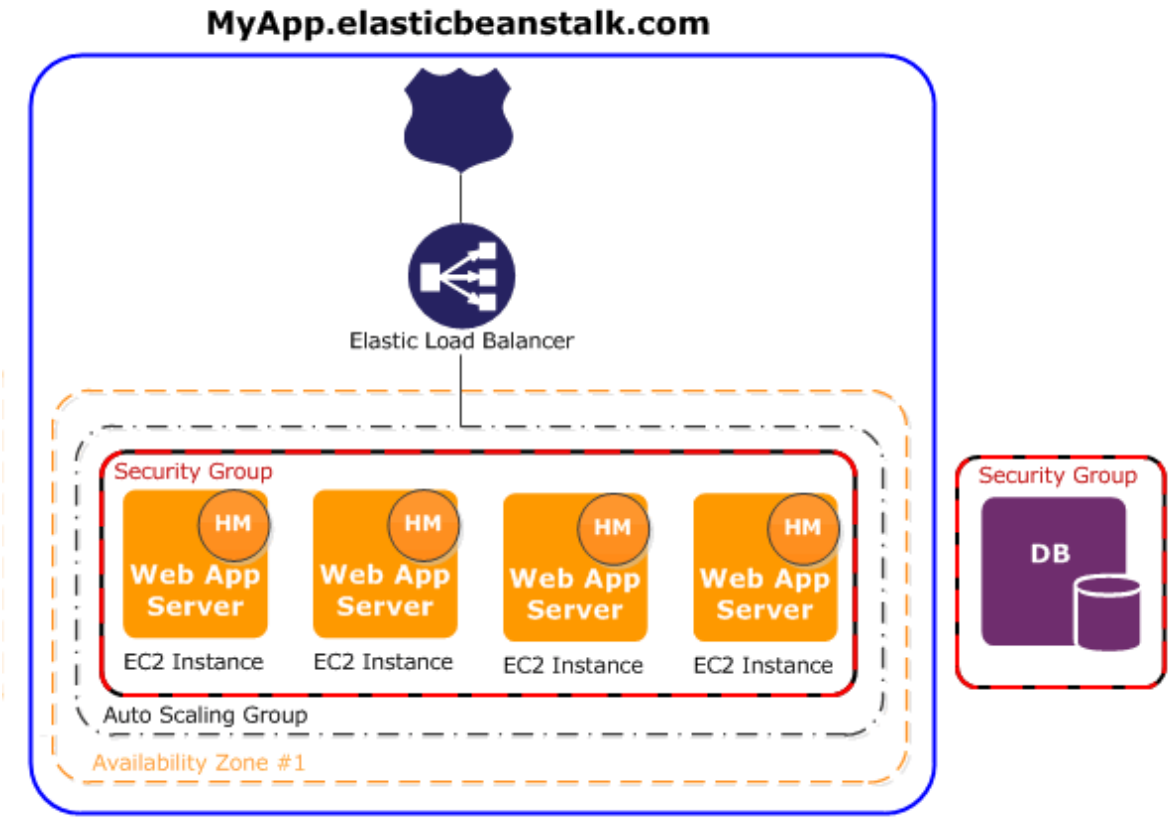
DEPLOYING DJANGO APPLICATIONS USING AWS ELASTIC BEANSTALK

Needless to say, to follow this guide you need to set up an AWS account. The process is quite straightforward and you should be up and running in a matter of minutes. But before we go any further, a word of caution: AWS is notorious for 'bill shock'. Even though its Free Tier is vast and this entire guide lies within its purview, a simple mis-click can cost you thousands of dollars if you aren't careful. This isn't meant to scare you, but rather to serve as a reminder to be vigilant.

All of the code for this guide is available at: https://github.com/chouhanaryan/deploying-django-beanstalk.

Let us begin by looking at how Beanstalk works.



For each project that you wish to deploy, you create an Application (eg. Inventory-Management-System). For each application, you can create different Environments depending upon your requirements. For example, you could create different environments for Development, Testing and Deployment with each configured differently. The blue line in the figure above corresponds to an environment. With each environment, among other things, you get an Elastic Load Balancer, an Auto Scaling Group and one or more EC2 instances.

