psql -h 127.0.0.1 -p $PG_PORT -c \"CREATE USER $PG_USER WITH PASS
            su postgres -c "$PG_BIN/createdb -h 127.0.0.1 -p $PG_PORT $PG_DB -O $PG_USER";
          else
            su postgres -c "$PG_BIN/pg_ctl -w -o '-F --port=$PG_PORT -k /tmp' start";
          fi
          rm /work/.dbcreation # Release lock
          sleep infinity
```

- – check if there is already a database created
- – otherwise just start the database

These changes will persist the database files inside `.db/data` on the project directory. We will not have any permission on that directory, so we would not be able to list its contents nor delete it, unless we are root.

Note that if we delete the `.db/data` directory, we will get the error:

```
Can't mount bind "/work/.db/data" to "/vagga/root/data": No such file or directory
```

To solve that, simply recreate `.db/data`.

Building a Laravel project

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

- *Creating the project structure*

- *Setup the database*
- *Adding some code*
- *Trying out memcached*
- *Deploying to a shared server*

Creating the project structure

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:

```
$ mkdir -p ~/projects/vagga-laravel-tutorial && cd ~/projects/vagga-laravel-tutorial
```

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

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

Here we are building a container from Ubuntu and and telling it to install PHP, setup Composer and install the Laravel installer. Now create our new project:

```
$ vagga _run laravel laravel new src
$ mv src/* src/.* .
$ rmdir src
```

We want our project's files to be in the current directory (the one containing `vagga.yaml`) but Laravel installer 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`.

You may see in the console `sh: composer: not found` because Laravel installer is trying to run `composer install`, but don't worry about it, vagga will take care of that for us.

Now that we have our project created, change our container as follows:

```
containers:
  laravel:
    environ: &env
      ENV_CONTAINER: 1
      APP_ENV: development
      APP_DEBUG: true
      APP_KEY: YourRandomGeneratedEncryptionKey
    setup:
    - !Ubuntu trusty
    - !Env { <<: *env }
    - !ComposerDependencies
```

- – tell our application we are running on a container.
- – the "environment" our application will run (development, testing, production).
- – enable debug mode.
- – a random, 32 character string used by encryption service.
- – inherit environment during build.
- – install dependencies from `composer.json`.

Laravel 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.

See that environment variable `ENV_CONTAINER`? With that, our application will be able to tell whether it's running in a container or not. We will need this to require the right `autoload.php` generated by Composer.

> **Warning:** Your composer dependencies will not be installed at the `./vendor` directory. Instead, the are installed globally at `/usr/local/lib/composer/vendor`, so be sure to follow this section to see how to require `autoload.php` from the right location.

THIS IS VERY IMPORTANT!

Now open `bootstrap/autoload.php` and change the line `require __DIR__.'/../vendor/autoload.php';` as follows:

```
<?php
// ...
if (getenv('ENV_CONTAINER')) {
    require '/usr/local/lib/composer/vendor/autoload.php';
} else {
    require __DIR__.'/../vendor/autoload.php';
}
// ...
```

This will enable our project to run either from a container (as we are doing here with vagga) or from a shared server.

> **Note:** If you are deploying your project to production using a container, you can just `require '/usr/local/lib/composer/vendor/autoload.php';` and ignore the environment variable we just set.

To test if everything is ok, let's add a command to run our project:

```
containers:
  # ...
commands:
  run: !Command
    container: laravel
    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 intefering on subsequent runs.

Now run our project:

```
$ vagga run
```

And visit `localhost:8000`. If everithing is OK, you will see Laravel default page saying "Laravel 5".

Setup the database

Every PHP project needs a database, and ours is not different, so let's create a container for our database:

```
containers:
  # ...
  mysql:
```

```
    setup:
    - !Alpine v3.3
    - !Install
      - mariadb
      - mariadb-client
      - php-cli
      - php-pdo_mysql
    - !EnsureDir /data
    - !EnsureDir /opt/adminer
    - !Download
      url: https://www.adminer.org/static/download/4.2.4/adminer-4.2.4-mysql.php
      path: /opt/adminer/index.php
    - !Download
      url: https://raw.githubusercontent.com/vrana/adminer/master/designs/nette/adminer.css
      path: /opt/adminer/adminer.css
    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: !Tmpfs
        size: 200M
        mode: 0o700
```

- – mariadb is a drop in replacement for mysql.
- – we need php to run adminer, a small database administration tool.
- – a better style for adminer.
- – set an yaml anchor so we can reference it in our run command.

