# Chapter 5. An Example Application Stack

Because this book covers how to run a production application in AWS, it is useful to have an example application stack that can demonstrate the various principles and strategies introduced in later chapters. Therefore, this chapter describes how to plan an application deployment and gradually builds up a working application using AWS components. As the chapter progresses, we will create Puppet manifests and a CloudFormation stack template that, when combined, can be used to deploy the entire application as a single self-contained stack.

By the end of this chapter, you will have a web-based application running in AWS, provisioned using CloudFormation and Puppet. This application consists of multiple services: a web application, a background task-processing service, a database, and a caching service.

This stack is something Mike has deployed many times, and is similar to those used by popular websites like Pinterest. You can, of course, replace the components we describe with your own application’s components. This chapter is concerned with general processes you must go through when planning a new application, rather than the specific technologies used in this stack.

# Overview of Application Components

This example deploys a content management system (CMS). A CMS provides a GUI-based interface for bloggers, journalists, and others to create and update web pages without requiring knowledge of HTML or other markup languages.

The infrastructure required to operate a CMS (or indeed, most web-based applications) consists of these components:

| **Component** | **Role** |
| --- | --- |
| Application layer | Handle incoming HTTP requests |
| Task-processing layer | Perform scheduled and ad hoc application tasks |
| Database layer | Provide persistent storage |
| Caching layer | Provide temporary quick-access storage |

We’ll build the infrastructure in this chapter using a combination of AWS services. The application and task-processing layers will be handled by EC2 instances. The database layer will use AWS RDS (a hosted database service), and the caching layer will use AWS ElastiCache.

## The Web Application

We will use the open source [Mezzanine](http://mezzanine.jupo.org/) CMS, which is based on [Django](https://www.djangoproject.com/), a Python web development framework. We chose Mezzanine because it provides a good compromise between ease of installation and customization: we get a working CMS out of the box, without having to spend too much time installing it.

The Mezzanine application will be served by the Nginx web server because it offers better performance than Mezzanine’s built-in HTTP server.

## Database and Caching

This application requires database and caching servers, which usually means installing software such as MySQL or Memcache. Amazon provides services that act as replacements for these software packages. Instead of running your own MySQL database, you can use Amazon’s Relational Database Service (RDS).

Memcache can be replaced with ElastiCache, which is a protocol-compatible replacement: that is, any valid Memcache client will work with ElastiCache without any modification.

The point of this chapter is to get an application up and running, not to spend time installing and configuring software. Therefore, the example application stack will use RDS and ElastiCache instead of installing the corresponding standalone software packages.

## Background Task Processing

Many applications require some form of background task processing. We want our websites to be as fast as possible from the user’s perspective, so waiting around for slow tasks to complete is not an option. Today’s web applications rarely live in isolation, and it is common for one website to interact with many others through the use of external API requests.

For example, your website might give users the opportunity to invite their friends from social networks such as Facebook or Twitter. This requires API requests to query these services for a list of the users’ friends, and to send out the invitations.

These API requests can take some time to complete, and making the user wait around until they do so does not make a good user experience. The best practice in this case is to move long-running tasks out of the HTTP request/response cycle and into a dedicated background processing application. This way, the tasks can be processed asynchronously. From the user’s perspective, the action is completed immediately, but, in fact, all the work happens in another process.

For blogs that accept comments from users, there is another popular use case for asynchronous task processing: anti-spam checks. We want to keep our blog’s comments spam-free, which means every new comment must be checked by an anti-spam service.

Our example blog will use background task processing to check posted comments for spam. For this application, we will use [Celery](http://www.celeryproject.org/), a distributed task queue application written in Python. Celery works extremely well with Django, and is the de facto standard task processing application for many Django and Python developers.

Celery works by sending messages between your application and the processes that are actually executing your task. It requires a message broker to store these messages. One of the most common (and most efficient) Celery message brokers is RabbitMQ, which operates using the Advanced Message Queueing Protocol (AMQP).

Celery can also work with Amazon’s Simple Queueing Service (SQS), which is a highly scalable message-queueing service. SQS can act as a replacement to tools like RabbitMQ. By using SQS as our Celery message broker, we do not need to install or maintain a RabbitMQ cluster.

In this architecture, we have to install and maintain only a single EC2 instance, which runs both the web application and the Celery task-processing application. The rest of the services are provided by AWS.

# Installing the Web Application

The first step is to launch and configure the EC2 instance that will run the web and task-processing applications. Once it is configured and working properly, create an Amazon Machine Image so it can be used as part of a CloudFormation stack.

First, we will install the software manually, and then “translate” these manual steps into Puppet manifests.

Start by creating a new security group named web. This security group should allow inbound TCP traffic from 0.0.0.0/0 on ports 8000 and 80.

$ **aws ec2 create-security-group --group-name web --description "global web \**

**server access"**

{

"GroupId": "sg-b7d15acd"

}

$ **aws ec2 authorize-security-group-ingress--group-name web --protocol tcp \**

**--port 80 --cidr 0.0.0.0/0**

$ **aws ec2 authorize-security-group-ingress**

**--group-name web --protocol tcp \**

**--port 8000 --cidr 0.0.0.0/0**

$ **aws ec2 describe-security-groups --group-names web**

SECURITYGROUPS global webserver access sg-b7d15acd web 740376006796 vpc-934935f7

IPPERMISSIONS 80 tcp 80

IPRANGES 0.0.0.0/0

IPPERMISSIONS 8000 tcp 8000

IPRANGES 0.0.0.0/0

IPPERMISSIONSEGRESS -1

IPRANGES 0.0.0.0/0

Next, launch an EC2 instance using the Ubuntu 14.04 AMI, making sure that this instance is a member of the web and ssh security groups. Once the instance has launched, log in with SSH.

$ **aws ec2 run-instances --image-id *ami-c80b0aa2* --region us-east-1 \**

**--key *federico* --security-groups ssh web --instance-type t2.micro**

[output truncated]

SECURITYGROUPS sg-4ebd8b36 ssh

SECURITYGROUPS sg-b7d15acd web

STATE 0 pending

STATEREASON pending pending

Mezzanine is a Python package available from [PyPI](https://pypi.python.org/pypi), Python’s package management system. Python packages can be installed with the pip command. First, we need to install pip itself in our new instance, along with the Python development libraries:

sudo apt install python-pip python-dev

sudo apt install libtiff5-dev libjpeg8-dev zlib1g-dev libfreetype6-dev

Once this is done, Mezzanine itself can be installed with pip. This will also install Django, as well as any other package dependencies required to run Mezzanine:

sudo pip install Mezzanine

We need to create a directory to store the files that make up the Mezzanine project. For security purposes, we will create a new user to own this directory:

sudo useradd mezzanine

sudo mkdir /srv/mezzanine

sudo chown mezzanine /srv/mezzanine

sudo -u mezzanine mezzanine-project myblog /srv/mezzanine

cd /srv/mezzanine

Unless otherwise configured, Mezzanine will use [SQLite](http://www.sqlite.org/) as its database. SQLite is a self-contained database engine: unlike MySQL or PostgreSQL, it does not require a dedicated database server. Instead, the entire database is contained in a C library file, which can be embedded into a compiled program.

Because database access is not mediated by a server program, concurrent writes are not SQLite’s forte. This makes it unsuitable for many production applications, but SQLite remains a great choice for development, as there is no need to spend time setting up a database server when beginning the project. So we will use SQLite to make sure Mezzanine is working, before moving to an RDS-based database.

Mezzanine provides a createdb command that initializes the database and populates it with some example pages. The command will also prompt you to create a superuser account, which is required to access the admin interface of the site and begin making changes. Make sure to have the public DNS name of the instance, then execute the following, binding to the public DNS and port 8000 when prompted:

$ **sudo -u mezzanine python manage.py createdb**

[output truncated]

A site record is required.

Please enter the domain and optional port in the format 'domain:port'.

For example 'localhost:8000' or 'www.example.com'.

Hit enter to use the default (127.0.0.1:8000): **ec2-54-172-21-112.compute-1.amazonaws.com:8000**

Creating default site record: ec2-54-172-21-112.compute-1.amazonaws.com:8000 ...

Creating default account ...

Username (leave blank to use 'mezzanine'):

Email address: **federico@ubuntu.com**

Password: **••••••**

Password (again): **••••••**

Superuser created successfully.

Installed 2 object(s) from 1 fixture(s)

Would you like to install some initial demo pages?

Eg: About us, Contact form, Gallery. (yes/no): **yes**

Creating demo pages: About us, Contact form, Gallery ...

Installed 16 object(s) from 3 fixture(s)

Once this command is complete, Mezzanine is ready to run.

###### WARNING

If you have to interrupt Mezzanine’s createdb command because of an error entering configuration settings, you will need to remove the file /srv/mezzanine/dev.db first to be able to run the command again. Mezzanine has built-in safeguards designed to prevent accidental database overwrites, something you will come to appreciate in highly automated environments.

To ease development, Django has a built-in HTTP server that can be used to quickly test pages, without having to set up Apache or Nginx. While in the /srv/mezzanine/ directory, the server can be started as follows:

sudo -u mezzanine python manage.py runserver 0.0.0.0:8000

Open your browser and visit the public IP address of your EC2 instance on port 8000—for example, http://ec2-54-172-21-112.compute-1.amazonaws.com:8000/-do not use SSL, the built-in server does not support it. You should now see the Mezzanine welcome page, welcoming you to your newly created website. You could validate the CMS system’s functionality by logging into the administrative interface and publishing a short test blog.

The Django development server is not fast enough for production use, but saves plenty of time in the development phase. In production, a Web Server Gateway Interface (WSGI) server such as Gunicorn is used to serve the Python application, and traffic is proxied by a web server such as Nginx or Apache. These servers are much better at dealing with higher numbers of concurrent users, while still providing fast responses to user requests.

To make this example more closely match a production environment, we will set up Nginx and configure it to serve the Mezzanine blog application instead of using Django’s development server. In this configuration, Nginx acts as a proxy to the actual application. The user’s request is received by Nginx, which then forwards the request to the application server before returning the result to the client.

Nginx can communicate with the application in a few ways, two of the most popular being HTTP and Unix sockets. Unix sockets can offer improved performance over HTTP, but they require Nginx and the application to run on the same physical server (or virtual server instance, in the case of AWS). Using HTTP to communicate with the proxy involves a little more overhead—network sockets must be created, for example—but allows Nginx and the application to run on separate servers, increasing resilience and scalability.

Install Nginx with the following:

sudo apt install nginx

###### TIP

If any apt sofware install ends unsuccessfully, your first check should be for stale metadata: run the following and try the install command once more:

sudo apt update

Remove the link to the placeholder configuration file found in /etc/nginx/sites-available/default -as indicated in its comments, one may keep it as a reference:

sudo unlink /etc/nginx/sites-enabled/default

[Example 5-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#nginx_conf) shows a simple Nginx virtual host definition. This configures Nginx to act as a proxy server and relay traffic to an upstream application server, running on port 8000 on the same host.

##### ***Example 5-1. Nginx configuration***

upstream myblog\_app {

server localhost:8000;

}

server {

listen \*:80 default;

server\_name blog.example.com;

access\_log /var/log/nginx/blog.example.com.access.log;

location / {

proxy\_pass http://myblog\_app;

proxy\_read\_timeout 90;

proxy\_set\_header Host $http\_host;

}

}

Save this configuration to /etc/nginx/sites-available/myblog.conf. Soft-link that file from /etc/nginx/sites-enabled/myblog.conf and restart Nginx:

sudo ln -s /etc/nginx/sites-available/myblog.conf /etc/nginx/sites-enabled/myblog.conf

sudo service nginx restart

Make sure the Django development server is not already running and then start it:

sudo -u mezzanine python manage.py runserver

Without parameters, Mezzanine will bind to localhost only, port 8000. Now visit the public hostname of the EC2 instance in your browser again, this time on port 80. You should again see the Mezzanine welcome page, but this time it is being served by Nginx instead of Django’s development server.