An EC2 instance is nothing but a virtual computer that you are provided access to. Your application runs within the EC2 instance. However, each instance is capable of handling only a certain amount of traffic. If your application ends up requiring more than one, you will probably need to pay for it. This guide will not be venturing into that territory.

A Load Balancer is used to route traffic to different EC2 instances in situations where your application is experiencing a lot of load.

An Auto Scaling Group is used to dynamically increase the number of EC2 instances in case your load exceeds capacity. In cases where there isn't much traffic, unrequired instances still exist, but are shut down.

A Security Group is used to define protocols that dictate access points to the application. If your Security Group allows only HTTP access, and you try to SSH into your EC2 instance – it won't work.

Your database is kept separate from all of this. There are two different services you need to be aware of when it comes to databases and Beanstalk – S3 and RDS.

An S3 bucket is where Beanstalk stores application versions along with other meta-data such as log files and temporary configuration files. Beanstalk does this automatically, and you have no control over this.

RDS on the other hand is an EC2 instance that is specially configured to simulate a database environment such as Postgresql, MySQL, etc. Depending upon the requirements of your application, you can manually set up an RDS instance and link it to your environment, or you can do this from within the environment console itself. If you choose the latter, the lifecycle of your RDS instance is tied to the lifecycle of your environment which is not ideal for a production environment. Manually setting up the link allows multiple environments to be linked to the same database and this can allow you to switch environments with no downtime, among other things. However, for this guide, we shall be not be setting up RDS manually. It is useful to note that both S3 and RDS also have independent Security Groups that can customized.

NOTE

A few things:

1. I will be using PowerShell as my terminal for this guide.
2. Our primary objective shall be to get everything up and running first. Hence, we shall work with SQLite for the most part. I recommend you try deploying an app with SQLite before moving to another database.

SET UP

First, begin by installing awsebcli. Install this globally.

```
PS D:\eb-django\project> pip install awsebcli
```

Next, set up a virtual environment and activate it. You can use any library for this purpose. I happen to prefer virtualenv.

```
PS D:\eb-django > virtualenv env
PS D:\eb-django > env\Scripts\activate
```

You should now see the `(env)` prefix before your terminal commands.

```
(env) PS D:\eb-django > virtualenv env
```

Now, let us install Django. Note that Beanstalk only supports Django 2.1. For the purposes of this guide, we will be using Django 2.1.4. You are free to choose any other patch.

```
(env) PS D:\eb-django> pip install django==2.1.4
```

You can now start working on your project.

Ensure that you have at least one app within it – this will come into play later on.

You can also install any other packages you might require.

Add `STATIC_ROOT = 'static'` to the `settings.py` file.

Make sure that everything is working the way you want it to, and then proceed.

Next, create a `requirements.txt` file within the project.

```
(env) PS D:\eb-django\project> pip freeze > requirements.txt
```

Your project directory should now roughly look like this:

```
|   manage.py
|   requirements.txt
├───app
|   |   admin.py
|   |   apps.py
|   |   models.py
|   |   tests.py
|   |   views.py
|   |   __init__.py
|   └───migrations
|           __init__.py
└───project
        settings.py
        urls.py
        wsgi.py
        __init__.py
```

UNDERSTANDING BEANSTALK AND DATABASES

Now that our essentials are covered, let us pause and think about our future steps.
When our site goes live, we will need a way to interact with all of our data. Usually while testing, we set up superusers locally and we directly work with our data via the default Django admin console. How then, will we manage to do this when our site lives in the cloud?

Let us take a step back and dive deeper into how Beanstalk works.
Every time you deploy on Beanstalk, a .zip file of your entire source code (called the 'application version') is created and pushed to your EC2 instance (actually stored on S3 bucket). This is then extracted and run. We can also specify certain instructions to execute immediately before or after deployment to help you set certain things up.