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: laravel
        environ: *db_config
        run: |
            touch /work/.dbcreation # Create lock file
            while [ -f /work/.dbcreation ]; do sleep 0.2; done # Acquire lock
            php artisan cache:clear
            php artisan config:clear
            php artisan serve
      db: !Command
        container: mysql
        run: |
            mysql_install_db --datadir=$DB_DATA_DIR
            mkdir /run/mysqld
            mysqld_safe --user=root --datadir=$DB_DATA_DIR \
              --bind-address=$DB_HOST --port=$DB_PORT \
              --no-auto-restart --no-watch
            while [ ! -S /run/mysqld/mysqld.sock ]; do sleep 0.2; done # wait for server to be ready
            mysqladmin create $DB_DATABASE
```

```
            mysql -e "CREATE USER '$DB_USERNAME'@'localhost' IDENTIFIED BY '$DB_PASSWORD';"
            mysql -e "GRANT ALL PRIVILEGES ON $DB_DATABASE.* TO '$DB_USERNAME'@'localhost';"
            mysql -e "FLUSH PRIVILEGES;"
            rm /work/.dbcreation # Release lock
            php -S 127.0.0.1:8800 -t /opt/adminer # run adminer
```

And run our project:

```
$ vagga run
```

To access adminer, visit `localhost:8800`, fill in the `server` field with `127.0.0.1:3307` and the other fields with "vagga" (the username and password we defined).

Adding some code

Now that we have our project working and our database is ready, let's add some.

Note: Let's add a shortcut command for running artisan

```
commands:
  # ...
  artisan: !Command
    description: Shortcut for running php artisan
    container: laravel
    run:
    - php
    - artisan
```

First, 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

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');
```

```
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::drop('articles');
    }
}
```

Open `app/routes.php` and setup routing:

```
<?php
Route::auth();

Route::get('/', 'ArticleController@index');
Route::resource('/article', 'ArticleController');

Route::get('/home', 'HomeController@index');
```

Create our controller:

```
$ 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\Http\Requests;
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('/');
    }

    public function show(Article $article)
    {
        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>
            <a href="{{ url('article/create') }}" class="btn">
                <i class="fa fa-btn fa-plus"></i>New Article
            </a>
            @if (count($articles) > 0)
            <table class="table table-bordered table-striped">
                <thead>
                    <th>id</th>
                    <th>title</a></th>
                    <th>actions</th>
                </thead>
                <tbody>
                    @foreach($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/'.$article->id.'/edit') }}" class="btn btn-prima
                                <i class="fa fa-btn fa-pencil"></i>Edit
                            </a>
                            <form action="{{ url('article/'.$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')
<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <h2>Create Article</h2>
            @include('common.errors')
            <form action="{{ url('article') }}" method="post">
                {!! csrf_field() !!}
                <div class="form-group">
```

```
                    <label for="id-title">Title:</label>
                    <input id="id-title" class="form-control" type="text" name="title" />
                </div>
                <div class="form-group">
                    <label for="id-body">Title:</label>
                    <textarea id="id-body" class="form-control" name="body"></textarea>
                </div>
                <button type="submit" class="btn btn-primary">Save</button>
            </form>
        </div>
    </div>
</div>
@endsection
```

```
<!-- resources/views/article/edit.blade.php -->
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <h2>Edit Article</h2>
            @include('common.errors')
            <form action="{{ url('article/'.$article->id) }}" method="post">
                {!! csrf_field() !!}
                {!! method_field('PUT') !!}
                <div class="form-group">
                    <label for="id-title">Title:</label>
                    <input id="id-title" class="form-control"
                           type="text" name="title" value="{{ $article->title }}" />
                </div>
                <div class="form-group">
                    <label for="id-body">Title:</label>
                    <textarea id="id-body" class="form-control" name="body">{{ $article->body }}</tex
                </div>
                <button type="submit" class="btn btn-primary">Save</button>
            </form>
        </div>
    </div>
</div>
@endsection
```

```
<!-- resources/views/common/errors.blade.php -->
@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

use Illuminate\Database\Seeder;

use App\Article;

class ArticleSeeder extends Seeder
{
    public function run()
    {
        $articles = [
            ['title' => 'Article 1', 'body' => 'Lorem ipsum dolor sit amet'],
            ['title' => 'Article 2', 'body' => 'Lorem ipsum dolor sit amet'],
            ['title' => 'Article 3', 'body' => 'Lorem ipsum dolor sit amet'],
            ['title' => 'Article 4', 'body' => 'Lorem ipsum dolor sit amet'],
            ['title' => 'Article 5', 'body' => 'Lorem ipsum dolor sit amet']
        ];
        foreach ($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
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    public function run()
    {
        $this->call(ArticleSeeder::class);
    }
}
```

