
Squadron Documentation

Release 0.3.1

Squadron Leader

March 12, 2014

Contents:

Getting Started

Squadron configures your service. It install packages, writes out templates, and tests them.

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 if you don't feel like it.

Just do this and you'll be at the end of the first part of the getting started guide:

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

1.1 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

From there it reads the location of its state directory.

1.2 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!

1.3 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.json
  -- defaults.json
  -- react.json
  -- root/
  -- schema.json
  -- state.json
  -- 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.json` to install `apache2` for our simple website:

```
{
  "apt": ["apache2"]
}
```

Now when we later run `squadron`, it'll make sure that Apache is installed.

1.3.1 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

1.3.2 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.json`, which is prepopulated 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 JSON 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.json`. 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.json` for our website would be:

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

1.3.3 Schema

Squadron includes one very useful file with every service description called `services/website/0.0.1/schema.json`. 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.

1.4 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.

1.5 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
```

1.6 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.
```

1.7 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*.

1.8 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.json
|-- prod
|   |-- website.json
$ diff -u config/dev/website.json config/prod/website.json
$
```

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.

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.

2.1 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.json
|   |-- prod/
|       |-- website.json
|-- libraries/
|-- nodes/
|   |-- dev%
|   |-- prod%
|-- services/
|   |-- website/
|       |-- 0.0.1/
|           |-- actions.json
|           |-- defaults.json
|           |-- react.json
|           |-- root/
|               |-- main~git
|               |-- robots.txt~tpl
|           |-- schema.json
|           |-- state.json
|           |-- 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.json
{
  "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.

2.1.1 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.

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.

2.1.2 Apache module

We also need to make sure that PHP is installed:

```
$ cd ..
$ cat > state.json
{
  "apt": ["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.json* and *react.json*.

The file *actions.json* describes the possible actions that can take place. These are commands that are run. Sometimes restarting the service, sometimes starting it. 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"]
  }
}
```

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.json*.

The file *react.json* 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: /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 127.0.1.1:80
... waiting apache2: Could not reliably determine the server's fully qualified domain name, using 127.0.1.1:80
```

```
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!

2.1.3 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.json
{
  "apt": ["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.
```

2.2 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":"f680e220f5e58408b233b700d0106b70582765937ca983e7969fd9b66dee5",
    "website/var/www/main/README.md":"0b3b1635d69e0e501e82d9ec70d15d650f17febc4ea3d4a47adb07a6025a7",
    "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
```

2.3 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.

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

3.1 Extensions

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

3.1.1 dir

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

Contents

None

3.1.2 download

The 'download' extension downloads a file over HTTP.

Contents

A single line of the HTTP endpoint with an optional SHA256 prefix hash of the file. Will have a template applied to it, so variable substitution and logic is possible.

Examples:

```
http://www.example.com/filename.ext a7898bc
```

or:

```
http://no.sha256.here.com/file.to.download.txt
```

3.1.3 extract

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

Contents

The extract extension handler takes a 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.
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.

3.1.4 git

The ‘git’ extension clones git repositories.

Contents

A 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:

```
{
  "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 <i>Resources</i> section for more information. Optional.
args	Command line arguments to pass to git clone. Optional.

3.1.5 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>
```

3.1.6 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.

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
```

3.2 Libraries

Libraries are Python modules which are applied through *state.json*.

3.2.1 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(inpuhashes):
    return []

def apply(inpuhashes, 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.

3.2.2 Included libraries

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

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:

```
{
  "apt": ["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:

```
{
  "group": [
    {
      "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:

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

3.3 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.

3.3.1 Actions

Actions are described in *actions.json* 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

3.3.2 Reactions

Reaction trigger actions in this service or other services based on files being created or modified. The reactions are described in *react.json* 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 <i>exitcode_not</i>
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

3.4 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 -lF
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/
    `-- script.sh

```

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
http://example.com/repo.git master 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.

3.5 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.

3.6 Global Configuration

Squadron keeps the global config in the default location `/home/user/.squadron` If you are root the default is `/etc/squadron`

It also looks for config in the following places:

- /etc/squadron/config
- /usr/local/etc/squadron/config
- ~/.squadron/config

Let's go over some of the configuration values and sections:

3.6.1 Daemon

- polltime - frequency in seconds that we check the git repo for changes

3.6.2 Squadron

- keydir - where we store any ssh keys
- nodename - name you want for the node, used to determine which node config applies to this machine
- statedir - directory to keep previous state of squadron
- send_status - whether or not to send node status to remote server defined in [status] section

3.6.3 Status

- status_host where to send status to
- status_apikey - key used for identity
- status_secret - shared secret to verify identity

3.6.4 Log

This section is a bit special, 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
```

- debugonly - just a friendly name, not used for anything **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, in this case the file to log

to

We support three types of logs at the moment

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: rotate = DEBUG rotatingfile /tmp/rot 500 2