

Deploying a Django Application to Elastic Beanstalk

NT

Nik Tomazic

[Twitter](#) [Reddit](#) [Hacker News](#) [Facebook](#)

In this tutorial, we'll walk through the process of deploying a production-ready [Django](#) application to [AWS Elastic Beanstalk](#).

Objectives

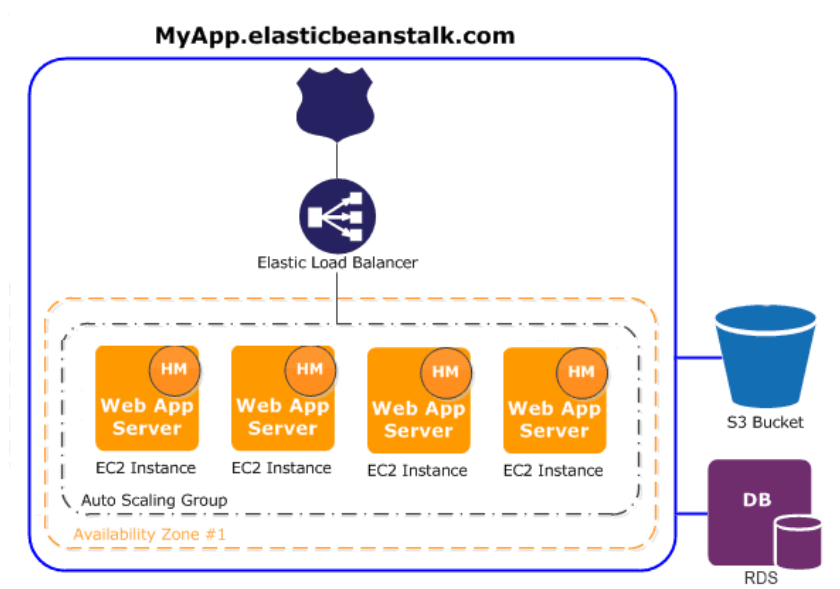
By the end of this tutorial, you'll be able to:

1. Explain what Elastic Beanstalk is
2. Initialize and configure Elastic Beanstalk
3. Troubleshoot an application running on Elastic Beanstalk
4. Integrate Elastic Beanstalk with RDS
5. Configure S3 for static and media file support
6. Obtain an SSL certificate via AWS Certificate Manager
7. Use an SSL certificate to serve your application on HTTPS

What is Elastic Beanstalk?

[AWS Elastic Beanstalk](#) (EB) is an easy-to-use service for deploying and scaling web applications. It connects multiple AWS services, like compute instances ([EC2](#)), databases ([RDS](#)), load balancers ([Application Load Balancer](#)), and file storage systems ([S3](#)), to name a few. EB allows you to quickly develop and deploy your web application without thinking about the underlying infrastructure. It [supports](#) applications developed in Go, Java, .NET, Node.js, PHP, Python, and Ruby. EB also supports Docker if you need to configure your own software stack or deploy an application developed in a language (or version) that EB doesn't currently support.

Typical Elastic Beanstalk setup:



Feedback

There's no additional charge for AWS Elastic Beanstalk. You only pay for the resources that your application consumes.

To learn more about Elastic Beanstalk, check out [What is AWS Elastic Beanstalk?](#) from the [official AWS Elastic Beanstalk documentation](#).

Elastic Beanstalk Concepts

Before diving into tutorial itself, let's look at a few key [concepts](#) related to Elastic Beanstalk:

1. An [application](#) is a logical collection of Elastic Beanstalk components, including environments, versions, and environment configurations. An application can have multiple [versions](#).
2. An [environment](#) is a collection of AWS resources running an application version.
3. A [platform](#) is a combination of an operating system, programming language runtime, web server, application server, and Elastic Beanstalk components.

These terms will be used throughout the tutorial.

Project Setup

We'll be deploying a simple image hosting application called [django-images](#) in this tutorial.

Check your understanding by deploying your own application as you follow along with the tutorial.

First, grab the code from the [repository on GitHub](#).

```
$ git clone git@github.com:duplxy/django-images.git
$ cd django-images
```

Create a new virtual environment and activate it:

```
$ python3 -m venv venv && source venv/bin/activate
```

Install the requirements and migrate the database:

```
(venv)$ pip install -r requirements.txt
(venv)$ python manage.py migrate
```

Run the server:


```
(venv)$ python manage.py runserver
```

Open your favorite web browser and navigate to <http://localhost:8000>. Make sure everything works correctly by using the form on the right to upload an image. After you upload an image you should see it displayed in the table:

django-images

127.0.0.1:8000

django-images

| ID | Title | File | Uploaded at | Preview |
|----|-------|--------------------------|----------------------|---|
| 1 | Cat | cat3.jpg | 02/09/2022 9:19 p.m. |  |

Upload

Fill out the form and press submit in order to upload a new image.

Title*

Feedback

Image*

Choose File No file chosen

Upload

Elastic Beanstalk CLI

Be sure to [register](#) for an AWS account before continuing. By creating an account you might also be eligible for the [AWS Free Tier](#).

The [Elastic Beanstalk command line interface](#) (EB CLI) allows you to perform a variety of operations to deploy and manage your Elastic Beanstalk applications and environments.

There are two ways of installing EB CLI:

1. Via the [EB CLI installer](#)
2. With [pip \(awsebcli\)](#)

It's recommended to install the EB CLI globally (outside any specific virtual environment) using the installer (first option) to avoid possible dependency conflicts. Refer to [this explanation](#) for more details.

After you've installed the EB CLI, you can check the version by running:

```
$ eb --version  
  
EB CLI 3.20.3 (Python 3.10.)
```

If the command doesn't work, you may need to add the EB CLI to `$PATH`.

