Django Deploy Starter Guide

This guide will walk you through the process of launching a barebones Django Application on Aptible. See it deployed live here.

📘 Deploy this starter template now and start building your app in minutes! This is the easiest and quickest way to get started with Django. You can start building your app without having to worry about the details of deployment.

H1: Table of Contents

H1: Deploy a Django App on Aptible

Deploying a Django app using Docker Compose simplifies the setup by orchestrating the app, its dependencies, and configurations in a harmonized environment. Docker Compose enables developers to define multi-container applications, ensuring consistency and eliminating compatibility issues. Users clone the app's repository and set up a `docker-compose.yml` file detailing services and configurations.

Launching the entire application environment in Aptible is then just a matter of running a single Docker Compose command. This method streamlines scaling, maintenance, and ensures

consistency across different environments, making Django app deployment more efficient and reliable.

H2: Prerequisites

This Django deployment guide has the following prerequisites:

a. Create an Aptible account
b. Install Git
c. Install pip
d. Install the Aptible CLI
e. Add an SSH public key to your Aptible user account.
f. Install Docker and Docker Compose.

H2: Clone the Repo

Clone the template-django repo to your local machine.

To get a copy of the Django app's source code on your local machine, use the git clone command:

```
Unset
git clone https://github.com/your_username/your_django_app.git
```

The code above will create a directory named **`your_django_app`** in your current directory, containing all the files from the repository.

Make sure to replace your_username and **`your_django_app`** with the appropriate username and repository name. If you're using a Git platform other than GitHub, like GitLab or Bitbucket, adjust the URL accordingly.

H2: Static and Media Files Handling

Handling static and media files correctly is a common challenge when deploying Django applications. Here's a step-by-step guide for setting up static and media files handling for a Django app on Aptible.

1. **Set up File Storage on Amazon S3:**

   Due to the ephemeral nature of containers in platforms like Aptible, it's common to use cloud storage solutions like Amazon S3 for serving media files.

   a. Create an S3 bucket via the AWS Management Console or AWS CLI. Take note of your S3 bucket name because we'll use it in a later step.

   b. Note your AWS credentials (we'll use them later): **`AWS_ACCESS_KEY_ID`** and **`AWS_SECRET_ACCESS_KEY`**.

2. **Install Required Packages**

   a. To integrate Amazon S3 with your Django app, you'll need some packages:

```
Unset
pip install boto3 django-storages
```

3. **Configure Django to Use S3:**

   a. Modify your Django **`settings.py`**:

```
Unset
INSTALLED_APPS += ['storages']

AWS_ACCESS_KEY_ID = os.environ.get('AWS_ACCESS_KEY_ID')
AWS_SECRET_ACCESS_KEY = os.environ.get('AWS_SECRET_ACCESS_KEY')
AWS_STORAGE_BUCKET_NAME = os.environ.get('AWS_STORAGE_BUCKET_NAME')
AWS_S3_CUSTOM_DOMAIN = '%s.s3.amazonaws.com' % AWS_STORAGE_BUCKET_NAME
AWS_S3_OBJECT_PARAMETERS = {
  'CacheControl': 'max-age=86400',
}

# Static files (CSS, JavaScript, images)
AWS_STATIC_LOCATION = 'static'
STATIC_URL = 'https://%s/%s/' % (AWS_S3_CUSTOM_DOMAIN, AWS_STATIC_LOCATION)
STATICFILES_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'

# Media files (uploads)
AWS_PUBLIC_MEDIA_LOCATION = 'media/public'
DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
MEDIA_URL = 'https://%s/%s/' % (AWS_S3_CUSTOM_DOMAIN, AWS_PUBLIC_MEDIA_LOCATION)
```

4. **Set Environment Variables in Aptible**

    a. Update the environment variables on Aptible to include your AWS credentials (replace `your_access_key` below with your **`AWS_SECRET_ACCESS_KEY`**) and the S3 bucket name (replace `your_bucket_name` below):

```
Unset
aptible config:set --app your-app-name AWS_ACCESS_KEY_ID=your_access_key
AWS_SECRET_ACCESS_KEY=your_secret_key AWS_STORAGE_BUCKET_NAME=your_bucket_name
```

5. **Run Collectstatic**

    a. Whenever you deploy or make changes to static files, run the **`collectstatic`** command to push them to S3:

```
Unset
python manage.py collectstatic
```

    b. This will collect all your static files and upload them to your S3 bucket.

