# Velociraptor

***Release 4.1***

**May 06, 2017**

# Contents

Open-source Platform as a Service (Paas).

CHAPTER 1

# Overview

Velociraptor (VR) seeks to create repeatable deployments by technical and non-technical users across a suite of languages, providing the service commonly knows as Platform as a Service or PaaS.

VR is inspired by Heroku and in particular the 12-factor methodology. VR supports the use of buildpacks for flexible, pluggable building of source code into apps suitable for deployment into Linux containers (lxc).

## Getting Started

In September 2015, Jason presented Velociraptor to DCPython. It's long (45 minutes!) but provides an overview of what VR is and how it works.

Alternatively, view the presentation slides (PDF) for an outline of the talk.

CHAPTER 2

# Balancers

Velociraptor has the ability keep your load balancer configuration up to date as nodes come and go on different hosts and ports. This is done through Velociraptor's "balancer" interface. When you define a Swarm you have the option to pick a pool name and balancer for routing traffic to that swarm. The list of available balancers is configured in the Django settings:

```
BALANCERS = {
    'default': {
        'BACKEND': 'vr.common.balancer.dummy.DummyBalancer',
    }
}
```

In the above example, a balancer named "default" is configured, by setting the BACKEND parameter to the dotted-path location of the DummyBalancer class in the Python path. The no-op "dummy" balancer doesn't actually route anything. It just stubs out the balancer methods for running in tests and development. Besides the dummy balancer, there are several real balancer backends available. The rest of the configuration examples use YAML, as you would use if you deploy Velociraptor itself from Velociraptor.

## Nginx

The nginx balancer backend can be configured like so:

```
BALANCERS:
  my_nginx_balancer:
    BACKEND: vr.common.balancer.nginx.NginxBalancer
    user: some_user_with_sudo
    password: some_password
    include_dir: /etc/nginx/sites-enabled/
    reload_cmd: /etc/init.d/nginx reload
    tmpdir: /tmp
    hosts:
    - frontend1.mydomain.com
    - frontend2.mydomain.com
```

One by one, here's what those config values mean:

- my_nginx_balancer: An artbitrary name for this balancer. It will be saved with swarm records in the database. If you change it later, you'll have to update those records with the new name.
- user: The username to be used by Velociraptor when SSHing to the nginx hosts.
- password: The password to be used by Velociraptor when SSHing to the nginx hosts.
- include_dir: The path to a folder that nginx has been configured to use for config includes. The balancer backend will write files there to define pools. It's set to the Ubuntu location by default, so if you're on that OS you can omit this setting.
- reload_cmd: The command to be used to tell nginx to reload its config. By default this uses the command for the Ubuntu init script.
- tmpdir: A place to put temporary files. Defaults to /tmp, so you can omit it if you don't need to customize it.
- hosts: The list of nginx hosts whose config should be updated.

If you were to select this balancer under the swarm form's Routing tab and provide a pool name of "my_app", Velociraptor would create a "/etc/nginx/sites-enabled/my_app.conf" file on each of the specified balancer hosts, with these contents:

```
upstream my_app {
  server host3.my-squad.somewhere.com:5010;
  server host7.my-squad.somewhere.com:5011;
}
```

(The hosts and port numbers were automatically selected by Velociraptor while swarming.)

Though Velociraptor creates that nginx routing pool for you, it will not automatically create an nginx 'server' directive mapping that pool to a publicly exposed hostname, directory, or port. You can do that in /etc/nginx/sites-enabled/default, like this:

```
server{
  listen 80;
  server_name my-app.my-public-domain.com;
  location / {
    proxy_pass http://my_app;
  }
}
```

You will only have to create that server directive the first time you swarm my_app. In subsequent swarmings, Velociraptor will automatically reload the nginx config after writing the new my_app.conf file.

## Varnish

Like the nginx balancer, the Varnish balancer connects to hosts over SSH in order to read/write files and run commands. It shares a common base class with the nginx balancer, so its config is very similar:

```
BALANCERS:
  my_varnish_balancer:
    BACKEND: vr.common.balancer.varnish.VarnishBalancer
    user: some_user_with_sudo
    password: some_password
    include_dir: /etc/varnish/
    reload_cmd: /etc/init.d/varnish reload
    tmpdir: /tmp
```

```
    hosts:
    - cache1.mydomain.com
    - cache2.mydomain.com
```

It also uses Ubuntu defaults for include_dir and reload_cmd.

## Stingray/ZXTM

The balancer backend for Stingray (fka ZXTM) connects over SOAP rather than using SSH, so its config looks different:

```
BALANCERS:
  my_stingray_balancer:
    BACKEND: vr.common.balancer.zxtm.ZXTMBalancer
    URL: https://traffic.yougov.local:9090/soap
    USER: api_user
    PASSWORD: api_user_password
    POOL_PREFIX: vr-
```

Those parameters are:

- URL: The URL to the SOAP interface.
- USER: username to be used for the SOAP connection.
- PASSWORD: password to be used for the SOAP connection.
- POOL_PREFIX: All pools created by Velociraptor will be prefixed with this name. This is useful if you have both automatically- and manually-created pools.