A list of EB CLI commands and their descriptions can be found in the [EB CLI command reference](#).

Initialize Elastic Beanstalk

Once we have the EB CLI running we can start interacting with Elastic Beanstalk. Let's initialize a new project along with an EB Environment.

Init

Within the project root ("django-images"), run:

```
$ eb init
```

You'll be prompted with a number of questions.

Default Region

The AWS region of your Elastic Beanstalk environment (and resources). If you're not familiar with the different AWS re

Feedback

check out [AWS Regions and Availability Zones](#). Generally, you should pick the region that's closest to your customers. Keep in mind that resource prices vary from region to region.

Application Name

This is the name of your Elastic Beanstalk application. I recommend just pressing enter and going with the default: "django-images".

Platform and Platform Branch

The EB CLI will detect that you're using a Python environment. After that, it'll give you different Python versions and Amazon Linux versions you can work with. Pick "Python 3.8 running on 64bit Amazon Linux 2".

CodeCommit

[CodeCommit](#) is a secure, highly scalable, managed source control service that hosts private Git repositories. We won't be using it since we're already using GitHub for source control. So say "no".

SSH

To connect to the EC2 instances later we need to set up SSH. Say "yes" when prompted.

Keypair

To connect to EC2 instances, we'll need an RSA keypair. Go ahead and generate, which will be added to your "~/.ssh" folder.

After you answer all the questions, you'll notice a hidden directory inside your project root named ".elasticbeanstalk". The directory should contain a *config.yml* file, with all the data you've just provided.

```
.elasticbeanstalk
└─ config.yml
```

The file should contain something similar to:

```
branch-defaults:
  master:
    environment: null
    group_suffix: null
global:
  application_name: django-images
  branch: null
  default_ec2_keyname: aws-eb
  default_platform: Python 3.8 running on 64bit Amazon Linux 2
  default_region: us-west-2
  include_git_submodules: true
  instance_profile: null
  platform_name: null
  platform_version: null
  profile: eb-cli
  repository: null
  sc: git
  workspace_type: Application
```

Create

Next, let's create the Elastic Beanstalk environment and deploy the application:

```
$ eb create
```

Again, you'll be prompted with a few questions.

Environment Name

This represents the name of the EB environment. I'd recommend sticking with the default: "django-images-env".

Feedback

It's considered good practice to add `L-env` or `L-dev` suffix to your environments so you can easily differentiate EB apps from environments.

DNS CNAME Prefix

Your web application will be accessible at `%cname%.%region%.elasticbeanstalk.com`. Again, use the default.

Load balancer

A load balancer distributes traffic among your environment's instances. Select "application".

If you want to learn about the different load balancer types, review [Load balancer for your Elastic Beanstalk Environment](#)

Spot Fleet Requests

[Spot Fleet](#) requests allow you to launch instances on-demand based on your criteria. We won't be using them in this tutorial, so say "no".

With that, the environment will be spun up:

1. Your code will be zipped up and uploaded to a new S3 Bucket
2. After that, the various AWS resources will be created, like the load balancer, security and auto-scaling groups, and EC2 instances

A new application will be deployed as well.

This will take about three minutes so feel free to grab a cup of coffee.

After the deployment is done, the EB CLI will modify `.elasticbeanstalk/config.yml`.

Your project structure should now look like this:

```
|-- .elasticbeanstalk
|   |-- config.yml
|-- .gitignore
|-- README.md
|-- core
|   |-- __init__.py
|   |-- asgi.py
|   |-- settings.py
|   |-- urls.py
|   |-- wsgi.py
|-- db.sqlite3
|-- images
|   |-- __init__.py
|   |-- admin.py
|   |-- apps.py
|   |-- forms.py
|   |-- migrations
|   |-- 0001_initial.py
|   |-- __init__.py
|   |-- models.py
|   |-- tables.py
|   |-- templates
|   |-- images
|       |-- index.html
|   |-- tests.py
|   |-- urls.py
|   |-- views.py
|-- manage.py
|-- requirements.txt
```

Status

Once you've deployed your app you can check its status by running:

Feedback

```
$ eb status
```

```
Environment details for: django-images-env
Application name: django-images
Region: us-west-2
Deployed Version: app-93ec-220218_095635133296
Environment ID: e-z7dmesipvc
Platform: arn:aws:elasticbeanstalk:us-west-2::platform/Python 3.8 running on 64bit Amazon Linux 2/3.3.10
Tier: WebServer-Standard-1.0
CNAME: django-images-env.us-west-2.elasticbeanstalk.com
Updated: 2022-02-18 16:00:24.954000+00:00
Status: Ready
Health: Red
```

You can see that our environment's current health is **Red**, which means that something went wrong. Don't worry about this just yet, we'll fix it in the next steps.

You can also see that AWS assigned us a CNAME which is our EB environment's domain name. We can access the web application by opening a browser and navigating to the CNAME.

Open

```
$ eb open
```

This command will open your default browser and navigate to the CNAME domain. You'll see **502 Bad Gateway**, which we'll fix here shortly

Console

```
$ eb console
```

This command will open the Elastic Beanstalk console in your default browser:

The screenshot shows the AWS Elastic Beanstalk console interface. The browser address bar indicates the URL: `eu-west-3.console.aws.amazon.com/elasticbeanstalk/home?region=eu-west-3#/environment/dashboard?applicationName=django-images2&environmentName=django-images-env`. The console page title is "Elastic Beanstalk". The left sidebar shows the navigation menu with "Environments" selected. The main content area displays the details for the "django-images-env" environment. The "Health" section shows a red exclamation mark icon and the status "Severe". The "Running version" section shows "app-4e77-220215_180105" and a button to "Upload and deploy". The "Platform" section shows the Python logo and "Python 3.8 running on 64bit Amazon Linux 2/3.3.10" with a "Change" button. Below these sections is a "Recent events" table with two entries: a "WARN" event from 2022-02-15 18:05:37 UTC+0100 stating "Environment health has transitioned from Degraded to Severe. 100.0 % of the requests are failing with HTTP 5xx. ELB processes are not healthy on all instances. Initialization completed 2 minutes ago and took 3 minutes. ELB health is failing or not available for all instances. Impaired services on all instances." and an "INFO" event from 2022-02-15 18:05:07 UTC+0100 stating "Successfully launched environment: django-images-env". A "Feedback" button is visible in the bottom right corner.

