School Inspector Documentation

Release 0.1

Code for Durham

September 23, 2014

Contents

Contents:

CHAPTER 1

Server Setup

1.1 Provisioning

The server provisioning is managed using Salt Stack. The base states are managed in a common repo and additional states specific to this project are contained within the `conf` directory at the root of the repository.

For more information see the doc:*provisioning guide </provisioning>*.

1.2 Layout

Below is the server layout created by this provisioning process:

```
/var/www/school_inspector/
    source/
    env/
    log/
    public/
        static/
        media/
    ssl/
```

`source` contains the source code of the project. `env` is the virtualenv for Python requirements. `log` stores the Nginx, Gunicorn and other logs used by the project. `public` holds the static resources (css/js) for the project and the uploaded user media. `public/static/` and `public/media/` map to the `STATIC_ROOT` and `MEDIA_ROOT` settings. `ssl` contains the SSL key and certificate pair.

1.3 Deployment

For deployment, each developer connects to the Salt master as their own user. Each developer has SSH access via their public key. These users are created/managed by the Salt provisioning. The deployment itself is automated with Fabric. To deploy, a developer simply runs:

```
# Deploy updates to staging
fab staging deploy
# Deploy updates to production
fab production deploy
```

This runs the Salt highstate for the given environment. This handles both the configuration of the server as well as updating the latest source code. This can take a few minutes and does not produce any output while it is running. Once it has finished the output should be checked for errors.

CHAPTER 2

Server Provisioning

2.1 Overview

School_Inspector is deployed on the following stack.

- OS: Ubuntu 12.04 LTS
- Python: 3.4
- Database: Postgres 9.1
- Application Server: Gunicorn
- Frontend Server: Nginx
- Cache: Memcached

These services can configured to run together on a single machine or on different machines. Supervisord manages the application server process.

2.2 Initial Setup

Before your project can be deployed to a server, the code needs to be accessible in a git repository. Once that is done you should update `conf/pillar/<environment>/env.sls` to set the repo and branch for the environment. E.g., change this:

```
# FIXME: Update to the correct project repo
repo:
  url: git@github.com:CHANGEME/CHANGEME.git
  branch: master
```

to this:

```
repo:
  url: git@github.com:codefordurham/school-inspector.git
  branch: master
```

The repo will also need a deployment key generated so that the Salt minion can access the repository. You can generate a deployment key locally for the new server like so:

```
ssh-keygen -t rsa -b 4096 -f <servername>
```

This will generate two files named `<servername>` and `<servername>.pub`. The first file contains the private key and the second file contains the public key. The public key needs to be added to the "Deploy keys" in the GitHub repository. For more information, see the Github docs on managing deploy keys: https://help.github.com/articles/managing-deploy-keys

The text in the private key file should be added to *conf/pillar/<environment>/secrets.sls*' under the label *github_deploy_key*, e.g.:

```
github_deploy_key: |
  -----BEGIN RSA PRIVATE KEY-----
  foobar
  -----END RSA PRIVATE KEY-----
```

There will be more information on the secrets in a later section. You may choose to include the public SSH key inside the repo itself as well, but this is not strictly required.

You also need to set `project_name` and `python_version` in `conf/pillar/project.sls`. Currently we support using Python 2.7 or Python 3.3. The project template is set up for 2.7 by default. If you want to use 3.3, you will need to change `python_version` and make a few changes to requirements. In `requirements/production.txt`, change python-memcached to python3-memcached. In `requirements/dev.txt`, remove Fabric and all its dependencies. Instead you will need Fabric installed on your laptop "globally" so that when you run `fab`, it will not be found in your virtualenv, but will then be found in your global environment.

For the environment you want to setup you will need to set the `domain` in `conf/pillar/<environment>/env.sls`.

You will also need add the developer's user names and SSH keys to `conf/pillar/devs.sls`. Each user record (under the parent `users:` key) should match the format:

```
example-user:
  public_key:
   - ssh-rsa <Full SSH Public Key would go here>
```

Additional developers can be added later, but you will need to create at least one user for yourself.

2.3 Managing Secrets

Secret information such as passwords and API keys should never be committed to the source repository. Instead, each environment manages its secrets in `conf/pillar/<environment>/secrets.sls`. These `secrets.sls` files are excluded from the source control and need to be passed to the developers out of band. There are example files given in `conf/pillar/<environment>/secrets.ex`. They have the format:

```
secrets:
  DB_PASSWORD: XXXXXX
```

Each key/value pair given in the `secrets` dictionary will be added to the OS environment and can retrieved in the Python code via:

```
import os

password = os.environ['DB_PASSWORD']
```

Secrets for other environments will not be available. That is, the staging server will not have access to the production secrets. As such there is no need to namespace the secrets by their environment.