Of course, running the Django development server manually is inconvenient. We don’t want to have to start it manually every time the server starts, nor do we want to have to restart it if it crashes. Therefore, stop the server and turn to our next step, which is a superserver that starts and monitors other processes.

[Supervisor](http://supervisord.org/) is a process-control system that can help solve this problem. It will automatically start processes when the instance is launched, and will restart them if they crash unexpectedly. Supervisor is just one example of many tools that perform a similar function. It can be installed as follows:

sudo apt install supervisor

[Example 5-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#supervisor_config) shows the Supervisor configuration file required to start our Django development server. This file provides all the information Supervisor needs to run the server. The process will be started automatically when the instance boots, and it will be automatically restarted if it crashes unexpectedly.

##### ***Example 5-2. Supervisor configuration file***

[program:myblog\_app]

command=/usr/bin/python /srv/mezzanine/manage.py runserver

autostart=true

autorestart=unexpected

stopwaitsecs=10

stopasgroup=true

killasgroup=true

user=mezzanine

Save this file to /etc/supervisor/conf.d/myblog\_web.conf and issue the sudo supervisorctl update command, instructing Supervisor to read and process the new configuration file. Make sure you stop the manually launched development server before doing this. Otherwise, the Supervisor-launched process will not start correctly because it will be unable to bind to port 8000 which is already in use.

$ **sudo supervisorctl update**

myblog\_app: added process group

$ **sudo supervisorctl status**

myblog\_app STARTING

$ **sudo supervisorctl status**

myblog\_app RUNNING pid 13012, uptime 0:00:10

Confirm that everything is working by reloading the welcome page in your web browser. Once more, the page should be displayed—only this time, the Django development server process is being managed by Supervisor. When the instance starts, the development server will be started. If the server process crashes for some reason, it will be automaticallyrestarted.

# Preparing Puppet and CloudFormation

Now that the server is configured correctly, we can retrace our steps and convert this manual process into a Puppet manifest. We will also begin creating the CloudFormation stack that describes all the EC2 resources required to provision the application.

## Puppet Files

Let’s first recap the steps we took:

1. Install some packages from Apt and pip repositories.
2. Create a directory to store the application, and a user to own the application files.
3. Initialize the application and database.
4. Create configuration files for Nginx and Supervisor.

To make this a repeatable process across any number of AWS instances, we will use a Puppet module that performs all of these configuration steps. We will call this module myblog. The Puppet Style Guide recommends that modules consist of multiple classes, each responsible for a subset of the module’s functionality. Therefore, the logic to complete the preceding tasks will be spread across multiple classes:

* Installing the Apt and pip requirements will be handled by themyblog::requirements class.
* The logic specific to the web application server will be contained in myblog::web.
* Later, when we add the Celery server to the stack, its Puppet configuration will be handled by the myblog::celery class.

Because both the web and Celery servers have the same basic set of requirements, both of these classes can include the myblog::requirements class instead of duplicating the requirements list.

To save time, we will use modules from the Puppet Forge where possible; this saves us from having to reinvent the wheel. Puppet modules are available for Nginx and Supervisor, and the rest of the configuration can be handled with Puppet’s built-in capabilities.

Begin by creating a new repository in your revision control system and setting up the initial directory structure for Puppet’s configuration files. We will be using Git for these examples.

sudo apt install git

git config --global user.name *"Federico Lucifredi"*

git config --global user.email *federico@ubuntu.com*

git init ~/myblog

cd ~/myblog

mkdir -p puppet/{manifests,modules}

mkdir puppet/modules/myblog

###### NOTE

The [Puppet Forge](https://forge.puppetlabs.com/) is a repository of reusable Puppet modules that you can use in your Puppet manifests. Many of these are incredibly high quality, and will give you a lot of control over how the underlying software or service is configured. You will find modules for a huge range of open source and proprietary software, as well as physical devices such as Juniper network switches.

Starting in Puppet version 2.7.14, modules can be installed with the puppet module command, for example:

puppet module install puppetlabs/stdlib

In previous versions, module files were manually placed in the/etc/puppet/modules/ directory.

[Example 5-3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#basic_site.pp) contains the basic site.pp file used by Puppet to control which configurations are applied to each node.

##### ***Example 5-3. Puppet role assignment with EC2 user data***

require stdlib

node default {

$userdata = parsejson($ec2\_userdata)

# Set variables from userdata

$role = $userdata['role']

case $role {

"web": { $role\_class = "myblog::web" }

default: { fail("Unrecognized role: $role") }

}

# Main myblog class

class { "myblog":

}

# Role-specific class, e.g. myblog::web

class { $role\_class:

}

}

This file uses EC2 user data, which you first learned about in [“EC2 Instance Details and User Data”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch02.html#ec2_instance_user_data), to determine which configuration should be applied to a node. This method was described in further detail in [“User Data and Tags”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch04.html#user_data_instance_tags). Here, we are setting the $role variable from user data. $role is then used in the case statement to set the $role\_data variable to the name of the class that should be used for this node.

Finally, the main myblog class is declared, along with the role-specific class (myblog::web).

Save this file to puppet/manifests/site.pp and commit your changes to the repository:

git add puppet/site.pp

git commit -m 'Added site.pp'

Next, we can add the [Nginx](https://forge.puppet.com/jfryman/nginx/readme) and [Supervisor](https://github.com/plathrop/puppet-module-supervisor) Puppet modules to our repository. We will do this using the git subtree command, which pulls the content of these external Git repositories into our own repository. Execute the following commands:

git subtree add --prefix puppet/modules/supervisor \

https://github.com/plathrop/puppet-module-supervisor.git master --squash

git subtree add --prefix puppet/modules/nginx \

https://github.com/jfryman/puppet-nginx.git master --squash

###### NOTE

As we have previously installed Puppet modules using the convenientpuppet module install command, questions naturally arise around our changed strategy. puppet module install name will deliver the latest version of a module, which can lead to unexpected code changes not acceptable in our automatically provisioned infrastructure. git subtreeimports a copy of the module into our version control system instead, ensuring such updates only occur at a time of our choosing.

##### PROVISIONING PUPPET RUNTIME AND MODULES

Deploying a consistent Puppet environment complete with one’s choice of modules is a broad topic that cannot be exhausted here. A number of alternatives exist, we introduce here the most popular options and leave a deeper exploration of the trade-offs to our readers.

[r10k](https://github.com/puppetlabs/r10k)is currently Puppet’s default choice for environment and module deployment. It provides no dependency resolution, meaning that modules used within other modules must be declared explicitly. This creates more work, but the net positive result is that all module versions, including the dependencies', can be pinned.

[puppet module install](https://docs.puppet.com/puppet/latest/reference/modules_installing.html?) is the simplest and fastest approach to installng a set of Puppet modules. It performs automatic dependency resolution, so while the modules installed explicitly can be pinned to specific versions, automatically resolved dependencies will default to the newest version available.

git subtree imports the code of the module into an application’s own Git repository, ensuring no version change occurs unless a developer initiates it first. The (very reasonable) trade-off is the need to master the appropriate [Git merge strategy](http://blogs.atlassian.com/2013/05/alternatives-to-git-submodule-git-subtree/).

[librarian-puppet](https://github.com/voxpupuli/librarian-puppet) provides dependency resolution as well a version pinning through a puppetfile.lock file. While some cosider librarian-puppet slow and would see it replaced with r10k, it remains the tool of choice in masterless Puppet environments.

The Nginx module uses Puppet’s stdlib (standard library) module, which provides useful functions such as variable validation. This can be installed with the following:

git subtree add --prefix puppet/modules/stdlib \

https://github.com/puppetlabs/puppetlabs-stdlib.git master --squash

Now we can begin writing our custom module, myblog, which will use the Nginx and Supervisor modules, as well as performing the rest of the configuration required to run the application.

The [Puppet Style Guide](https://docs.puppet.com/guides/style_guide.html) suggests that, when modules contain multiple Puppet classes, each class should be saved to its own file. This is best illustrated with an example.

[Example 5-4](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#myblog_class_def) shows the top-level myblog class, which is the class we referenced in site.pp.

##### ***Example 5-4. Initial MyBlog Puppet manifest***

class myblog {

$app\_path = "/srv/mezzanine"

class {"supervisor": }

require myblog::requirements

}

This class references a subclass, myblog::requirements. It also sets the $app\_pathvariable, which is used in the other classes. If we wanted to change the location of the Mezzanine project files, we would need to update only this variable instead of making changes in multiple files.

Save this file to puppet/modules/myblog/manifests/init.pp.

[Example 5-5](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#requirements_class_def) contains the myblog::requirements class. This class installs all of the software packages required to run Mezzanine. It also creates the mezzanine user, and creates a directory to store the project files.

##### ***Example 5-5. MyBlog requirements Puppet manifest***

class myblog::requirements {

$packages = ["python-dev", "python-pip", "libtiff5-dev", "libjpeg8-dev", "zlib1g-dev", "libfreetype6-dev"]

package { $packages:

ensure => installed

}

$pip\_packages = ["Mezzanine"]

package { $pip\_packages:

ensure => installed,

provider => pip,

require => Package[$packages]

}

user { "mezzanine":

ensure => present

}

file { "$myblog::app\_path":

ensure => "directory",

owner => "mezzanine",

group => "mezzanine"

}

}

Save this file to puppet/modules/myblog/manifests/requirements.pp.

The next subclass actually launches some servers. It is shown in [Example 5-6](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#create_project).

##### ***Example 5-6. MyBlog initialization Puppet manifests***

class myblog::create\_project {

# Create the Mezzanine project

exec { "init-mezzanine-project":

command => "/usr/local/bin/mezzanine-project myblog $myblog::app\_path",

user => "mezzanine",

creates => "$myblog::app\_path/\_\_init\_\_.py",

notify => Exec["init-mezzanine-db"]

}

# Create the development SQLite database

exec { "init-mezzanine-db":

command => "/usr/bin/python manage.py createdb --noinput",

user => "mezzanine",

cwd => "$myblog::app\_path",

refreshonly => true

}

}

This class uses Puppet’s Exec resource type to execute two commands. The mezzanine-project command creates the initial Mezzanine project, which will set up a simple website with example pages. The createdb command creates the SQLite database used in development.

###### WARNING

Be careful when creating database resources from tools like Puppet. An incorrect configuration could result in a database being unintentionally dropped and replaced with a freshly created database.

For this reason, it can be desirable to create the database outside the configuration management tool.

The Exec resource type accepts parameters that determine when it should be executed. The init-mezzanine-project Exec uses the creates parameter, which informs Puppet that executing this command will create a particular file and prevents the Exec from executing if that file already exists.

Thus, this command will execute only if /srv/mezzanine/\_\_init\_\_.py does not exist. Because we know that mezzanine-project will always create this file, it is a reliable method of ensuring we do not overwrite an existing project.

The init-mezzanine-db Exec uses another of Puppet’s control methods. It sets therefreshonly parameter to true, which means it will be executed only if explicitly requested by another resource. In this example, the init-mezzanine-project Exec notifies init-mezzanine-db, causing the latter to execute when init-mezzanine-project is executed.

