Contents

Docker on Azure

Docker is a popular container management and imaging platform that allows you to quickly work with containers on Linux and Windows. Learn how to leverage Docker on Azure with our quickstarts and tutorials.

5-Minute Quickstarts

Learn how to deploy Docker on Azure:

Deploy Docker on Ubuntu Quickstart template

Step-by-Step Tutorials

Learn how to use Docker in your application and as part of your CI/CD pipeline:

1. Build Python and PostgreSQL with Docker
2. CI/CD with Docker Swarm and Azure DevOps Services
3. Integrate Docker, Jenkins, and Linux VMs

Resources

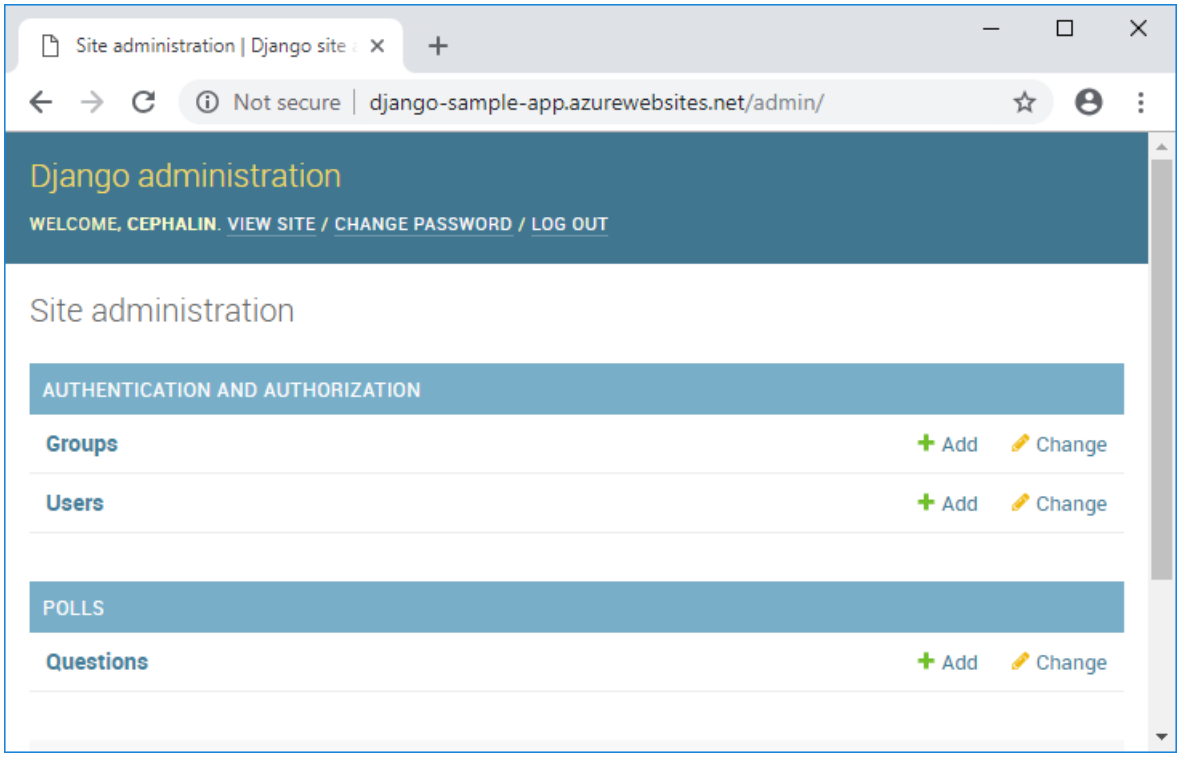
See the following links for additional reference materials and tools for working with Docker on Azure.

Docker Azure site
Docker Azure driver
Learn about Docker

Build a Python and PostgreSQL app in Azure App Service

4/16/2019 • 14 minutes to read • Edit Online

App Service on Linux provides a highly scalable, self-patching web hosting service. This tutorial shows how to create a data-driven Python app, using PostgreSQL as the database back end. When you are done, you have a Django application running in App Service on Linux.



In this tutorial, you learn how to:

- Create a PostgreSQL database in Azure
- Connect a Python app to PostgreSQL
- Deploy the app to Azure
- View diagnostic logs
- Manage the app in the Azure portal

> **NOTE**
> Before creating an Azure Database for PostgreSQL, please check which compute generation is available in your region.