Add a the php mysql module to our container:

```
containers:
  laravel:
    environ: &env
      ENV_CONTAINER: 1
      APP_ENV: development
      APP_DEBUG: true
      APP_KEY: YourRandomGeneratedEncryptionKey
    setup:
    - !Ubuntu trusty
    - !Env { <<: *env }
    - !Install
      - php5-mysql
    - !ComposerDependencies
```

Change the `run` command to execute the migrations and seed our database:

```
commands:
  run: !Supervise
```

```
    description: run the laravel development server
    children:
      app: !Command
        container: laravel
        environ: *db_config
        run: |
            touch /work/.dbcreation # Create lock file
            while [ -f /work/.dbcreation ]; do sleep 0.2; done # Acquire lock
            php artisan cache:clear
            php artisan config:clear
            php artisan migrate
            php artisan db:seed
            php artisan serve
      db: !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:8800` to inspect the database.

Trying out memcached

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

Activate Universe repository and add `php5-memcached`, to our container:

```
containers:
  laravel:
    environ: &env
      ENV_CONTAINER: 1
      APP_ENV: development
      APP_DEBUG: true
      APP_KEY: YourRandomGeneratedEncryptionKey
    setup:
    - !Ubuntu trusty
    - !UbuntuUniverse
    - !Env { <<: *env }
    - !Install
      - php5-mysql
      - php5-memcached
    - !ComposerDependencies
```

Create a container for `memcached`:

```
containers:
  # ...
  memcached:
    setup:
    - !Alpine v3.3
    - !Install [memcached]
```

Add some yaml anchors on the `run` command so we can avoid repetition:

```
commands:
  run: !Supervise
    description: run the laravel development server
    children:
      app: !Command
        container: laravel
```

```
        environ: *db_config
        run: &run_app |
            # ...
      db: !Command
        container: mysql
        run: &run_db |
            # ...
```

- – set an anchor at the `app` child command
- – set an anchor at the `db` child command

Create the command to run with caching:

```
commands:
  # ...
  run-cached: !Supervise
    description: Start the laravel development server alongside memcached
    children:
      cache: !Command
        container: memcached
        run: memcached -u memcached -vv
      app: !Command
        container: laravel
        environ:
          <<: *db_config
          CACHE_DRIVER: memcached
          MEMCACHED_HOST: 127.0.0.1
          MEMCACHED_PORT: 11211
        run: *run_app
      db: !Command
        container: mysql
        run: *run_db
```

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

Now let's change our controller to use caching:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;
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', [
            'article' => $article
        ]);
    }

    public function edit($id)
    {
        $article = Cache::rememberForever('article:'.$id, function() use ($id) {
            return Article::find($id);
        });
        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 memcached.

Deploying to a shared server

It's still common to deploy a php application to a shared server running a LAMP stack (Linux, Apache, MySQL and PHP), but our container in its current state isn't compatible with that approach. To solve this, we will create a command to export our project almost ready to be deployed.

Before going to the command part, we will need a new container for this task:

```
containers:
  # ...
  exporter:
    setup:
    - !Ubuntu trusty
    - !Depends composer.json
    - !Depends composer.lock
    - !EnsureDir /usr/local/src/
    - !Copy
      source: /work
      path: /usr/local/src/work
    - !ComposerInstall
    - !Env
      COMPOSER_VENDOR_DIR: /usr/local/src/work/vendor
    - !Sh |
        cd /usr/local/src/work
        rm -f export.tar.gz
        composer install \
          --no-dev --prefer-dist --optimize-autoloader
    volumes:
      /usr/local/src/work: !Snapshot
```

- – rebuild the container if dependencies change.
- – copy our project into a directory inside the container.
- – require Composer to be available.
- – install composer dependencies into the directory we just copied.
- – call `composer` binary directly, because using `!ComposerDependencies` would make vagga try to find `composer.json` before starting the build.
- – create a volume so we can manipulate the files in the copied directory.

Now let's create the command to export our container:

```
commands:
  # ...
  export: !Command
    container: exporter
    description: export project into tarball
    run: |
        cd /usr/local/src/work
        rm -f .env
        rm -f database/database.sqlite
        php artisan cache:clear
        php artisan config:clear
        php artisan route:clear
        php artisan view:clear
        rm storage/framework/sessions/*
        rm -rf tests
        echo APP_ENV=production >> .env
        echo APP_KEY=random >> .env
        php artisan key:generate
        php artisan optimize
        php artisan route:cache
        php artisan config:cache
        php artisan vendor:publish
        tar -czf export.tar.gz .env *
        cp -f export.tar.gz /work/
```

Note: Take this command as a mere example, hence you are encouraged to change it in order to better suit your needs.

The shell in the `export` command will make some cleanup, remove tests (we don't need them in production) and create a minimal .env file with an APP_KEY generated. Then it will compress everything into a file called `export.tar.gz` and copy it to our project directory.

Since the `export` command is quite long, it is a good candidate to be moved to a separate file, for example:

```
commands:
  # ...
  export: !Command
    container: exporter
    description: export project into tarball
    run: [sh, export.sh]
```

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:

```
$ 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:

```
containers:
  rails:
    setup:
    - !Alpine v3.3
    - !Install
      - libxml2
      - libxslt
      - zlib
    - !BuildDeps
      - libxml2-dev
      - libxslt-dev
      - zlib-dev
    - !Env
      NOKOGIRI_USE_SYSTEM_LIBRARIES: 1
    - !GemInstall [rails]
    environ:
      HOME: /tmp
```

- – `rails` depends on nokogiri, which needs these libs during build and runtime.
- – `nokogiri` ships its own versions of `libxml2` and `libxslt` in order to make it easier to build, but here we are instructing it to use the versions provided by Alpine. Refer to nokogiri docs for details.
- – tell `gem` to install `rails`.
- – The `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.

And now run:

```
$ 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`:

```
containers:
  rails:
    setup:
    - !Alpine v3.3
    - !Install
      - libxml2
      - libxslt
      - zlib
      - sqlite-libs
      - nodejs
    - !BuildDeps
      - libxml2-dev
      - libxslt-dev
      - zlib-dev
```

```
      - sqlite-dev
    - !Env
      NOKOGIRI_USE_SYSTEM_LIBRARIES: 1
    - !GemBundle
```

- – we need `sqlite` for the development database and `nodejs` for the asset pipeline (specifically, the `uglifier` gem).
- – install dependencies from `Gemfile` using `bundle install`.

Before we test our project, let's add two gems into the `Gemfile`:

```
# Gemfile
# ...
gem 'bigdecimal'
gem 'tzinfo-data'
# ...
```

Without these two gems, you may run into import errors.

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/databse.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:

```
$ vagga _run rails rails g scaffold article title:string:index body:text
```

Rails scaffolding will generate everything we need, we just have to run the migrations:

```
$ vagga _run rails rake db:migrate
```

Now we need to tell rails to use our articles index page as the root of our project. Change `config/routes.rb` as follows:

```
# config/routes.rb

Rails.application.routes.draw do
  root 'articles#index'
  resources :articles
  # ...
end
```

Run the project now:

```
$ vagga run
```

You should see the articles list page rails generated for us.

Caching with memcached

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

First, add `dalli`, a pure ruby memcached client, to our `Gemfile`:

```
gem 'dalli'
```

Then, open `config/environments/production.rb` and add the following:

```
# config/environments/production.rb
Rails.application.configure do
  # ...
  if ENV['MEMCACHED_URL']
      config.cache_store = :dalli_store, ENV['MEMCACHED_URL']
  end
  # ...
end
```

Create a container for memcached:

```
containers:
  # ...
  memcached:
    setup:
    - !Alpine v3.3
    - !Install [memcached]
```

Create the command to run with caching:

```
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
          RAILS_ENV: production
          SECRET_KEY_BASE: my_secret_key
          RAILS_SERVE_STATIC_FILES: 1
        run: |
            rake assets:precompile
            rails server
```

- – run memcached as verbose so we see can see the cache working
- – set the cache url
- – tell rails to run in production environment
- – production environment requires a secret key
- – tell rails to serve static files on production environment
- – precompile assets

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?' } %>
        </tr>
      <% end %>
```

```
    <% end %>
  </tbody>
</table>
<%# ... %>
```

Run the project with caching:

```
$ 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`