## Using Multiple Balancers:

You can use multiple balancers by having multiple entries in the BALANCERS setting:

```
BALANCERS:
  my_varnish_balancer:
    BACKEND: vr.common.balancer.varnish.VarnishBalancer
    user: some_user_with_sudo
    password: some_password
    hosts:
    - varnish.mydomain.com
  my_nginx_balancer:
    BACKEND: vr.common.balancer.nginx.NginxBalancer
    user: some_user_with_sudo
    password: some_password
    hosts:
    - nginx.mydomain.com
```

The above example includes both an nginx and varnish balancer. (It also omits the settings that have Ubuntu defaults, so if you're not on Ubuntu you'll have to fill those in.)

## Routing Rules and Other Intentional Omissions

Load balancers/traffic managers have an eclectic and bewildering array of features, and wildly different interfaces and config languages for driving them. Velociraptor does *not* attempt to provide an abstraction over all those features. The balancer interface is concerned solely with creating and updating pools. It's up to you to add rules telling your load balancer which hostnames/ports/paths/etc should map to which pools.

## Concurrency Caveats

When you add nodes using one of the SSH-based balancers (nginx and Varnish), it will do the following:

1. Get the current list of nodes by reading the remote balancer's config.
2. Add the new nodes to that list.
3. Write a new config file (or files).
4. Tell the remote service to reload its config.

If two processes are both making changes at the same time, there's opportunity for the first one's changes to be overwritten by the second's.

In the nginx balancer, this risk is mitigated somewhat by use of a separate file for each pool. So you'll only have problems if two workers are both trying to update the same pool at the same time.

Varnish, however, does not support a glob-style include of all files in a directory as nginx does, so the Varnish balancer maintains a pools.vcl file with include directives for all of the pool-specific files. The pools.vcl file is updated only when new pools are created. So there is additional risk of overwritten config with the Varnish balancer if two Velociraptor workers are trying to create new pools at the same time. (This is *probably* an extremely rare occurence, but it will depend on the size of your Velociraptor installation.)

Additionally, if you have multiple nginx or Varnish instances configured for a balancer, there will be a few seconds of lag between when the first and last one get their new config. (SSHing and reading/writing files takes time.)

The ZXTM/Stingray balancer does not suffer from the same concurrency risks as the SSH-based balancers, because the underlying SOAP API provides atomic methods for add_nodes and delete_nodes.

## Creating New Balancer Backends

A balancer is a Python class that implements the `vr.common.balancer.Balancer` interface.

Here's a hand-wavy hypothetical example.

```
# the abstract base class in the lib doesn't actually provide any
# behavior but does help ensure you've implemented the right methods.

from vr.common.balancer import Balancer
from mythical.tightrope.api imort go_get_a_pool


class TightRopeBalancer(Balancer):
    def __init__(self, config):
        """
        The `config` is the dict representation of YAML config.

        For example: ::
```

```
        # YAML
        BALANCERS:
          my_tightrope_balancer:
            BACKEND: deployment.balancer.tightrope.Balancer
            user: some_user_with_sudo
            password: some_password
            hosts:
            - tightrope.mydomain.com

        # config argument
        {'user': 'some_user',
         'password': 'some_password',
         'hosts': ['tightrope.mydomain.com']}
    """
    self.config = config

def get_nodes(self, pool_name):
    """
    Find the list of nodes that exist in a pool.

    Args:
     - pool_name: string argument for the name of
                  the pool

    Return a list of nodes, which are strings in the form
    "hostname:port".

    If the pool does not exist, this should return an empty
    list.
    """
    try:
        pool = go_get_a_pool(pool_name)
        return pool.nodes
    except PoolDoesNotExist:
        return []

def add_nodes(self, pool_name, nodes):
    """
    Add nodes to the current pool.

    Args:
     - pool_name: the name of the pool as a string
     - nodes: list of strings in the form "hostname:port"

    If the pool does not exist, it should be automatically
    created.
    """

    try:
        pool = go_get_a_pool(pool_name)
        pool.add_nodes(nodes)
    except PoolDoesNotExist:
        go_create_a_pool(pool_name, nodes)

def delete_nodes(self, pool_name, nodes):
    """
    Delete a node from the pool.
```

```
        Args:
         - pool_name: the name of the pool as a string
         - nodes: list of nodes as strings in the form "hostname:port"

        This should return successfully even if the pool
        or one of the nodes does not exist.
        """
        try:
            pool = go_get_a_pool(pool_name)
            pool.delete_nodes(nodes)
        except PoolDoesNotExist:
            pass
```

Velociraptor doesn't yet have balancer backends for Apache or HAProxy. It probably should! Patches are welcome if you'd like to submit an additional balancer backend.

CHAPTER 3

# Builder

Velociraptor builds apps using Heroku buildpacks. Builds are done in temporary containers where the untrusted app and buildpack code can't do harm to the rest of the system.

## CLI

Velociraptor's build tool is called vr.builder, and can be installed like a normal Python package:

```
pip install vr.builder
```

That package provides a 'vbuild' command line tool that takes two arguments:

1. A subcommand of either 'build' or 'shell'.
2. The path to a yaml file that specifies the app, version, and buildpacks to be used to do the build.

Here's an example invocation of the vbuild command:

```
vbuild build my_app.yaml
```

The vbuild tool must be run as root, as root permissions are (currently) required to launch LXC containers.

## The YAML file

The yaml file given to vbuild should have the following keys

- app_name: Should be both filesystem-safe and have no dashes or spaces.
- app_repo_type: Should be either 'git' or 'hg'.
- app_repo_url: The location of the app's Git or Mercurial repository.
- version: The tag, branch, or revision hash to check out from the app repository. *This must be a string.* If your version number looks like a float (e.g. 9.0) then you must enclose it in quotes to make YAML treat it as a string.