This leaves us with two options -

1. You could simply create a database file (`db.sqlite3`) locally via migrations and directly upload it with the source code. This would vastly simplify things as you could create superusers locally and later, directly access the Django admin console because superusers already exist!
   For this, all you need to do is run your migrations, create a superuser and sit tight. The `db.sqlite3` file should be created in your project directory and you are good to go.
2. Or – you could choose NOT to upload the database file directly and instead write commands that create the database then and there. This is a tad bit complicated since it forces us to automate the superuser creation process, but if you wish to switch to another database later on, it becomes easier. If you wish to learn more about the way Django works and such, I recommend going down this path.

Let us now look at how we can use configuration files (`.ebextensions`).

Configuration files are YAML or JSON formatted documents with a `.config` extension. They are stored in a folder called `.ebextensions` inside the project directory. They contain instructions that can be used to configure our environments and customize the AWS resources that they contain. These files are the key to automating deployment and are one of the reasons the Beanstalk service is so powerful. You can name these files anything (`*.config`) and can even have multiple such files.

Let us look at the config file which we will require if we choose **Option 2** above:

```
container_commands:
  01_migrate:
    command: "python manage.py migrate"
    leader_only: true
  02_sqlite_permissions:
    command: "sudo chmod 777 db.sqlite3"
    leader_only: true
```

```
  03_collectstatic:
    command: "python manage.py collectstatic --noinput"
    leader_only: true
  04_create_superuser:
    command: "python manage.py createsu"
    leader_only: true
  05_wsgi_pass:
    command: "echo 'WSGIPassAuthorization On' >> ../wsgi.conf"

option_settings:
  aws:elasticbeanstalk:application:environment:
    DJANGO_SETTINGS_MODULE: project.settings
  aws:elasticbeanstalk:container:python:
    WSGIPath: project/wsgi.py
```

This file consists of two parts:

1. `container_commands` – used to execute commands that affect the applications source code. The above options specify the following commands to be run, in an alphabetical order (which is why it is a good idea to prepend them with numbers)

    - Run migrations and create database

    - Change access permissions to the database file created

        The db.sqlite3 file that is created after migrations, is read-only. This command changes its access permissions to allow for external access.
        This is very, very bad in a production environment, however – this is the best we can do for now.

    - Collect static files

        The default Django admin console requires CSS files and hence, we need to collect them into a static folder.
        This is why we added `STATIC_ROOT` above.

    - Create a superuser

        The Django createsuperuser command runs in interactive mode by default. You could try to specify parameters in the command itself, but unfortunately, that doesn't allow you to enter a password and it simply doesn't work the way we need it to. This is by design and is one of the reasons why Django is so secure.

        To circumvent this, we need to create our own createsuperuser command (I named it – createsu), that creates a superuser by running the create_superuser()

function that every User model has. The function allows us to specify a username, email and password, and creates a superuser accordingly.

How do we do this? Create a Python package called `management` and another Python package within it called `commands` (a package is nothing but a folder with `__init__.py` in it). Within this folder, create a `supersu.py` file. The code for this is in the repository.

Now place the `management` package in the folder of any app of your project. Locally, you could try running `python manage.py createsu`, and test if it works.

- Set `WSGIPassAuthorization On` in the `wsgi.conf` file

  If you are using the Django Rest Framework to create a RESTful API and have endpoints that receive POST requests, you might want to add this setting. WSGI strips all POST requests of their headers by default and this changes that behaviour.

2. `option_settings` – used to modify configurations.
   The above options specify the following –

   - Set the `DJANGO_SETTINGS_MODULE` of the environment to `project.settings` (settings.py file of our project)

   - Set the `WSGIPath` of the Python container to our projects `wsgi.py` file

     WSGI is to Python what Servlets are to Java. It acts like a gatekeeper and codifies behaviour for different web servers (eg. Nginx and Apache) and application frameworks (eg. Django and Flask) to interact with each other based on a common API. WSGI makes sure that you do not have to learn different rules for every permutation of a possible server-application interaction.
     With this command, we specify the `wsgi.py` file that dictates these rules.