| Time | Type | Details |
|------------------------------|------|--|
| 2022-02-15 18:05:37 UTC+0100 | WARN | Environment health has transitioned from Degraded to Severe. 100.0 % of the requests are failing with HTTP 5xx. ELB processes are not healthy on all instances. Initialization completed 2 minutes ago and took 3 minutes. ELB health is failing or not available for all instances. Impaired services on all instances. |
| 2022-02-15 18:05:07 UTC+0100 | INFO | Successfully launched environment: django-images-env |

| | | |
|------------------------------------|------|---|
| 2022-02-15 18:04:37 UTC+0100 | INFO | Added instance [i-0974e53102debe6c5] to your environment. |
| 2022-02-15 18:04:37 UTC+0100 | WARN | Environment health has transitioned from Pending to Degraded. 100.0 % of the requests are failing with HTTP 5xx. Initialization completed 31 seconds ago and took 3 minutes. Impaired |

Feedback
English (US)
© 2022, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

Again, you can see that the health of the environment is "Severe", which we'll fix in the next step.

Configure an Environment

In the previous step, we tried accessing our application and it returned `502 Bad Gateway`. There are a few reasons behind it:

1. Python needs `PYTHONPATH` in order to find modules in our application.
2. By default, Elastic Beanstalk attempts to launch the WSGI application from `application.py`, which doesn't exist.
3. Django needs `DJANGO_SETTINGS_MODULE` to know which settings to use.

By default Elastic Beanstalk serves Python applications with [Gunicorn](#). EB automatically installs Gunicorn in the deployment process, hence we do not have to add it to `requirements.txt`. If you want to swap Gunicorn with something else, take a look at [Configuring the WSGI server with a Procfile](#).

Let's fix these errors.

Create a new folder in the project root called `".ebextensions"`. Within the newly created folder create a file named `01_django.config`:

```
# .ebextensions/01_django.config

option_settings:
  aws:elasticbeanstalk:application:environment:
    DJANGO_SETTINGS_MODULE: "core.settings"
    PYTHONPATH: "/var/app/current:$PYTHONPATH"
  aws:elasticbeanstalk:container:python:
    WSGIPath: "core.wsgi:application"
```

Notes:

1. We set the `PYTHONPATH` to the Python path on our EC2 instance ([docs](#)).
2. We pointed `DJANGO_SETTINGS_MODULE` to our Django settings ([docs](#)).
3. We changed the `WSGIPath` to our WSGI application ([docs](#)).

How do EB `.config` files work?

1. You can have as many as you want.
2. They are loaded in the following order: `01_x`, `02_x`, `03_x`, etc.
3. You do not have to memorize these settings; you can list all your environmental settings by running `eb config`.

If you want to learn more about advanced environment customization check out [Advanced environment customization with configuration files](#).

At this point your project structure should look like this:

```

|-- .ebextensions
|   |-- 01_django.config
|-- .elasticbeanstalk
|   |-- config.yml
|-- .gitignore
|-- README.md
|-- core
|   |-- __init__.py
|   |-- asgi.py
|   |-- settings.py
|   |-- urls.py
|   |-- wsgi.py
|-- db.sqlite3
|-- images
|   |-- __init__.py
|   |-- admin.py
|   |-- apps.py
|   |-- forms.py
|   |-- migrations
|   |   |-- 0001_initial.py
|   |   |-- __init__.py
|   |-- models.py
|   |-- tables.py
|   |-- templates
|   |   |-- images
|   |   |-- index.html
|   |-- tests.py
|   |-- urls.py
|   |-- views.py
|-- manage.py
|-- requirements.txt

```

Another thing we have to do before redeploying is to add our CNAME to the `ALLOWED_HOSTS` in `core/settings.py`:

```

# core/settings.py

ALLOWED_HOSTS = [
    'xyz.elasticbeanstalk.com', # make sure to replace it with your own EB CNAME
]

```

Alternatively, for testing, you could just use a wildcard: `ALLOWED_HOSTS = ['*']`. Just don't forget to change that after you're done testing!

Commit the changes to git and deploy:

```

$ git add .
$ git commit -m "updates for eb"

$ eb deploy

```

You'll notice that Elastic Beanstalk won't detect the changes if you don't commit. That's because EB integrates with git and only detects the committed (changed) files.

After the deployment is done, run `eb open` to see if everything worked

Ouch. We fixed the previous error, but there's a new one now:

```

NotSupportedError at /
deterministic=True requires SQLite 3.8.3 or higher

```

Don't worry. It's just an issue with SQLite, which shouldn't be used in production anyways. We'll swap it with Postgres here shortly.

Configure RDS

Django uses a [SQLite](#) database by [default](#). While this is perfect for development, you'll typically want to move to a more robust database, like Postgres or MySQL, for production. What's more, the current EB platform doesn't work well with SQLite, because of a version dependency conflict. Because of these two things we'll swap out SQLite for [Postgres](#).