Save this file to puppet/modules/myblog/manifests/create\_project.pp.

[Example 5-7](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#mynginx_class) shows the myblog::nginx class.

##### ***Example 5-7. Nginx Puppet module***

class myblog::mynginx {

class { "nginx": }

nginx::resource::upstream { "myblog\_app":

ensure => present,

members => [

'localhost:8000',

]

}

nginx::resource::vhost { "blog.example.com":

ensure => enable,

listen\_options => "default",

proxy => "http://myblog\_app"

}

}

Similar to the myblog::supervisor class, this class installs the Nginx package and writes the configuration file describing our desired Nginx setup. In this case, a single Nginx virtual host is created. This virtual host will proxy all traffic to the myblog\_app proxy, which is running on port 8000.

Because we already have a class named nginx in the Nginx module, we can’t call our class myblog::nginx. Instead, we call it myblog::mynginx to prevent a naming collision.

[Example 5-7](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#mynginx_class) should be saved to puppet/modules/myblog/manifests/mynginx.pp.

The final piece of the puzzle is the myblog::web class, shown in [Example 5-8](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#web_class).

##### ***Example 5-8. myblog::web class***

class myblog::web {

Class["myblog::web"] -> Class["myblog"]

require myblog::mynginx

supervisor::service { "myblog\_app":

ensure => present,

enable => true,

command => "/usr/bin/python ${myblog::app\_path}/manage.py runserver",

stopasgroup => true,

killasgroup => true,

user => "mezzanine",

group => "mezzanine"

}

}

This class contains everything specifically related to running an application server. It imports the myblog::nginx class to configure the web server. It also declares asupervisor::service resource, which will create a configuration file at /etc/supervisor/myblog\_web.ini, causing Supervisor to start the Mezzanine application when the instance launches.

Save this file to puppet/modules/myblog/manifests/web.pp.

Now the first step of the Puppet configuration is complete. The myblog module will take a fresh Ubuntu 14.04 instance and turn it into a working Mezzanine blog, served by Nginx.

## CloudFormation Files

Now we set up CloudFormation to provision the EC2 instance and security group.[Example 5-9](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#cloudformation_initial) shows the first version of the CloudFormation stack.

##### ***Example 5-9. Initial CloudFormation stack***

{

"AWSTemplateFormatVersion" : "2010-09-09",

"Description" : "Mezzanine-powered blog, served with Nginx.",

"Parameters" : {

"KeyName" : {

"Description" : "Name of an existing EC2 KeyPair to enable SSH access to the instance",

"Type" : "String"

}

},

"Resources" : {

"WebInstance" : {

"Type" : "AWS::EC2::Instance",

"Properties" : {

"SecurityGroups" : [ { "Ref" : "WebSecurityGroup" } ],

"KeyName" : "*federico*",

"ImageId" : "ami-c80b0aa2",

"UserData": {

"Fn::Base64": {

"{\"role\" : \"web\"}"

}

}

},

"WebSecurityGroup" : {

"Type" : "AWS::EC2::SecurityGroup",

"Properties" : {

"GroupDescription" : "Allow SSH and HTTP from anywhere",

"SecurityGroupIngress" : [

{

"IpProtocol" : "tcp",

"FromPort" : "22",

"ToPort" : "22",

"CidrIp" : "0.0.0.0/0"

},

{

"IpProtocol" : "tcp",

"FromPort" : "80",

"ToPort" : "80",

"CidrIp" : "0.0.0.0/0"

}

]

}

}

}

}

This stack does not deploy the Puppet manifests to the instance when it launches—we’ll add that later. Save this version of the stack template to myblog/cloudformation/myblog.json.

Another point to note is the use of the UserData parameter on the WebInstance resource. This provides the instance with a JSON string describing the role for the instance, which will be used in Puppet’s manifest/site.pp file to decide which classes to apply to the node during configuration. Because the CloudFormation manifest is itself written in JSON, we must escape quotation marks in the user data with a backslash to ensure they are not treated as part of the manifest.

Finally, add the Puppet modules and CloudFormation stack to the Git repository:

cd ~/myblog

git add -A

git commit -m 'added Puppet modules and CloudFormation stack'

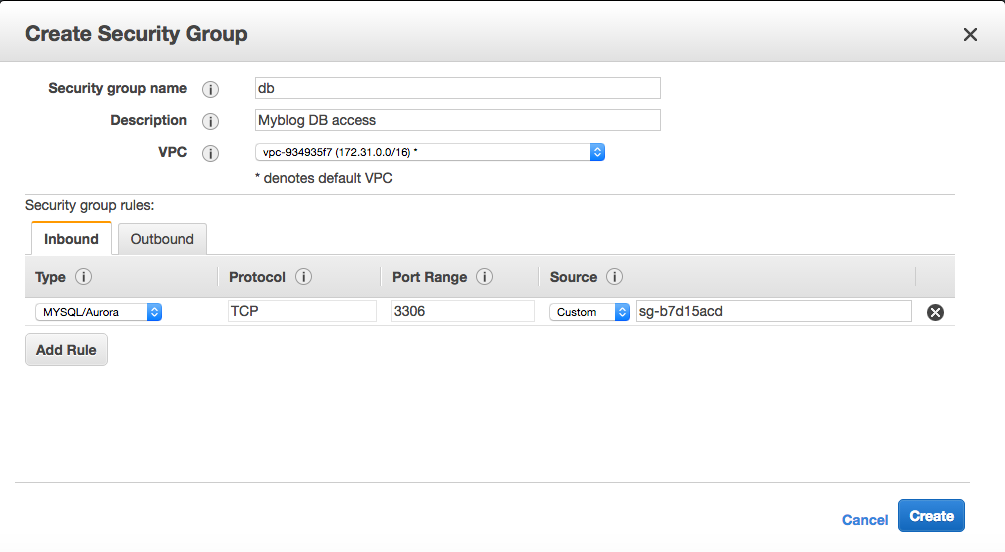
# Creating an RDS Database

The Mezzanine createdb command used in the previous step created a SQLite database. Now that the application is working, we can replace SQLite with Amazon’s Relational Database Service (RDS).

This involves two steps: creating the RDS database, and reconfiguring the Mezzanine application to use the new database instead of SQLite. First, we will perform these steps manually for testing purposes, before updating the Puppet manifests and CloudFormation stacks to reflect the updated application.

Before we can create the RDS database, we need to create a security group. This procedure can protect your database instances just as it protects EC2 instances. The two security groups perform exactly the same function: limiting access to resources, based on the source of the request.

For our purposes, we need a security group that permits access from our web server instances. We will do this with the AWS console for variety, but the task can be equally accomplished from the CLI as we have previously demonstrated.



###### *Figure 5-1. Creating the DB security group*

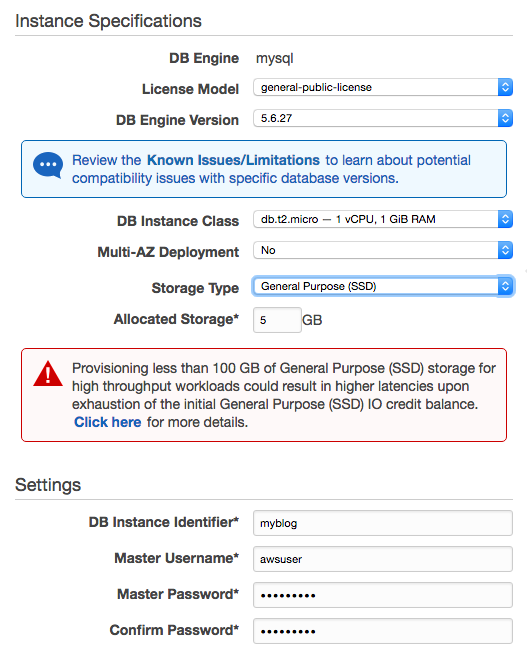
Create this group by visiting the [EC2 Security Groups](https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#SecurityGroups) page and clicking Create Security Group. Name the new group db and set the description to “Myblog DB access.”

After the group has been created, add an ingress rule that permits access for the web security group, as shown in [Figure 5-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#db_security_group). This will allow any member of the web security group to access the database.

Now we can create a new RDS database through the Management Console or command line. We will continue to use the Management Console here and later create the same database using CloudFormation. On the [RDS Console Dashboard page](https://console.aws.amazon.com/rds/home), click Launch a DB Instance, which will open a wizard that guides you through the process of creating a database.

The first screen presents you with a list of database engines—such as PostgreSQL, MySQL, Microsoft SQL Server, and Oracle—that can be used with RDS. Click the Select button on the tab of MySQL Community Edition.

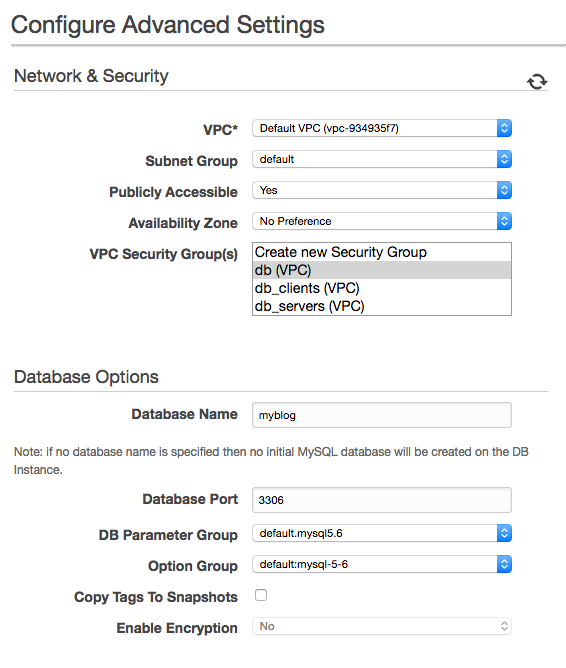
Multi-AZ Deployment allows you to deploy a master/slave database pair that spans multiple availability zones. This increases resilience to failure, but also increases cost, and it is not required for this test database. Select the MySQL “Dev/Test” option when prompted, ignoring the Multi-AZ and Amazon Aurora alternatives for now.



###### *Figure 5-2. DB instance details*

[Figure 5-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#db_instance_details) shows the third screen in the process, where the initial MySQL options are set.

The DB Instance Identifier, Master Username, and Master Password options can be set to any values you want—but keep track of them, as they will be needed in the next step. Then click Next Step to move on to the Configure Advanced Settings screen, shown in [Figure 5-3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#db_additional).



###### *Figure 5-3. Additional configuration*

Enter a name for your database (e.g., myblog). This database will be created automatically when the DB instance launches.

In the DB Security Groups box, select the db security group and click Next Step.

This screen lets you also configure the automatic backup settings. For the purposes of this test, we will disable automatic backups by defining a backup retention period of zero days. While we do not wish to incur additional data charges for our development instance, the automatic backup, tagging, minor version upgrade and monitoring functionality offered by AWS just in this screen make a clear case of why public cloud environments are not just convenient, but also effective: in a classic datacenter, you would be in charge of performing (or automating away) each of those tasks. You should absolutely make use of these facilities in production, particularly the automated backup option.

After reviewing your chosen settings, click Launch DB Instance.

Once the database instance has been created, its record in the console will be updated to include an endpoint. This is the hostname you will use to configure your MySQL clients—for example, myblog.cvqj2dqsvoab.us-east-1.rds.amazonaws.com.

Now that you have a database instance running, you can reconfigure Mezzanine to use this database instead of the local SQLite database.