6. **Handling User Uploaded Files (Media):**

    a. With the above configuration, any file that gets uploaded by a user will be stored in the **`media/public`** directory in your S3 bucket and served using an auto-generated S3 URL.

7. **Secure Media Files (Optional):**

    a. If you wish to serve media files privately, you can use Amazon S3's signed URLs. This requires changes to your Django views and models to generate signed URLs that expire after a given time. You can use **`boto3`** for secure S3 media delivery.

H2: Custom Configuration

When deploying a Dockerized Django application, custom configurations ensure the app runs efficiently and securely.

1. **Environment Variables:**

Aptible uses environment variables for configuration. This ensures that sensitive information isn't hard-coded in your codebase and provides a way to adjust configurations without changing the code.

a. Set basic Django Configurations

```
Unset
aptible config:set --app your-app-name SECRET_KEY=mysecretkey DEBUG=False
ALLOWED_HOSTS=mydomain.com
```

- **`SECRET_KEY:`** Django's secret key for cryptographic signing. The **`SECRET_KEY:`** is automatically generated for your project. It is essentially arbitrary in terms of its content but is designed to be cryptographically random and secure.
- **`DEBUG:`** Turn it off (False) in production for security reasons.
- **`ALLOWED_HOSTS:`** The domain(s) from which your app will be accessible.

b. Database Configuration

If you've set up a database for your Django app on Aptible, you'll get a **`DATABASE_URL`**. Use it to configure the database for Django:

```
Unset
aptible config:set --app your-app-name DATABASE_URL=your-database-url
```

In your **`settings.py:`**

```
Python
import dj_database_url

DATABASES = {
  'default': dj_database_url.config(default=os.environ.get('DATABASE_URL'))
}
```

2. Static and Media Files

a. If you're serving status and media files via Amazon S3, do the following: Update the Django settings (**`settings.py`**) accordingly and set the relevant environment variables on Aptible.

b. Example for Amazon S3

```
Unset
aptible config:set --app your-app-name AWS_STORAGE_BUCKET_NAME=mybucket
AWS_S3_REGION_NAME=region-name AWS_ACCESS_KEY_ID=myaccesskey
AWS_SECRET_ACCESS_KEY=mysecretkey
```

3. Email Configuration

a. If you're using a transactional email service like SendGrid:

```
Unset
aptible config:set --app your-app-name EMAIL_HOST=smtp.sendgrid.net
EMAIL_HOST_USER=apikey EMAIL_HOST_PASSWORD=sendgrid-api-key EMAIL_PORT=587
EMAIL_USE_TLS=True
```

Ensure that the Django settings.py is set up to use the aforementioned environment variables for sending emails. Doing so enables your app to fetch these variables to send emails using the configured email service (in this case, SendGrid). For example, in your **settings.py,** you might have something like: this:

```
Unset
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = os.environ.get('EMAIL_HOST')
EMAIL_HOST_USER = os.environ.get('EMAIL_HOST_USER')
EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_HOST_PASSWORD')
EMAIL_PORT = os.environ.get('EMAIL_PORT')
EMAIL_USE_TLS = os.environ.get('EMAIL_USE_TLS') == 'True'
```

4. Third-party Integration

a. If your Django app integrates with other services/APIs (e.g., payment gateways, third-party APIs), set the respective API keys or configuration as environment variables:

```
Unset
aptible config:set --app your-app-name STRIPE_API_KEY=mystripeapikey
ANOTHER_API_KEY=myotherapikey
```

5. Caching

a. If you're using Redis or Memcached for caching, ensure that the cache URL is set as an environment variable and Django's caching settings are appropriately configured.