Local Postgres

First, let's get Postgres running locally. You can either download it from [PostgreSQL Downloads](#) or spin up a Docker container:

```
$ docker run --name django-images-postgres -p 5432:5432 \
  -e POSTGRES_USER=django-images -e POSTGRES_PASSWORD=complexpassword123 \
  -e POSTGRES_DB=django-images -d postgres
```

Check if the container is running:

```
$ docker ps -f name=django-images-postgres
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS |
|------------------------|----------|--------------------------|--------------------|-------------------|------------------------|
| NAMES | | | | | |
| c05621dac852 | postgres | "docker-entrypoint.s..." | About a minute ago | Up About a minute | 0.0.0.0:5432->5432/tcp |
| django-images-postgres | | | | | |

Now, let's try connecting to it with our Django app. Inside *core/settings.py*, change the `DATABASE` config to the following:

```
# core/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'django-images',
        'USER': 'django-images',
        'PASSWORD': 'complexpassword123',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Next, install [psycopg2-binary](#), which is required for Postgres:

```
(venv)$ pip install psycopg2-binary==2.9.3
```

Add it to *requirements.txt*:

```
Django==4.0.2
Pillow==9.0.1
django-tables2==2.4.1
django-crispy-forms==1.14.0
psycopg2-binary==2.9.3
```

Create and apply the migrations:

```
(venv)$ python manage.py makemigrations
(venv)$ python manage.py migrate
```

Run the server:

```
(venv)$ python manage.py runserver
```

Make sure you can still upload an image at <http://localhost:8000>.

Feedback

If you get a `DisallowedHost` error, add `localhost` and `127.0.0.1` to `ALLOWED_HOSTS` inside `core/settings.py`.

AWS RDS Postgres

To set up Postgres for production, start by running the following command to open the AWS console:

```
$ eb console
```

Click "Configuration" on the left side bar, scroll down to "Database", and then click "Edit".

Create a DB with the following settings and click on "Apply":

- Engine: postgres
- Engine version: 12.9 (older Postgres version since db.t2.micro is not available with 13.1+)
- Instance class: db.t2.micro
- Storage: 5 GB (should be more than enough)
- Username: pick a username
- Password: pick a strong password

If you want to stay within the [AWS Free Tier](#) make sure you pick db.t2.micro. RDS prices increase exponentially based on the instance class you pick. If you don't want to go with `micro` make sure to review [AWS PostgreSQL pricing](#).

The screenshot shows the AWS Elastic Beanstalk console interface. On the left, the 'Elastic Beanstalk' sidebar is visible with a list of environments. The 'django-images-env' environment is selected, and the 'Configuration' tab is active. The main panel displays the 'Database settings' for this environment. The settings are as follows:

- Engine:** postgres
- Engine version:** 12.9
- Instance class:** db.t2.micro
- Storage:** 10 GB (Choose a number between 5 GB and 1024 GB)
- Username:** postgres
- Password:** [Redacted]
- Availability:** Low (one AZ)
- Database deletion policy:** Create snapshot (selected). The description states: 'Elastic Beanstalk saves a snapshot of the database and then deletes it. You can restore a database from a snapshot when you add a DB to an Elastic Beanstalk environment or when you create a standalone database. You might incur charges for storing database snapshots.'
- Retain:** (Unselected). The description states: 'The decoupled database will remain available and operational external to Elastic Beanstalk.'

The bottom of the console shows the footer with 'Feedback', 'English (US)', '© 2022, Amazon Web Services, Inc. or its affiliates.', 'Privacy', 'Terms', and 'Cookie preferences'.

After the environmental update is done, EB will automatically pass the following DB credentials to our Django app:

Feedback

```
RDS_DB_NAME
RDS_USERNAME
RDS_PASSWORD
RDS_HOSTNAME
RDS_PORT
```

We can now use these variables in `core/settings.py` to set up the `DATABASE`:

```
# core/settings.py

if 'RDS_DB_NAME' in os.environ:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.postgresql_psycopg2',
            '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.postgresql_psycopg2',
            'NAME': 'django-images',
            'USER': 'django-images',
            'PASSWORD': 'complexpassword123',
            'HOST': 'localhost',
            'PORT': '5432',
        }
    }
```

Don't forget to import the `os` package at the top of `core/settings.py`:

```
import os
```

Next, we have to tell Elastic Beanstalk to run `makemigrations` and `migrate` when a new application version gets deployed. We can do that by editing the `.ebextensions/01_django.config` file. Add the following to the bottom of the file:

```
# .ebextensions/01_django.config

container_commands:
  01_makemigrations:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py makemigrations --noinput"
    leader_only: true
  02_migrate:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py migrate --noinput"
    leader_only: true
```

The EB environment will now execute the above commands every time we deploy a new application version. We used `leader_only`, so only the first EC2 instance executes them (in case our EB environment runs multiple EC2 instances).

Elastic Beanstalk configs support two different command sections, `commands` and `container_commands`. The main difference between them is when they are run in the deployment process:

1. `commands` run before the application and web server are set up and the application version file is extracted.
2. `container_commands` run after the application and web server have been set up and the application version archive has been extracted, but before the application version is deployed (before the files are moved from the staging folder to their final location).

Let's also add a command to create a superuser. We can use Django's intuitive [custom command framework](#) to add a new command. Within the "images" app create the following files and folders:

Feedback

```

L-- images
  L-- management
    |-- __init__.py
    L-- commands
      |-- __init__.py
      L-- createsu.py

```

createsu.py:

```

# images/management/commands/createsu.py

from django.contrib.auth.models import User
from django.core.management.base import BaseCommand

class Command(BaseCommand):
    help = 'Creates a superuser.'

    def handle(self, *args, **options):
        if not User.objects.filter(username='admin').exists():
            User.objects.create_superuser(
                username='admin',
                password='complexpassword123'
            )
        print('Superuser has been created.')