You can follow the steps in this article on macOS. Linux and Windows instructions are the same in most cases, but the differences are not detailed in this tutorial.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

To complete this tutorial:

1. Install Git
2. Install Python
3. Install and run PostgreSQL

Test local PostgreSQL installation and create a database

In a local terminal window, run `psql` to connect to your local PostgreSQL server.

```
sudo -u postgres psql postgres
```

If you get an error message similar to `unknown user: postgres`, your PostgreSQL installation may be configured with your logged in username. Try the following command instead.

```
psql postgres
```

If your connection is successful, your PostgreSQL database is running. If not, make sure that your local PostgresQL database is started by following the instructions for your operating system at Downloads - PostgreSQL Core Distribution.

Create a database called *pollsdb* and set up a separate database user named *manager* with password *supersecretpass*.

```
CREATE DATABASE pollsdb;
CREATE USER manager WITH PASSWORD 'supersecretpass';
GRANT ALL PRIVILEGES ON DATABASE pollsdb TO manager;
```

Type `\q` to exit the PostgreSQL client.

Create local Python app

In this step, you set up the local Python Django project.

Clone the sample app

Open the terminal window, and `CD` to a working directory.

Run the following commands to clone the sample repository.

```
git clone https://github.com/Azure-Samples/djangoapp.git
cd djangoapp
```

This sample repository contains a Django application. It's the same data-driven app you would get by following the getting started tutorial in the Django documentation. This tutorial doesn't teach you Django, but shows you how to take deploy and run a Django app (or another data-driven Python app) to App Service.

Configure environment

Create a Python virtual environment and use a script to set the database connection settings.

```
# Bash
python3 -m venv venv
source venv/bin/activate
source ./env.sh

# PowerShell
py -3 -m venv venv
venv\scripts\activate
.\env.ps1
```

The environment variables defined in *env.sh* and *env.ps1* are used in *azuresite/settings.py* to define the database settings.

Run app locally

Install the required packages, run Django migrations and create an admin user.

```
pip install -r requirements.txt
python manage.py migrate
python manage.py createsuperuser
```

Once the admin user is created, run the Django server.

```
python manage.py runserver
```

