RUNNING DJANGO ON AWS

Using Python 2.7 and Django >= 1.6 on AWS

This is a step-by-step guide to deploying a Django app to AWS. Note, I mention Python 2.7 specifically. This is because AWS has Python 2.6 as the default version for it's linux distribution (as of writing this guide) . Django version 1.6 is the last one to support Python 2.6 and with Django 1.7 already out in the wild, it seems prudent to set up our stack for Python 2.7 (if not 3+).

Things we'll set up:

a. Using Amazon's EC2 Management console to provision a server. For this, you need an AWS account with Amazon (aws.amazon.com).

b. Install Apache and mod_wsgi. Python 2.7 requires a few additional steps to setup and compile mod_wsgi because the mod_wsgi that EC2 currently comes with is compiled against Python 2.6.

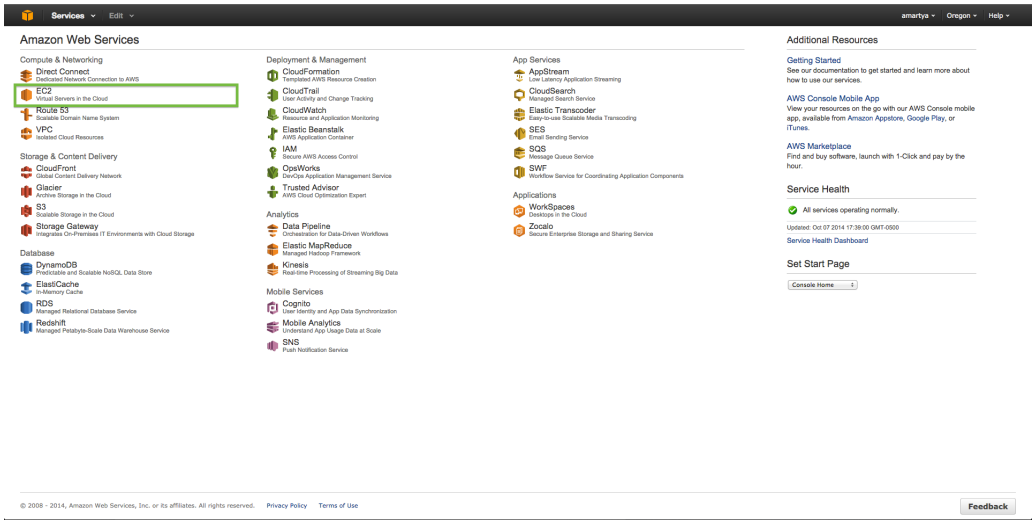
c. Set up a database. In this case, MySQL

d. Install Django

e. Create a Django app and configure it to be served via Apache

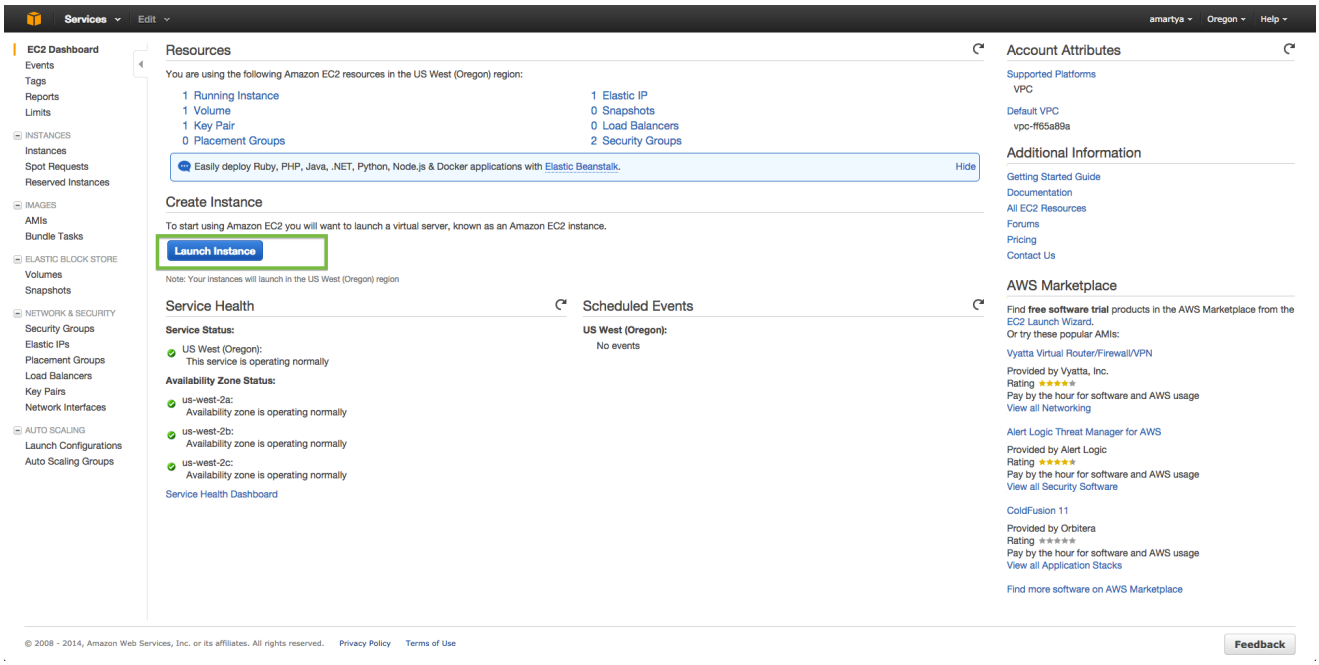
Starting an AWS instance

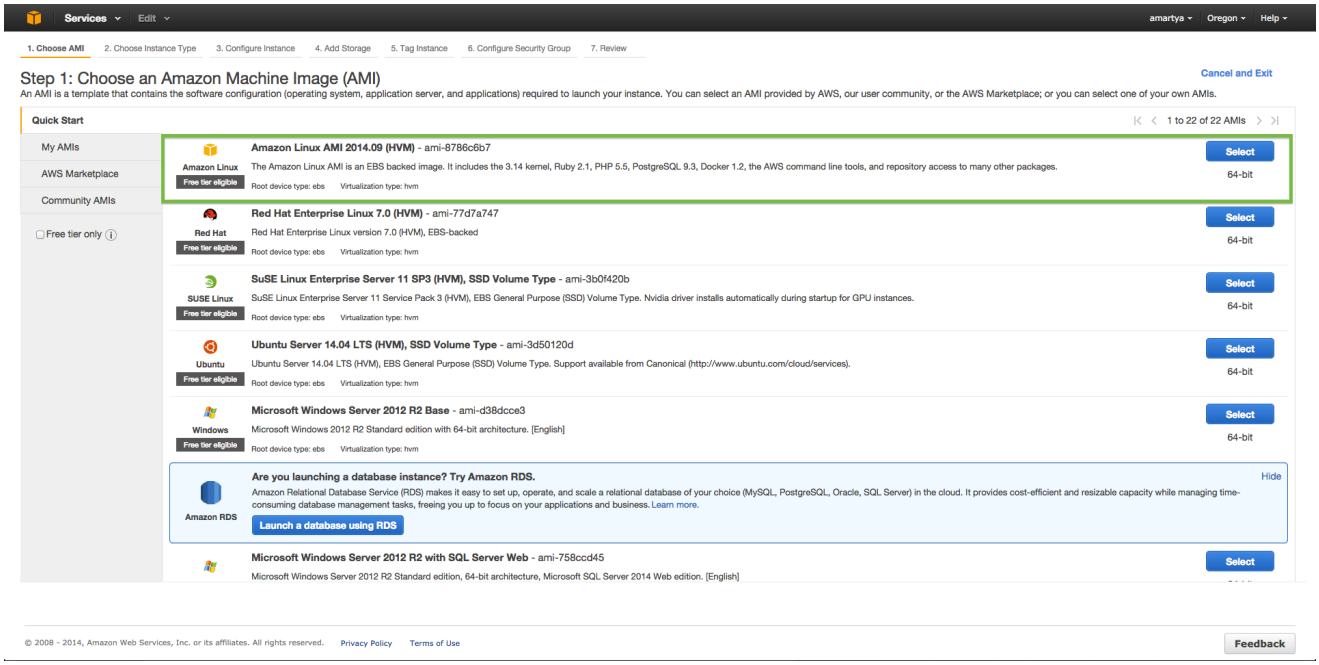
1.