- buildpack_urls: A list of URLs to the buildpacks that are allowed to build the app. Each buildpack's 'detect' script will be run against the app in order to determine which buildpack to run. If none matches, the vbuild command will exit with an error. To specify a particular version of a buildpack, include the revision hash as the fragment portion of the URL.

Here's an example of a valid build yaml file:

```
app_name: vr_python_example
app_repo_type: hg
app_repo_url: https://github.com/btubbs/vr_python_example
buildpack_urls:
- https://github.com/heroku/heroku-buildpack-nodejs.git
- https://github.com/heroku/heroku-buildpack-scala.git
- https://github.com/yougov/yg-buildpack-python2.git
version: v3
```

If you know which buildpack you want you can specify it directly with the buildpack_url field:

```
app_name: vr_python_example
app_repo_type: hg
app_repo_url: https://github.com/btubbs/vr_python_example
buildpack_url: https://github.com/yougov/yg-buildpack-python2.git
version: v3
```

If you specify both a buildpack_url and a buildpack_urls list, the singular buildpack_url setting will take precedence.

When you launch a build from Velociraptor's web interface, the build start message will include the YAML used to do the build (click the wrench to see the dialog with the YAML). You can copy/paste this into a local YAML file and run vbuild yourself to debug any problems with the build.

## Output

When the build has completed there should be three new files in your current working directory:

- build.tar.gz will contain the compiled result of the build.
- build_result.yaml will contain metadata about the build. A build_result.yaml file can be fed back to vbuild to do the build over again with the same versions, buildpacks, etc.
- compile.log will contain all the output from running the buildpack's 'compile' script on your app. (In cases where none of the listed buildpacks can detect your app, compile.log will not be present.)

## The Shell Command and Environment

In addition to the 'build' subcommand, the vbuild tool provides a 'shell' subcommand. You run it like this:

```
vbuild shell my_app.yaml
```

The shell subcommand works with the same YAML format as the build subcommand.

If you find that a build is failing for some reason you can use 'vbuild shell' to get a shell in the build environment and debug the problem. You will have exactly the same environment variables set, and the buildpacks and app code will be checked out and mounted into your container exactly as they are at build time.

You can also run the same script that Velociraptor runs inside the container to execute the buildpack:

```
/builder.sh /build/* /cache/buildpack_cache
```

Read the comments and source in builder.sh for more details.

## Checkouts and Caches

The vbuild tool keeps caches and copies of repositories on the local filesystem in order to speed up compilation on subsequent builds. All of these are kept under /apps/builder.

- Applications are kept under /apps/builder/repo, and should be identifiable by the app_name provided in the YAML file.
- Buildpacks are kept under /apps/builder/buildpacks.
- The caches used by buildpacks are kept under /apps/builder/cache

In all cases, folder names will be appended with an MD5 hash of the app's or buildpack's URL. This avoids collisions when two apps or buildpacks use the same name.

It is safe to delete the app repos, buildpacks, or caches saved by vbuild. The vbuild tool will re-download any repositories or re-create any directories it needs. Deletion of buildpack caches in particular is sometimes necessary if a buildpack gets a cache into a weird state.

# CHAPTER 4

# Containers

Velociraptor apps are deployed to LXC containers that give them an isolated process space and filesystem root. One way to think of them is "virtual machines that share a kernel with the host". Another way to think of them is "chroot on steroids".

At the time of writing (May 2013) Velociraptor's LXC containers are only minimally isolated. Though apps cannot see each other's code or config, essential system folders are bind-mounted from the host into the container and shared between apps. The host's network interface is shared inside the container. There are no caps on per proc resource usage (which LXC supports using Linux cgroups). As development continues, Velociraptor's containers will be made more isolated and secure. For now you should *not* run untrusted 3rd party code on Velociraptor.

The use of containers enforces the 12 Factor App rules that require state to be maintained in backing services (databases, caches, etc.) rather than on the application host. Any local files written by an app are likely to be deleted when the app is restarted, and *certain* to be deleted when a different release of the app is dispatched.

CHAPTER 5

# Runners

When you deploy an application with Velociraptor, it runs in a container. Those containers are set up and launched by a 'runner', which is a small command line application on the application host.

You can do some neat things using runners:

- Run a container on one host that will uptest a proc on another host (by setting the 'host' and 'port' keys in your local proc.yaml appropriately).
- Get a shell on a production host in the exact same environment in which your production instance is running.
- Easily start up a local instance of a proc that exactly matches what's running production.

The interface between Velociraptor and its runners is specified in terms of command line arguments and a yaml file.

A runner is launched with two command line arguments:

1. A command. ('setup', 'run', 'uptest', 'teardown', or 'shell')
2. The path to a proc.yaml file containing all the information necessary to set up the container with the application's build and config.

Using the 'setup' command might look like this:

```
some_runner setup /home/dave/myproc.yaml
```

## The proc.yaml file

The proc.yaml file contains the following keys:

- port: The network port on which the application should listen, if applicable.
- user: The name of the user account under which the application should be run. The application's files will have their owner set to this user.
- group: The application's files will have their group set to this.
- env: A dictionary of environment variables to be set before launching the application.