Mezzanine settings are controlled by a Python file located at /srv/mezzanine/myblog/settings.py. Because it is quite common for different environments to require different settings (for example, a local database is used during development, and a remote database is used during production), Django and Mezzanine make it easier to override individual settings.

If a file named local\_settings.py exists, any settings it contains will override those set in the main settings.py file. The settings that are consistent across all environments can be stored in settings.py, and any custom settings in local\_settings.py.

###### NOTE

There are many ways to set different configuration options, depending on the environment in which the instance is running. [The Twelve-Factor App](https://12factor.net/config) describes one such alternative method that uses a system-level environment variable to control which settings file is used.

[Example 5-10](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#local_settings_rds) shows a local\_settings.py file that specifies the use of an RDS database. Modify this file to reflect your database endpoint, as well as the username and password settings. The latter two should be set to the values you chose for Master Username and Master Password, respectively.

##### ***Example 5-10. Mezzanine database configuration***

ALLOWED\_HOSTS = "\*"

DEBUG = True

DATABASES = {

"default": {

"ENGINE": "django.db.backends.mysql",

"NAME": "*myblog*",

"USER": "*awsuser*",

"PASSWORD": "*foobarbaz*",

"HOST": "*myblog.cvqj2dqsvoab.us-east-1.rds.amazonaws.com*",

"PORT": "3306"

}

}

###### WARNING

Do not use the master user in your production applications, create for each a separate user with limited access. Doing so will limit the impact of a security breach.

###### WARNING

Setting ALLOWED\_HOSTS to \* will make our example application work regardless of the domain name in use. This is convenient in our example, but for production deployments you will want to limit the IP addresses and domain names that your project is allowed to serve. [ALLOWED\_HOSTS](https://docs.djangoproject.com/en/dev/ref/settings/#allowed-hosts) is a security feature meant to counteract [HTTP host header attacks](https://docs.djangoproject.com/en/dev/topics/security/#host-headers-virtual-hosting).

After making the required changes, save this file to /srv/mezzanine/myblog/local\_settings.new with appropriate user and group ownership, then execute the following to preserve the hash salt variable generated at project creation:

sudo su mezzanine

cd /srv/mezzanine/myblog

mv local\_settings.py local\_settings.old

cat local\_settings.old | grep 'SECRET\_KEY =\|NEVERCACHE\_KEY =' > local\_settings.py

cat local\_settings.new >> local\_settings.py

exit

We no longer need the SQLite database file, so proceed to delete it:

sudo rm /srv/mezzanine/dev.db

Because we will now be using MySQL instead of SQLite, we must ensure that the Python client libraries for MySQL are installed:

sudo apt install python-mysqldb mysql-client-5.6 libmysqlclient-dev

Before we can use the new database, we must create the initial table structure and example setup by running createdb again:

sudo -u mezzanine python /srv/mezzanine/manage.py createdb --noinput

As we have just replaced the existing database with a fresh one, do not expect to see any of your test blog entries. Further, you will need to re-initialize access with the following-make sure to use the same password for the default user:

sudo -u mezzanine python manage.py createsuperuser

The Mezzanine application must be restarted to pick up the changed settings file. This is done with Supervisor’s supervisorctl command:

sudo supervisorctl restart myblog\_app

Once the process has been restarted, use your web browser to verify once again that the Mezzanine site is still working.

You may notice that the web page feels a bit slower than it did previously. This is because the database is no longer on the same machine as the web application, which introduces some delay. After updating the Puppet and CloudFormation files with the changes we have just made, we will add a caching server to alleviate some of this delay.

## RDS: Updating Puppet and CloudFormation

Now that we have completed and tested our changes manually, it is time to update the Puppet manifests to reflect them.

As part of the manual process, we created a database with the Management Console and hard-coded its endpoint into the settings file, along with the username and password. Our end goal is to have a stack that can be deployed without any manual configuration, which means that the RDS database will need to be created by CloudFormation. Hard-coding the connection details will not be an option for a dynamically created database.

How do we solve the problem of dynamically changing configuration files based on other resources in the CloudFormation stack? [“User Data and Tags”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch04.html#user_data_instance_tags) demonstrated one way of solving this problem.

CloudFormation’s Fn::GetAtt function can access the attributes of resources in the stack template. So we can use this function to send the database’s endpoint as user data to the instance. Puppet can then access that user data and use it writing the configuration files.

[Example 5-11](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#myblog_stack_rds) shows an updated version of the myblog.json stack template.

##### ***Example 5-11. MyBlog CloudFormation stack with RDS database***

{

"AWSTemplateFormatVersion" : "2010-09-09",

"Description" : "Mezzanine-powered blog with RDS, served with Nginx.",

"Parameters" : {

"KeyName" : {

"Description" : "Name of an existing EC2 KeyPair to enable SSH access to the instance",

"Type" : "String"

},

"WebAMI": {

"Type": "String"

},

"KeyPair": {

"Type": "String"

},

"DBUser": {

"Type": "String"

},

"DBPassword": {

"Type": "String",

"NoEcho": "TRUE"

}

},

"Resources" : {

"BlogDB" : { [https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/1.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#callout_blog_db_co)

"Type" : "AWS::RDS::DBInstance",

"Properties" : {

"DBSecurityGroups" : [ {"Ref" : "DBSecurityGroup"} ],

"DBName" : "myblog",

"AllocatedStorage" : 5,

"DBInstanceClass" : "t2.micro",

"Engine" : "MySQL",

"EngineVersion" : "5.5",

"MasterUsername" : { "Ref" : "DBUser" }, [https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/2.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#callout_db_user_co)

"MasterUserPassword" : { "Ref" : "DBPassword" }

},

"DeletionPolicy" : "Snapshot" [https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/3.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#callout_snapshot_co)

},

"DBSecurityGroup" : { [https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/4.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#callout_db_sec_group_co)

"Type" : "AWS::EC2::SecurityGroup",

"Properties" : {

"GroupDescription" : "Allow inbound MySQL access from web instances",

"SecurityGroupIngress" : [

{

"IpProtocol" : "tcp",

"FromPort" : "3306",

"ToPort" : "3306",

"SourceSecurityGroupName" : { "Ref" : "WebSecurityGroup" }

}

]

},

"WebInstance" : {

"Type" : "AWS::EC2::Instance",

"Properties" : {

"SecurityGroups" : [ { "Ref" : "WebSecurityGroup" } ],

"KeyName" : { "Ref" : "KeyPair",

"ImageId" : { "Ref" : "WebAMI" },

"UserData" : { [https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/5.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#callout_user_data_2_co)

"Fn::Base64" : {

"Fn::Join" : [ "", [

"{\"db\_endpoint\": \"",

{ "Fn::GetAtt": [ "BlogDB", "Endpoint.Address" ] }, "\",",

" \"db\_user\": \"", { "Ref": "DBUser" }, "\",",

" \"db\_password\": \"", { "Ref": "DBPassword" }, "\" }"

] ]

}

}

}

},

"WebSecurityGroup" : {

"Type" : "AWS::EC2::SecurityGroup",

"Properties" : {

"GroupDescription" : "Allow SSH and HTTP from anywhere",

"SecurityGroupIngress" : [

{

"IpProtocol" : "tcp",

"FromPort" : "22",

"ToPort" : "22",

"CidrIp" : "0.0.0.0/0"

},

{

"IpProtocol" : "tcp",

"FromPort" : "80",

"ToPort" : "80",

"CidrIp" : "0.0.0.0/0"

}

]

}

}

}

}

The key items in the stack are as follows:

[https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/1.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#co_blog_db_co)

The BlogDB resource is the RDS database instance, using the same settings as the database we created using the Management Console.

[https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/2.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#co_db_user_co)

The MasterUsername database parameter is used in two places. First, it is used when creating the RDS instance, and it is also passed to the WebInstance resource as part of its user data, making it accessible to scripts running on that instance.

[https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/3.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#co_snapshot_co)

The DeletionPolicy database parameter ensures that when the stack is deleted, your data is not lost. Before CloudFormation terminates the database, it will perform a final snapshot.

[https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/4.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#co_db_sec_group_co)

The DBSecurityGroup allows members of the WebSecurityGroup to access the database instance.

[https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/assets/5.png](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#co_user_data_2_co)

The user data of the WebInstance contains the database hostname, username, and password.

Update ~/myblog/cloudformation/myblog.json to reflect the updates. The WebSecurityGroup is unchanged, so you won’t have to update that part of the file.

Now that the database name will be sent as user data, we need to update the Puppet manifests so that the local\_settings.py file contains the correct endpoint, username, and password settings.

###### NOTE

The user data is JSON-formatted to make it easy to read in Puppet. Because CloudFormation stacks are themselves JSON-formatted, it means the user data must be escaped in order to be valid JSON. This, admittedly, can lead to rather ugly syntax in stack templates.

Here is an example of the user data produced by this statement:

{ "db\_endpoint": "myblog.cvqj2dqsvoab.us-east-1.rds.amazonaws.com",

"db\_user": "awsuser",

"db\_password": "foobarbaz"

}

Currently, this is the process for setting up a new Mezzanine project with Puppet:

1. Initialize the project with mezzanine-project.
2. Create the database with createdb.

We need to insert an additional step in the middle:

* Create a local\_settings.py file based on the given user data.

This must be done before running createdb; otherwise, the default Mezzanine settings will be used, and a SQLite database will be created.

Create the new file by creating a File resource in the Puppet manifest and using a template to populate the contents of this file. The template uses variables that are set in the Puppet manifests by querying the user data for the instance.

Although this is a small change, implementing it cleanly requires changing a few of the Puppet manifest files. We could access the user data variable directly from the myblog::create\_project class, but this goes against Puppet’s best practice guidelines.

Instead, we will convert the top-level myblog class to a parameterized class, which takes the DB endpoint, username, and password as parameters. Placing variables in a class is the recommended way to introduce variation into Puppet templates, as it helps make modules a lot more reusable by avoiding variable scoping issues.

[Example 5-12](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#myblog_variable) shows an updated version of the myblog class that accepts the required parameters.

##### ***Example 5-12. MyBlog Puppet manifest with parameters***

class myblog ( $db\_endpoint, $db\_user, $db\_password ) {

$app\_path = "/srv/mezzanine"

class {"supervisor": }

require myblog::requirements

}

Update puppet/modules/myblog/manifests/init.pp with the contents of this example. The first line is changed to include a list of parameters that must be set when declaring an instance of the myblog class.

The next step is to modify site.pp so that it retrieves the new parameters from user data and passes them to the myblog class, as shown in [Example 5-13](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#site.pp_variable). The file reads these parameters from the $userdata variable, which is a JSON object created by reading the $ec2\_userdata string variable. parsejson is a function provided by Puppet’s stdlib.

##### ***Example 5-13. Puppet site.pp file for MyBlog***

require stdlib

node default {

$userdata = parsejson($ec2\_userdata)

# Set variables from userdata

$role = $userdata['role']

**$db\_endpoint = $userdata['db\_endpoint']**

**$db\_user = $userdata['db\_user']**

**$db\_password = $userdata['db\_password']**

case $role {

"web": { $role\_class = "myblog::web" }

default: { fail("Unrecognized role: $role") }

}

# Main myblog class, takes all params

class { "myblog":

**db\_endpoint => $db\_endpoint,**

**db\_user => $db\_user,**

**db\_password => $db\_password**

}

# Role-specific class, e.g. myblog::web

class { $role\_class: }

}

Update puppet/manifests/site.pp with the contents of this example.