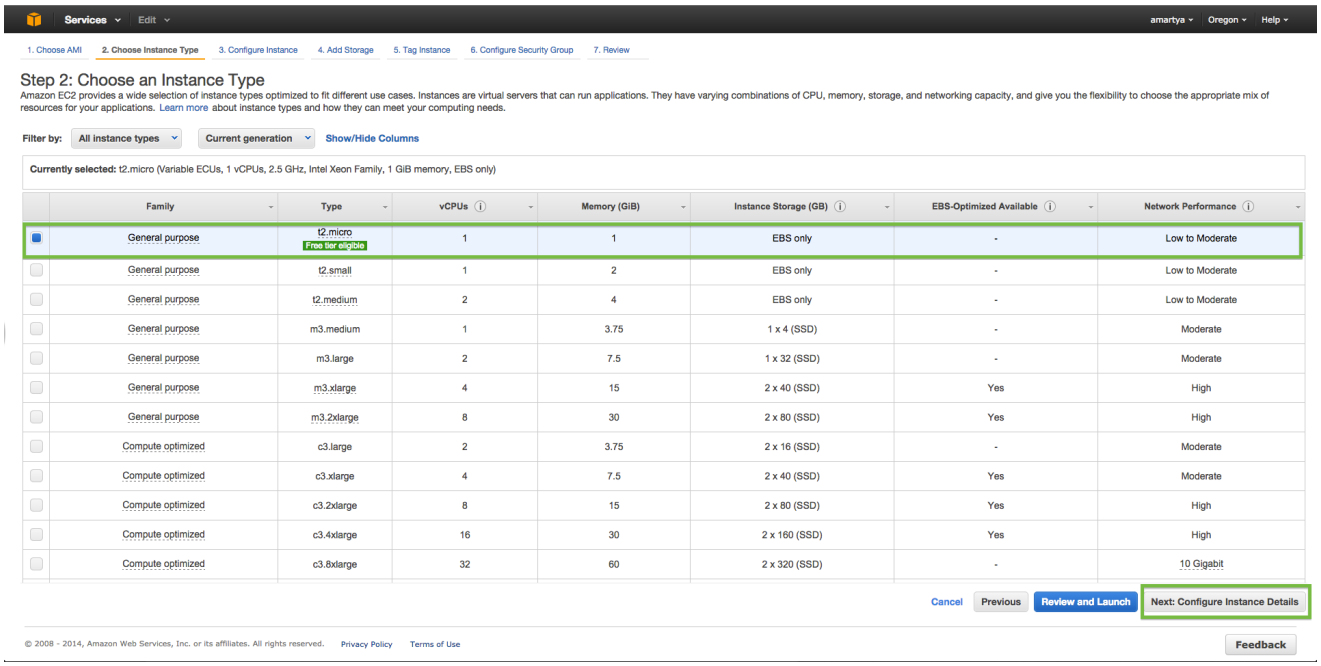
After logging in to the AWS management console, select EC2

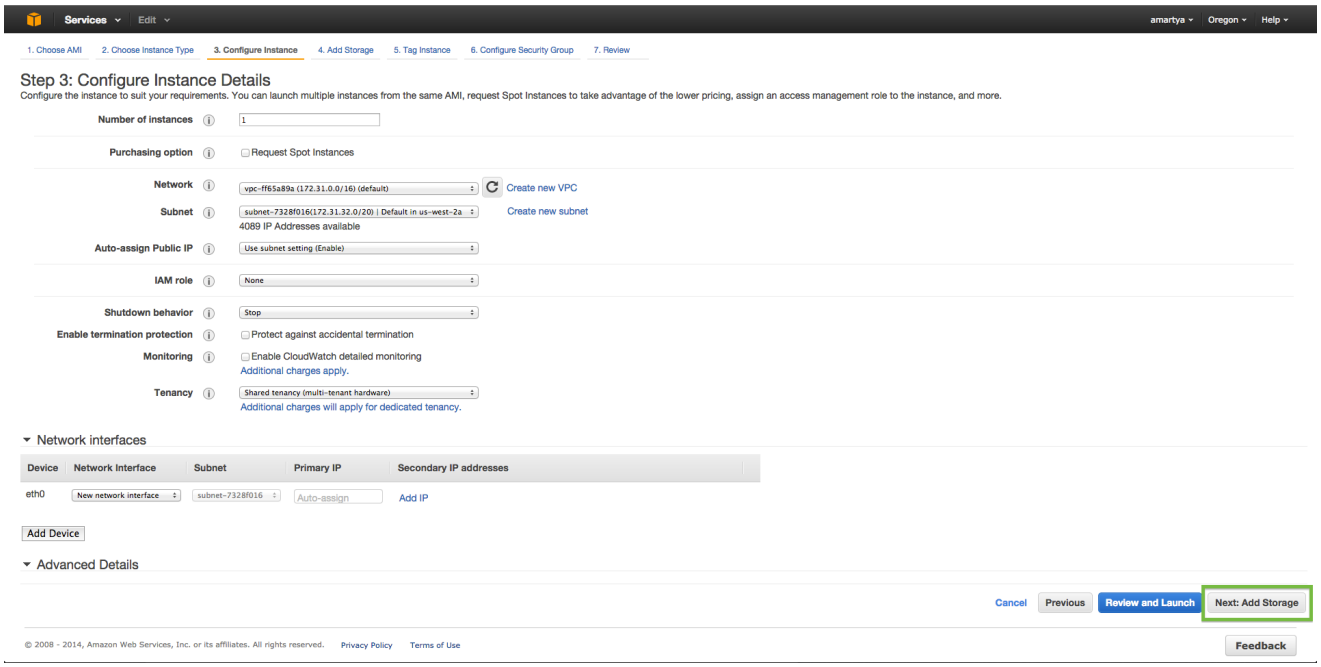
2. Follow the screenshots below to get an instance up and running.