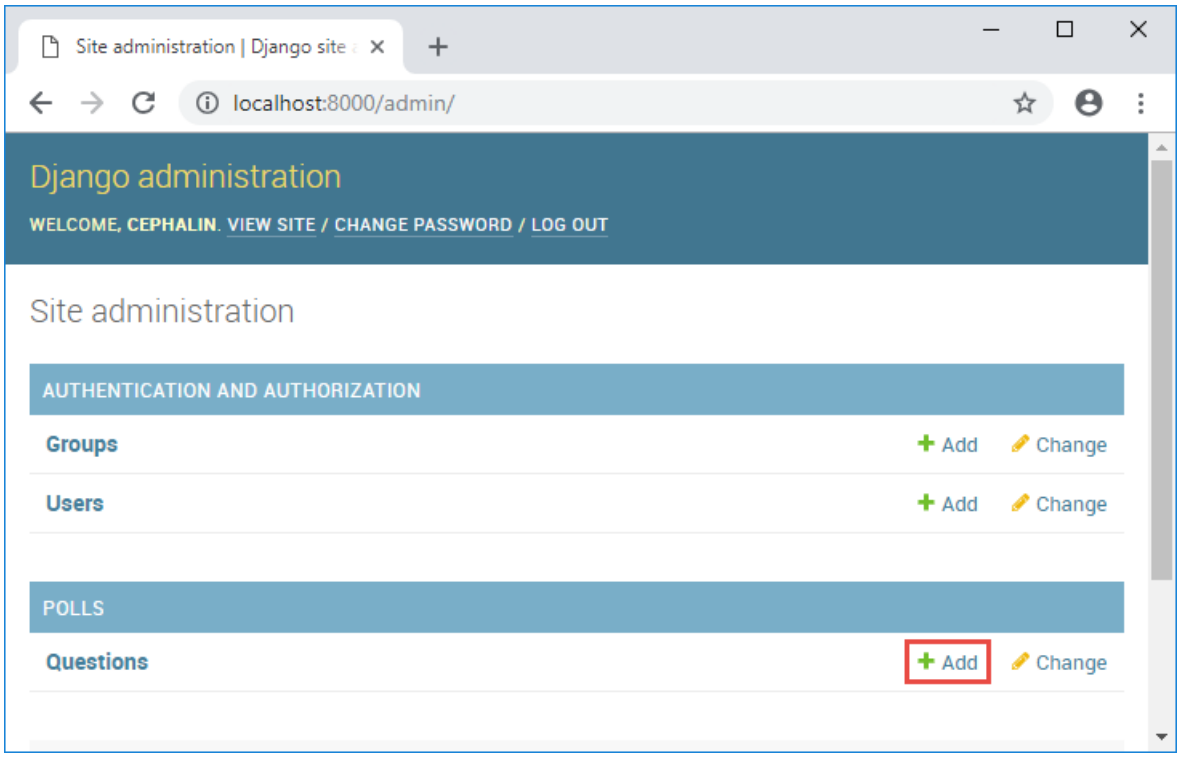
When the app is fully loaded, you see something similar to the following message:

```
Performing system checks...

System check identified no issues (0 silenced).
October 26, 2018 - 10:54:59
Django version 2.1.2, using settings 'azuresite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Navigate to `http://localhost:8000` in a browser. You should see the message `No polls are available.`.

Navigate to `http://localhost:8000/admin` and sign in using the admin user you created in the last step. Click **Add** next to **Questions** and create a poll question with some choices.



Navigate to `http://localhost:8000` again and see the poll question displayed.

The Django sample application stores user data in the database. If you are successful at adding a poll question, your app is writing data to the local PostgreSQL database.

To stop the Django server at anytime, type Ctrl+C in the terminal.

Create a production PostgreSQL database

In this step, you create a PostgreSQL database in Azure. When your app is deployed to Azure, it uses this cloud database.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Select **Copy** to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select **Try It** in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the **Cloud Shell** button on the menu in the upper-right corner of the Azure portal.	

Create a resource group

A resource group is a logical container into which Azure resources like web apps, databases, and storage accounts are deployed and managed. For example, you can choose to delete the entire resource group in one simple step later.

In the Cloud Shell, create a resource group with the `az group create` command. The following example creates a

resource group named *myResourceGroup* in the *West Europe* location. To see all supported locations for App Service on Linux in **Basic** tier, run the `az appservice list-locations --sku B1 --linux-workers-enabled` command.

```
az group create --name myResourceGroup --location "West Europe"
```

You generally create your resource group and the resources in a region near you.

When the command finishes, a JSON output shows you the resource group properties.

Create an Azure Database for PostgreSQL server

Create a PostgreSQL server with the `az postgres server create` command in the Cloud Shell.

In the following example command, replace *<postgresql-name>* with a unique server name, and replace *<admin-username>* and *<admin-password>* with the desired user credentials. The user credentials are for the database administrator account. The server name is used as part of your PostgreSQL endpoint (`https://<postgresql-name>.postgres.database.azure.com`), so the name needs to be unique across all servers in Azure.

```
az postgres server create --resource-group myResourceGroup --name <postgresql-name> --location "West Europe" --
admin-user <admin-username> --admin-password <admin-password> --sku-name B_Gen4_1
```

When the Azure Database for PostgreSQL server is created, the Azure CLI shows information similar to the following example:

```
{
  "administratorLogin": "<admin-username>",
  "fullyQualifiedDomainName": "<postgresql-name>.postgres.database.azure.com",
  "id": "/subscriptions/00000000-0000-0000-0000-
000000000000/resourceGroups/myResourceGroup/providers/Microsoft.DBforPostgreSQL/servers/<postgresql-name>",
  "location": "westus",
  "name": "<postgresql-name>",
  "resourceGroup": "myResourceGroup",
  "sku": {
    "capacity": 1,
    "family": "Gen4",
    "name": "B_Gen4_1",
    "size": null,
    "tier": "Basic"
  },
  < JSON data removed for brevity. >
}
```

> **NOTE**
>
> Remember <admin-username> and <admin-password> for later. You need them to sign in to the Postgre server and its databases.

Create firewall rules for the PostgreSQL server

In the Cloud Shell, run the following Azure CLI commands to allow access to the database from Azure resources.

```
az postgres server firewall-rule create --resource-group myResourceGroup --server-name <postgresql-name> --
start-ip-address=0.0.0.0 --end-ip-address=0.0.0.0 --name AllowAllAzureIPs
```

NOTE

This setting allows network connections from all IPs within the Azure network. For production use, try to configure the most restrictive firewall rules possible by using only the outbound IP addresses your app uses.

In the Cloud Shell, run the command again to allow access from your local computer by replacing <*your-ip-address*> with your local IPv4 IP address.