- settings: A dictionary of config values to be written to settings.yaml inside the container. The location of that file will be passed to the application using the APP_SETTINGS_YAML environment variable.
- cmd OR proc_name: If there is a 'cmd' key in proc.yaml, it will be used as the command to launch inside the container. If there is no 'cmd' key, then the 'proc_name' key should have an entry like 'web' or 'worker' that points to a line in the application's Procfile.
- build_url: The HTTP URL of the build tarball.
- build_md5: Optional. If supplied, and the runner sees that the build tarball has already been downloaded, its md5sum will be checked against build_md5 from the proc.yaml.

The file doesn't actually have to be named proc.yaml. It can have any name you like.

Note: String config values that start with an @ sign and also contain single quote characters get serialized in a special way by the underlying YAML library. For instance, "@How's it going?" gets serialized as '@How"s it going?'.

# Commands

Runners support the following commands:

## Setup

Example:

```
some_runner setup /home/dave/myproc.yaml
```

The `setup` command will read the proc.yaml file, download the build (if necessary), and create necessary scripts, directories and LXC config files for the container.

If proc.yaml contains a 'cmd' key, this will be written into the startup script created during setup. If there is no 'cmd' key, the runner will use the 'proc_name' key to determine which line from the application's Procfile should be executed.

This command locks the proc.yaml file so other locking runner commands cannot run on this file at the same time.

## Teardown

Example:

```
some_runner teardown /home/dave/myproc.yaml
```

The `teardown` command should remove the proc folder and related files from the filesystem. If the runner has done other changes to the host, such as creating special network interfaces for the container, it should clean those up too.

Note: It is permissible for teardown to leave a copy of the build tarball in /apps/builds even after teardown is called. (There's no way for teardown to know whether you have other containers based on the same build.)

The teardown command locks the proc.yaml file while running.

## Run

Example:

```
some_runner run /home/dave/myproc.yaml
```

The `run` command starts your process inside the container. The process should not daemonize. When the process exists, the container will stop with it.

The run command locks the proc.yaml while running.

### Uptest

Example:

```
some_runner uptest /home/dave/myproc.yaml
```

The `uptest` command relies on the presence of a proc_name key in proc.yaml. It looks for any scripts in <app_dir>/uptests/<proc_name> and will execute each one, passing host and port on the command line (as specified in the uptests spec). The host and port settings passed to the uptests will be pulled from the host and port keys in the proc.yaml.

The results from the uptests will be written to stdout as a JSON array of objects (one object for each uptest result). The uptest command must *not* emit any other output besides the JSON results.

Uptests should be run in an environment identical to the proc being tested (same os, build, settings, environment variables, etc.).

The uptest command does not lock the proc.yaml while running.

### Shell

Example:

```
some_runner shell /home/dave/myproc.yaml
```

The `shell` command creates and starts a container identical to the one running for the proc, but starts /bin/bash in it instead of the proc's command. This is useful for debugging pesky problems that only seem to show up in production.

The shell command does not lock the proc.yaml while running.

## Runner Variants

Velociraptor provides two runner implementations.

### vrun_precise

The `vrun_precise` runner is specific to Ubuntu 12.04 (Precise) hosts. It creates bind mounts of the host's essential system folders inside the container. This matches Velociraptor's original container implementation.

### vrun

The `vrun` runner supports specifying an OS image tarball to be used inside the container. It uses the following additional keys in proc.yaml:

- image_name: This should be a filesystem-safe name for the image to be used in the container. Example: ubuntu-core-12.04.3-amd64
- image_url: An http URL from which the image tarball can be downloaded.
- image_md5 (optional): If provided, this checksum will be used to determine whether an already-downloaded tarball is correct. If there's a mismatch, the image will be re-downloaded.

Here's a working example of those three proc.yaml lines:

```
image_url: http://cdimage.ubuntu.com/ubuntu-core/releases/12.04/release/ubuntu-core-
↪12.04.3-core-amd64.tar.gz
image_md5: ea978e31902dfbf4fc0dac5863d77988
image_name: ubuntu-core-12.04.3-amd64
```

(That Ubuntu core image is only 34MB!)

Image tarballs must be compressed with either gzip or bzip2 compression, and use the appropriate extension in their filenames.

The `vrun` runner uses an overlayfs mount of the unpacked build inside each container, so the same image can be used by many containers without using any more disk space.

Other runner implementations may be added in the future, or created as separate projects.

CHAPTER 6

# Stacks

Velociraptor's concept of a "stack" is more or less the same as Heroku's:

```
A stack is a complete deployment environment including the base operating
system, the language runtime and associated libraries. As a result,
different stacks support different runtime environments.
```

## OS Images are to Stacks as Builds are to Apps

Unlike Heroku, Velociraptor lets you create your own stacks, and provides tools to make it fairly simple. When using the Velociraptor UI, you can select Platform -> Stacks from the menu. To create a new stack, you will need to provide the URL for a base image tarball and a script to run inside the base image to install the things you need. Put those pieces of information in the form and click "Save and Build". Velociraptor will tell one of its workers to do the following:

1. Download your base image and unzip it.
2. Start a container with your base image mounted in read/write mode.
3. Run your provisioning script inside the container.
4. Once the provisioning script is finished, tar up the result and save it to Velociraptor's file store.

After those steps are complete, you should be to go to Platform -> Apps, create a new application or select an existing one, and link it to your stack. When you next build the app, the build will occur inside your newly built OS image. When you deploy the app, the image will be downloaded and unpacked just like the build is.