6. Custom Domains

a. If you have a custom domain for your app:

```
Unset
aptible config:set --app your-app-name ALLOWED_HOSTS=your-custom-domain.com
```

7. Security Settings

a. To enhance security, configure other Django security settings using environment variables (e.g., `SECRET_KEY, DEBUG, ALLOWED_HOSTS'`).

```
Unset
aptible config:set --app your-app-name SECURE_SSL_REDIRECT=True
CSRF_COOKIE_SECURE=True SESSION_COOKIE_SECURE=True
```

8. Scaling and Performance

a. Based on your app's needs, set environment variables for performance, like the number of worker processes for Gunicorn.

```
Unset
aptible config:set --app your-app-name WEB_CONCURRENCY=4
```

9. Periodic Tasks

a. If you use `Celery` for asynchronous tasks, ensure that the broker (like RabbitMQ) is set up, and its URL is provided as an environment variable.

Always test your configurations in a staging environment before applying them to production. Also, keep sensitive keys and configurations secret and avoid hardcoding them in the app or version control system. See our best practices guide for more.

H2: Using Docker Compose

Docker Compose ensures that developers work in a consistent environment that closely mirrors production, minimizing the risk of deployment problems. Aptible offers a secure and compliant platform that is optimized for the deployment of containerized applications.

Together, these tools create an environment where Django apps can be developed, tested, and deployed with maximum efficiency and reliability. This makes the combination of Docker Compose and Aptible a popular choice for Django developers.

H3: Local Development with Docker Compose:

1. **Django Docker Configuration:**
   a. Django Dockerfile:

      Create a `Dockerfile` in your Django app's root directory:

```
Unset
FROM python:3.8-slim

WORKDIR /app

COPY requirements.txt /app/
RUN pip install --no-cache-dir -r requirements.txt

COPY . /app/

CMD ["gunicorn", "myapp.wsgi:application", "--bind", "0.0.0.0:8000"]
```

      Make sure to replace `myapp` with the name of your Django project.

      Note: not all Django apps use `gunicorn.` It's just one option among several. When you run a Django app in development mode using `python manage.py runserver,` it uses

Django''s built-in development server. This server is suitable for development but is not recommended for production use due to performance and security concerns.

For production deployments, users can deploy `uWSGI` and `mod_wsgi` (an Apache module There are also several other servers and methods that exist to serve Django apps.

b. Requirements:
Ensure you have a `requirements.txt` in your root directory that lists your Python dependencies.

2. Docker Compose Setup:

a. Create a `docker-compose.yml` in the root directory:

```
Unset
version: '3'

services:
  web:
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - db
  db:
    image: postgres:13
    environment:
      - POSTGRES_DB=mydatabase
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
```

3. Run your Django app with Docker Compose

```
Unset
docker-compose up
```

a. You should be able to access your Django app at http://localhost:8000.

H3: Deploying to Aptible

1. Push your Docker Image to Aptible:

a. Build your Docker Image

```
Unset
docker build -t your-django-image-name /path/to/your/build
```

b. Push the image to Aptible's Registry

docker tag: This command assigns a new tag to an existing Docker image. It's useful when you want to have multiple tags pointing to the same image ID, or when you want to rename tags, or point existing tags to a different image.

```
Unset
docker tag your-django-image-name:latest
registry.aptible.com/my-aptible-app/your-django-image-name:latest
```

In this command, you're creating (or overwriting) a tag `registry.aptible.com/my-aptible-app/your-django-image-name:latest` for the image `your-django-image-name:latest.`

docker push: This command pushes a Docker image or a repository to a registry.

```
Unset
docker push registry.aptible.com/my-aptible-app/your-django-image-name:latest
```

After tagging the image in the previous step, you're now pushing the image with its new tag to the Aptible registry.