```
az postgres server firewall-rule create --resource-group myResourceGroup --server-name <postgresql-name> --
start-ip-address=<your-ip-address> --end-ip-address=<your-ip-address> --name AllowLocalClient
```

Connect Python app to production database

In this step, you connect your Django sample app to the Azure Database for PostgreSQL server you created.

Create empty database and user access

In the Cloud Shell, connect to the database by running the command below. When prompted for your admin password, use the same password you specified in Create an Azure Database for PostgreSQL server.

```
psql -h <postgresql-name>.postgres.database.azure.com -U <admin-username>@<postgresql-name> postgres
```

Just like in your local Postgres server, create the database and user in the Azure Postgres server.

```
CREATE DATABASE pollsdb;
CREATE USER manager WITH PASSWORD 'supersecretpass';
GRANT ALL PRIVILEGES ON DATABASE pollsdb TO manager;
```

Type `\q` to exit the PostgreSQL client.

NOTE

It's best practice to create database users with restricted permissions for specific applications, instead of using the admin user. In this example, the `manager` user has full privileges to *only* the `pollsdb` database.

Test app connectivity to production database

In the local terminal window, change the database environment variables (which you configured earlier by running *env.sh* or *env.ps1*):

```
# Bash
export DBHOST="<postgresql-name>.postgres.database.azure.com"
export DBUSER="manager@<postgresql-name>"
export DBNAME="pollsdb"
export DBPASS="supersecretpass"

# PowerShell
$Env:DBHOST = "<postgresql-name>.postgres.database.azure.com"
$Env:DBUSER = "manager@<postgresql-name>"
$Env:DBNAME = "pollsdb"
$Env:DBPASS = "supersecretpass"
```

Run Django migration to the Azure database and create an admin user.

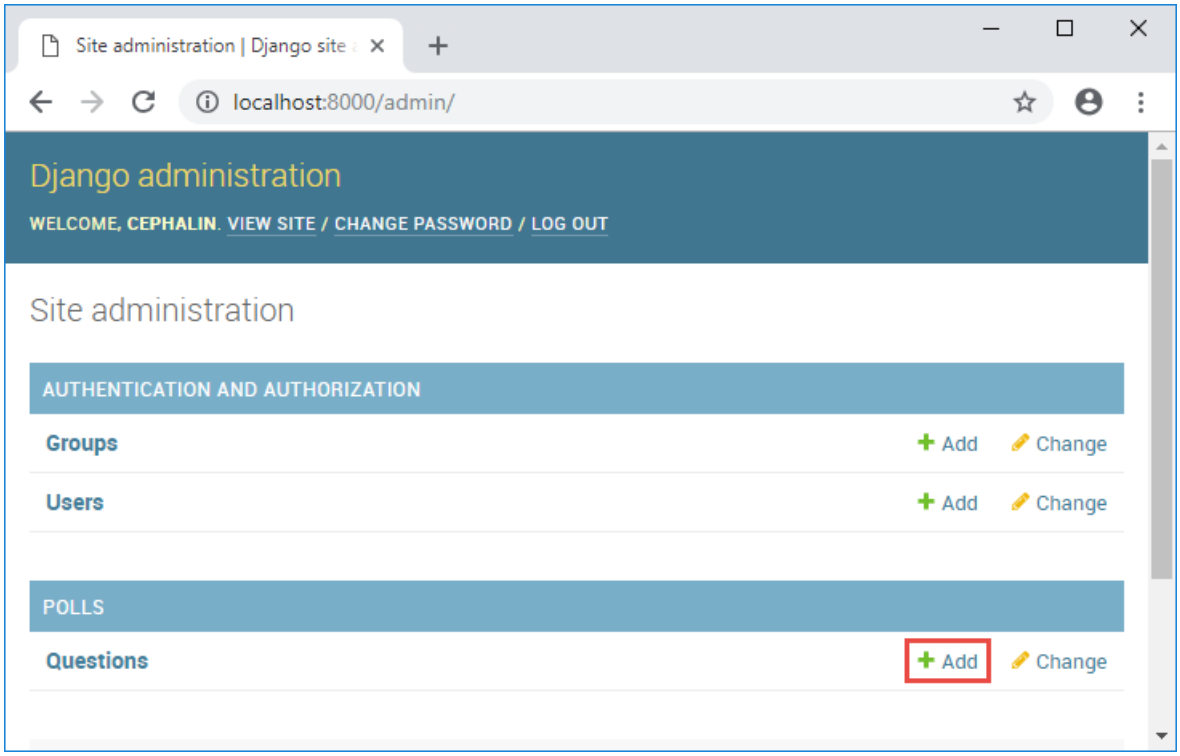
```
python manage.py migrate
python manage.py createsuperuser
```

Once the admin user is created, run the Django server.