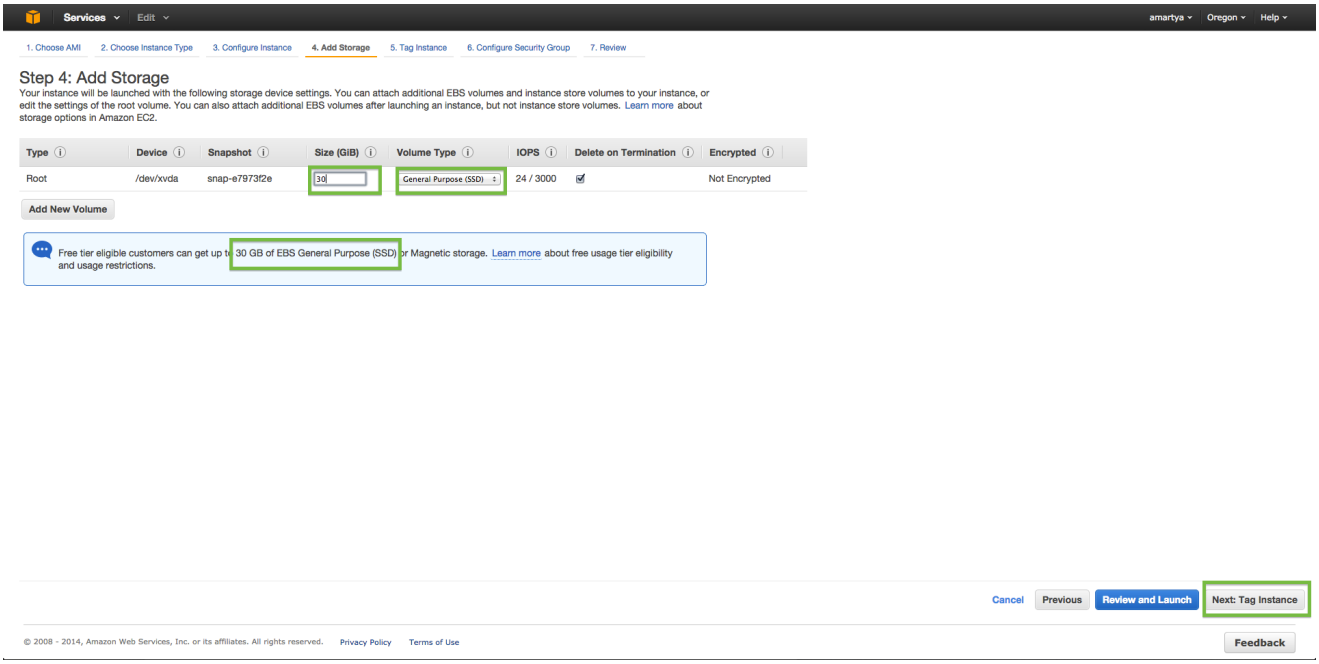


We'll go with the Amazon flavoured Linux distro because they include a lot of the stuff we need by default.

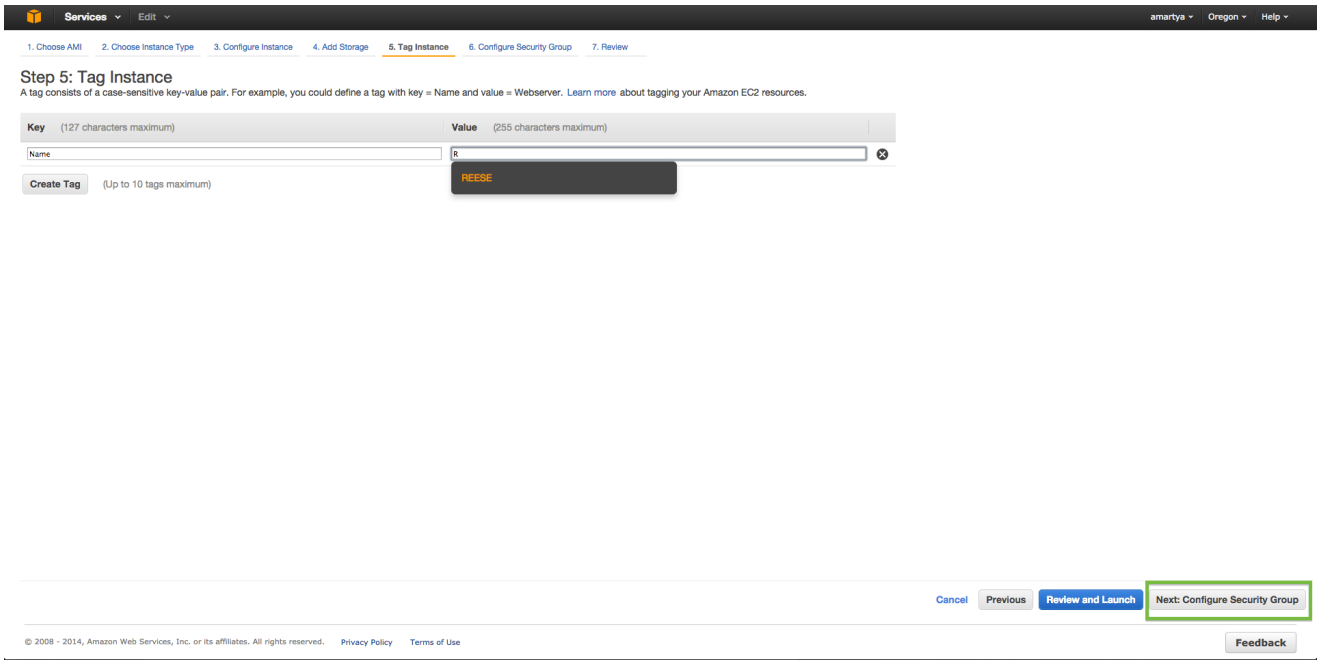




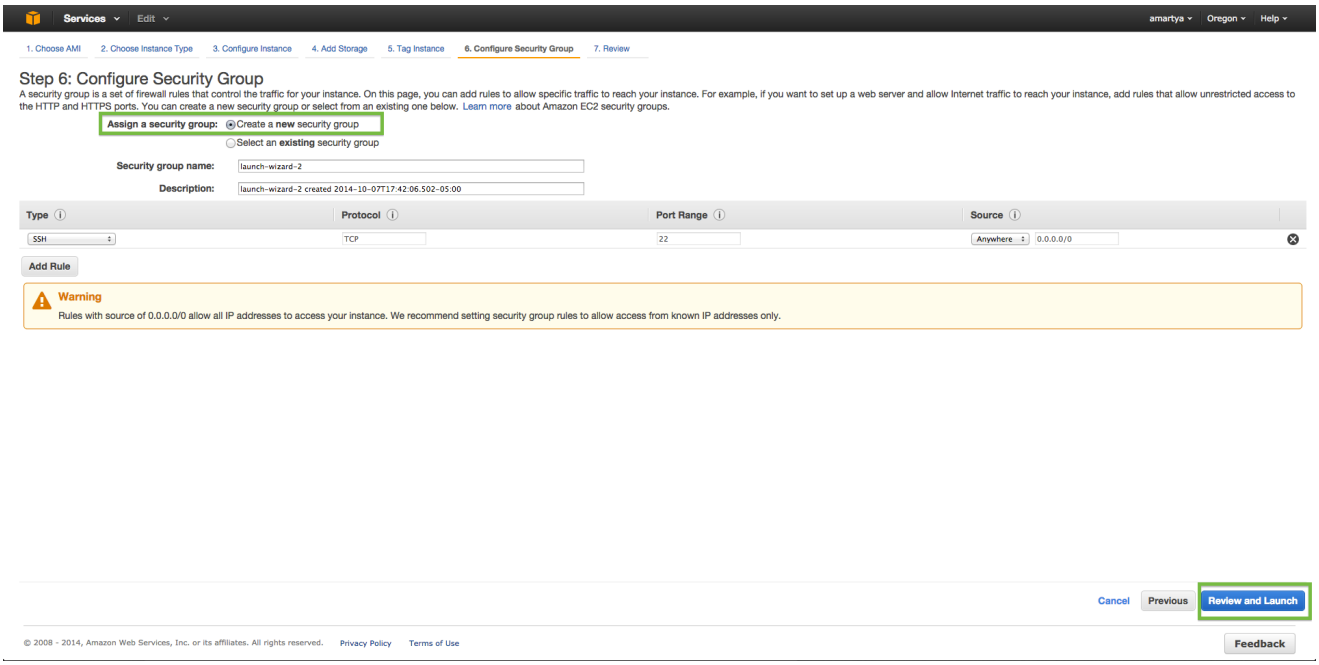
Nothing to do on this particular screen, move on to the next screen (“Add Storage”).