3. Deploy your Docker Image

```
Unset
aptible deploy --app APP_HANDLE --docker-image
registry.aptible.com/my-aptible-app/your-django-image-name:latest.
```

4. Configure Environmental Variables:

a. Set up environment variables for your Django app, especially for **`SECRET_KEY`**, **`DEBUG`**, **`ALLOWED_HOSTS`**, and database configurations.

```
Unset
aptible config:set --app APP_HANDLE SECRET_KEY=mysecretkey DEBUG=False
ALLOWED_HOSTS=mydomain.com
```

b. The SECRET_KEY in Django is a critical setting used for cryptographic signing. It's essential for the security of a Django application and should be kept secret.

1. **Purpose of SECRET_KEY**: The SECRET_KEY is used for session verification, CSRF protection, and more. It's vital for ensuring data integrity and security.
2. **Generation**: When starting a new Django project using the django-admin startproject command, Django automatically generates a unique SECRET_KEY for you. This is typically found in the `settings.py` file of your project.
3. **Setting for Deployment**: For deployments, especially in production, it's essential not to hard-code this key in `settings.py` or any other public file (e.g., version control). Instead, it should be set as an environment variable, ensuring it's not exposed or easily accessible.
4. **Generating a New SECRET_KEY**: If a user needs to generate a new SECRET_KEY (for instance, if they believe their existing key might have been compromised), there are several ways to do so:
    - Use Django's built-in utilities:
    - pythonCopy code
    - from django.core.management.utils import get_random_secret_key
        - print(get_random_secret_key())
        - Use online tools or other methods to generate a 50-character random string.
5. **Safety Precautions**: Once a SECRET_KEY is set, it should not be changed without care, as doing so will invalidate certain data tied to the original key, like existing user sessions or reset tokens.

H3: Database Setup:

1. **Create a Database in Aptible:**

```
Unset
aptible db:create my-django-db --type postgres --environment ENV_HANDLE
```

This command will provide you with a postgres database URL. Aptible also supports the following database types:

- MySQL
- MongoDB
- CouchDB
- Elasticsearch
- InfluxDB
- Redis
- RabbitMQ

Learn more about databases in Aptible.

2. **Configure Django to Use the Aptible Database:**

a. First, install the Aptible CLI:

```
Unset
curl -s https://www.aptible.com/cli/aptible-cli-latest.tar.gz | tar -C
/usr/local/bin -xzf -
```

b. Log in to your Aptible dashboard

```
Unset
aptible login
```

c. Retrieve the DATABASE_URL.for your database, replacing `<ENVIRONMENT>` with your environment name, and `<DATABASE_HANDLE>` with your database's handle:

```
Unset
aptible config --app <ENVIRONMENT> | grep DATABASE_URL
```

d. Configure Django to use the `DATABASE_URL`.

   First, install the necessary package:

```
Unset
pip install dj-database-url
```

   Ensure you have `dj-database-url` in your `requirements.txt`.

e. In your Django `settings.py`, use a package like `dj-database-url` to parse the `DATABASE_URL`

```
Java
import dj_database_url
DATABASES = {
  'default': dj_database_url.config(default=os.environ.get('DATABASE_URL'))
}
```

f. Make sure to have **`dj-database-url`** in your **`requirements.txt`**.

H2: Testing and Quality Assurance

Ensuring the reliability, functionality, and performance of your Dockerized Django app is more than just a technical requirement. It is also about safeguarding the user experience and your brand's reputation. When deploying on a platform like Aptible, these aspects of quality assurance take on even greater importance. Aptible is designed to handle secure and compliant deployments, making it a popular choice for sensitive applications. Therefore, your testing must be thorough, covering not just app-specific checks but also the hosting environment and platform-specific nuances.

As you proceed with the deployment of your Django app on Aptible, it is essential to adopt a holistic approach to testing and quality assurance. This will ensure that the app not only functions optimally but also takes advantage of Aptible's unique features and benefits.