Next, we need to update the myblog::create\_project class so that it creates the local\_settings.py file. This is shown in [Example 5-14](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#myblog_create_local_settings).

##### ***Example 5-14. Updated MyBlog initialization Puppet manifest***

class myblog::create\_project {

# Create the Mezzanine project

exec { "init-mezzanine-project":

command => "/usr/local/bin/mezzanine-project $myblog::app\_path",

user => "mezzanine",

creates => "$myblog::app\_path/\_\_init\_\_.py"

}

**# Create the local\_settings.py file**

**file { "$myblog::app\_path/myblog/local\_settings.py":**

**ensure => present,**

**content => template("myblog/local\_settings.py.erb"),**

**owner => "mezzanine",**

**group => "mezzanine",**

**require => Exec["init-mezzanine-project"],**

**notify => Exec["init-mezzanine-db"]**

**}**

# Create the database

exec { "init-mezzanine-db":

command => "/usr/bin/python manage.py createdb --noinput",

user => "mezzanine",

cwd => "$myblog::app\_path",

refreshonly => true

}

This file should replace puppet/modules/myblog/manifests/create\_project.pp. The main change is to add the File resource that creates local\_settings.py. Its contents will be based on the template file named local\_settings.py.erb.

Although we specify the template name as myblog/local\_settings.py.erb, Puppet will look for the file in puppet/modules/myblog/templates/local\_settings.py.erb.

As before, the require and notify parameters control the ordering of Puppet resources-note the repositioning of the notify line in the updated file. The local\_settings.py file must be created before createdb is executed.

Finally, we need to create the template file that will be used populate the local\_settings.pyfile. This is shown in [Example 5-15](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#local_settings_template).

##### ***Example 5-15. Updated MyBlog database manifest***

ALLOWED\_HOSTS = "\*"

DEBUG = True

DATABASES = {

"default": {

"ENGINE": "django.db.backends.mysql",

"NAME": "*mydb*",

"USER": "<%= @db\_user %>",

"PASSWORD": "<%= @db\_password %>",

"HOST": "<%= @db\_endpoint %>",

"PORT": "3306"

}

}

Save this file to puppet/modules/myblog/templates/local\_settings.py.erb. The content is almost exactly the same as the local\_settings.py file we created manually, except that Puppet will replace the variables with information taken from user data.

###### NOTE

For more information on Puppet templates, see the documentation on [Using Puppet Templates](https://docs.puppet.com/puppet/latest/reference/lang_template.html).

[Example 5-16](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#requirements_class_def_upd) contains the updated myblog::requirements class, now including the MySql support packages.

##### ***Example 5-16. Updated MyBlog requirements Puppet manifest***

class myblog::requirements {

$packages = ["python-dev", "python-pip", "libtiff5-dev", "libjpeg8-dev",

"zlib1g-dev", "libfreetype6-dev", **"python-mysqldb", "mysql-client-5.6",**

**"libmysqlclient-dev"**]

package { $packages:

ensure => installed

}

Commit the updated files to the Git repository:

git add -A

git commit -am 'added RDS database to stack'

With that step complete, we can move on to the caching server.

###### NOTE

Mezzanine’s built-in web server has the capability to selectively deliver only dynamically-generated content, while delegating the application’s static content delivery to a more efficient external web server. The DEBUG flag controls this behavior: once it is set to false, Django will no longer answer requests for any static content and images, CSS and Javascript components of the application templates need to be delivered through another mechanism.

Running the manage.py collectstatic command will copy all files from the static directory of each Django application to the location defined in STATIC\_ROOT. We leave as an exercise for the reader to map the URL defined by STATIC\_URL to serve files from that directory. There is something to be learned by correctly sequencing the dependencies required to set up this configuration automatically. The authors chose to focus here on scale-out caching for the application as a whole as illustrated in the next section, making this detail less important as the caching layer can be relied to serve static files even more effectively.

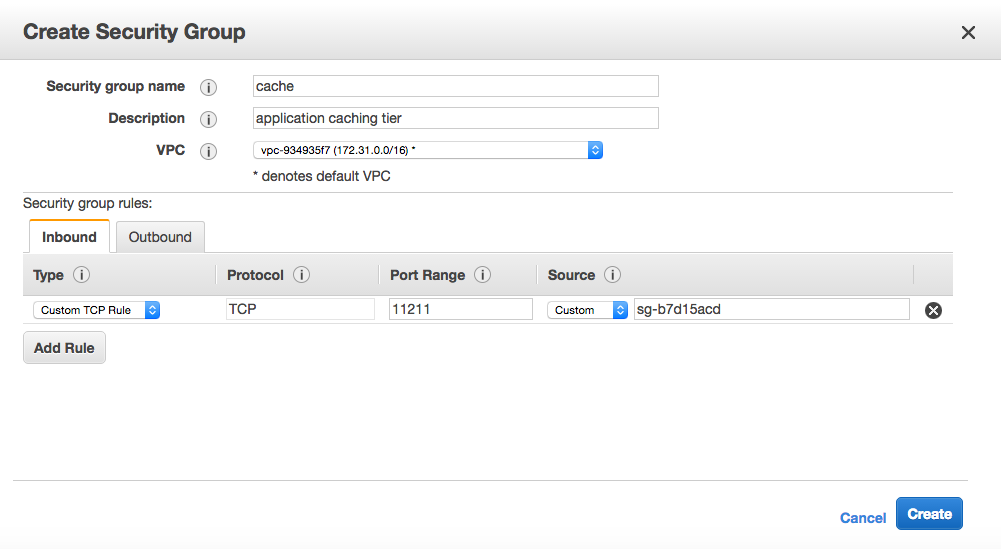
# Creating an ElastiCache Node

Caching is a required part of any high-traffic web application, so it makes sense to include some caching in the example stack. We will use Amazon’s ElastiCache service instead of running our own Memcache cluster. ElastiCache is a drop-in replacement for Memcache, which means minimal work is required on the client side.

If a cache server is specified in Mezzanine’s settings file, unauthenticated page views will be automatically cached—i.e., anonymous users will see cached copies of the Mezzanine home page. To enable caching—and start using it—we simply need to let Mezzanine know there is a caching server available, which means modifying the local\_settings.py file again.

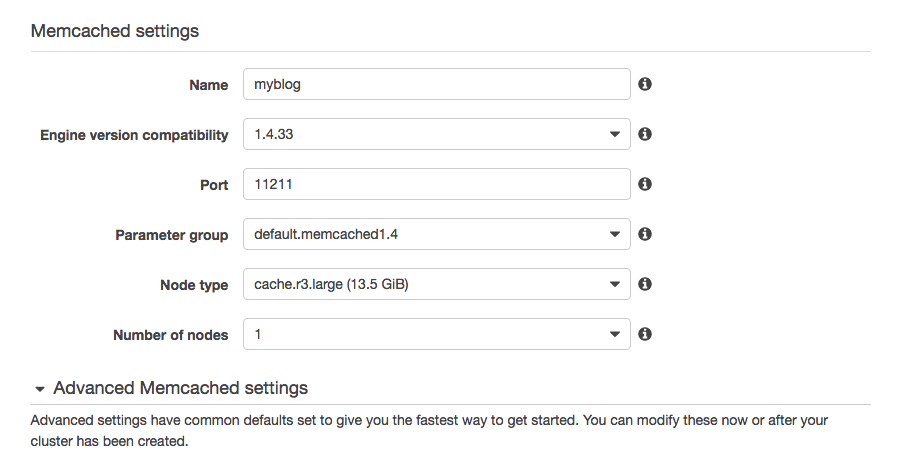
Just as we did earlier in the chapter, we begin by manually setting up an ElastiCache node and then automating this configuration with Puppet and CloudFormation files.

First, visit the EC2 Security Groups page and create a new security group named cache. Grant access to members of the web EC2 security group, as shown in [Figure 5-4](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#set_cache_security).



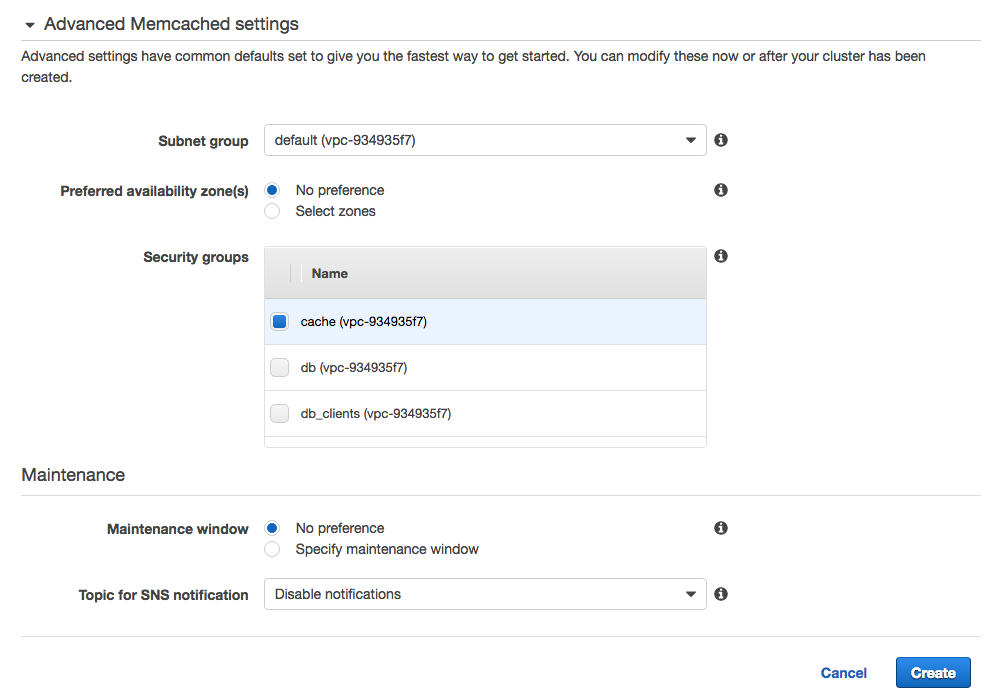
###### *Figure 5-4. Setting Cache Security Group permissions*

After creating the cache security group and setting its permissions, go to the [ElastiCache](https://console.aws.amazon.com/elasticache/home)page and click Launch Cluster to open the Launch Cluster Wizard, which will first prompt you to choose between Memcached and Redis-select Memcached and click Next. The second section of the wizard is shown in [Figure 5-5](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#launch_cluster_wizard).



###### *Figure 5-5. Launch Cluster Wizard*

The key settings here are the name of the cluster and the number of nodes. We are choosing to launch a cache cluster containing a single node, which is not ideal for reliability or resilience, but is perfect to minimize costs for testing.



###### *Figure 5-6. Setting the security group*

After selecting the options shown in the screenshot, open the Advanced Memcached setting to access the next group of settings. This is shown in [Figure 5-6](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#set_security_group).

This screen has only one setting that is important for our purposes: the Security Group. Select the security group you created (cache) in addition to your pre-selected default subnet, and click Create.

Once the ElastiCache cluster has launched, you will see it in the Cache Clusters list. Click the name of the cache cluster to go to the description page. Here you will see a list of information describing the cache cluster, including its configuration endpoint.

##### ELASTICACHE AUTO DISCOVERY

The number of nodes in an ElastiCache cluster can change at any time, due to either planned events such as launching new nodes to increase the size of the cluster, or unplanned events such as a reduction in capacity caused by a crashed node. This can lead to problems when client applications try to connect to nodes that no longer exist, or never connect to new nodes because the client is not aware of the node’s existence.