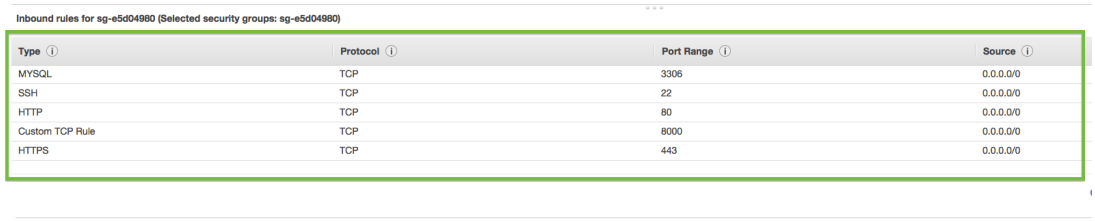
Maxing out the default storage for now. Amazon has something called RDS that allows us to attach resizable storage (you can see the option in the very first screen where we chose EC-2). However, it seems to be an overkill for our purposes and it seems prudent to have as much storage assigned to EC2 initially as possible.

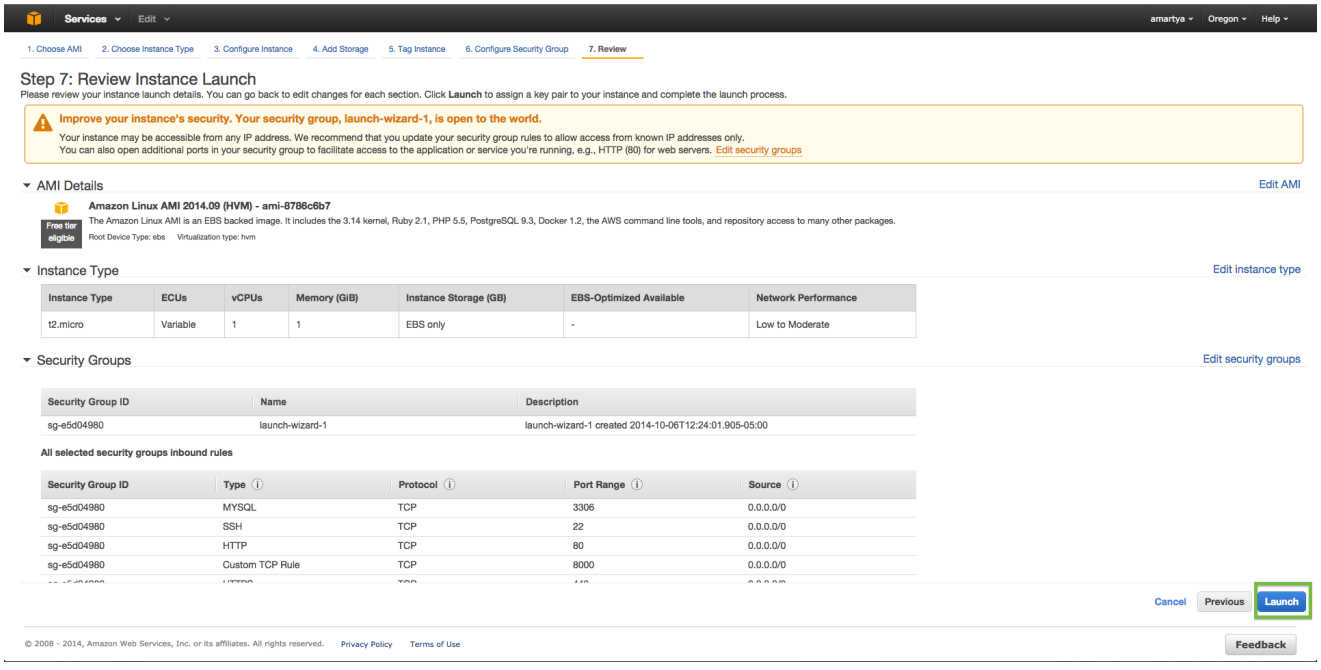


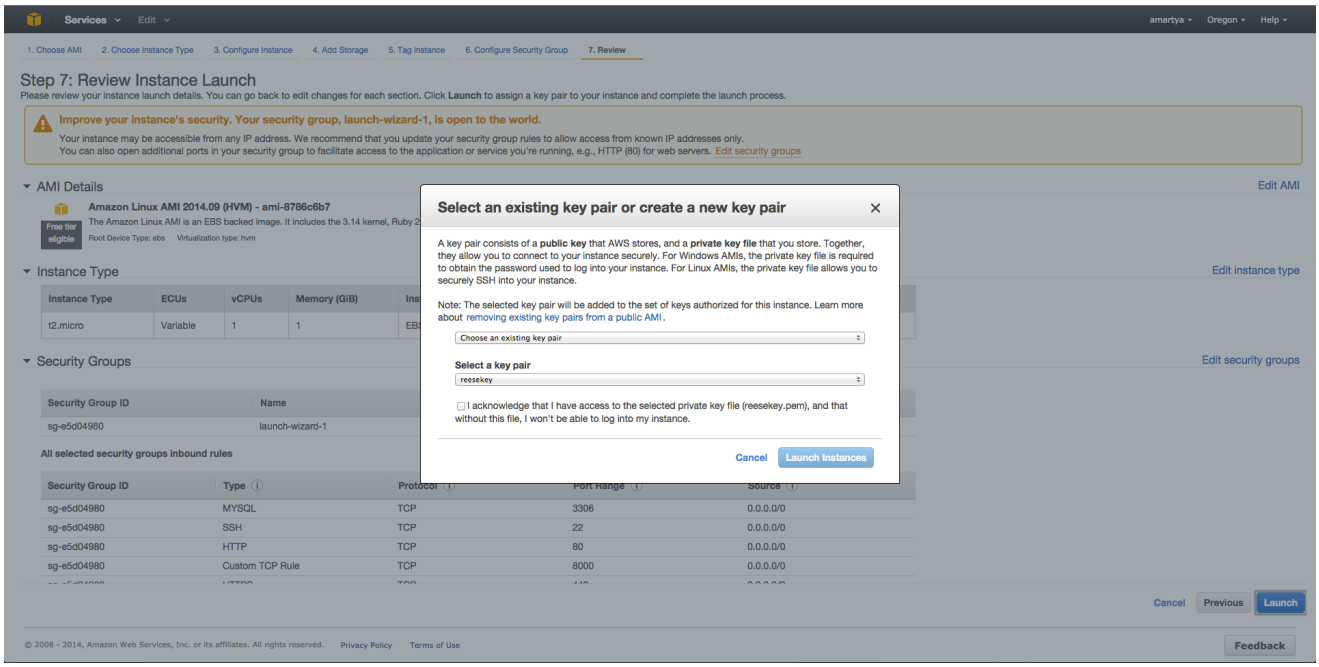
Give this instance a Name key to make it easier to track in the EC2 management console.



The security group setting is IMPORTANT. Copy the configuration shown below. This basically says what kind of traffic/ ports should the server allow. If you already have a security group and wish to reuse it, just add all these rules.