Note: Replace **`[APP_HANDLE]`** and **`[ENVIRONMENT_HANDLE]`** with appropriate names for your application and environment.

1. **Setting up your Test Environment on Aptible:**

a. Start by creating a new environment for testing on Aptible. This ensures that your main app remains unaffected during testing.
b. Use the **`aptible apps:create [APP_HANDLE] --environment [ENVIRONMENT_HANDLE]`** command to set up a separate testing environment.

2. **Deploying to the Test Environment**
   a. After setting up your tests locally, push your Dockerized Django app to your Aptible test environment.
   b. Use the `aptible deploy --app [APP_HANDLE]` command after making sure your Dockerfile is correctly configured. In other words, ensure you're:
      i. Using an appropriate base image for your Django app,
      ii. That all your project's dependencies are installed,
      iii. That your app reads all required environment variables like `SECRET_KEY`, `DEBUG`, `ALLOWED_HOSTS`
      iv. Configurations related to serving static and media files are correctly set up
      v. The CMD or ENTRYPOINT in your Dockerfile should correctly start your Django app.
      vi. The Dockerfile does not contain sensitive information or secrets
      vii. Sending your application logs to `stdout` or `stderr`
      viii. Creating a mechanism to handle database migrations.

3. **Running Unit Test:**
   a. Once deployed, run your **`unittest`** or **`pytest`** tests using:

```
Unset
aptible ssh --app [APP_HANDLE] --command "python manage.py test [app_name]"
```

4. **Integration and Functional Tests:**
   a. Ensure your database and other required services are linked.
   b. Run integration and functional tests using the above SSH command, but specify the appropriate test files.
5. **Automate Testing Workflow**

a. Consider using Aptible's CI/CD integrations to automatically run tests when changes are pushed to your codebase.

6. **Performance Testing**
   a. Deploy tools like 'Locust' or 'JMeter' on Aptible and simulate user loads to gauge performance.

7. **Security Testing:**
   a. Regularly scan your deployed app on Aptible using the Aptible security scan.

8. **Docker-specific Testing on Aptible:**
   a. Ensure that your Django app interacts correctly with other services in the test environment on Aptible by running integration tests that touch all linked services.

9. **Regression Testing**
   a. Regularly redeploy and retest your app using the above steps whenever there are changes to your codebase.

10. **Cleanup**
    a. Once testing is complete and you're satisfied, ensure to remove the test environment to avoid unnecessary charges.

```
Unset
aptible apps:deprovision --app [APP_HANDLE]
```

Tips

- Continuously review and improve your test suite as your app evolves.
- If your Django app has a frontend component, consider using browser testing platforms that integrate with Aptible to test across different browsers and devices.
- See our Best Practices Guide for more information on setting up your Aptible account with a production ready environment.

<Replace `[APP_HANDLE]` and `[ENVIRONMENT_HANDLE]` with appropriate names for your application and environment. >

H2: Scaling and Maintenance

Applications hosted on Aptible may encounter fluctuating demands. To manage this, users can either allocate additional resources to their environment or extend their deployment to include more containers.