```

Next, add the third container command to *.ebextensions/01_django.config*:

```

# .ebextensions/01_django.config

container_commands:
  01_makemigrations:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py makemigrations --noinput"
    leader_only: true
  02_migrate:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py migrate --noinput"
    leader_only: true
  # ----- new -----
  03_superuser:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py createsu"
    leader_only: true
  # ----- end of new -----

```

An alternative to creating a `createsu` command is to SSH into one of the EC2 instances and run Django's default `createsuperuser` command.

Commit the changes to git and deploy:

```

$ git add .
$ git commit -m "updates for eb"

$ eb deploy

```

Wait for the deployment to finish. Once done, run `eb open` to open your app in a new browser tab. Your app should now work. Make sure you can upload an image.

S3 for File Storage

Check out the deployed version of your admin dashboard. The static files aren't being served correctly. Further, we don't want static or media files stored locally on an EC2 instance since EB applications should be as stateless, which makes it much easier to [scale](#) your applications out to multiple EC2 instances.

While AWS provides a number of [persistent storage](#) services, [S3](#) is arguably the most popular and easiest to work with.

Feedback

To configure S3, we'll need to:

1. Create an S3 Bucket
2. Create an IAM group and user for S3 Bucket management
3. Set Elastic Beanstalk S3 environment variables
4. Configure Django static and media settings

Create an S3 Bucket

To start, let's create a new S3 Bucket. Navigate to the [AWS S3 Console](#) and click on "Create Bucket". Give the Bucket a unique name and set the AWS region. Use the default config for everything else. Press "Create".

IAM Group and User

Navigate to the [IAM Console](#). On the left side of the screen, select "User groups". Create a new group with the "AmazonS3FullAccess" permission:

The screenshot shows the AWS IAM console interface. The left sidebar contains the 'Identity and Access Management (IAM)' menu with options like 'Dashboard', 'Access management', 'Users', 'Roles', 'Policies', 'Identity providers', 'Account settings', 'Access reports', 'Access analyzer', 'Archive rules', 'Analyzers', 'Settings', 'Credential report', 'Organization activity', and 'Service control policies (SCPs)'. The main content area is titled 'Create user group' and includes a 'Name the group' section where 'S3FullAccess' is entered. Below this, the 'Attach permissions policies - Optional' section shows a list of policies. The 'AmazonS3FullAccess' policy is selected, and a table of available policies is displayed below.

| | Policy name | Type | Description |
|-------------------------------------|---|-------------|---------------------|
| <input type="checkbox"/> | AmazonDMSRedshiftS3Role | AWS managed | Provides access to |
| <input checked="" type="checkbox"/> | AmazonS3FullAccess | AWS managed | Provides full acces |
| <input type="checkbox"/> | QuickSightAccessForS3StorageManagementAnal... | AWS managed | Policy used by Qui |
| <input type="checkbox"/> | AmazonS3ReadOnlyAccess | AWS managed | Provides read only |
| <input type="checkbox"/> | AmazonS3OutpostsFullAccess | AWS managed | Provides full acces |
| <input type="checkbox"/> | AmazonS3ObjectLambdaExecutionRolePolicy | AWS managed | Provides AWS I an |

Then, create a new user with "Programmatic access" and assign that group to the user:

The screenshot shows the 'Add user' page in the AWS IAM console. The page has a progress bar at the bottom with five steps: 1 (highlighted), 2, 3, 4, and 5. The first step is labeled 'Add user'. The page also includes a 'Feedback' button in the bottom right corner.

Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name* S3Admin

[+ Add another user](#)

Select AWS access type

Select how these users will primarily access AWS. If you choose only programmatic access, it does NOT prevent users from accessing the console using an assumed role. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

Select AWS credential type*



Access key - Programmatic access

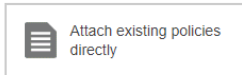
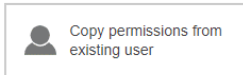
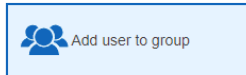
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.



Password - AWS Management Console access

Enables a **password** that allows users to sign-in to the AWS Management Console.

Set permissions



Add user to an existing group or create a new one. Using groups is a best-practice way to manage user's permissions by job functions. [Learn more](#)

Add user to group

Create group

Refresh

| Search | | Showing 1 result |
|--|-------------------------------|------------------|
| Group | Attached policies | |
| <input checked="" type="checkbox"/> S3FullAccess | AmazonS3FullAccess and 1 more | |

* Required

Cancel

Next: Permissions

Feedback English (US)

© 2022, Amazon Web Services, Inc. or its affiliates.

Privacy

Terms

Cookie preferences

AWS will generate authentication credentials for you. Download the provided .csv file. We'll need to pass them to our Elastic Beanstalk environment in the next step.

Set EB Environment Variables

Next, we need to set the following environmental variables:

| | |
|-------------------------|---------------------------|
| AWS_ACCESS_KEY_ID | - your ACCESS_KEY_ID |
| AWS_SECRET_ACCESS_KEY | - your SECRET_ACCESS_KEY |
| AWS_S3_REGION_NAME | - your selected S3 region |
| AWS_STORAGE_BUCKET_NAME | - your bucket name |

Navigate to your Elastic Beanstalk console. Click "Configuration". Then, within the "Software" category, click "Edit" and scroll down to the "Environment properties" section. Add the four variables.

The screenshot shows the AWS Elastic Beanstalk console for the 'django-images-env' environment. The 'Configuration' tab is selected, and the 'Environment properties' section is expanded. The table below shows the environment variables being set:

| Name | Value |
|--------------------|------------|
| AWS_ACCESS_KEY_ID | [Redacted] |
| AWS_S3_REGION_NAME | eu-west-3 |

Other visible settings include 'Retention' set to 7 days and 'Lifecycle' set to 'Keep logs after terminating environment'.