```
python manage.py runserver
```

Navigate to `http://localhost:8000` in again. You should see the message `No polls are available.` again.

Navigate to `http://localhost:8000/admin` and sign in using the admin user you created, and create a poll question like before.



Navigate to `http://localhost:8000` again and see the poll question displayed. Your app is now writing data to the database in Azure.

Deploy to Azure

In this step, you deploy the Postgres-connected Python application to Azure App Service.

Configure repository

Django validates the `HTTP_HOST` header in incoming requests. For your Django app to work in App Service, you need to add the full-qualified domain name of the app to the allowed hosts. Open *azuresite/settings.py* and find the `ALLOWED_HOSTS` setting. Change the line to:

```
ALLOWED_HOSTS = [os.environ['WEBSITE_SITE_NAME'] + '.azurewebsites.net', '127.0.0.1'] if 'WEBSITE_SITE_NAME' in os.environ else []
```

Next, Django doesn't support serving static files in production, so you need to enable this manually. For this tutorial, you use WhiteNoise. The WhiteNoise package is already included in *requirements.txt*. You just need to configure Django to use it.

In *azuresite/settings.py*, find the `MIDDLEWARE` setting, and add the `whitenoise.middleware.WhiteNoiseMiddleware` middleware to the list, just below the `django.middleware.security.SecurityMiddleware` middleware. Your `MIDDLEWARE`

setting should look like this:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',
    ...
]
```

At the end of *azuresite/settings.py*, add the following lines.

```
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'

STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

For more information on configuring WhiteNoise, see the WhiteNoise documentation.

> **IMPORTANT**
> The database settings section already follows the security best practice of using environment variables. For the complete deployment recommendations, see Django Documentation: deployment checklist.

Commit your changes into the repository.

```
git commit -am "configure for App Service"
```

Configure deployment user

In the Azure Cloud Shell, configure deployment credentials with the `az webapp deployment user set` command. This deployment user is required for FTP and local Git deployment to a web app. The username and password are account level. *They're different from your Azure subscription credentials.*

In the following example, replace <*username*> and <*password*>, including the brackets, with a new username and password. The username must be unique within Azure. The password must be at least eight characters long, with two of the following three elements: letters, numbers, and symbols.

```
az webapp deployment user set --user-name <username> --password <password>
```

You get a JSON output with the password shown as `null`. If you get a `'Conflict'. Details: 409` error, change the username. If you get a `'Bad Request'. Details: 400` error, use a stronger password. The deployment username must not contain '@' symbol for local Git pushes.

You configure this deployment user only once. You can use it for all your Azure deployments.

> **NOTE**
> Record the username and password. You need them to deploy the web app later.

Create App Service plan

In the Cloud Shell, create an App Service plan in the resource group with the `az appservice plan create` command.

The following example creates an App Service plan named `myAppServicePlan` in the **Basic** pricing tier (`--sku B1`) and in a Linux container (`--is-linux`).

```
az appservice plan create --name myAppServicePlan --resource-group myResourceGroup --sku B1 --is-linux
```

When the App Service plan has been created, the Azure CLI shows information similar to the following example:

```
{
  "adminSiteName": null,
  "appServicePlanName": "myAppServicePlan",
  "geoRegion": "West Europe",
  "hostingEnvironmentProfile": null,
  "id": "/subscriptions/0000-
0000/resourceGroups/myResourceGroup/providers/Microsoft.Web/serverfarms/myAppServicePlan",
  "kind": "linux",
  "location": "West Europe",
  "maximumNumberOfWorkers": 1,
  "name": "myAppServicePlan",
  < JSON data removed for brevity. >
  "targetWorkerSizeId": 0,
  "type": "Microsoft.Web/serverfarms",
  "workerTierName": null
}
```

Create a web app

Create a web app in the `myAppServicePlan` App Service plan.

In the Cloud Shell, you can use the `az webapp create` command. In the following example, replace `<app-name>` with a globally unique app name (valid characters are `a-z`, `0-9`, and `-`). The runtime is set to `PYTHON|3.7`. To see all supported runtimes, run `az webapp list-runtimes --linux`.