`leader_only: true` ensures that the command runs on one instance only during environment creation and deployments. If you have multiple EC2 instances and you don't specify this command, they will run every single time a new one is provisioned or updated.

If you chose **Option 1** instead, remove command `02` and `04` from `container_commands`, and copy the rest to a config file (eg. `django.config`) inside `project/.ebextensions`. We remove these commands because the appropriate permissions are already applied to your database file and a superuser already exists.

Your file structure should now look like –

```
|   db.sqlite3
|   manage.py
|   requirements.txt
├──.ebextensions
|       django.config
├──app
|   |   admin.py
|   |   apps.py
|   |   models.py
|   |   tests.py
|   |   views.py
|   |   __init__.py
|   └──migrations
|           __init__.py
└──project-
        settings.py
        urls.py
        wsgi.py
        __init__.py
```

You might also have some `__pycache__` folders.

If you chose **Option 2,** copy the above code into a config file (eg. `django.config`) inside `project/.ebextensions`. Remember to delete your migrations folder within each app and the database file, and you should be good to go.

Your file structure should look like –

```
|   manage.py
|   requirements.txt
├──.ebextensions
|       django.config
├──app
|   |   admin.py
```

```
|   |   apps.py
|   |   models.py
|   |   tests.py
|   |   views.py
|   |   __init__.py
|   ├──management
|   |   |   __init__.py
|   |   └──commands
|   |           createsu.py
|   |           __init__.py
|   └──migrations
|           __init__.py
└──project
        settings.py
        urls.py
        wsgi.py
        __init__.py
```

UP, UP AND AWAY!

After deactivating your terminal, run the following command –

```
PS D:\eb-django\project> eb init -i
```

- Select your region (eg. 6)
- Create a new application (eg. django-test)
- Select Python 3.6
- Select Yes when asked if you would like to set up SSH (allows for debugging later)
- To find your keypair
    - Go to My Security Credentials in your AWS Account
    - Click on Access keys
    - Create New Access Key
    - Copy & paste the Access Key ID and Secret Access Key in your terminal

Now, type in –

```
PS D:\eb-django\project> eb create
```

- Create a new environment (eg. project-test)

- DNS CNAME prefix – Beanstalk will generate a link for your web app which will have this DNS CNAME prepended to it (eg. project-test.dmjsengw6a.ap-south-1.elasticbeanstalk.com/)
- Select load balancer type as default (Application)
- Select No when asked if you would like to enable Spot Fleet requests

Wait for the environment to be created – this can take up to 5 minutes.
If you get any errors, rectify them and run –

```
PS D:\eb-django\project> eb deploy
```

Once your environment is successfully created for the first time, run –

```
PS D:\eb-django\project> eb status
```

Copy the `CNAME` and paste it into the `ALLOWED_HOSTS` list in your `settings.py` file.

```
ALLOWED_HOSTS = ['<CNAME>']
```

Now, run –

```
PS D:\eb-django\project> eb deploy
```

This should take a while to complete. If it executed successfully, run –

```
PS D:\eb-django\project> eb open
```

And your website should show up!

You should also be able to go `/admin` and login using the credentials you entered in `createsu.py`!

To SSH into your instance, run –

```
PS D:\eb-django\project> eb ssh
```

and you should be able to access the filesystem too! Even though changing files directly isn't recommended because it defeats the purpose of automated deployment, it can be a handy tool to debug pesky errors.

Another setting you might want to customize at this stage is the Security Group.

- Go to VPC via your AWS Management Console.
  VPC stands for Virtual Private Cloud – a service that allows you to isolate a section of the cloud for yourself, and dictate how the network of the AWS resources that you provision within it, operates.
- Go to Security Groups and add Inbound Rules to the appropriate instances that will allow HTTPS and SSH access (if not configured previously). Note that this is different from truly enabling HTTPS access on your website as that requires an SSL Certificate from AWS Certificate Manager.

POSTGRESQL/MYSQL – DATABASE CONNECTION

However, we aren't done yet – using an SQLite database in a production environment is nothing short of blasphemous.