Monitoring
Alarms
Managed updates
Events
Tags

▼ Recent environments
django-images-env

| | | |
|-------------------------|-------------------------------|---|
| AWS_SECRET_ACCESS_KEY | [REDACTED] | ✕ |
| AWS_STORAGE_BUCKET_NAME | django-images-bucket | ✕ |
| DJANGO_SETTINGS_MODULE | core.settings | ✕ |
| PYTHONPATH | /var/app/current:\$PYTHONPATH | ✕ |
| | | |

Cancel Continue **Apply**

Feedback English (US) © 2022, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

After you've added all the variables click "Apply".

Configure Django Static and Media Settings

Next, in order for Django to communicate with our S3 Bucket, we need to install the [django-storages](#) and [boto3](#) packages.

Add them to the *requirements.txt* file:

```
Django==4.0.2
Pillow==9.0.1
django-tables2==2.4.1
django-crispy-forms==1.14.0
psycopg2-binary==2.9.3
boto3==1.21.3
django-storages==1.12.3
```

Next, add the newly installed app to `INSTALLED_APPS` in *core/settings.py*:

```
# core/settings.py

INSTALLED_APPS = [
    # ...
    'storages',
]
```

Configure *django-storages* to use the environmental variables passed by Elastic Beanstalk:

```
if 'AWS_STORAGE_BUCKET_NAME' in os.environ:
    STATICFILES_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
    DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'

    AWS_STORAGE_BUCKET_NAME = os.environ['AWS_STORAGE_BUCKET_NAME']
    AWS_S3_REGION_NAME = os.environ['AWS_S3_REGION_NAME']

    AWS_S3_ACCESS_KEY_ID = os.environ['AWS_ACCESS_KEY_ID']
    AWS_S3_SECRET_ACCESS_KEY = os.environ['AWS_SECRET_ACCESS_KEY']
```

Lastly, we need to run the `collectstatic` command after deployment is complete, so add the following to the bottom of *01_django.config*:

```
# .ebextensions/01_django.config
# ...

container_commands:
    # ...
    04_collectstatic:
        command: "source /var/app/venv/*/bin/activate && python3 manage.py collectstatic --noinput"
        leader_only: true
```

The full file should now look like this:

```
# .ebextensions/01_django.config

option_settings:
  aws:elasticbeanstalk:application:environment:
    DJANGO_SETTINGS_MODULE: "core.settings"
    PYTHONPATH: "/var/app/current:$PYTHONPATH"
  aws:elasticbeanstalk:container:python:
    WSGIPath: "core.wsgi:application"

container_commands:
  01_makemigrations:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py makemigrations --noinput"
    leader_only: true
  02_migrate:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py migrate --noinput"
    leader_only: true
  03_superuser:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py createsuperuser"
    leader_only: true
  04_collectstatic:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py collectstatic --noinput"
    leader_only: true
```

Commit the changes to git and deploy:

```
$ git add .
$ git commit -m "updates for eb"

$ eb deploy
```

Confirm that the static and media files are now stored on S3.

If you get a `Signature mismatch` error, you might want to add the following setting to `core/settings.py`:

```
AWS_S3_ADDRESSING_STYLE = "virtual"
```

For more details, refer to [this GitHub issue](#).

To learn more about static and media file storage on AWS S3, take a look at the [Storing Django Static and Media Files on Amazon S3](#) article.

HTTPS with Certificate Manager

This part of the tutorial requires that you have a domain name.

Need a cheap domain to practice with? Several domain registrars have specials on '.xyz' domains. Alternatively, you can create a free domain at [Freenom](#). If you don't own a domain name, but would still like to use HTTPS you can [create and sign with an X509 certificate](#).

To serve your application via HTTPS, we'll need to:

1. Request and validate an SSL/TLS certificate
2. Point your domain name to your EB CNAME
3. Modify the load balancer to serve HTTPS
4. Modify your application settings

Request and Validate an SSL/TLS Certificate

Navigate to the [AWS Certificate Manager console](#). Click "Request a certificate". Set the certificate type to "Public" and click "Next". Enter your [fully qualified domain name](#) into the form input, set the "Validation method" to "DNS validation", and click "Request".

Feedback

You'll then be redirected to a page where you can see all your certificates. The certificate that you just created should have a status of "Pending validation".

For AWS to issue a certificate, you first have to prove that you're the owner of the domain. In the table, click on the certificate to view the "Certificate details". Take note of the "CNAME name" and "CNAME value". To validate the ownership of the domain, you'll need to create a CNAME Record in your domain's DNS settings. Use the "CNAME name" and "CNAME value" for this. Once done, it will take a few minutes for Amazon to pick up the domain changes and issue the certificate. The status should change from "Pending validation" to "Issued".

Point the Domain Name to the EB CNAME

Next, you need to point your domain (or subdomain) to your EB environment CNAME. Back in your domain's DNS settings, add another CNAME record with the value being your EB CNAME -- e.g., `django-images-dev.us-west-2.elasticbeanstalk.com`.

Wait a few minutes for your DNS to refresh before testing things out from the `http://` flavor of your domain name in your browser.

Modify the Load Balancer to serve HTTPS

Back in the Elastic Beanstalk console, click "Configuration". Then, within the "Load balancer" category, click "Edit". Click "Add listener" and create a listener with the following details:

1. Port - 443
2. Protocol - HTTPS
3. SSL certificate - select the certificate that you just created

Click "Add". Then, scroll to the bottom of the page and click "Apply". It will take a few minutes for the environment to update.

Modify your Application Settings

Next, we need to make a few changes to our Django application.

