Squadron Documentation

Release 0.6.4

Squadron Leader

Apr 18, 2018

Contents

Contents:

CHAPTER 1

Overview

Squadron helps you deploy and configure your software. It grabs your software, sets it up right, starts it up, and has built-in tests to make sure this goes off without a hitch.

Squadron is good for:

- Deploying complex websites
- Releasing your new software-as-a-service application
- Deploying testing and production versions of your code
- Testing your configuration

The basic process of using Squadron is: make a Squadron repo, configure your service after learning how Squadron works, commit your changes, and then run *squadron* on your servers.

1.1 Features

Squadron has a bunch of features to help you download, configure, and test your software and its dependencies.

1.1.1 Multiple ways to get your code

With Squadron, you can download binaries, download tarballs of your code, grab your source with git. With *apt*, *virtualenv*, and *npm* support, you can get your code's dependencies.

After grabbing your code, Squadron can run *Built-in Tests* and use *Atomic Deployments* to deploy your code.

1.1.2 Easy dependency management

Squadron supports apt, npm, and pip via virtualenv, and we're always adding more. Adding a new package to install is super easy, and you can have different packages in different *Environments*.

1.1.3 Easy Templating

Squadron uses a simple templating library called Quik to write configuration templates. They look like this:

```
[log]
debuglog = @loglevel rotatinglog @logdir 5000 5
#if @output:
outputlog = DEBUG stderr
#end
```

This is used to write out your configuration files for your software. Values are set through defaults and service configurations and can be different for different *Environments*.

1.1.4 Atomic Deployments

Did you know that deploying your code takes time? And during that time your users might get inconsistent results? Why bother with that headache. Deploy your code atomically!

Squadron can easily deploy your code atomically via a symlink so that its either all the new version or doesn't reflect any changes at all.

1.1.5 Environments

You want to have a development and production environments? Easy. You want to have six different QA environments? Done. You need two production environments for A/B testing? Trivial.

Environments are easy with Squadron.

1.1.6 Built-in Tests

Squadron has two types of built-in tests: automatic tests you don't have to write, and an easy framework for testing your deployed code.

If you write a JSON file as part of a template, Squadron will check to make sure it's a valid JSON file. In the future, XML files written out by templates will be similarly checked, and if they have a schema, that will be verified.

The second type of tests are the kind you write because you know your service. Do you write a PHP script to hit a URL? Or a simple bash script to check if your service is running? Whatever you want, we'll run it.

1.1.7 Rollback

What if your deployment goes wrong?

With traditional configuration management software, it's difficult or impossible to rollback to exactly how it was before the deployment. With Squadron, that's built-in.

1.1.8 No programming

And, maybe best of all, there's no programming involved. There's no Domain Specific Language to learn. It's all just config that is rigorously checked by Squadron when you run it.

No programming means less testing and little to no debugging so you can get what you want done faster and easier.

1.2 Open source

Squadron is open source software written in Python. Our project source and issue tracker is on Github. If you'd like to contribute, please do!

You can get in touch with us via email, on Twitter or via IRC:

```
irc.freenode.net #squadron
```

We'd love to hear from you!

1.3 Sound good?

Does it sound like Squadron fits your use case? If so, head on over to the *Getting Started* section and try Squadron out!

CHAPTER 2

Getting Started

Squadron configures your software service. It install packages, writes out configuration templates, and tests them. This getting started guide tells you how to do all that. If you'd like to know more about what Squadron can do, see the *Overview* of Squadron.

If you want to follow along with this guide, we've made a git repo for you so you don't have to type out all these commands:

```
$ git clone -b simple2 https://github.com/gosquadron/example-squadron-repo.git
```

You can also play with a Vagrant box we made to test out Squadron.

2.1 Install Squadron

First, get the prerequisites:

```
$ sudo apt-get install git python python-pip
```

or, if you're on OS X:

```
$ brew install python python-pip git
```

Now let's install squadron:

```
$ pip install squadron
$ squadron setup
Location for config [/home/user/.squadron]:
Location for state [/home/user/.squadron/state]:
Initializing config dir /home/user/.squadron
Initializing state dir /home/user/.squadron/state
```

Squadron can be installed into a virtualenv. It uses a directory to store *Global Configuration* and another one to store the state change of your services.

Squadron looks for config in the following places:

- /etc/squadron/config
- /usr/local/etc/squadron/config
- ~/.squadron/config
- or in .squadron/config in the Squadron repository itself

From there it reads the location of its state directory. The setup step is optional if you want to keep the config in the repository and make the state directory on your own.

2.2 Start a Squadron repository

It's not too hard:

```
$ mkdir squadron
$ cd squadron
$ squadron init
$ ls
config/ services/ nodes/ libraries/
$ git rev-parse --is-inside-work-tree
true
```

Squadron uses git for everything, so squadron init automatically makes a git repository for you!

2.3 Describe your service

So, to deploy a service, you need to tell Squadron how to do it. We're going to deploy a simple website as an example.

To make a service, we need to provide a service version. This isn't the version of our website, but instead the version of this deployment configuration:

```
$ squadron init --service website --version 0.0.1
$ tree -F services/website
website/
`-- 0.0.1/
    |-- actions
    |-- copy
    |-- defaults
    |-- react
    |-- root/
    |-- schema
    |-- state
    `-- tests/
```

We won't need all these files yet, and Squadron gives you sensible defaults if you don't need the features they provide.