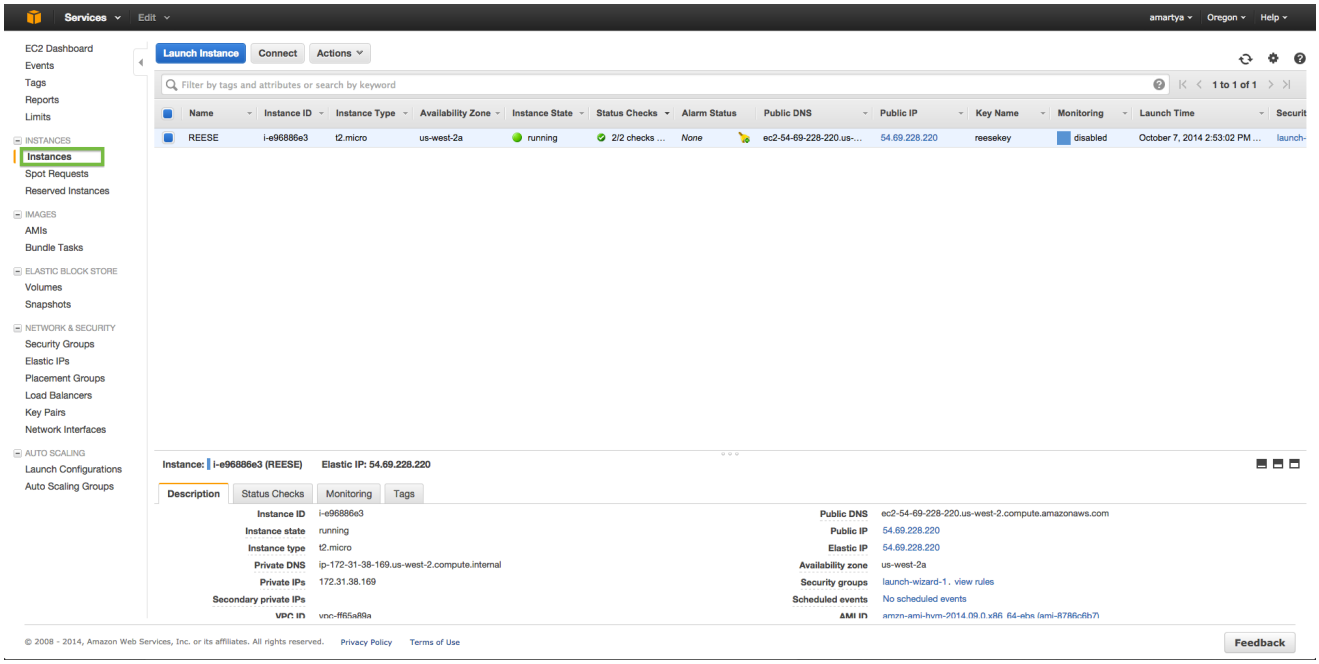




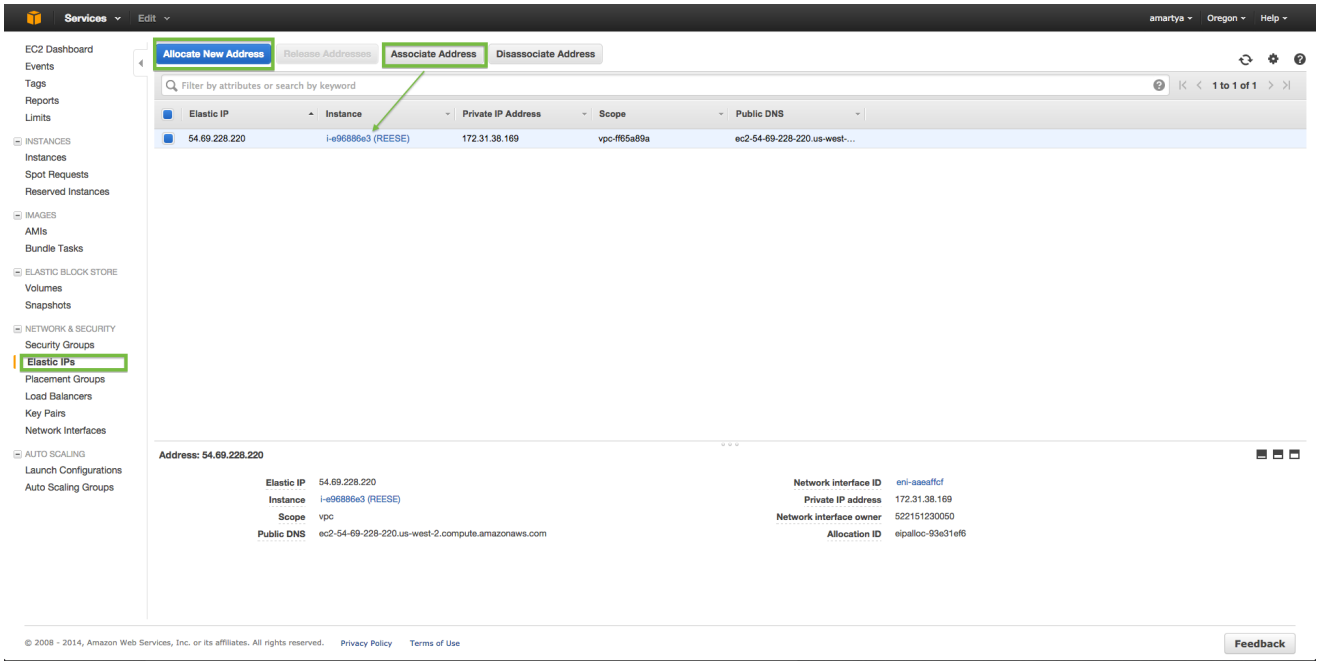


I already had a key setup. However, you can create a new key by selecting “Create a new Key Pair” and giving it a name. Download the key (I have it in a folder on Dropbox). We’ll have to edit the folder and permissions for this key.

We are almost done, just Launch Instance and navigate to the “Instances” section under the EC2 management console. You should see something similar to the screenshot below.



AWS assigns us a public IP and a DNS. However, these are for a machine in the cloud that might be restarted. Which implies that this IP could randomly change. To fix that, we need to create an "Elastic IP" and "Associate" it with the EC2 instance that we just created.



That's it! Once we have an elastic IP pointing to our instance, a server restart on Amazon's end would no longer result in our work being lost in the ether :).

Moving on to setting up the setup that includes apache, django etc. from the next page.

Installing Apache and Mod_Wsgi

1. First, we have to login to our EC2 instance that we set up. Fire up Terminal on the Mac and navigate to the folder containing the key you downloaded (the *.pem file). You can copy paste the commands from the text boxes that follow:

```
#Login
#Can use the domain name (ec2-aws....) instead of the I.P. too
chmod 400 reesekey.pem (change file permissions to be able to add the reesekey.pem as an ssh key)
ssh-add reesekey.pem
cd path_to_ #keypair location
ssh -i reesekey.pem ec2-user@54.69.228.220 #use elastic IP.

#if it throws a warning saying that the "remote host identification has changed", run "ssh-keygen -R
54.69.228.220"
#and then follow if up with the "ssh -i reesekey.pem ec2-user@54.69.228.220" command
```

2. Once we are logged in, try using the command “pwd” (present working directory). It should say something like “/home/ec2-user”. Now run “python -V”. Currently EC2 comes with Python 2.6, we’ll start by updating it to Python 2.7 for future Django versions. We also install Pip, a fantastic python package manager :).

```
# install build tools
sudo yum install make automake gcc gcc-c++ kernel-devel git-core -y

# install python 2.7 and change default python symlink
sudo yum install python27-devel -y
sudo rm /usr/bin/python
sudo ln -s /usr/bin/python2.7 /usr/bin/python

# yum still needs 2.6, so write it in and backup script
sudo cp /usr/bin/yum /usr/bin/_yum_before_27
sudo sed -i s/python/python2.6/g /usr/bin/yum
sudo sed -i s/python2.6/python2.6/g /usr/bin/yum

# should now display 2.7.5 or later instead of 2.6xx:
python -V

# now install pip for 2.7
#Just to keep things neat, create a downloads directory in ec2-user
# for all application packages (rpm, tar etc.)
cd /home/ec2-user
mkdir downloads
cd downloads
wget https://bootstrap.pypa.io/get-pip.py
sudo python get-pip.py

# should display current version of pip:
pip -V
```

3. Install apache

```
#Apache
sudo yum install httpd #installs apache as a service

sudo service httpd start
#starts apache, test by going to http://<your elastic IP> here it was http://54.69.228.220
sudo service httpd stop #stops apache
sudo service httpd restart #restarts apache
sudo chkconfig httpd on #starts the Apache service whenever the server reboots
```

You should see the Apache default welcome screen.