To help users build more resilient applications, Amazon has extended ElastiCache with a feature known as Auto Discovery. This allows clients to connect to a single address—the configuration endpoint—and retrieve information about all nodes in the cluster.

Using Auto Discovery requires a compatible client, because this feature is not part of the vanilla Memcache specification. Amazon has released compatible clients for PHP and Java, and plans to add clients for other languages in the future.

At a technical level, a configuration endpoint address is simply a CNAME DNS record. Resolving this address will return the hostname for one of the nodes in the ElastiCache cluster.

Amazon ensures that all nodes in the cluster contain up-to-date information about their sibling nodes. Auto Discovery−compatible clients use the endpoint address to connect to one of the target nodes, from which they can retrieve a list of other nodes in the cluster.

If you are not using Java or PHP, you can still use Auto Discovery, albeit with a bit more effort. You will need to periodically connect to the configuration endpoint to retrieve information about members of the cluster and update your local Memcache client configuration.

In some cases, the configuration endpoint is all you need. If you maintain an ElastiCache cluster containing a single node, you can add the configuration endpoint directly to your client configuration. Because you have only one node, the configuration endpoint CNAME will always resolve to the same node address.

When running clusters with two or more nodes, using the configuration endpoint directly from an incompatible client has two drawbacks. Depending on how your client caches DNS records, traffic might become unevenly distributed between your nodes, leading to underutilized cache nodes. You will also have no control over which node your data resides on, leading to an unnecessarily high cache miss rate.

Because our cache cluster contains only a single node, we can use the configuration endpoint to configure our Memcache client.

Configuring Mezzanine to use caching is simple: if a cache host is defined in the settings file, Mezzanine will use it to cache unauthenticated page views. Of course, we first need to install the Python Memcache library so Mezzanine can communicate with ElastiCache. We will use the python-memcached library. Install it as follows:

sudo pip install python-memcached

The next step is to add the cache configuration information to Mezzanine’s settings. Append the following text to /srv/mezzanine/myblog/local\_settings.py, replacing the hostname with your ElastiCache cluster’s configuration endpoint:

CACHES = {

"default": {

"BACKEND": "django.core.cache.backends.memcached.MemcachedCache",

"LOCATION": "*myblog.gdkr4r.cfg.use1.cache.amazonaws.com*:11211"

}

}

With the configuration file updated, restart the Mezzanine process:

sudo supervisorctl restart myblog\_app

Visit the Mezzanine page in your web browser. Everything should look exactly the same. However, if you refresh the page, you may notice it feels faster on subsequent requests. Of course, it feels faster is not the most scientific of tests, so we should verify that data is, in fact, being written to the Memcache node. We can do this by connecting to the node with Telnet and checking that it contains a record of the cached page content.

Open a connection to the node:

telnet myblog.gdkr4r.cfg.use1.cache.amazonaws.com 11211

Memcache does not contain a built-in command to list all stored keys, so we must take a slightly roundabout approach here. Within Memcache, data is stored in a series of slabs. We can list the slabs with the stats slabs command:

$ **stats slabs**

STAT 20:chunk\_size 7104

STAT 20:chunks\_per\_page 147

STAT 20:total\_pages 1

STAT 20:total\_chunks 147

STAT 20:used\_chunks 1

STAT 20:free\_chunks 146

STAT 20:free\_chunks\_end 0

[output truncated]

The number in the first column after STAT is the slab ID; in this example, it’s 20. We can then dump the keys that belong to that slab with the stats cachedump command. This accepts two arguments: the slab ID and the maximum number of keys to dump. Execute this command within your Telnet session:

$ **stats cachedump *20* 100**

ITEM :1:e67b2514a4f5d0b3253baa637db8ba01 [6811 b; 1475464297 s]

END

A single key is dumped, verifying that our visit to the web page cached its data. You can delete this key:

$ **delete *:1:e67b2514a4f5d0b3253baa637db8ba01***

After deleting the key, refresh the Mezzanine page and use the cachedump command to verify that a new key has been written.

###### TIP

You can exit this telnet session by entering Ctrl-] and using the telnet client’s quit command, like so:

$ **^]**

telnet> **quit**

Connection closed.

Now that ElastiCache has been tested, we can make the relevant changes to the Puppet and CloudFormation files.

# ElastiCache: Updating Puppet and CloudFormation

Because we laid much of the groundwork when setting up RDS, updating Puppet and CloudFormation to use Puppet will be a lot simpler than the previous section.

We will begin by ensuring that the python-memcached library is installed when the instance is provisioned. The puppet/modules/myblog/manifests/requirements.pp file contains the following line:

$pip\_packages = ["Mezzanine"]

Replace it with the following:

$pip\_packages = ["Mezzanine", **"python-memcached"**]

Next, we need to add a new parameter to the myblog Puppet module, which will be used to store the configuration endpoint of the cache cluster. Update puppet/modules/myblog/manifests/init.pp, changing the class signature to this:

class myblog ( $db\_endpoint, $db\_user, $db\_password**, $cache\_endpoint** ) {

The puppet/manifests/site.pp file must also be updated so that this parameter is passed when the myblog class is declared. Update this file with the following content:

require stdlib

node default {

$userdata = parsejson($ec2\_userdata)

# Set variables from userdata

$role = $userdata['role']

$db\_endpoint = $userdata['db\_endpoint']

$db\_user = $userdata['db\_user']

$db\_password = $userdata['db\_password']

**$cache\_endpoint = $userdata['cache\_endpoint']**

case $role {

"web": { $role\_class = "myblog::web" }

default: { fail("Unrecognized role: $role") }

}

class { "myblog":

db\_endpoint => $db\_endpoint,

db\_user => $db\_user,

db\_password => $db\_password,

**cache\_endpoint => $cache\_endpoint**

}

# Role-specific class, e.g. myblog::web

class { $role\_class: }

}

Finally, update puppet/modules/myblog/templates/local\_settings.py.erb and append the cache configuration:

CACHES = {

"default": {

"BACKEND": "django.core.cache.backends.memcached.MemcachedCache",

"LOCATION": "<%= @cache\_endpoint %>:11211"

}

}

Those are the changes required for the Puppet side of the equation. Updating the CloudFormation stack is equally straightforward. Rather than replicating the entire stack template again, we will just include those sections that need to be updated.

[Example 5-17](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#elasticache_template_resources) shows the CloudFormation stack template section that declares the resources we need to use ElastiCache: the cache cluster itself, an ElastiCache security group, and a security group ingress rule that grants our web server instance access to the cache cluster nodes as a member of the WebSecurityGroup security group. Insert the text into the stack template, located at cloudformation/myblog.json, before the existing BlogDB resource.

##### ***Example 5-17. ElastiCache CloudFormation stack***

"Resources" : {

**"CacheCluster": {**

**"Type" : "AWS::ElastiCache::CacheCluster",**

**"Properties" : {**

**"CacheNodeType" : "cache.r3.large",**

**"CacheSecurityGroupNames" : [ "CacheSecurityGroup" ],**

**"Engine" : "memcached",**

**"NumCacheNodes" : "1"**

**}**

**},**

**"CacheSecurityGroup": {**

**"Type": "AWS::ElastiCache::SecurityGroup",**

**"Properties": {**

**"Description" : "Allow access from Web instances"**

**}**

**},**

**"CacheSecurityGroupIngress": {**

**"Type": "AWS::ElastiCache::SecurityGroupIngress",**

**"Properties": {**

**"CacheSecurityGroupName" : { "Ref" : "CacheSecurityGroup" },**

**"EC2SecurityGroupName" : { "Ref" : "WebSecurityGroup" }**

**}**

**},**

"BlogDB" : {

Because the cache cluster configuration endpoint address is passed to the web server instances as user data, we also need to modify the EC2 instance resource. We will again use the Fn::GetAtt function, this time to retrieve the configuration endpoint address.

[Example 5-18](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#userdata_elasticache) shows the updated user data—note the addition of the cache\_endpointattribute. Replace the UserData attribute of the WebInstance resource with [Example 5-18](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#userdata_elasticache).

##### ***Example 5-18. User data with ElastiCache and database parameters***

"UserData" : {

"Fn::Base64" : {

"Fn::Join" : [ "", [

"{\"db\_endpoint\": \"", { "Fn::GetAtt": [ "BlogDB", "Endpoint.Address" ] }, "\",",

" \"db\_user\": \"", { "Ref": "DBUser" }, "\",",

" \"db\_password\": \"", { "Ref": "DBPassword" }, "\",",

**" \"cache\_endpoint\": \"",**

**{ "Fn::GetAtt": [ "CacheCluster", "ConfigurationEndpoint.Address" ] }, "\"}"**

] ]

}

}

Now, when the stack is created, the configuration endpoint will be passed to the instance as user data and used to populate the local\_settings.py file.

As always, finish by committing the changes to the Git repository:

git add -A

git commit -am 'added ElastiCache cluster'

# Installing Celery with Simple Queueing Service

The final component in the example application stack is the background task-processing service, which will be implemented using Celery.

Mezzanine does not, by default, use Celery, so we will briefly digress into the realm of Python programming and build a simple task that will give Celery something useful to do as part of the demonstration. In this case, something useful means checking user submitted comments for spam content.

Whenever a new comment is posted on the site, Mezzanine will send a signal to notify other components in the application, letting them take action on the new comment. [Signals](https://docs.djangoproject.com/en/dev/topics/signals/) are afeature of Django, and are an implementation of the Observer software design pattern.

We will write some code that listens for this signal. When a new comment is posted, it will be checked by our extremely simple spam filter function.

When we launch the final stack containing this infrastructure, Celery and Mezzanine will be running on separate EC2 instances. However, it is a waste of time (and money) to launch another development machine to configure Celery when we could instead use the web application instance we have used in the previous steps. So, for testing purposes, perform the steps in this section on the web application instance.

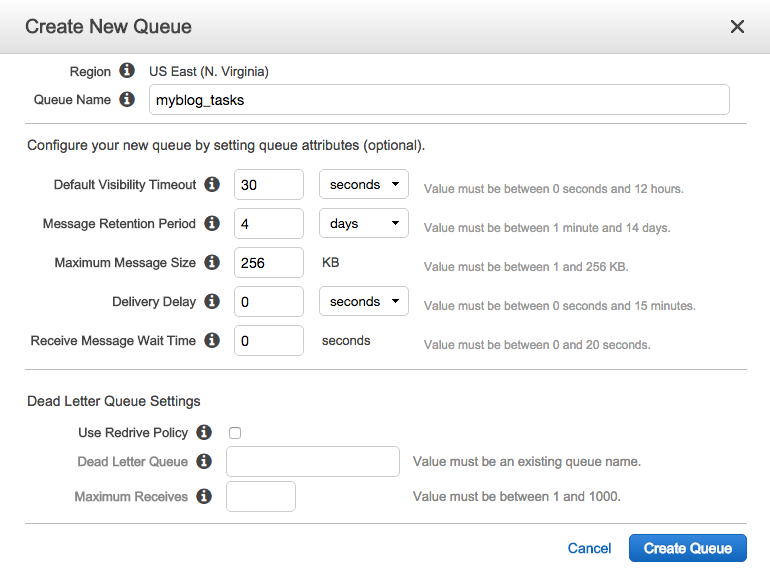
Celery works by passing messages between your application processes and the background worker processes. These messages describe the task that should be executed, along with any parameters required by the task. Celery needs somewhere to store these messages, so it uses a message broker for this purpose. In keeping with the strategy of using Amazon services for as many tasks as possible in this example stack, we will use SQS, Amazon’s own message queuing service, as the message broker. Because Celery has built-in support for SQS, it is really simple to use.