Let's make a *state* to install apache2 for our simple website. The format is YAML or JSON, both are supported:

```
[
    {
        "name":"apt",
        "parameters":["apache2"]
    }
]
```

Now when we later run squadron, it'll make sure that Apache is installed via apt-get.

2.3.1 Templating

Squadron takes whatever files you have in root/ and deploys them to the correct directory (which in this):

```
$ cd services/website/0.0.1
$ tree -F root
root/
    ├── main~git
    └── robots.txt~tpl
```

So we've got two strange looking filenames. The tilde (~) means that Squadron will apply that handler to that file. The '~tpl' extension is how you make files via a template.

Squadron uses the Quik templating library, so robots.txt~tpl will look something like this:

```
User-agent: *
#for @d in @disallow:
Disallow: @d
#end
Allow: /humans.txt
```

So the variable @disallow, which is an array, is looped over and so there are as many Disallow directives as elements in the array.

main~git looks like this:

```
{
    "url":"https://github.com/cxxr/example-squadron-repo.git",
    "refspec":"@release"
}
```

Squadron will clone this repo when it runs, checkout the refspec simple (which is a branch, a tag, or a hash) and place it in the 'main' directory. The variable '@release' will be replaced by whatever we set that to later

2.3.2 Configuration

How do all those values get set? They're set in two ways.

The first is from the service configuration for each environment. Back in the top level of the Squadron directory, there's a directory called config/. In it are your environments.

Environments are distinct places you can deploy your code to that don't interact with each other. This allows you to have multiple testing environments that don't affect your customers. Let's make a development environment for our website:

```
$ cd -
$ ls
config/ services/ nodes/ libraries/
$ squadron init --env dev
```

Now there's a file called config/dev/website, which is pre-populated with the latest version number. Let's add the disallow config so the file looks like this:

```
{
    "base_dir": "/var/www",
    "config": {
        "disallow":["/secret/*","/admin/*"],
        "release":"master"
    },
    "version": "0.0.1"
}
```

The "version" field tells Squadron which service description version to use. Different environments can use different service description versions at the same time.

The "config" field is a object that will be given to your service. These fields can be used in templates. If you have config that is often the same between environments, you can put it in another place.

The "base_dir" field tells Squadron where the root/ directory should be written to. Since we're just deploying files to our web root, it's */var/www*.

The second way in which these values are set is via *defaults*. This file can be used to set default values in case none are set. Keys are the key in question, and the values are the values you would set in the config.

An equivalent *defaults* file for our website would be:

```
{
    "disallow":["/secret/*","/admin/*"]
}
```

2.3.3 Schema

Squadron includes one very useful file with every service description called *schema*. This is a JSON schema describing the configuration that your service accepts. For our service it looks like this:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type" : "object",
    "properties" : {
        "disallow" : {
            "description" : "a list of disallow directives",
            "type" : "array",
            "items": {
                "type": "string"
            },
            "uniqueItems": true
        },
        "release" : {
            "description" : "what to checkout from the git repository",
            "type" : "string"
        }
    },
    "required": ["disallow", "release"]
}
```

This allows you to be sure that you passed in the correct types of input in your config files and in your defaults. If you don't supply a JSON Schema, everything will still work, but it won't be checked, either.

You can do some fairly advanced things with JSON Schema, such as regular expression matching. With this you could ensure that "release" met some tag pattern or similar.

2.4 Nodes

Now, how can you make sure that each node which runs Squadron runs the correct stuff? That the database node doesn't install Apache? Enter the nodes directory:

```
$ ls
config/ services/ nodes/ libraries/
$ cd nodes
```

This directory should have in it exact domain name matches (FQDN, to be precise) of the machine, or you can use glob style matching with percent (%) being the glob marker, instead of the usual asterisk (*). Files would look like these:

```
$ ls
dev-01.nyc.example.com # Only matches the machine with that name
dev-%.example.com      # Matches all dev machines
%-db%.example.com      # Matches all database machines
%                      # Matches all machines
```

Node files look like this:

```
$ cat %
{
    "env":"dev",
    "services":["website"]
}
```

Any node will run website in the dev environment unless overridden by another, more specific node file. All node files that match are sorted by length ascending, and applied in that order.

2.5 Testing your changes locally

We want to make sure that our configuration works as expected. Squadron allows you to see the result of your configuration before even touching a remote server.

Here we will pretend that we are the machine mywebserver.com and see the results locally without modifying our system:

```
$ squadron check
Staging directory: /tmp/squadron-s70Rjh
Would process apache2 through apt
Dry run changes
===============
Paths changed:

New paths:
    website/robots.txt
    website/main/LICENSE
    website/main/README.md
    website/main/index.html

$ tree -F /tmp/squadron-s70Rjh
/tmp/squadron-s70Rjh
`-- website/
    |-- main/
```

```
    |   |-- index.html
    |   |-- LICENSE
    |   `-- README.md
    `-- robots.txt
```

Our template was applied as well:

```
$ cat /tmp/squadron-s70Rjh/website/robots.txt
User-agent: *

Disallow: /secret/*

Disallow: /admin/*
Allow: /humans.txt
```

2.6 Deploying your changes (locally)

Now, if the machine you're developing on is the machine you'd like to set up your website on (which is unlikely), you can just apply your changes:

```
$ sudo squadron apply
Staging directory: /var/squadron/tmp/sq-0
Processing apache2 through apt
Applying changes
Successfully deployed to /var/squadron/tmp/sq-0
===============
Paths changed:

New paths:
    website/main/README.md
    website/robots.txt
    website/main/index.html
    website/main/LICENSE
```

And you can see that this won't work twice in a row, as nothing has changed:

```
$ sudo squadron apply
Staging directory: /var/squadron/tmp/sq-1
Processing apache2 through apt
Nothing changed.
```

Notice how the staging directory was increased by one. This lets you have several staged (but not deployed) versions in case of test or deployment failures. This is also how auto-rollback works.

Running squadron check produces similar results:

```
$ squadron check
Staging directory: /tmp/squadron-H1Vym2
Would process apache2 through apt
Nothing changed.
```

2.7 Deploying your changes (remotely)

Squadron will work regardless of how you get your files to your remote servers. If you SCP them over each time and then run squadron apply, it'll work, but that's not very convenient.

The standard way is polling the git repository.

You'll need a git server and then the squadron daemon running on your web server.

Set up git:

```
$ git remote origin add your_origin
$ git add files you changed
$ git commit # automatically runs squadron check for you!
$ git push # deploys!
```

Then set up the daemon:

```
$ squadron daemon
```

It's really that easy. Any node running the Squadron daemon will pull down your changes over the next 30 seconds.

You can configure the poll interval and logging for the daemon using the system config file described in *Global Configuration*.

2.8 More environments

Now that you've tested your website in your development environment, it's time for it to go to production:

```
$ squadron init --env prod --copyfrom dev
Initialized environment prod copied from dev
```

This is another way to initialize environments. It copies all the config from the dev environment to the prod environment. Now we have this in *config*:

```
$ tree -F config
config/
|-- dev
|   `-- website
`-- prod
    `-- website
$ diff -u config/dev/website config/prod/website
$
```

No differences because they're the same!

Let's change our nodes so that nodes can choose to be dev or production:

```
$ cd nodes
$ mv % dev%
$ cat > prod%
{
    "env":"prod",
    "services":["website"]
}
```

Any node whose name begins with dev will get the dev environment, while any node that begins with prod will get the prod environment. This allows you to test your changes before making them live.

CHAPTER 3

Next steps

Now that you have a basic Squadron setup going, it's time to make it do more for you.

If you want to follow along with this guide, we've made a git repo for you so you don't have to type out all these commands.

Just do this and you'll be at the start of the next steps guide (which is the end of the getting started guide):

```
$ git clone -b simple2 https://github.com/gosquadron/example-squadron-repo.git
```

The end result of this page is in the *nextsteps2* branch of the same repo.

3.1 Apache configuration

Now we want our website to be in PHP. So let's look at what we're starting with:

```
$ tree -F
.
|-- config/
|   |-- dev/
|   |   `-- website
|   `-- prod/
|       `-- website
|-- libraries/
|-- nodes/
|   |-- dev%
|   `-- prod%
`-- services/
    `-- website/
        `-- 0.0.1/
            |-- actions
            |-- copy
            |-- defaults
            |-- react
```

```
            |-- root/
            |   |-- main~git
            |   `-- robots.txt~tpl
            |-- schema
            |-- state
            `-- tests/
```

Okay, so let's make a new dev version of our service:

```
$ cp -r services/website/0.0.1 services/website/1.0.0
```

It's important to use semantic versioning for your services, as it communicates vital information to the user. This new version will be backwards incompatible. Let's update our dev environment:

```
$ cat > config/dev/website
{
    "base_dir": "/",
    "config": {
        "disallow":["/secret/*","/admin/*"],
        "release":"master"
    },
    "version": "1.0.0"
}
```

We've changed the *base_dir* to be root because we're going to need to be updating a lot of different paths. We've also increased the version to match our latest version.

3.1.1 New root

Let's make our new root directory:

```
$ cd services/website/1.0.0/root
$ mkdir -p var/www/
$ mv main~git var/www/
$ mv robots.txt~tpl var/www/
```

You can see how this reflects our new *base_dir* of root. It would also be nice if we released our web root atomically so that if anyone happens to load it while we're copying over, they don't get half new and half old assets. Fortunately, this is really easy with Squadron:

```
$ cat > config.sq
var/www/ atomic:true user:nobody group:nobody
```

The *config.sq* file in the *root* directory of a service is special. It's not copied to your *base_dir*, but instead configures some metadata, such as setting the user, group, or mode for a file or directory. For more information, see the *Config.sq* section of the *User Guide*.

What we've done here is to tell Squadron to do an atomic deploy of *var/www/*, which means it will use a symbolic link from /var/www/ to Squadron's deployment directory.

If we wanted *var/www/* to be configurable on a per-environment basis, we could put it into a variable and template the path like so:

```
$ mkdir -p var/@www_location/
$ cat > config.sq
var/@www_location/ atomic:true user:nobody group:nobody
```

Then you could define *www_location* in the configuration or in this service's defauts. You can only template one directory level at a time at the moment.

3.1.2 Apache module

We also need to make sure that PHP is installed:

```
$ cd ..
$ cat > state
[
    {
        "name":"apt",
        "parameters":["apache2", "libapache2-mod-php5"]
    }
]
```

Now we need to run a2enmod when this is installed. We actually need to set up two files for this: *actions* and *react*.

The file *actions* describes the possible actions that can take place. These are commands that are run, such as starting or restarting the service. Ours will look like this:

```
{
    "run a2enmod php": {
        "commands": ["a2enmod php5", "/etc/init.d/apache2 restart"],
    },
    "start" : {
        "commands" : ["/etc/init.d/apache2 start"]
    },
    "reload" : {
        "commands" : ["/etc/init.d/apache2 reload"],
        "not_after" : ["start", "restart"]
    },
    "restart" : {
        "commands" : ["/etc/init.d/apache2 restart"],
        "not_after" : ["start"]
    }
}
```

This file can be written in YAML or JSON.

So we have four actions. Three are easy enough to understand: they control the running of the service. Starting apache, reloading it, and restarting it. The *not_after* property means that if there are several actions to run for a deployment, that these should not be run after successful invocations of those. This will be more clear after understanding *react*.

The file *react* describes how to react to various events. It gives criteria for the events and then which actions to execute. Ours looks like this:

```
[
    {
        "execute": ["run a2enmod php"],
        "when" : {
            "not_exist": "/etc/apache2/mods-enabled/php5"
        }
    },
    {
        "execute": ["start"],
        "when" : {
            "command": "/etc/init.d/apache2 status",
```

```
            "exitcode_not": 0
        }
    },
    {
        "execute" : ["reload"],
        "when" : {
            "files" : ["*.conf", "*/conf.d/*"]
        }
    }
]
```

For a complete description of actions and reactions, see *Action and reaction* in the *User Guide*.

Let's do it:

```
$ sudo squadron apply -n dev
Staging directory: /var/squadron/tmp/sq-8
Processing apache2, libapache2-mod-php5 through apt
Applying changes
Running action website.run a2enmod php in reaction {u'execute': [u'website.run␣
↪a2enmod php'], u'when': {u'not_exist': [u'/etc/apache2/mods-enabled/php5']}}
Module php5 already enabled
* Restarting web server apache2
    apache2: Could not reliably determine the server's fully qualified domain name,␣
↪using 127.0.1.1 for ServerName
... waiting apache2: Could not reliably determine the server's fully qualified domain␣
↪name, using 127.0.1.1 for ServerName   [ OK ]
Apache2 is running (pid 2332).
Successfully deployed to /var/squadron/tmp/sq-8
===============
Paths changed:

New paths:
    website/var/www/main/LICENSE
    website/var/www/main/index.html
    website/var/www/main/README.md
    website/var/www/robots.txt
$ ls -l /var/www
lrwxrwxrwx 1 root root 39 Jan 01 00:00 /var/www -> /var/squadron/tmp/sq-8/website/var/
↪www/
```

And navigating to http://localhost works!

3.1.3 Testing

An important part of deploying software is making sure it's correct. For our purposes, we want to check that PHP is working and that Apache was set up correctly.

In Squadron, *Tests* are located in the service's *tests* directory. Let's make one now:

```
$ mkdir -p services/website/1.0.0/tests
$ cat > services/website/1.0.0/tests/check_php.sh
#!/bin/bash

while read line; do
    true
done
```

```
OUTPUT=`curl http://localhost/main/test.php 2>/dev/null`

if [ "$?" -eq "0" ]; then
    if [[ $OUTPUT == *php* ]]; then
        echo "PHP not enabled"
        exit 1
    fi
else
    echo "Couldn't connect"
    exit 1
fi
```

Tests must read in the JSON object passed via standard in. For our test, we don't care about the configuration, so we just throw it away.

We then test that the connection worked via the exit code flag *$?*. If curl was successful, we check to make sure the output didn't have the string "php" in it, which would indicate that PHP wasn't configured properly.

Almost done. We just need to make sure this test is executable and that curl is installed:

```
$ chmod +x services/website/1.0.0/tests/check_php.sh
$ cat > services/website/1.0.0/state
[
    {
        "name":"apt",
        "parameters":["apache2", "libapache2-mod-php5", "curl"]
    }
]
```

And now we're done. Let's run it:

```
$ sudo squadron apply -n dev
Staging directory: /var/squadron/tmp/sq-11
Processing apache2, libapache2-mod-php5, curl through apt
Running 1 tests for website v1.0.0
Nothing changed.
```

3.2 Keeping state between runs

Squadron keeps a file in the state directory (*/var/squadron/info.json* for some nodes) which describes what the last successful run did. Here is the *info.json* file from our last run:

```
{
  "commit":{
    "website":{
      "version":"1.0.0",
      "config":{
        "release":"master",
        "disallow":[
          "/secret/*",
          "/admin/*"
        ]
      },
      "atomic":{
        "var/www/":true
```

```
      },
      "dir":"/var/squadron/tmp/sq-8/website",
      "base_dir":"/"
    }
  },
  "dir":"/var/squadron/tmp/sq-8",
  "checksum":{
    "website/var/www/main/LICENSE":
↪"3d8f45ba8ca6ebf6e9990f580df8387d49f3e72e9119ff19e63393c12d236aff",
    "website/var/www/main/index.html":
↪"f680e220f5e58408b233b700d0106b70582765937ca983e7969fd9b66dee599e",
    "website/var/www/main/README.md":
↪"0b3b1635d69e0e501e82d9ec70d15d650f17febc4ea3d4a47adbd07a6025a739",
    "website/var/www/robots.txt":
↪"1bb88650e0ac17db58a556033c0e9cda3534902f8c9cef87ffa8ac4ca6e0635f"
  }
}
```

The *commit* block describes what was committed. It is a dictionary of all services, what version was deployed, and what configuration was used. We can see that we deployed version 1.0.0 of our website service description, with the expected configuration. It's also shown that *var/www/* was deployed atomically.

There is also a checksum dictionary which keeps the SHA-256 sum of each file it deploys. If Squadron notices that one of the next run's files has a different SHA-256 sum, it will replace it.

If we try to rerun Squadron it won't reapply anything because nothing tracked by Squadron is different:

```
$ !sudo
sudo squadron apply -n dev
Staging directory: /var/squadron/tmp/sq-9
Processing apache2, libapache2-mod-php5 through apt
Nothing changed.
```

You can grab the completed example for this section by checking out the nextsteps2 branch from the example repo:

```
$ git clone -b nextsteps2 https://github.com/gosquadron/example-squadron-repo.git
```

3.3 Where to go from here

The *User Guide* describes all of the functionality of Squadron. If you're looking for more extension handlers or more state libraries, that's the place to go. You could even write your own.

CHAPTER 4

Examples

Here are examples of various problems solved with Squadron.

4.1 Escape hatch

Do you need to do something that Squadron can't do? Squadron wasn't designed to support every single possible action to deploy your software, so what do you do when you need to do something Squadron doesn't support?

Use an escape hatch!

Let's say you need to decrypt a file using GPG after downloading it. Squadron doesn't support this, so you'll need to do it yourself:

```
$ cd services/escape-hatch/0.0.1/
$ ls -F root/
config.sq  decrypt.sh~tpl  encrypted.sh.gpg~download
```

The file encrypted.sh.gpg will be downloaded from the URL contained therein. And the script decrypt.sh will be used to decrypt it. Let's look at that script:

```
#!/bin/bash
PASSPHRASE="@passphrase"
echo -n "$PASSPHRASE" | gpg -d --passphrase-fd 0 -o run.sh encrypted.sh.gpg
chmod +x run.sh
```

Since this is a template, the variable @passphrase will be supplied by Squadron. We have an action (in *actions.json*)to run decrypt:

```
{
    "decrypt": {
        "commands": ["./decrypt.sh"],
        "chdir": "."
    }
}
```

And a reaction (in *react.json*) to execute it when appropriate:

```
[
    {
        "execute":["decrypt"],
        "when": {
            "files": ["encrypted.sh.gpg"],
            "not_exists": ["run.sh"]
        }
    }
]
```

If you want to see the complete repository, simply checkout the *escape-hatch* branch from the example Squadron repository:

```
git clone -b escape-hatch https://github.com/gosquadron/example-squadron-repo.git
```

4.2 Node.js

Got a Node.js project? Well we've set up an example Squadron repository which does a lot of the setup you'll need to do in your Node.js project. Simply clone the *node* branch of the example Squadron repository:

```
git clone -b node https://github.com/gosquadron/example-squadron-repo.git
```

The best part about this example is the *Action and reaction*. Here is *actions.json*:

```
{
    "install npm deps" : {
        "commands" : ["npm install"],
        "chdir" : "code"
    },
    "rebuild" : {
        "commands" : ["npm rebuild"],
        "chdir" : "code"
    },
    "start" : {
        "commands" : ["forever start index.js"],
        "chdir" : "code"
    },
    "restart" : {
        "commands" : ["forever restart index.js"],
        "chdir" : "code",
        "not_after" : ["start"]
    }
}
```

Contained in *actions.json* is almost everything you'll want to do with your Node.js project: install dependences, rebuild for different platforms, and start or restart your service when appropriate. To go with this is *react.json*:

```
[
    {
        "execute":["rebuild"],
        "when": {
            "always": true
        }
    },
```

```
    {
        "execute":["install npm deps"],
        "when": {
            "files":["code/package.json"],
            "not_exist":["code/node_modules"]
        }
    },
    {
        "execute":["start"],
        "when": {
            "command":"forever list | grep index.js",
            "exitcode_not": 0
        }
    },
    {
        "execute":["restart"],
        "when": {
            "files":["*"]
        }
    }
]
```

This says:

1. Always run npm rebuild in the *code* directory
2. Run npm install in the *code* directory when the *packages.json* file changes, or *code/node_modules/* doesn't exist
3. Start our service if it's not already running
4. If our service was already running (checked via "not_after" in *actions.json*) then restart it if any files were changed.

One more bit is used to make our deployment fast. Using *copy.json* to copy *code/node_modules/* from previous runs so we don't need to redownload all of them every time:

```
[
    {
        "path": "code/node_modules/"
    }
]
```

See *Copying from previous runs* for more information about *copy.json*.

CHAPTER 5

User Guide

This is a reference guide to the various components of Squadron.

5.1 Config.sq

The file *config.sq* is a special file that can be put in the *root/* of a service. It can change the file mode, user or group ownership, and controls whether or not it is deployed atomically.

Here's an example:

```
var/www/ atomic:true user:nobody group:www-data
var/www/@file_path user:root mode:0600