First, add your fully qualified domain to `ALLOWED_HOSTS`:

```
# core/settings.py

ALLOWED_HOSTS = [
    # ...
    'yourdomain.com',
]
```

Last, we need to redirect all traffic from HTTP to HTTPS. There are multiple ways of doing this, but the easiest way is to set up [Apache](#) as a proxy host. We can achieve this programmatically by adding the following to the end of the `option_settings` in `.ebextensions/01_django.config`:

```
# .ebextensions/01_django.config

option_settings:
  # ...
  aws:elasticbeanstalk:environment:proxy: # new
    ProxyServer: apache                    # new
```

Your final `01_django.config` file should now look like this:

```
# .ebextensions/01_django.config

option_settings:
  aws:elasticbeanstalk:application:environment:
    DJANGO_SETTINGS_MODULE: "core.settings"
    PYTHONPATH: "/var/app/current:$PYTHONPATH"
  aws:elasticbeanstalk:container:python:
    WSGIPath: "core.wsgi:application"
  aws:elasticbeanstalk:environment:proxy:
    ProxyServer: apache

container_commands:
  01_makemigrations:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py makemigrations --noinput"
    leader_only: true
  02_migrate:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py migrate --noinput"
    leader_only: true
  03_superuser:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py createsuperuser"
    leader_only: true
  04_collectstatic:
    command: "source /var/app/venv/*/bin/activate && python3 manage.py collectstatic --noinput"
    leader_only: true
```

Next, create a ".platform" folder in the project root and add the following files and folders:

```
L-- .platform
  L-- httpd
    L-- conf.d
      L-- ssl_rewrite.conf
```

`ssl_rewrite.conf`:

```
# .platform/httpd/conf.d/ssl_rewrite.conf

RewriteEngine On
<If "-n %{HTTP:X-Forwarded-Proto}' && %{HTTP:X-Forwarded-Proto} != 'https'">
RewriteRule (.*) https://%{HTTP_HOST}%{REQUEST_URI} [R,L]
</If>
```

Your project structure should now look like this:

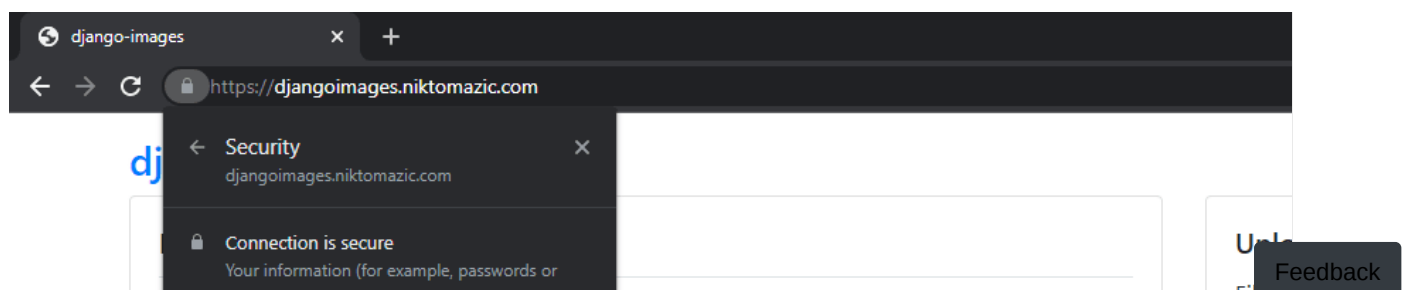
```
|-- .ebextensions
|   |-- 01_django.config
|-- .elasticbeanstalk
|   |-- config.yml
|-- .gitignore
|-- .platform
|   |-- httpd
|       |-- conf.d
|           |-- ssl_rewrite.conf
|-- README.md
|-- core
|   |-- __init__.py
|   |-- asgi.py
|   |-- settings.py
|   |-- urls.py
|   |-- wsgi.py
|-- db.sqlite3
|-- images
|   |-- __init__.py
|   |-- admin.py
|   |-- apps.py
|   |-- forms.py
|   |-- management
|       |-- __init__.py
|       |-- commands
|           |-- __init__.py
|           |-- createsu.py
|-- migrations
|   |-- 0001_initial.py
|   |-- __init__.py
|-- models.py
|-- tables.py
|-- templates
|   |-- images
|       |-- index.html
|-- tests.py
|-- urls.py
|-- views.py
|-- manage.py
|-- requirements.txt
```



Commit the changes to git and deploy:

```
$ git add .
$ git commit -m "updates for eb"

$ eb deploy
```

Now, in your browser, the `https://` flavor of your application should work. Try going to the `http://` flavor. You should be redirected to the `https://` flavor. Ensure the certificate is loaded properly as well:



| | | | Uploaded at | Preview |
|--|----------|--------------------------|----------------------|---|
| <div> <div>credit card numbers) is private when it is sent to this site. Learn more</div> <div>Certificate is valid</div> </div> | | | 02/07/2022 6:19 p.m. |  |
| 1 | Gray cat | cat2.jpg | 02/07/2022 5:48 p.m. |  |

Environment Variables

In production, it's [best to store environment-specific config in environment variables](#). With Elastic Beanstalk you can set custom environmental variables two different ways.

Environment Variables via EB CLI

Let's turn Django's `SECRET_KEY` and `DEBUG` settings into environmental variables.

Start by running:

```
$ eb setenv DJANGO_SECRET_KEY='<replace me with your own secret key>' \
           DJANGO_DEBUG='1'
```

You can set multiple environmental variables with one command by separating them with spaces. This is the recommended approach as it results in only a single update to the EB environment.

Change `core/settings.py` accordingly:

```
# core/settings.py

SECRET_KEY = os.environ.get(
    'DJANGO_SECRET_KEY',
    '<replace me with your own fallback secret key>'
)

DEBUG = os.environ.get('DJANGO_DEBUG', '1').lower() in ['true', 't', '1']
```

Commit the changes to git and deploy:

```
$ git add .
$ git commit -m "updates for eb"

$ eb deploy
```

Environment Variables via EB Console

Enter the Elastic Beanstalk console via [eb open](#). Navigate to "Configuration" > "Software" > "Edit". Then, scroll down to the "Environment properties".