In time you may realize that you want to change something in the OS image. Maybe you need to add a system-level package, or maybe there's an urgent security fix. You should modify your provisioning script to make the desired change, edit your stack in the Velociraptor UI, upload your new script, and click "Save and Build" as you did before. When your image is done building, it will be marked as the 'active' build in the stack, and will be used for all builds and swarms of your app.

## Base Images

Velociraptor needs a base image as a starting point. You can use an existing tarball provided by a Linux distribution. Ubuntu's website provides minimal base images that you can use.

But!

There is a bug in stock Ubuntu 14.04 (Trusty) and CentOS 6.5 images that makes them essentially unusable in containers, unless you disable PAM audit signals. Some kind souls have implemented that workaround for Docker images, and it works for Velociraptor as well. There is a Velociraptor-compatible base image for Ubuntu Trusty at http://cdn.yougov.com/build/ubuntu_trusty_pamfix.tar.gz.

Additionally, any Docker image can be made into a Velociraptor image by doing docker export and gzipping the result.

## Provisioning Scripts

The only requirement on provisioning scripts is that they be executable, but it's recommended that you write them in Bash.

The cedarish open source project provides a provisioning script that can be used to make a Heroku-compatible image.

## Using vimage

Most Velociraptor functions have both a high level graphical user interface and a lower level command line interface. OS images are no exception. The vr.imager Python package is used by the Velociraptor server to build OS images, and you can easily use it from the command line yourself.

These commands require that you run as root on a Linux host with LXC installed.

Install vr.imager:

```
pip install vr.imager
```

Create a file named my_image.yaml with contents like this:

```
base_image_url: http://cdn.yougov.com/build/ubuntu_trusty_pamfix.tar.gz
base_image_name: ubuntu_trusty_pamfix
new_image_name: my_awesome_image_20141031
script_url: /path/to/my_provisioning_script.sh
env:
  PATH: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

(As you may have guessed, any of the "_url" fields in that YAML file may be given either an http url or a local file path.)

Tell vimage to build it:

```
vimage build my_image.yaml
```

You should see all the steps in your provisioning script get executed in your terminal. When it's all done, you should have two new files in your current directory:

```
root@vagrant-ubuntu-trusty-64:~# ls -1
my_awesome_image_20141031.log
```

```
my_awesome_image_20141031.tar.gz
my_image.yaml
```

If something goes wrong while running your provisioning script, you might want to get into the container and debug interactively. You can do so like this:

```
vimage shell my_image.yaml
```

CHAPTER 7

# Supervisor is Awesome

Velociraptor relies heavily on Supervisor to manage processes. It calls Supervisor's XML RPC interface in order stop and start things, and to report on which processes are running on which hosts. You should make sure you have the XML RPC interface enabled in supervisord.conf:

```
[inet_http_server]
port = *:9001
username = fakeuser
password = fakepassword
```

See the Supervisor docs for how to *configure this section*, including an option for including a SHA1 hash of a password instead of plaintext.

## The Velociraptor Event Listener

Velociraptor includes a Supervisor event listener plugin to watch for any changes in process state and put a message on a Redis pubsub when they happen. The Velociraptor web interface relies on these messages to stay up to date.

You'll need to install the 'vr.agent' package on each of your hosts and configure Supervisor to start the 'proc_publisher' event plugin, as shown in this sample supervisord.conf snippet:

```
[eventlistener:proc_publisher]
command=proc_publisher
events=PROCESS_STATE,PROCESS_GROUP,TICK_60
environment=REDIS_URL="redis://localhost:6379/0",HOSTNAME=precise64
```

*command*

The 'command' parameter here should point to the proc_publisher script installed by the vr.agent package.

*events*

Here you configure the events that Supervisor should send to the plugin. Set it as above.

*environment*

Here you set environment variables to be passed in to the event plugin process. The REDIS_URL variable must be set in order for the plugin to know where to post events.

The HOSTNAME variable is optional. If provided, it will be the hostname included in messages placed on the Redis pubsub. If not provided, the event plugin will guess a hostname by calling Python's "socket.getfqdn()" function. You should set the HOSTNAME variable if the name used in your Velociraptor web interface isn't the same as the one returned by socket.getfqdn(). If they don't match, and you don't set HOSTNAME, you'll see duplicate procs on your dashboard.

## Version

Supervisor 3.1.0 or later is required for the event listener support that Velociraptor needs.

CHAPTER 8

# Uptests

Uptests are an essential part of Velociraptor's promise of zero-downtime deployment. The idea is that a deployment should look like this:

1. You have some instances of your app already deployed, but you want to update the code or config.
2. You deploy new instances with the new code or config, while leaving the old instances up and still handling traffic. The new instances don't handle any traffic yet.
3. You run tests on the new instances to ensure that they're running happily and ready to serve traffic.
4. If the tests pass, then you change your routing rules to serve traffic from the new instances instead of the old ones.
5. Finally, you take down the old instances.

Velociraptor automates all of the above steps when you swarm. All you have to do as a developer is include some uptests.

## Definition

An uptest is a small script or program that checks whether a single instance of an app is running correctly.

## Running

Uptests scripts must be executable files. They accept two command line arguments: "host" and "port". Uptests are run in an environment identical to production (same build, same environment variables), but not necessarily on the same host as the proc being tested. The uptest should exercise the designated proc in some way to check whether it's ready to accept production traffic.

## Results

If successful, the uptest script must exit with status code 0. Any other exit code signifies a failure. The script may emit debug information to stdout or stderr.

## Organization

Uptests live in your app's source code repo. Each proc in an app has different uptests, organized by subfolders of an 'uptests' folder in the project root. In the example below, the web proc has 3 uptests, which will be executed in the order listed by the OS.