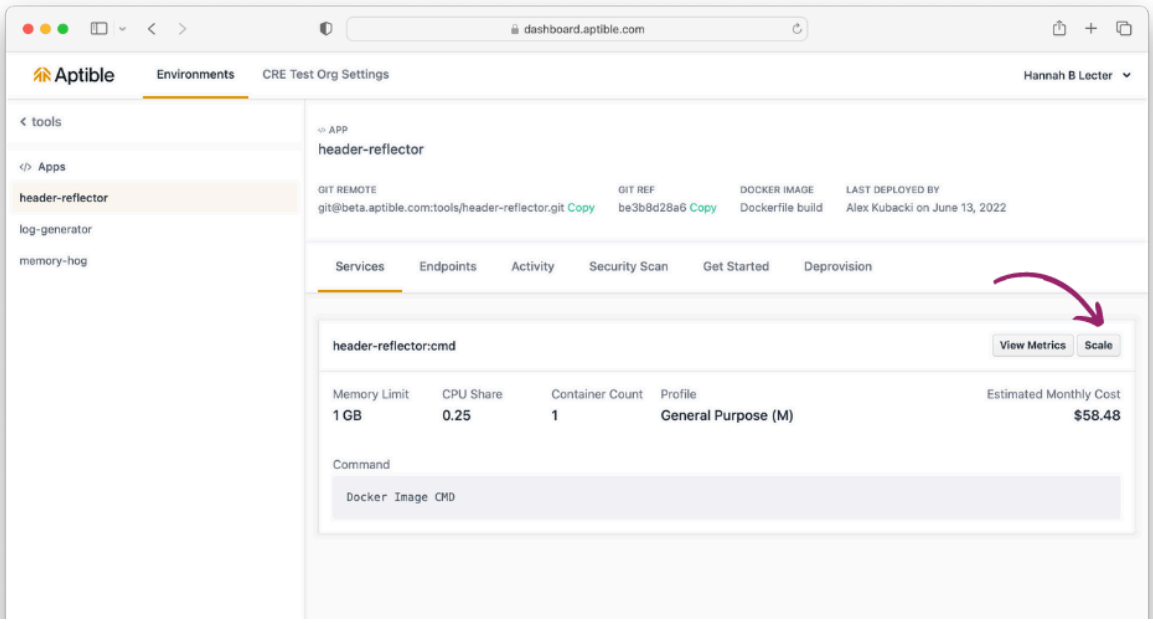
Aptible uses a load balancer to manage traffic across multiple containers, ensuring efficient distribution of network requests. Aptible's scaling configurations support various methods, which are detailed in the subsequent sections.

Learn more about how scaling works on Aptible.

H3: Manual Scaling

In Aptible, manual scaling offers users the ability to intentionally adjust resources based on anticipated needs. This means you can pre-emptively allocate more containers or resources when expecting higher traffic or scale down during off-peak periods to conserve resources.

To manually scale your app, navigate to the deployment settings in the Aptible dashboard. Navigate to the Environment in which your App lives, select the **Apps** tab, select the Django App, then select **Scale**.



To scale your Aptible app, you can use the aptible apps:scale command, the Aptible dashboard, or the Aptible Terraform Provider.

From here, you can specify the number of containers or the amount of resources to allocate. This method provides greater control over the resources available to your application, but it can also be more expensive. You should carefully consider your application's demands and monitor its performance closely to ensure that you are not over-provisioning resources.

In addition to scaling horizontally, you can also scale vertically by increasing the resources available to a single container. This can be done by increasing the amount of CPU, memory, or storage allocated to the container. Scaling vertically can be a more cost-effective way to scale your application, but it can also be more complex to manage.

H3: Step-by-Step Instructions

Note: Remember to replace placeholders like `[APP_HANDLE], [DATABASE_HANDLE], [DESIRED_SIZE], and [NEW_VERSION]` with the appropriate names or values for your application and environment.

1. **Monitoring Performance Metrics:**
    a. Before scaling, it's important to know the current performance of your app.
    b. Use Aptible's built-in metrics or metric drains.

2. **Scale Containers Vertically (Resources per Container):**
    a. Scale your Django apps vertically by changing the size of Containers, i.e., changing their Memory Limits and CPU Limits.
    b. The available sizes are determined by the Container Profile. Aptible offers a variety of container profiles: general purpose, CPU optimized, and RAM optimized.
    c. If your app requires more resources, you can increase the container size.

```
Unset
aptible apps:scale --app "$APP_HANDLE" SERVICE \
```

```
   --container-size SIZE_MB
```

3. **Scale Containers Horizontally (Number of Containers):**
   a. Scale your Django apps horizontally by adding more Containers to a given Service. Services scale up to 32 Containers.
   b. Note that apps scaled to 2 or more Containers are automatically deployed in a high-availability configuration, with Containers deployed in separate AWS Availability Zones.
   c. If you need to handle more concurrent connections or distribute traffic, add more containers. For example, to scale the `web` Service to 2 containers, you would use the following command:

```
Unset
aptible apps:scale --container-count [DESIRED_NUMBER]
```

   d. Replace `[DESIRED_NUMBER]` with the number of containers you want.