We will begin by creating the SQS queue that will hold our messages. In the Management Console, go to the [SQS page](https://console.aws.amazon.com/sqs/home) and click Create New Queue. [Figure 5-7](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#create_sqs_queue) shows the Create Queue screen.

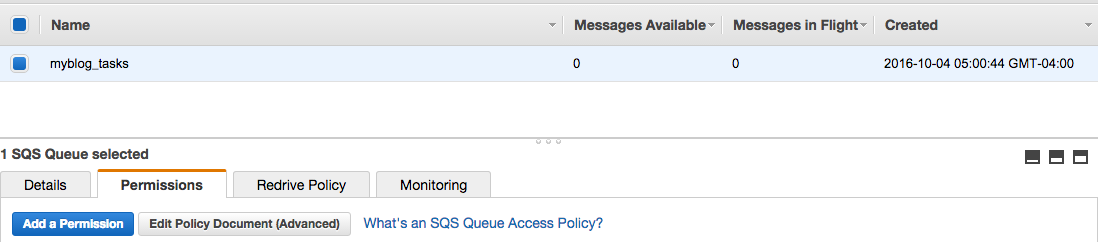
Enter **myblog\_tasks** as the queue name and click Create Queue-the remaining settings are default values at the time of writing.

Once the queue has been created, you will be returned to the Queues page, where you can see the details of the new queue. Now we need to configure the queue’s access permissions so we can read from and write to it.

Select the Permissions tab and click Add a Permission, as shown in [Figure 5-8](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#add_queue_permission) (alternatively you may use the Queue Actions pull-down menu).



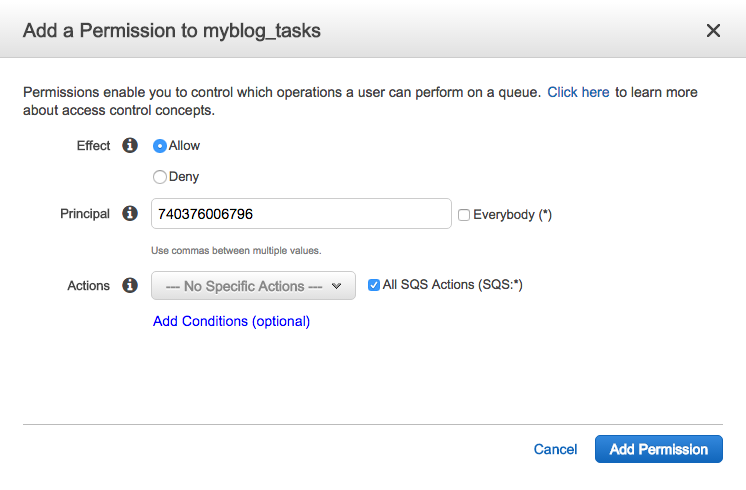
###### *Figure 5-7. Creating a new SQS queue*



###### *Figure 5-8. Queue permissions*

[Figure 5-9](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#adding_queue_permission) shows the next screen.

Select the options shown in [Figure 5-9](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#adding_queue_permission), replacing 740376006796 with your 12-digit AWS account ID, and click Add Permission. Now anyone in your AWS account has full permissions on this SQS queue. In the next chapter, we will define a more restrictive security policy using IAM.



###### *Figure 5-9. Adding a queue permission*

With the queue created, we can move on to installing and configuring Celery. First, install the django-celery library:

sudo pip install django-celery

This package will also install the core celery library. django-celery is a convenience package, making it easy to use Celery from within Django-powered applications such as Mezzanine.

Celery tasks are simply Python functions. By convention, these are stored in a file named celery.py.

[Example 5-19](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#python_signal) shows the Python code that handles the signal received when a new comment is posted.

##### ***Example 5-19. The simple asynchronous spam check***

from celery import Celery

from django.dispatch import receiver

from django.db.models.signals import post\_save

from mezzanine.generic.models import ThreadedComment

app = Celery('tasks', broker='amqp://guest@localhost//')

def is\_comment\_spam(comment):

# This check is just an example!

if "spam" in comment.comment:

return True

@app.task

def process\_comment\_async(comment):

print "Processing comment"

if is\_comment\_spam(comment):

# The comment is spam, so hide it

ThreadedComment.objects.filter(id=comment.id).update(is\_public=False)

@receiver(post\_save, sender=ThreadedComment)

def process\_comment(sender, \*\*kwargs):

process\_comment\_async.delay(kwargs['instance'])

Save this code to /srv/mezzanine/myblog/tasks.py, making sure it is owned by the user mezzanine like all others in this directory. Before moving on with the configuration, let’s look at what this code is doing. This requires a brief description of a useful Django feature: signals.

Signals can be thought of as hooks for your custom code to connect to. For example, every time a database object is saved, a post\_save signal is sent out by Django. The @receiver function decorator informs Django that whenever a ThreadedComment object sends the post\_save signal (i.e., a new comment is posted to the blog), the process\_comment function is called.

The process\_comment function calls process\_Comment\_async.delay. This does not execute the code immediately—instead, it posts a message to the Celery queue. This message is picked up by a Celery worker, and the code in process\_comment\_async is executed by the worker.

This means that whenever a comment is posted to the blog, it will be initially displayed. After a worker picks up the job from the queue, the message will be hidden if it contains spammy content, as defined by the is\_comment\_spam function. In this trivial case, we simply check whether the string spam exists in the comment text. Alas, real spammers are not so easy to catch. You might want to update this function to perform a more reliable spam check, such as submitting the comment to [Akismet’s spam-checking service](http://akismet.com/).

Because we are using the django-celery package, we can configure Celery by updating /srv/mezzanine/myblogs/local\_settings.py. Append the code in [Example 5-20](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#celery_local_settings_py) to that file, replacing the AWS access key and secret with your own.

##### ***Example 5-20. Adding Celery settings to local\_settings.py***

BROKER\_URL = 'sqs://*your-aws-access-key*:*your-aws-access-secret*@'

# BROKER\_TRANSPORT\_OPTIONS = {'region': '*us-east-1*'}

CELERY\_IMPORTS = ('tasks')

INSTALLED\_APPS = (

"django.contrib.admin",

"django.contrib.auth",

"django.contrib.contenttypes",

"django.contrib.redirects",

"django.contrib.sessions",

"django.contrib.sites",

"django.contrib.sitemaps",

"django.contrib.staticfiles",

"mezzanine.boot",

"mezzanine.conf",

"mezzanine.core",

"mezzanine.generic",

"mezzanine.blog",

"mezzanine.forms",

"mezzanine.pages",

"mezzanine.galleries",

"mezzanine.twitter",

#"mezzanine.accounts",

#"mezzanine.mobile",

"djcelery"

)

###### WARNING

Note the @ at the end of the BROKER\_URL setting—this is required, and that the AWS access secret may contain unsafe characters that require URL-encoding. This can be easily accomplished with the ever-handy jq:

$ **echo -n 'hCJ/Fn3nE378Hb7WjGpSYHa9TRCsia/4UcAd+MG7' | jq -R -r @uri**

hCJ%2fFn3nE378Hb7WjGpSYHa9TRCsia%2f4UcAd%2bMG7

Login credentials can be provided using the environment variablesAWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEY, in which case the broker url may be simply set to sqs://. This is really convenient during development, while leaving account credentials in the shell environment may be quite ill advised in production.

Because we created our queue in the default us-east-1 region, we do not need to add the BROKER\_TRANSPORT\_OPTIONS setting. If you choose to create your queue in another region, you will need to uncomment this setting.

The final Celery setting (CELERY\_IMPORTS) makes sure that Celery loads the task from your tasks.py file. By convention, Celery tasks are contained within a Python module named tasks. Django applications can contain their own tasks module that provides tasks relevant to the application’s purpose. For example, a Django application that integrates with Twitter might provide tasks to asynchronously post new tweets or scrape profile information.

Celery can autodiscover task modules that are stored within Django applications. Because your tasks.py file lives in the root of the web application directory, you must explicitly import it.

INSTALLED\_APPS is a core Django setting that lists the applications that make up the project: Python libraries that are automatically loaded and used by Django. In order to add djcelery, which is the Django module provided by django-celery, to this list, you need to duplicate the entire list in the local\_settings.py file. This is one of many ways to customize Django settings.

With that done, you can launch the Celery worker process, which will wait for new tasks to be written to the SQS queue. Launch the process:

sudo -u mezzanine python manage.py celery -A tasks worker --loglevel=info

sudo -u mezzanine celery -A myblog worker -l info

If everything is configured correctly, you will see output similar to the following:

root@ip-10-32-34-116:/srv/myblog# **python manage.py celery worker --loglevel=info**

-------------- celery@celery v3.1.13 (Cipater)

---- \*\*\*\* -----

--- \* \*\*\* \* -- Linux-3.2.0-40-generic-x86\_64-with-Ubuntu-14.04-precise

-- \* - \*\*\*\* ---

- \*\* ---------- [config]

- \*\* ---------- .> broker: sqs://ABCDEFG@localhost//

- \*\* ---------- .> app: default:0x298b250 (djcelery.loaders.DjangoLoader)

- \*\* ---------- .> concurrency: 1 (processes)

- \*\*\* --- \* --- .> events: OFF (enable -E to monitor this worker)

-- \*\*\*\*\*\*\* ----

--- \*\*\*\*\* ----- [queues]

-------------- .> celery: exchange:celery(direct) binding:celery

[Tasks]

. myblog.tasks.process\_comment\_async

The final line of this output shows that your tasks.py module has been loaded successfully.

With Celery running, create a new blog post in your Mezzanine site by visiting a URL of the form http://example.com/admin/blog/blogpost/add/ (substitute your own domain name). After creating the blog post, browse to the post’s page on the main site. If your blog post title was test, for example, this will be http://example.com/blog/test/.

Post a comment on the blog page by filling in the comment form. After clicking the Comment button, you will see some activity in the Celery console, as the task is received and executed.

Assuming your example comment was not particularly spammy, and did not contain the string spam, it will remain on the site after the Celery task has completed. Otherwise, it will be hidden shortly after being posted.

In this example application, we are choosing to display comments by default and hide them if they prove to be spam. Another option would be to hide comments by default and display them only if they are not spam.

# Celery: Updating Puppet and CloudFormation

With Celery working, we can update the Puppet and CloudFormation configurations. This will differ slightly from the changes required to add ElastiCache and RDS, because Celery and the web application will be running on separate instances. Therefore, we need to define a new role in our Puppet manifests so that the correct processes will be started on each instance.

Begin by updating the puppet/manifests/site.pp file, adding celery as one of the available role types:

case $role {

"web": { $role\_class = "myblog::web" }

**"celery": { $role\_class = "myblog::celery" }**

default: { fail("Unrecognised role: $role") }

}

Remember that we update $role\_class to dynamically include Puppet modules based on the instance’s user data, so instances with a $role of celery will use the myblog::celery module.

[Example 5-21](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#example_myblog_celery_pp) shows the myblog::celery module.

##### ***Example 5-21. Celery Puppet module***

class myblog::celery {

Class["myblog::celery"] -> Class["myblog"]

supervisor::service { "myblog\_celery":

ensure => present,

enable => true,

command => "/usr/bin/python ${myblog::app\_path}/manage.py celery",

user => "mezzanine",

group => "mezzanine"

}

}

Save this module to puppet/modules/myblog/manifests/celery.py. This simple module ensures that Supervisor starts the Celery process. All of the heavy lifting is done in other parts of the myblog module—one of the many reasons for separating Puppet modules into separate manifest files.