```
# Bash
az webapp create --resource-group myResourceGroup --plan myAppServicePlan --name <app-name> --runtime
"PYTHON|3.7" --deployment-local-git
# PowerShell
az --% webapp create --resource-group myResourceGroup --plan myAppServicePlan --name <app-name> --runtime
"PYTHON|3.7" --deployment-local-git
```

When the web app has been created, the Azure CLI shows output similar to the following example:

```
Local git is configured with url of 'https://<username>@<app-name>.scm.azurewebsites.net/<app-name>.git'
{
  "availabilityState": "Normal",
  "clientAffinityEnabled": true,
  "clientCertEnabled": false,
  "cloningInfo": null,
  "containerSize": 0,
  "dailyMemoryTimeQuota": 0,
  "defaultHostName": "<app-name>.azurewebsites.net",
  "deploymentLocalGitUrl": "https://<username>@<app-name>.scm.azurewebsites.net/<app-name>.git",
  "enabled": true,
  < JSON data removed for brevity. >
}
```

You've created an empty new web app, with git deployment enabled.

> **NOTE**
>
> The URL of the Git remote is shown in the `deploymentLocalGitUrl` property, with the format `https://<username>@<app-name>.scm.azurewebsites.net/<app-name>.git`. Save this URL as you need it later.

Configure environment variables

Earlier in the tutorial, you defined environment variables to connect to your PostgreSQL database.

In App Service, you set environment variables as *app settings* by using the `az webapp config appsettings set` command in Cloud Shell.

The following example specifies the database connection details as app settings.

```
az webapp config appsettings set --name <app-name> --resource-group myResourceGroup --settings DBHOST="
<postgresql-name>.postgres.database.azure.com" DBUSER="manager@<postgresql-name>" DBPASS="supersecretpass"
DBNAME="pollsdb"
```

For information on how these app settings are accessed in your code, see Access environment variables.

Push to Azure from Git

Back in the *local terminal window*, add an Azure remote to your local Git repository. Replace *<deploymentLocalGitUrl-from-create-step>* with the URL of the Git remote that you saved from Create a web app.

```
git remote add azure <deploymentLocalGitUrl-from-create-step>
```

Push to the Azure remote to deploy your app with the following command. When prompted for credentials by Git Credential Manager, make sure that you enter the credentials you created in Configure a deployment user, not the credentials you use to sign in to the Azure portal.

```
git push azure master
```

This command may take a few minutes to run. While running, it displays information similar to the following example:

```
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 775 bytes | 0 bytes/s, done.
Total 7 (delta 4), reused 0 (delta 0)
remote: Updating branch 'master'.
remote: Updating submodules.
remote: Preparing deployment for commit id '6520eeafcc'.
remote: Generating deployment script.
remote: Running deployment command...
remote: Python deployment.
remote: Kudu sync from: '/home/site/repository' to: '/home/site/wwwroot'
.
.
.
remote: Deployment successful.
remote: App container will begin restart within 10 seconds.
To https://<app-name>.scm.azurewebsites.net/<app-name>.git
   06b6df4..6520eea  master -> master
```

The App Service deployment server sees *requirements.txt* in the repository root and runs Python package management automatically after `git push`.

Browse to the Azure app

Browse to the deployed app. It takes some time to start because the container needs to be downloaded and run when the app is requested for the first time. If the page times out or displays an error message, wait a few minutes

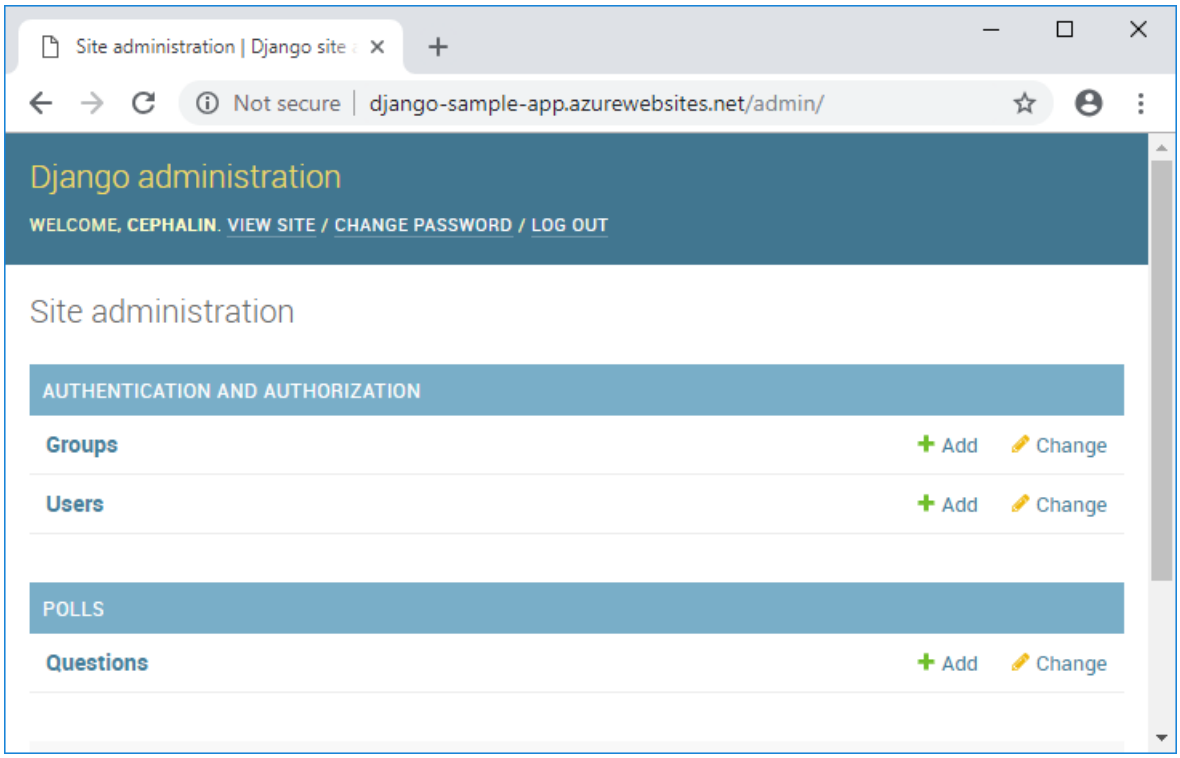
and refresh the page.

```
http://<app-name>.azurewebsites.net
```

You should see the poll question that you created earlier.

App Service detects a Django project in your repository by looking for a *wsgi.py* in each subdirectory, which is created by `manage.py startproject` by default. When it finds the file, it loads the Django app. For more information on how App Service loads Python apps, see Configure built-in Python image.

Navigate to `<app-name>.azurewebsites.net` and sign in using same admin user you created. If you like, try creating some more poll questions.



Congratulations! You're running a Python app in App Service for Linux.

Stream diagnostic logs

You can access the console logs generated from inside the container. First, turn on container logging by running the following command in the Cloud Shell:

```
az webapp log config --name <app-name> --resource-group myResourceGroup --docker-container-logging filesystem
```

Once container logging is turned on, run the following command to see the log stream:

```
az webapp log tail --name <app-name> --resource-group myResourceGroup
```

If you don't see console logs immediately, check again in 30 seconds.

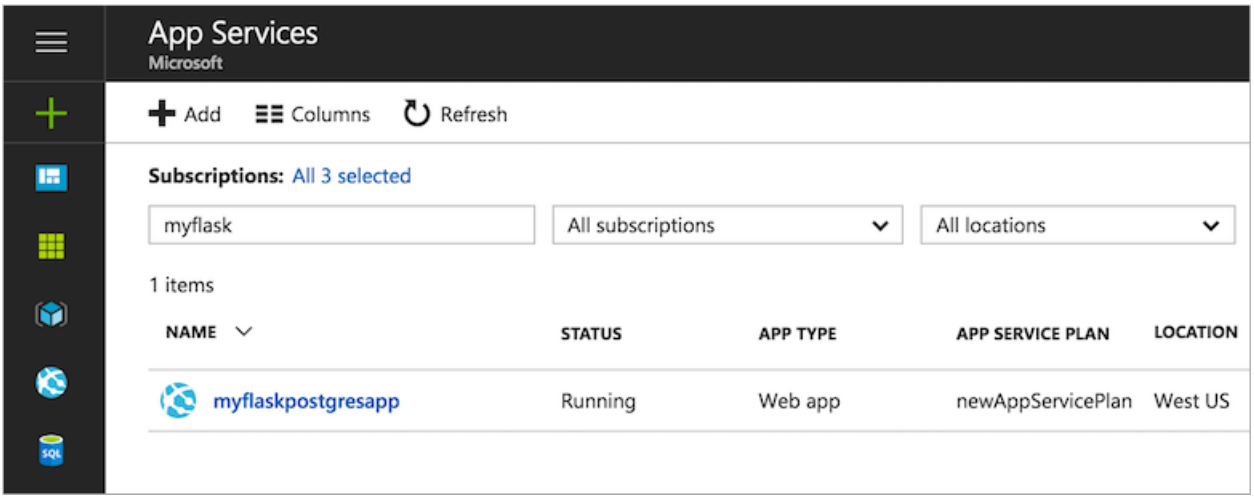
> **NOTE**
> You can also inspect the log files from the browser at `https://<app-name>.scm.azurewebsites.net/api/logs/docker`.

To stop log streaming at any time, type `Ctrl` + `C`.

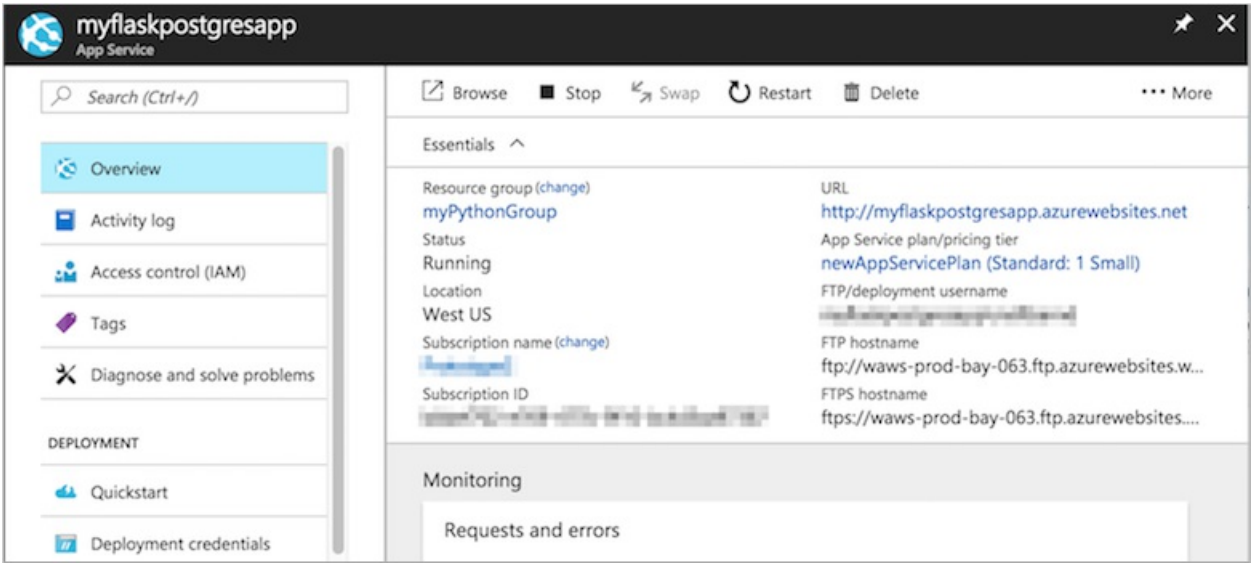
Manage your app in the Azure Portal

Go to the Azure portal to see the app you created.

From the left menu, click **App Services**, then click the name of your Azure app.



By default, the portal shows your app's **Overview** page. This page gives you a view of how your app is doing. Here, you can also perform basic management tasks like browse, stop, start, restart, and delete. The tabs on the left side of the page show the different configuration pages you can open.



Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, delete the resource group by running the following command in the Cloud Shell:

```
az group delete --name myResourceGroup
```

This command may take a minute to run.

Next steps

In this tutorial, you learned how to:

- Create a PostgreSQL database in Azure

- Connect a Python app to PostgreSQL
- Deploy the app to Azure
- View diagnostic logs
- Manage the app in the Azure portal

Advance to the next tutorial to learn how to map a custom DNS name to your app.

Tutorial: Map custom DNS name to your app

Or, check out other resources:

Configure Python app