```
|
+-- Procfile
|
+-- README.rst
|
+-- uptests/
   |
   +-- web/
      |
      +-- 01_its_alive.py
      |
      +-- 02_login_required.py
      |
      +-- 03_check_rss.py
```

## What to Test

Uptests do not replace the unit or functional tests that you write in the course of normal development. You still need those!

Uptests should test what those other tests can't:

- They can let you know if the production version pulled in a different (and buggy) dependency than staging.
- They can tell you if there's some system level dependency that's not met in production.
- They can catch fat finger errors you made when typing in config values like the location of your production database.

At minimum you should have an uptest that pings your app to check whether it's running. More robust uptests will check things like whether all the app's backing services are also up.

It is not recommended that uptests make any persistent changes. They shouldn't create or delete records.

## You Will Love Uptests

Uptests make it safe(r) to deploy your code a dozen times a day if you need to. If you take the time to write some now, your future self will thank you when you save him or her from bringing the whole site down because of some stupid slip.

CHAPTER 9

# Volumes

Velociraptor's volumes feature allows you to specify one or more directories on the host to be mounted inside the container.

## Caution

The volumes feature is a departure from Velociraptor's normal requirement that applications be completely stateless (i.e. 12 Factor-compliant). With volumes, applications may maintain some state between deployments by reading from and writing to persistent files on the local disk.

## Configuration

Volumes are specified in the user interface by entering YAML configuration into the Swarm or Release forms. In both cases, the YAML should be a list of host dir, mountpoint pairs:

```
- [/var/data, /data]
```

In that example, the host's /var/data folder will be mounted inside the application's container at /data.

You may specify multiple volumes for a container:

```
- [/var/data, /data]
- [/blahblah/cache, /cache]
```

Because YAML supports multiple ways of encoding the same structure, you are likely to see notation different from the above when viewing volumes in the UI. This format is equivalent to the above:

```
- - /var/data
  - /data
- - /blahblah/cache
  - /cache
```

## Permissions

Volumes are implemented using bind mounts written into the proc's LXC container configuration. They will *not* automatically modify any permissions on the files in the volume in order to make them readable or writable by your application. It is up to you to ensure that the permissions are appropriately set, and then use the 'Run as' field in the Swarm and Release forms to make your application run as the right user.

CHAPTER 10

# Deploying Velociraptor

This page serves as a placeholder for instructions and advice on deploying Velociraptor in a production environment.

## Setting Memory Limits

In order to support the memory limits on containers, the host kernel must be configured with one or both of the following command-line parameters:

swapaccount=1 cgroup_enable=memory

Please update this document if you can provide clarification on which parameters are actually required and how to specify those parameters.

## Monitoring with Flower

As Velociraptor uses celery queues to manage its tasks, it's often useful to have a tool for monitoring them. The Flower project implements one such tool.

To deploy it against your broker, add a proc to your VR Procfile like so:

```
flower: python -m vr.server.manage celery flower --broker=$BROKER --port=$PORT
```

Add the flower dependency in your requirements.txt (alongside other VR dependencies):

```
Flower =>0.9, <1
```

And deploy that proc using VR. You'll then have a web service configured to monitor Celery.

CHAPTER 11

# Administration

In the process of deploying apps into real-world systems, there are cases where actions in the UI can lead to surprising outcomes due to limitations that Velociraptor sets for itself in managing the systems.

This section provides some guidance on common admininstrative scenarios.

## Deploying Workers

Workers are not part of a swarm, as they do not bind to a port and must be specifically installed one per host. Also, workers can't be easily deployed without a working worker. Therefore, deploying workers takes special care.

First, swarm another process to the desired version. It could be the beat proc or web proc or another, but it should be using the same configuration as the workers use. After it's built properly, that will have created a release.

Next, manually stop one of your workers. Note the 'config' name of the worker. Then do a manual deploy step (Actions > Deploy), select the release that was just created (should be the first one in the list). Select 'worker' for the proc and give the config the same name as noted previously. Finally, select the same host as the stopped worker and a port of 0 and submit.

The proc should deploy and turn green and start completing tasks. Next, repeat the steps for the remaining workers.

After you're done, you'll have a new set of workers running and the old workers stopped. Additionally, if the last worker deploy worked, that worker would have been deployed using one of the new running workers, so you have some confidence that it's functional, so it's safe now to destroy the old workers.

## When a Squad Loses a Host

When a squad unexpectedly loses a host (or hosts), not only do all of the procs on that host cease to function, but also the supervisor will not respond to requests. At the time of this writing, that causes the dashboard to become unresponsive... at a time when the administrators desperately need the dashboard.

VR doesn't yet have a UI for deleting members of a squad, so use the Django admin. Navigate manually to /admin/squads, select the affected squad, then check the delete box for each failing host then click save. Repeat for additional squads if necessary.

Dependening on how many hosts died and how much slack you have on the other hosts, you may need to provision replacement hosts. If so, create the hosts using whatever technique you use to provision hosts, and then add those hosts using the VR UI by browsing to /host/add/, entering the FQDN of the host, and selecting the target squad in which it should be added. Repeat for each supplemental host needed.

Now your environment should be responsive and ready to recover your failed procs. Since VR does not keep a record of which procs are offline, your best option is to dispatch all swarms in the affected squad or squads. Any swarms that are already complete will be quickly fulfilled, and only those not matching the running configuration will be re-deployed and re-routed.

To identify the relevant swarms, there's no UI that will accomplish this for you. You must go to the database, find the relevant squad, and then query for swarms pointing to that squad. If that's too much trouble, you can consider simply dispatching all swarms, which you can list with the CLI:

```
$ vr.cli list-swarms '.*'
swarm1-config1-runner
swarm2-config2-runner
...
```

Once you have the list of swarms, you can readily dispatch each of those using the same version that was last indicated:

```
$ export RELEVANT_SWARM_PATTERN="swarm1-config1-runner|swarm2-config2-runner|..."
$ vr.cli swarm "$RELEVANT_SWARM_PATTERN" -
```

Important is the last '-' character, which is the version to dispatch. The dash simply indicates use the current version.

The CLI will then dispatch each of the indicated swarms, getting all the procs back to production levels.

## Proc Supervisor

Velociraptor delegates the process supervision to Supervisor. As a result, procs in the UI may not have been deployed by VR, but could have been deployed manually. Regardless of how a proc came to exist in supervisor, it is displayed in the UI.

## Defunct Procs

Although Velociraptor will tear down procs within the same swarm when they are no longer needed for that swarm, if the definition of the swarm changes, Velociraptor will no longer recognize the extant procs from a previous manifestation of a particular swarm.

For example, if one has a swarm for MyApp, but then changes the Application to be MyAppNG, Velociraptor will deploy new procs to service a swarm called MyAppNG-{ver}-{config}, but it will do nothing to eliminate the MyApp-{ver}-{config}. It will, if configured, update the routing to point to the new app, giving the desired behavior, i.e. that MyAppNG is the exposed service, but the rogue procs will continue to run.

It is unclear at this time if those rogue procs are recognized as consuming a port in Velociraptor's port accounting.

This condition can happen when mutating any of the following Swarm fields:

- App
- Proc name

- Config name
- Squad

Therefore, to avoid leaving rogue procs lying around, it is recommended that one of the following techniques be followed to clean up the orphaned procs:

- Manually delete them using the UI or CLI.
- Create another swarm matching the original and dispatch it with a size of 0.
- Before making the change, first dispatch the swarm with a size of 0 (this will result in downtime).
- Re-use the mutated swarm by mutating it back but also clear the routing fields (so that routing is not affected) and with a size of 0. Then, restore the desired settings in the swarm.

CHAPTER 12

# Development

## VM

The smoothest way to get running is to start up a VM with the included Vagrantfile. This requires having VirtualBox and Vagrant installed. Go do that now.

You'll need a local clone of the Velociraptor repo:

```
git clone --recursive https://github.com/yougov/velociraptor
```

Now launch vagrant:

```
cd velociraptor
vagrant up
```

Now go make a sandwich while you wait for the Ubuntu Trusty VM image to download (about 430MB).

Installation of system-level dependencies inside the VM is done automatically using Vagrant's Puppet provisioner. This includes some normal apt packages, (curl, Vim, Postgres), and some installed with pip (Mercurial and Virtualenv). You can see the Puppet manifest at puppet/manifests/vr.pp.

The first time you 'vagrant up', the Puppet provisioning could take about 5 minutes. It will be faster on later startups, since most packages will already be installed.

Once the image is all downloaded and Puppet has run, log in with:

```
vagrant ssh
```

You're now inside your new Vagrant VM! The Velociraptor repo will be at /vagrant. Now make a Python virtualenv for yourself. It will use Python 2.7 by default. Virtualenvwrapper is pre-installed to make this extra easy:

```
mkvirtualenv velo
```

## Python Dependencies

Velociraptor contains a dev_requirements.txt file listing its dev-time Python dependencies. You can install the dependencies with this:

```
cd /vagrant
pip install -r dev_requirements.txt
```

## Database

There is a dbsetup.sql file included that contains commands for creating the Postgres database used by Velociraptor:

```
psql -U postgres -f dbsetup.sql
```

Once your database is created, you'll need to create the tables:

```
python -m vr.server.manage syncdb --noinput
python -m vr.server.manage loaddata bootstrap.json
```

The schema is created with an initial user `admin` with password `password`.

As Velociraptor is developed and the DB schema changes, you can run *python -m vr.server.manage migrate* again to get your local DB schema in sync with the code.

## Dev Server

The Velociraptor server is composed of three processes:

1. The main Django web service.
2. A Celery daemon that starts and controls one or more workers.
3. A 'celerybeat' process that puts maintenance jobs on the Celery queue at preconfigured times.

There is a Procfile included with Velociraptor that can be used to run a development environment with these processes. You can use Foreman to read the Procfile and start the processes it lists:

```
foreman start -f Procfile.dev
```

That will start the Django dev server on port 8000, the Celery workers, and the celerybeat process.

Now open your web browser and type in http://localhost:8000. You should see Velociraptor. (The Vagrantfile is configured to forward ports 8000, 9001, and 5000-5009 to the VM. If you need these ports back for other development, you can stop your Vagrant VM with a *vagrant halt*.)

## Add Metadata

### Buildpacks

In order to build and deploy your apps, Velociraptor needs to be told where they are and how to build them. The 'how to build them' part is done with Heroku buildpacks. Go to http://localhost:8000/buildpack/add/ in your browser in order to add a buildpack. You will need to enter the git (or mercurial) repository URL, as well as an integer for

the 'order'. See the Heroku buildpack documentation to understand more about how buildpacks work and why order matters. For now, just add a single buildpack, and set its order to '0'. A good one to start is the NodeJS buildpack.

## Squads and Hosts

In order for Velociraptor to know where to deploy an application, it requires some hostnames. Velociraptor does load balanced deployments across a group of hosts called a "Squad". Go to http://localhost:8000/squad/add/ to create a new squad. Call it whatever you like (I suggest 'local'). Squad names must be unique. Then add a host; go to http://localhost:8000/host/add/ and give the squad a host named 'vr-master', which is the hostname of the Vagrant VM itself.

## Stacks and Images

Velociraptor uses a container based system for isolating the execution environments of each application.

A "legacy" stack is provided but deprecated.

Instead, create a trusty stack. Use the base trusty image per docs. http://cdn.yougov.com/build/ubuntu_trusty_pamfix.tar.gz

Provision the stack with the 'provision.sh' file from the Velociraptor repository. You must also provide name and description. Use "trusty" for both.

The provisioning script takes some time as it needs to download, expand, and mount the base image, run the provisioning script in a container for that image, collect the image back into an archive, and upload the image to the Velociraptor image repository.

Watch the "worker" log for progress and wait for a green cube icon in the UI. The process takes most of 20 minutes.

## Apps

Now tell Velociraptor about your code! Go to http://localhost:8000/app/add/ and give the name, repo url, and repo type (git or hg) of your application. If you don't have one around, try the vr_node_example app. The name you give to your app should have only letters, numbers, and underscores (no dashes or spaces).

You can leave the 'buildpack' field blank. Velociraptor will use the buildpacks' built-in 'detect' feature to determine which buildpack to use on your app.

Select "trusty" for the stack.

## Swarms

Swarms are where Velociraptor all comes together. A swarm is a group of processes all running the same code and config, and load balanced across one or more hosts. Go to http://localhost:8000/swarm/ to create yours. Here's what all the form fields mean:

- App: Select your app from this drop down.
- Tag: This is where you set the version of the code that Velociraptor should check out and build. You can use almost any tag, branch name, bookmark, or revision hash from your version control system (any valid 'git checkout' or 'hg update' target), as long as it does not contain invalid characters for use in file names/directory names (most notably, /). Use 'v5' for the vr_node_example.
- Proc name: The name of the proc that you want to run in this swarm (from the Procfile). Type in 'web' for vr_node_example.

- Config Name: This is a short name like 'prod' or 'europe' to distinguish between deployments of the same app. Must be filesystem-safe, with no dashes or spaces. Use 'demo' here for vr_node_example.
- Squad: Here you declare which group of hosts this swarm should run on. If you set up the squad as indicated earlier in this walkthrough, you should be able to select 'local' here.
- Size: The number of procs to put in the swarm. Try 2 for now.
- Config YAML: Here you can enter optional YAML text that will be written to the remote host when your app is deployed. Your app can find the location of this YAML file from the APP_SETTINGS_YAML environment variable.
- Env YAML: Here you can enter YAML text to specify additional environment variables to be passed in to your app.
- Pool: If your app accepts requests over a network, you can use this "pool" field to tell your load balancer what name to use for the routing pool. By default Velociraptor talks only to an in memory stub balancer called "Dummy". For the walkthrough, leave this field blank. To configure a real load balancer, see docs/balancers.rst in the Velociraptor repo. Velociraptor supports nginx, Varnish, and Stingray load balancers. This interface is pluggable, so you can also create your own.
- Balancer: Here you select which balancer should be told to route traffic to your swarm. For the walkthrough, leave this field blank.

Now click Swarm. Velociraptor will start a series of worker tasks to check out the buildpack, check out your code, download the image, compile your code in the image, save the resulting build, push it out to the hosts in the squad along with any config you've specified, and launch the code within the stack image. The Swarm Flow diagram in the docs folder illustrates the process.

## Tests

Run the tests with py.test from the root of the repo after installing the dev requirements:

```
cd /vagrant
pip install -r dev_requirements.txt
py.test
```

The tests will automatically set up and use separate databases from the default development ones.

While developing, you might want to speed up tests by skipping the database creation (and just re-using the database from the last run). You can do so like this:

```
py.test --nodb
```

This should be safe as long as we keep using randomly-generated usernames, etc., inside tests.

## Editing Code

Running the code inside a VM does not mean that you need to do your editing there. Since the project repo is mounted inside the VM, you can do your editing on the outside with your regular tools, and the code running on the inside will stay in sync.

## UI

All frontend interfaces rely on a 'VR' javascript object defined in deployment/static/js/vr.js. Individual pages add their own sub-namespaces like VR.Dash and VR.Squad, using vrdash.js and vrsquad.js, for example.

Velociraptor uses goatee.js templates (a Django-friendly fork of mustache.js). They are defined as HTML script blocks with type "text/goatee".

Velociraptor makes liberal use of jQuery, Backbone, and Underscore.

## Repositories (and Submodules)

Velociraptor is a suite of projects in the *vr* namespace. Each of these projects are a separate repository, linked by the parent repository https://github.com/yougov/velociraptor using git submodules.

If you're committing to the project, you'll want to first configure the parent repository to automatically push commits in subrepos referenced by the parent:

```
$ git config push.recurseSubmodules on-demand
```

CHAPTER 13

# Indices and tables

- genindex
- modindex
- search