When installing Celery, we made some changes to the local\_settings.py file. These changes must also be made to the template used for this file, with one modification. Append the code from [Example 5-20](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#celery_local_settings_py) to puppet/modules/myblog/templates/local\_settings.py.erb and then replace the BROKER\_URL setting with the following:

BROKER\_URL = 'sqs://'

This removes the AWS credentials from the broker URL, telling Celery to use the keys provided by the IAM role assigned to the EC2 instance.

Now we can add Celery to the CloudFormation stack. We want to make the following changes to the stack:

1. Create an SQS queue to store Celery messages.
2. Create an IAM policy that allows web and Celery instances to write to the SQS queue.
3. Create an EC2 instance to run Celery.
4. Update the ElastiCache and RDS security groups to permit Celery access.
5. Update the Web EC2 instance so it can use the SQS queue name as a dynamic setting.

This requires changes to cloudformation/myblog.json. For the sake of clarity, we will gradually update this file in a series of small steps.

Begin by adding the SQS queue:

"CeleryQueue": {

"Type": "AWS::SQS::Queue"

},

Add this text to the Resources section of myblog.json, at the same level as WebInstance.

Next, we will create the IAM policy and role that will set up the AWS access credentials that Celery and web instances use to access the SQS queue:

"MyBlogRole": {

"Type": "AWS::IAM::Role",

"Properties": {

"AssumeRolePolicyDocument": {

"Statement": [ {

"Effect": "Allow",

"Principal": {

"Service": [ "ec2.amazonaws.com" ]

},

"Action": [ "sts:AssumeRole" ]

} ]

},

"Path": "/"

}

},

"MyBlogRolePolicies": {

"Type": "AWS::IAM::Policy",

"Properties": {

"PolicyName": "MyBlogRole",

"PolicyDocument": {

"Statement" : [ {

"Effect" : "Allow",

"Action" : [ "sqs:\*" ],

"Resource" : "{ "Ref" : "CeleryQueue" }"

} ]

},

"Roles": [ { "Ref": "MyBlogRole" } ]

}

},

"MyBlogInstanceProfile": {

"Type": "AWS::IAM::InstanceProfile",

"Properties": {

"Path": "/",

"Roles": [ { "Ref": "MyBlogRole" } ]

}

},

Again, insert this text into myblog.json at the same level as WebInstance.

The final new resources are the Celery EC2 instance and associated security group:

"CeleryInstance" : {

"Type" : "AWS::EC2::Instance",

"Properties" : {

"SecurityGroups" : [ { "Ref" : "CelerySecurityGroup" } ],

"KeyName" : "my-ssh-keypair",

"ImageId" : { "Ref" : "CeleryAMI" },

"IamInstanceProfile": {

"Ref": "MyBlogInstanceProfile"

},

"UserData" : {

"Fn::Base64" : {

"Fn::Join" : [ "", [

"{\"role\": \"celery\",",

" \"db\_endpoint\": \"", { "Fn::GetAtt": [ "BlogDB", "Endpoint.Address" ] }, "\",",

" \"db\_user\": \"", { "Ref": "DBUser" }, "\",",

" \"db\_password\": \"", { "Ref": "DBPassword" }, "\",",

" \"cache\_endpoint\": \"", { "Fn::GetAtt": [ "CacheCluster", "ConfigurationEndpoint.Address" ] }, "\"}"

] ]

}

}

}

},

"CelerySecurityGroup" : {

"Type" : "AWS::EC2::SecurityGroup",

"Properties" : {

"GroupDescription" : "Allow SSH from anywhere",

"SecurityGroupIngress" : [

{

"IpProtocol" : "tcp",

"FromPort" : "22",

"ToPort" : "22",

"CidrIp" : "0.0.0.0/0"

}

]

}

},

"CelerySecurityGroupIngress": {

"Type": "AWS::ElastiCache::SecurityGroupIngress",

"Properties": {

"CacheSecurityGroupName" : { "Ref" : "CacheSecurityGroup" },

"EC2SecurityGroupName" : { "Ref" : "CelerySecurityGroup" }

}

},

Insert this text into the Resources section of myblog.json, at the same level as the WebInstance resource. The CeleryInstance resource also uses a reference to the AMI input, meaning it will use the same AMI as the WebInstance.

That’s it for the new resources. We need to make a few other changes to this file before it is complete.

The CelerySecurityGroupIngress resource gives the CeleryInstance access to the ElastiCache cluster. We also need to allow Celery to access the RDS database instance, which requires another modification to myblog.json:

"DBSecurityGroup" : {

"Type" : "AWS::EC2::SecurityGroup",

"Properties" : {

"GroupDescription" : "Allow inbound MySQL access from web instances",

"SecurityGroupIngress" : [

{

"IpProtocol" : "tcp",

"FromPort" : "3306",

"ToPort" : "3306",

"SourceSecurityGroupName" : { "Ref" : "WebSecurityGroup" }

}**,**

**{**

**"IpProtocol" : "tcp",**

**"FromPort" : "3306",**

**"ToPort" : "3306",**

**"SourceSecurityGroupName" : { "Ref" : "CelerySecurityGroup" }**

**}**

]

},

Update the DBSecurityGroup resource definition as shown here, so that the CelerySecurityGroup is listed in the DBSecurityGroupIngress attribute.

The WebInstance resource also requires some changes:

"WebInstance" : {

"Type" : "AWS::EC2::Instance",

"Properties" : {

"SecurityGroups" : [ { "Ref" : "WebSecurityGroup" } ],

"KeyName" : "my-ssh-keypair",

"ImageId" : { "Ref" : "WebAMI" },

**"IamInstanceProfile": {**

**"Ref": "MyBlogInstanceProfile"**

**},**

"UserData" : {

"Fn::Base64" : {

"Fn::Join" : [ "", [

"{\"role\": \"web\",",

" \"db\_endpoint\": \"", { "Fn::GetAtt": [ "BlogDB", "Endpoint.Address" ] }, "\",",

" \"db\_user\": \"", { "Ref": "DBUser" }, "\",",

" \"db\_password\": \"", { "Ref": "DBPassword" }, "\",",

" \"cache\_endpoint\": \"", { "Fn::GetAtt": [ "CacheCluster", "ConfigurationEndpoint.Address" ] }, "\"}"

] ]

}

}

}

},

This code shows the updated version of the WebInstance resource. Note the addition of the IamInstanceProfile property, which makes the EC2 instance inherit the permissions described in the IAM policy.

With that change complete, myblog.json now contains a full description of our stack resources.

# Building the AMIs

With the Puppet configuration complete, the next step is to create the AMI we will use for the EC2 instances. We will do this using the method described in [“Building AMIs with Packer”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch04.html#building_amis_with_packer), in the previous chapter.

For demonstration purposes, we will use Packer to build two AMIs: a web AMI and a Celery AMI. Because these images are similar, you could reduce the AMI management overhead by using a single AMI for each role.

Begin by creating a directory to store the Packer configuration files. This should be at the same level as the puppet directory:

mkdir ~myblog/packer

Copy [Example 4-4](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch04.html#install_puppet_sh) to packer/install\_puppet.sh, as described in [“Building AMIs with Packer”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch04.html#building_amis_with_packer).

First, we will create the configuration for the web AMI, which is shown in [Example 5-22](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#packer_conf_web). Save this file to packer/web.json.

##### ***Example 5-22. Packer configuration: Web***

{

"variables": {

"aws\_access\_key": "",

"aws\_secret\_key": ""

},

"provisioners": [

{

"type": "shell",

"script": "install\_puppet.sh"

},

{ "type": "puppet-masterless",

"manifest\_file": "puppet/manifests/site.pp",

"module\_paths": ["puppet/modules"]

}

],

"builders": [{

"type": "amazon-ebs",

"access\_key": "",

"secret\_key": "",

"region": "eu-west-1",

"source\_ami": "ami-89b1a3fd",

"instance\_type": "m1.small",

"ssh\_username": "ubuntu",

"associate\_public\_ip\_address": true,

"ami\_name": "myblog-web-",

"user\_data": "{\"role\": \"web\"}"

}]

}

This amazon-ebs object contains a user\_data parameter. This is passed to the instance that Packer uses to create the AMI. Puppet will use this user data to control which configuration classes are applied to the instance during the provisioning step. In this case, we want to build an image for the web role, so we provide a JSON string setting the role to web.

###### TIP

User data can also be stored in a separate file, rather than cluttering up your Packer configuration file. This is especially useful when your user data contains large JSON strings. To do this, set user\_data\_file to the path of the file containing your user data.

Now we can create the Packer configuration file for the Celery role. The only difference is the role value specified as part of the user\_data. Copy packer/web.json to packer/celery.json, changing the user\_data and ami\_name to read as follows:

"ami\_name": "myblog-celery-",

"user\_data": "{\"role\": \"celery\"}"

With the configuration files for each role created, we can now build the AMIs, starting with the web role:

packer build web.json

Once Packer finishes creating the AMI, it will output the AMI ID. Make note of this, as we will need it for the next section.

Now we can create an AMI for the Celery instance:

packer build celery.json

Again, make a note of the AMI ID output by Packer.

With the AMIs created, we can proceed with bringing up the CloudFormation stack.

# Creating the Stack with CloudFormation

Now that we have created the AMIs containing our application, we can launch the CloudFormation stack. We will do this using the aws command-line tool.

The stack accepts parameters that we can specify on the command line. Execute the following command to begin creating the stack, replacing the parameter values where necessary:

aws cloudformation create-stack --region eu-west-1 --stack-name myblog-stack \

--template-body file://myblog.json \

--parameters ParameterKey=CeleryAMI,ParameterValue=*ami-XXXXXXXX* \

ParameterKey=WebAMI,ParameterValue=*ami-*XXXXXXXX \

ParameterKey=DBUser,ParameterValue=*myblog\_user* \

ParameterKey=DBPassword,ParameterValue=*s3cr4t* \

ParameterKey=KeyName,ParameterValue=*mike-ryan*

###### TIP

If there are any syntax errors in the stack template, they will be highlighted here. Common errors include unbalanced parentheses and brackets, and misplaced commas.

Use the describe-stack-status command to check on the status of the stack:

aws cloudformation describe-stacks --stack-name myblog-stack | jq '.Stacks[0].StackStatus'

While the resources are being created, the stack’s status will be CREATE\_IN\_PROGRESS. Once the resources have been successfully created, this will change to CREATED. Any other status means an error has occurred, and more information will be available in the Events tab of the Management Console, or by running the describe-stack-events command:

aws cloudformation describe-stack-events --stack-name myblog-stack

Once the stack has been created, find the public DNS name of the web instance by querying the outputs of the stack with the describe-stacks command:

aws cloudformation describe-stacks --stack-name myblog-stack | jq '.Stacks[0].Outputs[]'

Open this address in your web browser, and you should be greeted with the Mezzanine welcome page.

Adding a more friendly DNS name with Route 53 is left as an exercise for you.

# Recap

Although Mezzanine was used an example, the core concepts in this chapter are applicable to nearly all applications.

Imagine you are a web design agency using Mezzanine as your CMS. Using the information in this chapter, you could set up a test environment for new clients in a few minutes, just by creating a new stack for each client.

If you follow the process shown in this chapter, incrementally building an application by first making manual changes before committing those to Puppet and CloudFormation, you may save lots of debugging time.