# The init script needs to be +x
etc/init.d/service-script mode:0755
```

Each line describes the file or directory that is being configured. Directories must end in a slash. Spaces or newlines in filenames should be escaped with a backslash. Comments start with a #. This file is templated, so you can use variables such as *@file_path* or conditionals or loops.

In this example, the directory *var/www/* in *root/* will be deployed atomically via symlinks, and it and all of its children with be owned by the *nobody* user and have its group set to *www-data*.

The file *var/www/file.txt* will be in the same atomic deployment as *var/www/*, and will have its user set to *root*, and its group set to *www-data*. Its file mode will be 0600.

The following table describes the options available to config.sq and their description:

Name	Description
atomic	Whether or not to deploy this directory via a symlink, so that it is atomic. Valid values are true or false. Defaults to false.
group	What group to chgrp this file or directory to. If a directory, all children are also set to this group. Default: group of the user running Squadron.
mode	The numeric mode of the file or directory. If a directory, all children are also set to this mode. Defaults to umask.
user	What user to chown this file or directory to. If a directory, all children are also set to this user. Default: the user running Squadron.

5.2 Extensions

Extensions are used in the root directory of a service to do some kind of transformation on them.

5.2.1 dir

The 'dir' extension creates an empty directory of that name.

Contents

None

5.2.2 download

The 'download' extension downloads a file over HTTP.

Contents

A YAML or JSON object of the HTTP endpoint, and optionally a username and password for HTTP BASIC authentication. Will have a template applied to it, so variable substitution and logic is possible.

Examples:

```
{
    "url": "http://example.com/file.txt"
}
```

or, for HTTP BASIC authentication with templating:

```
{
    "url": "http://www.example.com/@{file}.txt",
    "username": "@username",
    "password": "@password"
}
```

The full list of acceptable parameters for the config object are in the following table.

Name	Description
url	URL to download the file from. Required.
username	Username for HTTP BASIC authentication.
password	Password for HTTP BASIC authentication.

5.2.3 extract

The 'extract' extension downloads and extracts tarballs and zip files.

Contents

The extract extension handler takes a YAML or JSON object like this:

```
{
    "url": "http://www.example.com/dir/file.tar.bz2"
}
```

or with manually specifying the type:

```
{
    "url": "https://www.example.com/dir/filename_without_ext",
    "type": "tar.gz"
}
```

The extract handler can also copy files out of the tarball:

```
{
    "url": "https://www.example.com/dir/file.zip",
    "persist": false,
    "copy": [
        {
            "from": "test*",
            "to": "/etc/test/"
        },
        {
            "from": "dir/file.txt",
            "to": "../file.txt"
        }
    ]
}
```

The full list of supported fields is described in the following table.

Name	Description
url	URL to download the tarball from. Required.
username	Username for HTTP BASIC authentication.
password	Passwprd for HTTP BASIC authentication.
type	One of "tar.gz", "tar", "tar.bz2", or "zip". Optional if it can be inferred from filename.
persist	Whether or not to keep around the directory specified by this extension handler. If false, it should be used with copy, as non-copied files will be unavailable. Defaults to true.
copy	An array of copy objects, described in the following table.

The copy objects are described in the following table.

Name	Description
from	A glob match which matches files based on the path relative to the tarball.
to	The destination to copy the files to. If it's an absolute path, it copies it there. If it's a relative path, it's relative to the directory that would have been created by the extension handler if persist was true. Does not create directories.

5.2.4 git

The 'git' extension clones git repositories.

Contents

A YAML or JSON object with properties such as "url". Will have a template applied to it, so variable substitution and logic is possible.

Examples:

```
{
    "url":"https://github.com/gosquadron/squadron.git"
}
```

or:

```
{
    "url":"git@github.com:gosquadron/example-squadron-repo.git",
    "refspec":"experimental"
}
```

or even, in YAML:

```
---
  url: "git@github.com:gosquadron/example-squadron-repo.git"
  refspec: "@release"
  sshkey: "ssh_keys/deploy1"
  args: "--depth=2"
```

The last example requires that the *ssh_keys/deploy1* resource exists and is a private ssh key. See the *Resources* section for more information. It also does a shallow clone of the git repository via the –depth argument.

The properties allowed in the object are described in the following table:

Name	Description
url	URL to clone the git clone from. Required.
refspec	The branch, tag, or commit hash to checkout after clone. Optional.
sshkey	Relative path to the sshkey resource. See the *Resources* section for more information. Optional.
args	Command line arguments to pass to git clone. Optional.

5.2.5 tpl

The template extension simply applies a template to the given file.

Contents

The template is the content.

Example:

```
<html>
    <body>
        <h1>Hello, @user!</h1>
#for @p in @paragraphs:
        <p>@p</p>
#end
```

```
    </body>
</html>
```

5.2.6 virtualenv

Creates a Python virtualenv. The virtualenv and pip commands must be available and in the current user's PATH. Run through a template so variable substitution is possible.

Since it's typically expensive to install a virtualenv, you should copy the previous version with *copy.json*. See *Copying from previous runs* for more information.

Contents

The contents of this file are passed to pip as if they were a requirements.txt file.

Example:

```
Flask==@versions.flask
Jinja2==2.6
Werkzeug==0.8.3
certifi==0.0.8
chardet==1.0.1
distribute==0.6.24
gunicorn==0.14.2
requests==0.11.1
```

5.3 Libraries

Libraries are Python modules which are applied through *state.json*, *state.yml*, or plain old *state* in the service directory for each service. These state files can be written in YAML or JSON.

5.3.1 How to write a library

In the *libraries* directory of your Squadron repository, you can place a Python module.

The Python module should expose three functions:

```
def schema():
    return {}

def verify(inputhashes):
    return []

def apply(inputhashes, dry_run=True):
    return []
```

The schema function should return the Python representation of a JSON schema. It describes one object passed into the verify function.

The verify function takes a list of objects (of the type described in the schema). It then returns a list of objects that are not already in the state specified.

The apply function takes the list of objects that failed verification (weren't yet in the state they were supposed to be in) and a boolean dry_run. It returns a list of objects that couldn't be applied.

5.3.2 Included libraries

Some libraries are included with Squadron so you don't have to write them yourself.

The state.json file is passed through the template engine before being executed, so you can embed logic and variables within it like you would normally.

apt

Installs packages via apt. Takes a list of string names, each string is a package to be installed via apt.

Example state.json with apt:

```
[
    {
        "name":"apt",
        "parameters":["screen","tmux"]
    }
]
```

group

Creates groups. Takes an object with the following fields.

Field	Description
name	Required. Sets the group name
gid	Integer. Specific group id
system	Boolean. Is a system group?

Example state.json with group:

```
[
    {
        "name":"group",
        "parameters":[
            {
                "name": "newgroup"
            },
            {
                "name": "specificgroup",
                "gid": 555,
                "system": true
            }
        ]
    }
]
```

user

Creates users. Takes an object with the following fields.

Field	Description
username	Required. Sets the user name
shell	User's command shell
realname	User's real name
homedir	User's home directory
uid	Integer. Specific user id
gid	Integer. Specific group id
system	Boolean. Is a system user?

Example state.json with user:

```
[
    {
        "name":"user":
        "parameters":[
            {
                "username": "newuser"
            },
            {
                "username": "specificuser",
                "shell":"/bin/bash",
                "homedir":"/users/specificuser"
                "realname":"Specific User"
            },
            {
                "username":"windows",
                "uid":666,
                "system":true
            }
        ]
    }
]
```

5.4 Action and reaction

To perform actions when certain files are created or modified such as restart a service or run a command, you need to first create an action and then create a reaction to trigger it.

5.4.1 Actions

Actions are described in YAML/JSON in *actions.json*, *actions.yml* or just plain *actions* in each service. An action has a name, a list of commands to run, and a list of actions to not run this one after.

Here's what one might look like:

```
{
    "start" : {
        "commands" : ["/etc/init.d/service start"]
    },
    "reload" : {
        "commands" : ["killall -HUP service"],
        "not_after" : ["start", "restart"]
    },
```

```
    "restart" : {
        "commands" : ["/etc/init.d/service restart"],
        "not_after" : ["start"]
    }
}
```

So this service has three actions. The *start* command starts up the service. The *restart* command restarts it, but only if the *start* command didn't just succeed. This way you can avoid restarting a service immediately after starting it.

Here are the possible fields to put in an action:

Field	Description
commands	Required. A list of commands to run
not_after	A list of actions to not run this after

5.4.2 Reactions

Reaction trigger actions in this service or other services based on files being created or modified. The reactions are described in *react.json*, *react.yml* or *react* in each service.

One might look like this:

```
[
    {
        "execute": ["start", "apache2.restart"],
        "when" : {
            "command": "pidof service",
            "exitcode_not": 0
        }
    },
    {
        "execute" : ["restart"],
        "when" : {
            "files" : ["mods-enabled/*"]
        }
    },
    {
        "execute" : ["reload"],
        "when" : {
            "files" : ["*.conf", "conf.d/*"]
        }
    }
]
```

The first reaction starts this service and restarts another service called *apache2* when it's not running.

The second reaction restarts this service if there are any modules created or modified. You can use 'files-created' or 'files-modified' to narrow this scope.

The third reaction reloads this service when any of the config files change.

The executing actions must be defined in *actions.json* or an error will be raised.

Here is a list of fields the top level reaction object can contain:

Field	Description
execute	Required. A list of actions to run
when	Required. An object with fields described below

Here is a list of fields that a *when* object can contain:

Field	Description
command	Command to run, used with exitcode_not. This is passed through a shell, so you can use pipes and shell logic.
exitcode_not	Run action if exit code for command isn't this
files	List. Run if any of these files were created or modified by Squadron. Can be globs
files_created	List. Run if any of these files were created by Squadron. Can be globs
files_modified	List. Run if any of these files were modified by Squadron. Can be globs
always	Boolean. Whether or not to always run. Default: false
not_exist	List of globs/absolute paths to run if these files don't exist

5.5 Copying from previous runs

Squadron keeps around a history of your last few deployments. Since building some aspects of your deployment are expensive (such as installing npm or virtualenv dependencies), you can list which paths to copy from your last successful deployment.

You do this with YAML or JSON in *copy.json*, *copy.yml*, or plain old *copy* file in your service directory. It looks like this:

```
[
    {
        "path": "code/deps/"
    },
    {
        "path": "env/"
    }
]
```

So on each run of Squadron, first it checks if there was a previous version of your deployment that was successfully deployed. If so, it checks if the paths listed existed in the previous deployment. If they did, it copies them recursively.

This happens before *Action and reaction* are triggered. It is important to have an action that installs these dependencies if they weren't copied over, as it can't be guaranteed that there was a previous successful run.

5.6 Resources

Resources are files that are available to multiple services, such as ssh private keys, which allow Squadron to deploy software from a private git server.

Resources are located in the *resources* directory at the top level of Squadron:

```
$ ls -1F
config/
nodes/
resources/
services/
```

And inside *resources* can be any number of subdirectories and files. Like this:

```
$ tree -F resources/
resources/
|-- ssh_keys/
|   |-- deploy1
|   |-- deploy1.pub
|   `-- old_keys/
|       |-- deploy_key
|       `-- deploy_key.pub
|-- other/
|   `-- file
|-- test.sh
`-- this.py
```

So now, in ~git files within your *root* in a service, you can reference these keys by relative path.

Like this:

```
$ cat services/example/0.0.1/root/test~git
{
    "url":"git@example.com:user/repo.git",
    "refspec":"master",
    "sshkey":"ssh_keys/deploy1"
}
```

The ~git extension knows to look in the *resources* directory for the file *ssh_keys/deploy1*, which is the secret key needed to deploy that git repository.

You can also use resources with *Action and reaction*. Just specify the command like this:

```
{
    "run" : {
        "commands" : ["resources/test.sh"]
    },
    "go for it" : {
        "commands" : ["resources/other/file arg1 arg2", "resources/this.py", "touch /
↪tmp/out"]
    }
}
```

This defines two actions. The first, *run*, uses one resource called test.sh. The file resources/test.sh will be extracted to a temporary location, made executable, and then executed with no arguments.

The second action *go for it* defines three commands to run in order. The first two are resources. The first resource will have two command line arguments passed to it.

5.7 Tests

Testing is an important part of configuring software. Tests live in the *tests* directory of each service.

After the service is configured, applied, and the reactions trigger the actions, all executable files in this directory are run.

On standard input, a JSON string is provided which describes the various configuration options for this service. It looks like this:

```
{
    "version": "0.0.1",
    "config": {
        "debug": false,
        "workers": 100
    },
    "atomic": {},
    "dir": "/var/squadrontmp/sq-0/service",
    "base_dir": "/var/service/"
}
```

The test *must* read standard input even if it does not intend to use this information.

Returning a non-zero status code indicates a test failure.

5.8 Global Configuration

Squadron looks for config in the following places:

- /etc/squadron/config
- /usr/local/etc/squadron/config
- ~/.squadron/config
- And in .squadron/config in the Squadron repository itself

Here's an example a config file:

```
[squadron]
basedir = /config/squadron-repo
nodename = override.example.com
statedir = /var/squadron/state

[daemon]
polltime = 600

[status]
send_status = true
status_host = status.gosquadron.com
status_apikey = ABCDEF12345
status_secret = 8d4ce3db954ab1bed870ce682e6765ec24a1227352b3d2688170ecaefda1165c

[log]
infolog = INFO rotatingfile /var/squadron/logs/info.log 50000 10
runlog = DEBUG stream stderr
```

This example configuration does a few things. The squadron configuration repo resides in */config/squadron-repo*, so if it's not overridden by the *-i* flag, Squadron will look there for its configuration. It overrides its hostname to *override.example.com* and sets its between run state directory to */var/squadron/state*.

The daemon polls every 10 minutes, and will send status updates to status.gosquadron.com with the given API and secret key.

Two logs are configured, one which logs to a rotating log file, and another which only uses standard error. Squadron will use both of these log destinations simultaneously.

The description of the sections and their valid configuration items follows.

5.8.1 Daemon

This section configures the daemon.

Name	Description
polltime	How often in seconds that we poll the git repo for changes.

5.8.2 Squadron

This is the main configuration section, which configures most of Squadron's settings. It's used by the daemon and by running Squadron in standalone mode (via the check, apply, or init commands).

Name	Description
basedir	The location of the squadron config directory. Can be overridden with -i.
node-name	Name you want for this server, used to determine which node config applies to this machine. Default is the hostname.
statedir	The directory to keep previous state of squadron.

5.8.3 Status

This section controls the sending of status updates to a Squadron state server.

Name	Description
send_status	Boolean of whether or not to send node status to remote server defined in this section.
status_host	Hostname of where to connect to send status updates to.
status_apikey	API key used with the secret. Provided by Squadron status server admin.
status_secret	Secret hex string to verify identity

5.8.4 Log

This section is a bit different, as you can enter as many lines as you want here so long as they follow the following format defined in the example:

```
debugonly = DEBUG file /tmp/log
```

Which means:

- debugonly - Just an identifier. This log name **must be unique**.
- DEBUG - Level to log must match one of Python's log levels.
- file - Type of log, in this case this is a simple file log.
- /tmp/log - Parameter(s) for the type of log, which is, in this case, the file to log to.

Squadron supports three types of logs:

file:

- expects file to log to as parameter

stream:

- expects stdout or stderr as the parameter

rotatingfile:

- file to log to
- max file size in bytes
- max number of files to backup

Example of rotating file located at /var/log/squadron/log, which rotates every 5000 bytes and keeps two files in backup:

```
rotate = DEBUG rotatingfile /var/log/squadron/log 5000 2
```

5.9 Github Webhooks

If you'd like to trigger the squadron daemon sooner than polling, you can hook up Github Webhooks. You'll need to add this to your Squadron config file:

```
[webhook]
webhook = true
webhook_username = authusername
webhook_password = authpassword
webhook_listen = 0.0.0.0
webhook_port = 8888
```

You should modify the username and password to be random so that attackers cannot trigger Squadron on demand. You can also choose whatever port you would like.

Then run the squadron daemon like normal:

```
$ squadron daemon
Starting webhook server on 0.0.0.0:8888
```

Squadron will still poll for changes for however often you have set the polltime to in the configuration. However, now if Github (or anyone else, really) POSTs a Github webhook to that port, Squadron will restart.

For help on setting up Github webhooks for your repository, see this Github documentation page: Creating Webhooks.