4. **Scaling Databases:**
   a. Aptible users can restart a database to resize it. The command below restarts the database and resizes it to 2048 MB.

```
Unset
aptible db:restart "$DB_HANDLE" \
   --container-size 2048
```

5. **Regular Backups:**
   a. Aptible automatically backups your databases every 24 hours. You can also configure Aptible to retain additional daily backups, as well as monthly backups.
   b. For monthly backups, Aptible will retain the last automatic backup of each month. This means that you will always have at least one backup of your database that is no more than one month old.

c. You can view your database backups in two ways::

   i. Using the `aptible backup:list` command.

   ii. Within the Aptible Dashboard, by navigating to the **Database** > **Backup** tab.

6. **Database Maintenance & Upgrades:**

   a. Aptible performs maintenance for databases since they're managed.

7. **Periodic Reviews with Aptible Metrics**

   a. Routinely monitor the metrics provided by Aptible. This helps in understanding when to scale up/down and in ensuring optimal performance. The following code will get the list of all metrics for the current environment.

```python
import aptible

def get_metrics():
 """Gets the metrics for the current environment."""
 client = aptible.Client()
 metrics = client.metrics.list()

 return metrics

def main():
 """Periodically reviews Aptible metrics."""
 metrics = get_metrics()

 for metric in metrics:
  print(metric.name, metric.value)

if __name__ == "__main__":
 main()
```

8. **Handling Traffic Spikes:**

   a. For anticipated traffic spikes, pre-scale your app and database temporarily.

   b. Once the traffic normalizes, you can scale down to manage costs.

8. **Regular Security Reviews:**

a. Periodically check for updates and ensure your Django app and all dependencies are up to date.
b. Use Aptible's in-built features to run regular vulnerability scans.

9. **Cleanup and Optimization:**
    a. Remove unused services, endpoints, or databases to optimize costs and performance.

```
Unset
aptible apps:deprovision --app [APP_HANDLE]
```

    b. The code above needs to be specific to resource to deprovision
    c. Only run this if you are sure you want to remove the app permanently
    d. However, before running the command, take note:
        i. **Be Specific:** Ensure the `[APP_HANDLE]` is correctly specified to avoid accidentally deleting the wrong resource.
        ii. **Finality:** This action is irreversible. Once you deprovision an app, it's gone forever. Double-check and be 100% sure before proceeding.
        iii. **Backup:** Always have a backup of your data, especially if you're making significant changes or deletions

10. **Tips:**
    a. Set up metric drains to monitor your Django app's performance and receive alerts when metrics go beyond your desired thresholds.
    b. Always test changes in a staging environment before applying to production.

H2: Troubleshooting and Debugging

Troubleshooting and debugging your Dockerized Django app are inevitable. Doing so will help you maintain a reliable and smoothly functioning application. Follow these steps to identify and resolve issues quickly:

H3: Logging and Error Handling:

Implementing comprehensive logging in a Django app is vital for diagnosing issues and understanding the behavior of your application. Let's break down logging and error handling in a Django app in Aptible:

1. **Initialize Comprehensive Logging:**
    a. Setup Django Logging in `settings.py`:
        - Define formatters, handlers, and loggers as provided in your existing guide.
        - Ensure log files, such as `debug.log`, have correct permissions and are accessible.
    b. Implement App-level Logging:

```python
import logging
logger = logging.getLogger(__name__)
```

    c. **Handle Exceptions** in your views or other components:

```python
def some_view(request):
  try:
    # some code here
  except Exception as e:
    logger.error(f"Error encountered: {e}")
```