2.4 Environment Variables

Other environment variables which need to be configured but aren't secret can be added to the `env` dictionary in `conf/pillar/<environment>/env.sls`:

> # Additional public environment variables to set for the project env:
>
> FOO: BAR

For instance the default layout expects the cache server to listen at `127.0.0.1:11211` but if there is a dedicated cache server this can be changed via `CACHE_HOST`. Similarly the `DB_HOST/DB_PORT` defaults to `''/''`:

```
env:
  DB_HOST: 10.10.20.2
  CACHE_HOST: 10.10.20.1:11211
```

2.5 Setup Checklist

To summarize the steps above, you can use the following checklist

- `repo` is set in `conf/pillar/<environment>/env.sls`
- Developer user names and SSH keys have been added to `conf/pillar/devs.sls`
- Project name has been set in `conf/pillar/project.sls`
- Environment domain name has been set in `conf/pillar/<environment>/env.sls`
- Environment secrets including the deploy key have been set in `conf/pillar/<environment>/secrets.sls`

2.6 Salt Master

Each project needs to have at least one Salt Master. There can be one per environment or a single Master which manages both staging and production. The master is configured with Fabric. You will need to be able to connect to the server as a root user. How this is done will depend on where the server is hosted. VPS providers such as Linode will give you a username/password combination. Amazon's EC2 uses a private key. These credentials will be passed as command line arguments.:

```
# Template of the command
fab -H <fresh-server-ip> -u <root-user> setup_master
# Example of provisioning 33.33.33.10 as the Salt Master
fab -H 33.33.33.10 -u root setup_master
# Example DO setup
fab -H 107.170.136.182 -u root setup_master
# Example AWS setup
fab -H 54.86.14.136 -u ubuntu -i ~/.ssh/aws-cfa.pem setup_master
```

This will install salt-master and update the master configuration file. The master will use a set of base states from https://github.com/caktus/margarita using the gitfs root. Once the master has been provisioned you should set:

```
env.master = '<ip-of-master>'
```

in the top of the fabfile.

If each environment has its own master then it should be set with the environment setup function `staging` or `production`. In these case most commands will need to be preceded with the environment to ensure that `env.master` is set.

Additional states and pillar information are contained in this repo and must be rsync'd to the master via:

```
fab -u <root-user> sync
```

This must be done each time a state or pillar is updated. This will be called on each deploy to ensure they are always up to date.

To provision the master server itself with salt you need to create a minion on the master:

```
fab -H <ip-of-new-master> -u <root-user> --set environment=master setup_minion:salt-master
fab -u <root-user> accept_key:<server-name>
fab -u <root-user> --set environment=master deploy
# Example DO (may have to run a second time to catch key)
fab -H 107.170.136.182 -u root --set environment=master setup_minion:salt-master
fab -H 107.170.136.182 -u root --set environment=master deploy
# Example AWS setup
fab -H 54.86.14.136 -u ubuntu -i ~/.ssh/aws-cfa.pem --set environment=master setup_minion:salt-master
fab -H 54.86.14.136 -u ubuntu -i ~/.ssh/aws-cfa.pem --set environment=master deploy
```

This will create developer users on the master server so you will no longer have to connect as the root user.

2.7 Provision a Minion

Once you have completed the above steps, you are ready to provision a new server for a given environment. Again you will need to be able to connect to the server as a root user. This is to install the Salt Minion which will connect to the Master to complete the provisioning. To setup a minion you call the Fabric command:

```
fab <environment> setup_minion:<roles> -H <ip-of-new-server> -u <root-user>
fab staging setup_minion:web,balancer,db-master,cache -H  33.33.33.10 -u root
# Example DO
fab production setup_minion:web,balancer,db-master,cache,queue,worker -H 107.170.136.182
# Example AWS setup
fab production setup_minion:web,balancer,db-master,cache,queue,worker -H 54.86.14.136
```

The available roles are `salt-master`, `web`, `worker`, `balancer`, `db-master`, `queue` and `cache`. If you are running everything on a single server you need to enable the `web`, `balancer`, `db-master`, and `cache` roles. The `worker` and `queue` roles are only needed to run Celery which is explained in more detail later.

Additional roles can be added later to a server via `add_role`. Note that there is no corresponding `delete_role` command because deleting a role does not disable the services or remove the configuration files of the deleted role:

```
fab add_role:web -H  33.33.33.10
```

After that you can run the deploy/highstate to provision the new server:

```
fab <environment> deploy
```

The first time you run this command, it may complete before the server is set up. It is most likely still completing in the background. If the server does not become accessible or if you encounter errors during the process, review the Salt logs for any hints in `/var/log/salt` on the minion and/or master. For more information about deployment, see the *server setup </server-setup>* documentation.