What's more – connecting a production grade database, like MySQL or PostgreSQL is incredibly easy!

- First, edit your `settings.py` file
  - Change

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

  to

```
if 'RDS_DB_NAME' in os.environ:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.postgresql',
            'NAME': os.environ['RDS_DB_NAME'],
            'USER': os.environ['RDS_USERNAME'],
            'PASSWORD': os.environ['RDS_PASSWORD'],
            'HOST': os.environ['RDS_HOSTNAME'],
            'PORT': os.environ['RDS_PORT']
        }
    }
else:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.sqlite3',
            'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
        }
    }
```

  The engine will depend upon whether you are using Postgres or MySQL.

- Head over to your AWS Management Console and select Elastic Beanstalk (ensure that you are in the same region as that of the application you just deployed)
- Select the appropriate application and environment
- Go to Configuration and select Modify next to Database

- Enter the following settings:
    - Engine: postgres/mysql
    - Engine version: default
    - Instance class: default (DO NOT change this – other instances might be paid services)
    - Storage: default
    - Set an appropriate Username and Password
    - Retention: up to you

        If you want to retain your database in case you terminate your environment, select Create snapshot, else Delete
    - Availability: default (DO NOT change this – other options might be paid services)
- Now, run `eb deploy` once again – and your app should automatically be connected to the RDS instance!
- If you wish to access your database locally via pgAdmin, do the following –
    - Go to your AWS Management Console and select VPC
    - Select the appropriate Security Group that is related to your RDS instance
    - Click on Edit Rules under Inbound Rules
    - Add a rule with Source as Me and select a port (eg. 5432)
    - Copy your RDS endpoint
    - Open pgAdmin and select Add New Server
    - In the General tab, enter the port, username, password and endpoint into host

      You server should now be accessible from pgAdmin as well!

We're finally done!

DONE AND DUSTED

If you wish to, you can skip this section entirely. I address the following here:

- EC2 vs Beanstalk
- An interesting behaviour we glossed over earlier
- What next?

1. EC2 vs Beanstalk

   Like I said earlier, fundamentally – EC2 is just a virtual computer that you can connect to. AWS calls each such computer an instance, and so each time you wish to deploy an application, you have to create an EC2 instance and manually set up a server such as Nginx or Apache on your instance and link the server and your app via the Web Server Gateway Interface (WSGI). Another complication you need to consider is where do you store your source code – you could set up an S3 bucket and store it there, or store it right here on your EC2 instance. Both options have their respective pros & cons. Furthermore, if you wish to set up a database, you would have to manually

set up an RDS instance and connect it to your EC2 instance. It is clear that this is a very laborious process.

Beanstalk is actually just an abstraction over the entire EC2, RDS, S3, etc. stack. How? Each Beanstalk instance consists of an EC2 instance (or instances) that is preconfigured with Nginx, S3 as a bucket to store your code and several other services preconfigured to make your life simpler. All that you need to do is specify your settings in simple .config files.

Ultimately, it all depends upon your requirements. If your application requires extensive customizations that are server or database specific, it would be wise to go with EC2. In all other cases however – Beanstalk is probably what you might be looking for.

2. So, what was it that we missed earlier?
   If you go back and read the `createsu.py` file we created, we explicitly designed it to create a new user only and only if a superuser did not previously exist. Why was this important? While we were using SQLite, this had no consequence whatsoever since every time you deployed, a new database file was being created. However, when it comes to Postgres, the RDS instance it is connected to is the same. If we did not design the code keeping this in mind, each time we deploy – a new superuser would be added to the database! This behaviour would result in massive errors and obviously needs to be caught.

3. If you wish to dive deeper into AWS and explore more concepts, here are a few things you could try:
   - Deploying an application using EC2 directly
   - Using Route 53 to change your domain name
   - Deploying a Flask/Node.js/PHP app

THANK YOU!

Congratulations on making it this far and thank you for reading!

If you have any suggestions/questions, you can contact me at: **chouhanaryan@yahoo.com**.

Until next time!