```
gem 'pg'
```

Then add the system dependencies for gem `pg`

```
containers:
  rails:
    setup:
    - !Alpine v3.3
    - !Install
      - libxml2
      - libxslt
      - zlib
      - sqlite-libs
      - libpq
      - nodejs
    - !BuildDeps
      - libxml2-dev
      - libxslt-dev
      - zlib-dev
      - sqlite-dev
      - postgresql-dev
    - !Env
      NOKOGIRI_USE_SYSTEM_LIBRARIES: 1
    - !GemBundle
    environ:
      DATABASE_URL: sqlite3:db/development.sqlite3
```

- – runtime dependency
- – build dependency

Create the database container

```
containers:
  # ...
  postgres:
    setup:
    - !Ubuntu trusty
    - !Install [postgresql]
    - !EnsureDir /data
    environ:
      PGDATA: /data
```

```
      PG_PORT: 5433
      PG_DB: test
      PG_USER: vagga
      PG_PASSWORD: vagga
      PG_BIN: /usr/lib/postgresql/9.3/bin
    volumes:
      /data: !Tmpfs
        size: 100M
        mode: 0o700
```

And then add the command to run with Postgres:

```
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: |
            touch /work/.dbcreation # Create lock file
            while [ -f /work/.dbcreation ]; do sleep 0.2; done # Acquire lock
            rake db:migrate
            rails server
      db: !Command
        container: postgres
        run: |
            chown postgres:postgres $PGDATA;
            su postgres -c "$PG_BIN/pg_ctl initdb";
            su postgres -c "echo 'host all all all trust' >> $PGDATA/pg_hba.conf"
            su postgres -c "$PG_BIN/pg_ctl -w -o '-F --port=$PG_PORT -k /tmp' start";
            su postgres -c "$PG_BIN/psql -h 127.0.0.1 -p $PG_PORT -c \"CREATE USER $PG_USER WITH PASS
            su postgres -c "$PG_BIN/createdb -h 127.0.0.1 -p $PG_PORT $PG_DB -O $PG_USER";
            rm /work/.dbcreation # Release lock
            sleep infinity
```

Now run:

```
$ vagga run-postgres
```

We can also add some default records to the database, so we don't have to add them everytime we run our project. To do so, add the following to `db/seeds.rb`:

```
# db/seeds.rb
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' }
])
```

Now change the `run-postgres` command to seed the database:

```
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: |
          touch /work/.dbcreation # Create lock file
          while [ -f /work/.dbcreation ]; do sleep 0.2; done # Acquire lock
          rake db:migrate
          rake db:seed
          rails server
    db: !Command
      # ...
```

- – populate the database.

Now , everytime we run `run-postgres`, we will have our database populated.

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 posgres.

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

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

commands:
  psql: !Command
    description: Run postgres shell
    container: ubuntu
    # This long script initialized new empty postgres database each time
    # container is run
```

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

There is a more complicated example of postgres with alembic migrations

Redis

Simplest container with redis looks like this:

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

commands:

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

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

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

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

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

commands:

  redis-server: !Command
    description: Run instance of Redis server
```

```
    container: ubuntu
    run: |
      redis-server --daemonize no --port $REDIS_PORT1 --logfile "" --loglevel debug

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

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

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

Consul

```
containers:

  ubuntu-consul:
    setup:
    - !Ubuntu trusty
    - !Install [unzip, wget, ca-certificates]
    - !Sh |
        cd /tmp
        wget https://dl.bintray.com/mitchellh/consul/0.5.2_linux_amd64.zip
        unzip 0.5.2_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
```

Miscellaneous

Travis Gem

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

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

The vagga configuration for the command:

```
containers:
  travis:
    setup:
    - !Ubuntu trusty
    - !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:

```
containers:
  selenium:
    setup:
    - !Ubuntu trusty
    - !UbuntuUniverse
    - !Install [phantomjs]
    - !Py3Install [selenium, py, pytest]

commands:
  test: !Command
    description: Run selenium test
    container: selenium
    run: [py.test, test.py]
```

And the test may look like the following:

```
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
```

Firefox 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`:

```
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).

```
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:
  firefox: !Command
    container: browser
    environ: { HOME: /tmp }
    run: [firefox, --no-remote]
```

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):

```
containers:
  browser:
    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]
```

Documentation

Sphinx Documentation

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

```
containers:

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

commands:

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

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

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

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

A

B

C

P

R

S

T

U

V

W