4. Just having Apache is not going to work when we try to serve a Django view. We need another interface called mod_wsgi that acts as a conduit. While mod_wsgi comes with EC2 as defaultt, we'll have to do some grunt work to get mod_wsgi to work with the updated Python 2.7 that we got going (the original version that came with the EC2 instance was compiled for Python 2.6).

```
#Mod-Wsgi
#wsgi = Web Server Gateway Interface, a python spec. for server to application communication
# install development apache tools
yum install httpd-devel.x86_64

#downloads and compiles mod_wsgi to be used for python2.7, I recommend using the downloads folder
# you created while installing pip (step-2) before you run the wget
wget http://modwsgi.googlecode.com/files/mod_wsgi-3.4.tar.gz
tar -zvxf mod_wsgi-3.4.tar.gz
cd mod_wsgi-3.4
./configure --with-python=/usr/bin/python2.7
make
sudo make install
```

Installing MySQL

Next, we'll install MySQL. After installing MySQL, we need to ensure that we install the python driver for it, create a database along with a user (root) and password. The commands are all listed below:

```
#MySQL
sudo yum install mysql mysql-server MySQL-python mysql-devel
sudo pip install MySQL-python #python driver for MySQL
service mysqld start #start MySql server
/usr/bin/mysqladmin -u root password 'R33535' # Set password for MySQL root user'
mysql -u root -p -e 'CREATE DATABASE reese' (if it asks asks for pwd: R33535)
#Enter python shell and "import MySQLdb" to test installation
#if something went wrong and you wish to reinstall mysql
#sudo yum -y remove mysql-server
```

Installing and configuring Django App

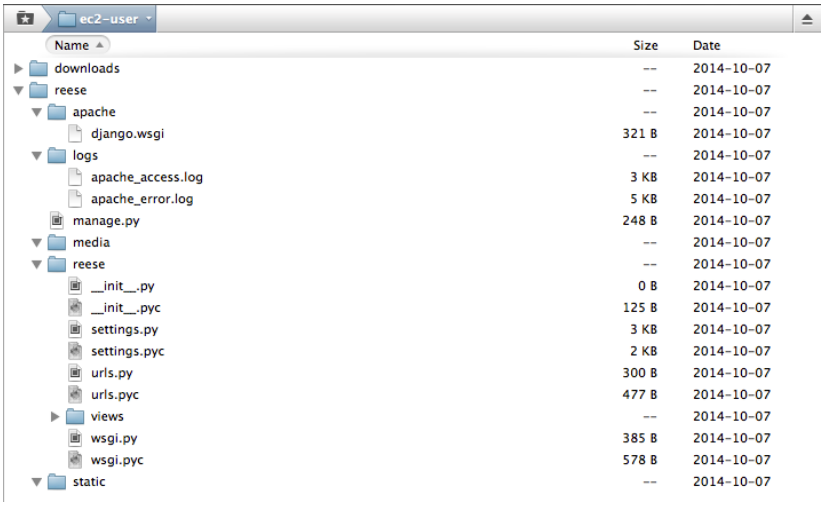
In this part of the guide, we'll install Django, create a basic Django app and then serve it via the mod_wsgi and Apache layers we'd added in the previous step.
Once you've installed the app, we need to configure our server to serve this Django app. Go to the next page for the apache mod_wsgi settings.

```
#Django
sudo pip install django==1.6

#If this fails, get the latest package from the django servers and follow the steps below
wget https://www.djangoproject.com/download/1.7/tarball/ #download Django
mv index.html Django-1.7.tar.gz #renaming file manually, giving a file name didn't work in the prev.
step
tar xvf Django-1.7.tar.gz #extract tar archive, x= get, v= verbose, w= interactive(not used here),
f=file/hostname
cd Django-1.7
python setup.py install #installs django
#Enter python shell and "import django" to test installation

#create basic Django project in a directory of choice, I created it in my /home/ec2-user directory
django-admin.py startproject reese

#If you go inside the reese directory, you'd notice that Django has created another directory called
#reese inside 'this' reese directory. That's normal :). In the outer reese directory, you should see
#a file called manage.py at the same level as another "reese" directory
```

Here's a slightly tricky part. It's tricky only because we need to pay extra attention to the file paths that we lay out. Bear with me :), it isn't as bad as it might seem. To start off, we need to access/edit Apache's config file (httpd.conf).

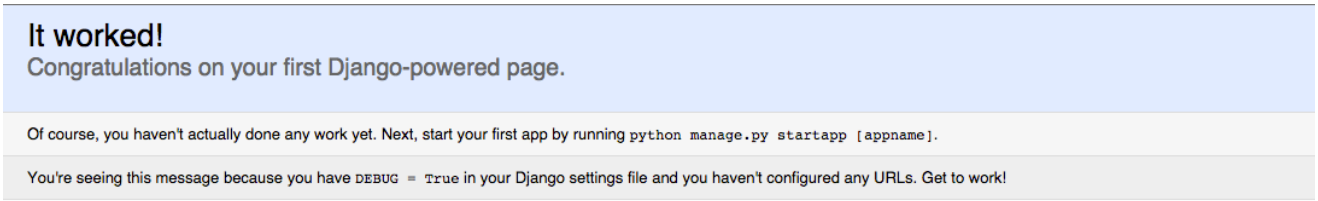
```
#Configure apache to serve from Django app
#Edit the httpd.conf file in /etc/httpd/conf/
sudo vim /etc/httpd/conf/httpd.conf

# Add to the end of the opened file. tip: use ctrl+d to scroll down & ctrl+u to scroll up
WSGIPythonHome /usr
WSGIPythonPath /home/ec2-user/reese:/usr/lib64/python2.7/site-packages:/usr/lib64/python2.7
<VirtualHost *:80>
        DocumentRoot /home/ec2-user/reese
        ErrorLog /home/ec2-user/reese/logs/apache_error.log
        CustomLog /home/ec2-user/reese/logs/apache_access.log combined
        WSGIScriptAlias / /home/ec2-user/reese/apache/django.wsgi
        Alias /static/ /home/ec2-user/reese/static/

        <Directory /home/ec2-user/reese/media>
                Order deny,allow
                Allow from all
        </Directory>

        <Directory /home/ec2-user/reese/apache>
                Order deny,allow
                Allow from all
        </Directory>

        LogLevel warn
</VirtualHost>

#THEN find a couple of lines that say User and Group in the same file, mine were line numbers 243
#and 244 and change them to ec2-user (screenshot below)

                # User/Group: The name (or #number) of the user/group to run httpd as.
                #  . On SCO (ODT 3) use "User nouser" and "Group nogroup".
                #  . On HPUX you may not be able to use shared memory as nobody, and the
                #    suggested workaround is to create a user www and use that user.
                #  NOTE that some kernels refuse to setgid(Group) or semctl(IPC_SET)
                #  when the value of (unsigned)Group is above 60000;
                #  don't use Group #-1 on these systems!
                #
                User ec2-user
                Group ec2-user

                ### Section 2: 'Main' server configuration
                #
                # The directives in this section set up the values used by the 'main'
                # server, which responds to any requests that aren't handled by a
                # <VirtualHost> definition.  These values also provide defaults for
                # any <VirtualHost> containers you may define later in the file.
                #
                -- INSERT --                                                    244,15        18%

#save and exit this file (:wq is the Vim command)
#Run apachectl configtest  to ensure that the file is syntactically correct

#We've done a few things here. We've told apache that it should redirect all HTTP traffic (port 80)
#to the /home/ec2-user/reese directory. You see a reference to a django.wsgi file, folders such as
#media, static, apache and logs. These directories don't exist right now, we'll create them :).

# Go to the parent reese folder (/home/ec2-user/reese), create 2 folders, logs and apache
# In the apache folder, create a django.wsgi file (note the WSGIScriptAlias variable in the httpd
conf. file above)
# Similarly, create blank log files called apache_error.log and apache_access.log in the logs folder
# Just ensure that the paths make sense w.r.t. the configuration shown above (httpd.conf)
```

Basically, your directory structure should be similar to the screenshot below (screenshot from FTP client):



Don't worry if you don't have a directory called views in there, that's something I created later on. After the https.conf file has been edited and saved, restart the server (sudo service https restart).