2. **Capture System-Level Logs**
    a. **Database Queries**: Configure `django.db.backends` logger.
    b. **Server Operations:** Ensure logs from Gunicorn, uWSGI, or other servers are captured.
3. **Integrate Error Reporting Tools:**
    a. Use tools like Sentry:
        - Install Sentry's SDK `pip install sentry-sdk`
        - Setup Sentry in your Django project:

```python
import sentry_sdk
from sentry_sdk.integrations.django import DjangoIntegration

sentry_sdk.init(
  dsn="YOUR DSN HERE",
  integrations=[DjangoIntegration()],
  traces_sample_rate=1.0,
  send_default_pii=True
)
```

4. **Review Configuration Settings**
    a. **Verify Debug Mode:** Set debug to `STDOUT` (local file isn't recommended). With `STDOUT` for errors, users can review log drain to check logs.
    b. **Database Settings:** Confirm database settings in **`DATABASES`** dictionary.
    c. **Domain Verification:** Confirm your domain is listed in **`ALLOWED_HOSTS`**.
    d. **Static/Media Files:** Check **`STATIC_URL`**, **`STATIC_ROOT`**, **`MEDIA_URL`**, and **`MEDIA_ROOT`**.
5. **Review Configuration Settings**
    a. Check server or container: **`echo $MY_ENV_VARIABLE.t`**.
    b. Confirm Django's accessibility to these variables using tools like **`python-decouple`** or **`django-environ`**.
6. **Confirm Third-Party Apps & Middleware:**
    a. Review **`INSTALLED_APPS`** and **`MIDDLEWARE`** for correct configurations.
    b. Ensure no missing database tables or middleware errors.
7. **Validate External Service Configurations:**
    a. Confirm connections to caching services, message brokers, or any third-party services.
    b. Ensure no missing database tables or middleware errors.
8. **Use Django's In-built Troubleshooter:**
    a. Run: **`python manage.py check`** and address any warnings or errors.
9. **Verify Custom Configurations:**

a. Ensure any custom settings (API keys, service URLs) are correctly set.

10. **Examine Logs on Aptible:**

a. To check your Django app's logs for any unusual behaviors or errors, you can use Aptible's built-in logging dashboard or CLI.

b. You can also use a log drain to send your app's logs to a third-party service for analysis. This can be useful if you want to use a more sophisticated logging solution than Aptible's built-in logging.

c. **Choose a Log Management Service:** Aptible doesn't store logs for you long term. Instead, you'll use a Log Drain to send logs to a third-party log storage and analysis platform.

   i. Navigate to the Aptible dashboard
   ii. Choose your environment
   iii. Click on the “Log Drains” option on the sidebar
   iv. Click on “New Log Drain”
   v. Select the type of Log Drain that corresponds to your log management service (3.g. “Syslog TLS,” “HTTPS,” etc.).
   vi. Provide the endpoint information. This is typically given to you by your log management service. For example for Syslog it could be something like `logsN.papertrailapp.com:XXXXX`

d. Ensure your Django App is Logging Properly

   i. By default, Django logs errors, but if you want more detailed logs, update your Django

settings.py

```
Unset
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
```

```
        'django': {
            'handlers': ['console'],
            'level': 'DEBUG',  # Change this as per your needs
            'propagate': True,
        },
    },
}
```

This configuration logs Django-related messages to the console, which Aptible captures and can forward to your log drain.

e. **Monitor and Analyze Your Logs**

Once logs start flowing into your log management service

- Set up alerts to be notified about critical errors or suspicious activities
- Regularly review logs to ensure the application is running smoothly
- Use the search and visualization tools of your log management service to dig deep into issues.

f. **Troubleshoot Issues**

When issues arise:

- Use filters to narrow down the logs to a specific time frame or severity level
- Identify patterns — Are errors coming from a specific part of your app?
- Correlate with other events> Were there any deployments or changes around the same time?
- Contact Aptible Support if you're stuck