2.8 Optional Configuration

The default template contains setup to help manage common configuration needs which are not enabled by default.

2.8.1 HTTP Auth

The `secrets.sls` can also contain a section to enable HTTP basic authentication. This is useful for staging environments where you want to limit who can see the site before it is ready. This will also prevent bots from crawling and indexing the pages. To enable basic auth simply add a section called `http_auth` in the relevant `conf/pillar/<environment>/secrets.sls`:

```
http_auth:
  admin: 123456
```

This should be a list of key/value pairs. The keys will serve as the usernames and the values will be the password. As with all password usage please pick a strong password.

2.8.2 Celery

Many Django projects make use of Celery for handling long running task outside of request/response cycle. Enabling a worker makes use of Django setup for Celery. As documented you should create/import your Celery app in `school_inspector/__init__.py` so that you can run the worker via:

```
celery -A school_inspector worker
```

Additionally you will need to configure the project settings for Celery:

```
# school_inspector.settings.staging.py
import os
from school_inspector.settings.base import *

# Other settings would be here
BROKER_URL = 'amqp://school_inspector_staging:%(BROKER_PASSWORD)s@%(BROKER_HOST)s/school_inspector_st
```

You will also need to add the `BROKER_URL` to the `school_inspector.settings.production` so that the vhost is set correctly. These are the minimal settings to make Celery work. Refer to the Celery documentation for additional configuration options.

`BROKER_HOST` defaults to `127.0.0.1:5672`. If the queue server is configured on a separate host that will need to be reflected in the `BROKER_URL` setting. This is done by setting the `BROKER_HOST` environment variable in the `env` dictionary of `conf/pillar/<environment>/env.sls`.

To add the states you should add the `worker` role when provisioning the minion. At least one server in the stack should be provisioned with the `queue` role as well. This will use RabbitMQ as the broker by default. The RabbitMQ user will be named school_inspector_<environment> and the vhost will be named school_inspector_<environment> for each environment. It requires that you add a password for the RabbitMQ user to each of the `conf/pillar/<environment>/secrets.sls`:

```
secrets:
  BROKER_PASSWORD: thisisapasswordforrabbitmq
```

The worker will run also run the `beat` process which allows for running periodic tasks.

CHAPTER 3

Vagrant Testing

3.1 Starting the VM

You can test the provisioning/deployment using Vagrant. This requires Vagrant 1.3+. The Vagrantfile is configured to install the Salt Master and Minion inside the VM once you've run `vagrant up`. The box will be installed if you don't have it already.:

```
vagrant up
```

The general provision workflow is the same as in the previous *provisioning guide* so here are notes of the Vagrant specifics.

3.2 Provisioning the VM

Set your environment variables and secrets in `conf/pillar/local.sls`. It is OK for this to be checked into version control because it can only be used on the developer's local machine. To finalize the provisioning you simply need to run:

```
fab vagrant deploy
```

The Vagrant box will use the current working copy of the project and the local.py settings. If you want to use this for development/testing it is helpful to change your local settings to extend from staging instead of dev:

```
# Example local.py
from school_inspector.settings.staging import *

# Override settings here
DATABASES['default']['NAME'] = 'school_inspector_local'
DATABASES['default']['USER'] = 'school_inspector_local'

DEBUG = True
```

This won't have the same nice features of the development server such as auto-reloading but it will run with a stack which is much closer to the production environment. Also beware that while `conf/pillar/local.sls` is checked into version control, `local.py` generally isn't, so it will be up to you to keep them in sync.

3.3 Testing on the VM

With the VM fully provisioned and deployed, you can access the VM at the IP address specified in the `Vagrantfile`, which is 33.33.33.10 by default. Since the Nginx configuration will only listen for the domain name in `conf/pillar/local.sls`, you will need to modify your `/etc/hosts` configuration to view it at one of those IP addresses. I recommend 33.33.33.10, otherwise the ports in the localhost URL cause the CSRF middleware to complain `REASON_BAD_REFERER` when testing over SSL. You will need to add:

```
33.33.33.10 <domain>
```

where `<domain>` matches the domain in `conf/pillar/local.sls`. For example, let's use dev.example.com:

```
33.33.33.10 dev.example.com
```

In your browser you can now view https://dev.example.com and see the VM running the full web stack.

Note that this `/etc/hosts` entry will prevent you from accessing the true dev.example.com. When your testing is complete, you should remove or comment out this entry.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*