With a all fingers crossed and a prayer, go to http:<your elastic I.P.> (in this case http://http://54.69.228.220/) and you'd should see this :)



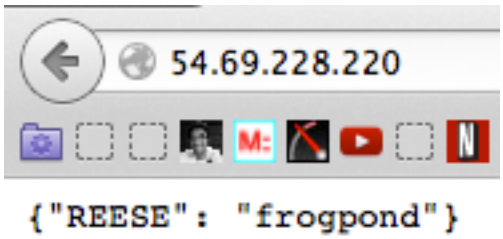
So right now Django is running in Debug mode. We need to turn off debug mode and setup Django to use the MySQL database we had set up. To do that, we need to edit the settings.py file

```
vim /home/ec2-user/reese/reese/settings.py

#find the DATABASE variable and change it from the default sqllite to MySQL

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql', # Add 'postgresql_psycopg2', 'mysql', or 'sqlite3'
        'NAME': 'reese',
        'USER': 'root',
        'PASSWORD': 'R33535',
        'HOST': '',
        'PORT': '', # Set to empty string for default.
        'OPTIONS': {
        "init_command": "SET foreign_key_checks = 0;",
        },
    }
}

#then find the ALLOWED_HOSTS variable and change it to:
ALLOWED_HOSTS = ['*']

#finally, find the variable called DEBUG and set it to false
DEBUG = False
```

To test if things are still working, create a view. I prefer having a folder that contains all my views. So I created a "views" folder in the same directory that contains the settings.py file. Make sure you create a file (empty file) called ***__init.py__*** inside this views folder, it tells django to look inside this directory for files. I created a view called "home" with a function called index that returns a json.

```
#This is the home.py file

from django.template import RequestContext,Context
from django.core.urlresolvers import reverse
from django.http import HttpResponse
from django.http import HttpResponseRedirect
from django.shortcuts import redirect
from django.shortcuts import render
from django.views.decorators.csrf import csrf_protect
from django.shortcuts import render_to_response
from django.conf import settings

import json

def index(request):
    return HttpResponse(json.dumps({'REESE' : 'frogpond'}), content_type="application/json")


#then edit the urls.py to something like

from django.conf.urls import patterns, include, url
from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    url(r'^$', 'reese.views.home.index', name='home'),
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),
)
```

Now, going to http://<your elastic I.P>, i.e. http://54.69.228.220/ in this case, should result in the following page:



Success!

Next: We look at how to setup Django to enable us to post data without running into cross-origin resource issues (e.g.when posting data from a standalone app to our end-point/server)

Getting Django and Dart to play nice

We want to be able to POST data to our server while getting around cross-site security issues that most browsers check against. Broadly, we'll do the following:

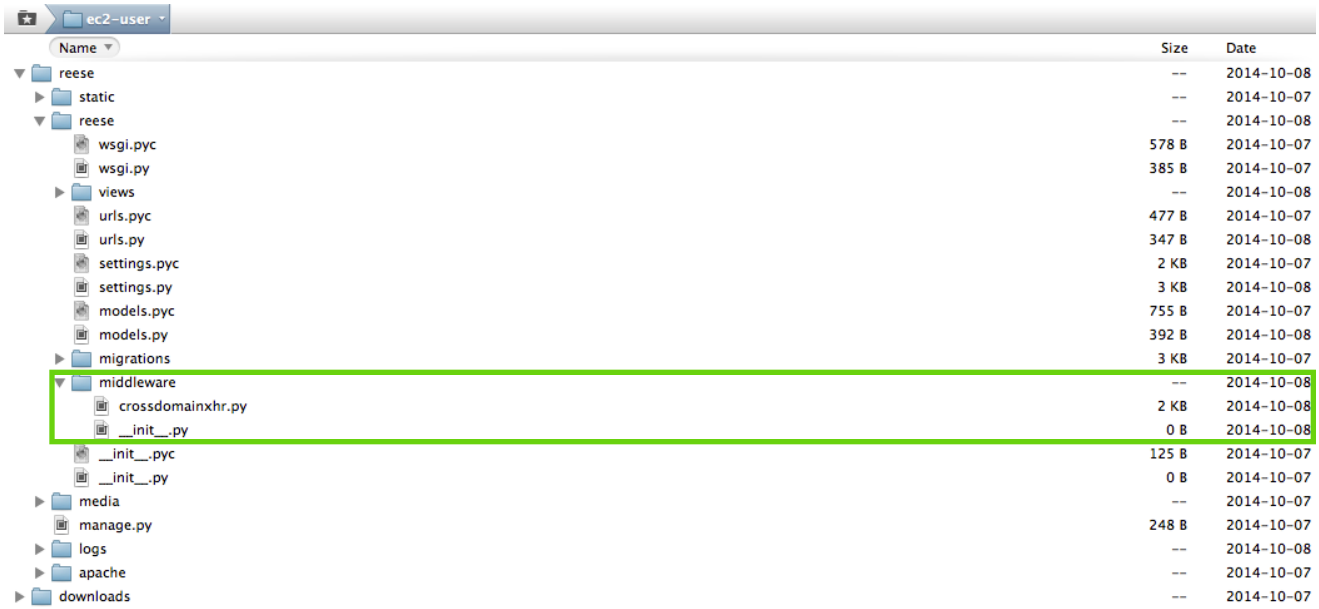
a. Setup Django according to the specs laid out in CORS (Cross-Origin Resource Sharing).
b. Create and POST a json request from Dart.
c. Look at Django's csrf (cross-site request forgery) security issue

Django Setup for CORS

While we could have manually set this up according to the CORS specs in each view, since Django supports plugins/middleware, it is neater to add CORS relevant code as a middleware.