Environments

Applications

Change history

▼ django-images

Application versions

Saved configurations

▼ django-images-env

Go to environment

Configuration

Logs

Health

Monitoring

Alarms

Managed updates

Events

Tags

▼ Recent environments

django-images-env

Instance log streaming to CloudWatch Logs

Configure the instances in your environment to stream logs to CloudWatch Logs. You can set the retention to up to ten years and configure Elastic Beanstalk to delete the logs when you terminate your environment.

Log streaming
(Standard CloudWatch charges apply.)

☐ Enabled

Retention
7 days

Lifecycle
Keep logs after terminating environment

Environment properties

The following properties are passed in the application as environment properties. [Learn more](#)

| Name | Value |
|---------------|----------------|
| VARIABLE_NAME | variable value |
| | |

Cancel Continue **Apply**

Feedback English (US) © 2022, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

After you're done, click "Apply" and your environment will update.

You can then access these variables in your Python environment via `os.environ`.

For example:

```
VARIABLE_NAME = os.environ['VARIABLE_NAME']
```

Debugging Elastic Beanstalk

When working with Elastic Beanstalk, it can be pretty frustrating to figure out what went wrong if you don't know how to access the log files. In this section will look at just that.

There are two ways to access the logs:

1. Elastic Beanstalk CLI or console
2. SSH into EC2 instance

From personal experience, I've been able to solve all issues with the first approach.

Elastic Beanstalk CLI or Console

CLI:

```
$ eb logs
```

This command will fetch the last 100 lines from the following files:

```
/var/log/web.stdout.log
/var/log/eb-hooks.log
/var/log/nginx/access.log
/var/log/nginx/error.log
/var/log/eb-engine.log
```

Running `eb logs` is equivalent to logging into the EB console and navigating to "Logs".

I recommend piping the logs to [CloudWatch](#). Run the following command to enable this:

```
$ eb logs --cloudwatch-logs enable
```

You'll typically find Django errors in `/var/log/web.stdout.log` or `/var/log/eb-engine.log`.

To learn more about Elastic Beanstalk logs check out [Viewing logs from Amazon EC2 instances](#).

SSH into EC2 Instance

To connect to an EC2 instance where your Django application is running, run:

```
$ eb ssh
```

You'll be prompted to add the host to your known hosts the first time. Say yes. With that, you'll now have full access to your EC2 instance. Feel free to experiment around with Django management commands and check some of the log files mentioned in the previous section.

Keep in mind that Elastic Beanstalk automatically scales and deploys new EC2 instances. The changes you make on this specific EC2 instance won't be reflected on newly launched EC2 instances. Once this specific EC2 instance is replaced, your changes will be wiped.

Conclusion

In this tutorial, we walked through the process of deploying a Django application to AWS Elastic Beanstalk. By now you should have a fair understanding of how Elastic Beanstalk works. Perform a quick self-check by reviewing the objectives at the beginning of the tutorial.

Next steps:

1. You should consider creating two separate EB environments (`dev` and `production`).
2. Review [Auto Scaling group for your Elastic Beanstalk environment](#) to learn about how to configure triggers for auto scaling your application.

To remove all the AWS resources we created throughout the tutorial, first terminate the Elastic Beanstalk environment:

```
$ eb terminate
```

You'll need to manually remove the S3 Bucket, SSL certificate, and IAM group and user.

Finally, you can find the final version of the code in the [django-elastic-beanstalk](#) repo on GitHub.

 [aws](#) [devops](#) [django](#)

Feedback



Nik Tomazic

Nik is a software developer from Slovenia. He's interested in object-oriented programming and web development. He likes learning new things and accepting new challenges. When he's not coding, Nik's either swimming or watching movies.



Featured Course

Test-Driven Development with Django, Django REST Framework, and Docker

Buy Now **\$30**

[View Course](#)

[Twitter](#) [Reddit](#) [Hacker News](#) [Facebook](#)

Search all tutorials

TUTORIAL TOPICS

[api](#) [architecture](#) [aws](#) [devops](#) [django](#) [django rest framework](#) [docker](#) [fastapi](#) [flask](#) [front-end](#) [heroku](#) [kubernetes](#)
[machine learning](#) [python](#) [react](#) [task queue](#) [testing](#) [vue](#) [web scraping](#)

RECOMMENDED TUTORIALS

[Deploying Django to AWS ECS with Terraform](#)



[Michael Herman](#)

Nov 29th, 2021

Deploy a Django app to AWS ECS with Terraform.



[aws](#) [devops](#) [django](#) [docker](#)

[Effectively Using Django REST Framework Serializers](#)

Feedback



Nik Tomazic

Mar 24th, 2021

Deep dive into Django REST Framework (DRF) serializers.

 [api](#) [django](#) [django rest framework](#)

[Dockerizing Django with Postgres, Gunicorn, and Nginx](#)



Michael Herman

Aug 27th, 2021

This tutorial details how to configure Django to run on Docker along with Postgres, Nginx, and Gunicorn.

 [django](#) [docker](#)

Stay Sharp with Course Updates

Join our mailing list to be notified about updates and new releases.

LEARN

[Courses](#) [Bundles](#) [Blog](#)

GUIDES

[Complete Python](#) [Django and Celery](#) [Deep Dive Into Flask](#)

ABOUT TESTDRIVEN.IO

[Support and Consulting](#) [What is Test-Driven Development?](#) [Testimonials](#) [Open Source Donations](#) [About Us](#)
[Meet the Authors](#) [Tips and Tricks](#)



TestDriven.io is a proud supporter of open source

10% of profits from each of our [FastAPI](#) courses and our [Flask Web Development](#) course will be donated to the FastAPI and Flask teams, respectively.

[Follow our contributions](#)