In your Django app, create a folder called "middleware". Inside it create 2 files, __init.py__ (empty file that tells Django that this directory is a part of your project) and crossdomainxhr.py

This is what the directory structure should look like



Inside the crossdomainxhr.py file, paste the following code

```
from django import http

try:
    from django.conf import settings
    XS_SHARING_ALLOWED_ORIGINS = settings.XS_SHARING_ALLOWED_ORIGINS
    XS_SHARING_ALLOWED_METHODS = settings.XS_SHARING_ALLOWED_METHODS
    XS_SHARING_ALLOWED_HEADERS = settings.XS_SHARING_ALLOWED_HEADERS
    XS_SHARING_ALLOWED_CREDENTIALS = settings.XS_SHARING_ALLOWED_CREDENTIALS
except AttributeError:
    XS_SHARING_ALLOWED_ORIGINS = '*'
    XS_SHARING_ALLOWED_METHODS = ['POST', 'GET', 'OPTIONS', 'PUT', 'DELETE']
    XS_SHARING_ALLOWED_HEADERS = ['Content-Type', '*']
    XS_SHARING_ALLOWED_CREDENTIALS = 'true'


class XsSharing(object):
    """
    This middleware allows cross-domain XHR using the html5 postMessage API.

    Access-Control-Allow-Origin: http://foo.example
    Access-Control-Allow-Methods: POST, GET, OPTIONS, PUT, DELETE

    Based off https://gist.github.com/426829
    """
    def process_request(self, request):
        if 'HTTP_ACCESS_CONTROL_REQUEST_METHOD' in request.META:
            response = http.HttpResponse()
            response['Access-Control-Allow-Origin']  = XS_SHARING_ALLOWED_ORIGINS
            response['Access-Control-Allow-Methods'] = ",".join( XS_SHARING_ALLOWED_METHODS )
            response['Access-Control-Allow-Headers'] = ",".join( XS_SHARING_ALLOWED_HEADERS )
            response['Access-Control-Allow-Credentials'] = XS_SHARING_ALLOWED_CREDENTIALS
            return response

        return None

    def process_response(self, request, response):
        response['Access-Control-Allow-Origin']  = XS_SHARING_ALLOWED_ORIGINS
        response['Access-Control-Allow-Methods'] = ",".join( XS_SHARING_ALLOWED_METHODS )
        response['Access-Control-Allow-Headers'] = ",".join( XS_SHARING_ALLOWED_HEADERS )
        response['Access-Control-Allow-Credentials'] = XS_SHARING_ALLOWED_CREDENTIALS

        return response
```

Now, we need to add this to our settings file (settings.py) and assign a couple of variables:

```
#Cross-site sharing related vars
XS_SHARING_ALLOWED_ORIGINS = '*'
XS_SHARING_ALLOWED_METHODS = ['POST','GET','OPTIONS']
XS_SHARING_ALLOWED_HEADERS = ['Content-Type', '*']

.
.
.
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'reese.middleware.crossdomainxhr.XsSharing', # JUST ADD THIS to MIDDLEWARE_CLASSES
)
```

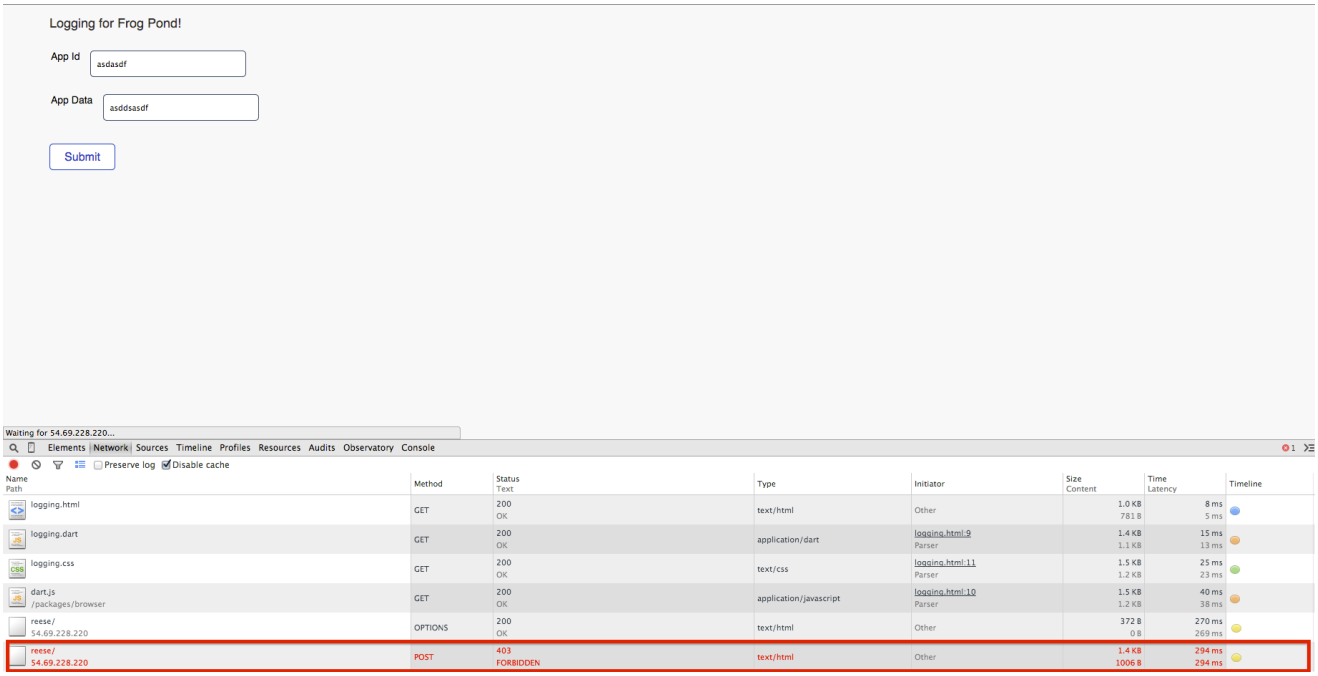
Basically, we just told our Django app to accept cross-origin requests of type POST/GET/ OPTIONS/ etc. Restart you server for good measure even though it's not mandatory ("sudo service https restart")

Dart Code

We can post from any machine. Notice the url we are posting to, it has to be a valid url defined in urls.py in Django.

```
import 'dart:html';
import 'dart:convert';

void main() {
  querySelector("#submit-btn")
      ..text = "Submit"
      ..onClick.listen(sendData);
}

void sendData(MouseEvent event) {
  InputElement appId = document.querySelector("#app-id");
  InputElement appData = document.querySelector("#app-data");

  HttpRequest request = new HttpRequest(); // create a new XHR

  // add an event handler that is called when the request finishes
  request.onReadyStateChange.listen((_) {
    if (request.readyState == HttpRequest.DONE &&
        (request.status == 200 || request.status == 0)) {
      // data saved OK.
      print(request.responseText); // output the response from the server
    }
  });

  //Store form data
  Map data = new Map();
  data['app_id'] = appId.value;
  data['app_data'] = appData.value;


  //setup to POST data to the server
  //request.open("POST","http://54.69.228.220/reese/", async: false);
  request.open("POST","http://54.69.228.220/reese/", async: true);

  //we can set the content type to "application/x-www-form-urlencoded; charset=UTF-8"
  //otherwise the Django side gets an empty POST Query dictionary,
  //If we set the content-type to application/json, we can use the request.body
  //variable in Django
  //request.setRequestHeader("Content-type","application/x-www-form-urlencoded;
charset=UTF-8");

  request.setRequestHeader("Content-type","application/json");

  request.send(JSON.encode(data)); // perform the async POST
}
```

One would expect that this would be enough, however, if you try to post something right now, you'd get a 403 response similar to the screenshot below



We need to do one last thing :).

Setting the Django csrf flag

It's also called a decorator in Django parlance. Adding a decorator "@csrf_exempt" just above your Django view would do it

```
from django.views.decorators.csrf import csrf_exempt

#include the decorator "@csrf_exempt" for all views that receive data without csrf token
@csrf_exempt
def reeselog(request):
    if request.method == "POST":
        try:
            #read request data as json from request body
            request_data = json.loads(request.body)

            #store the app_id and app_data payload
            app_id = request_data['app_id']
            app_data= request_data['app_data']
        except:
            app_id = "appID error"
            app_data = "appData error"
            return HttpResponse(json.dumps({'status': 'failed to saved data'}),
content_type="application/json")

        #create a log object and save it in the database
        log_data = FrogPondLog(name=app_id, data=app_data)
        log_data.save()

        return HttpResponse(json.dumps({'status': 'data saved successfully'}),
content_type="application/json")
```

Now, if you try to POST data from the remote/cross-site, it should work :)