Learning Objectives: Environment Variables

Learners will be able to...

- **Define an environment variable**
- **Store environment variables in an `.env` file**
- **Read environment variables into a Python program**
- **Set default value for an environment variable**
- **Cast environment variables as different data types**

info

Make Sure You Know

You are familiar with the command line, installing packages, and basic Python.

Limitations

Environment variables are only mentioned with respect to this Django project. They are also discussed using the `environs` package.

Environment Variables

What are Environment Variables?

The first step in getting our Django project ready for production is to separate out information for our code base. We are separating information because it is sensitive information we do not want others to see. For example, every Django project has a secret key. As its name implies, we do not want to share this with the world. We would do exactly that if we upload our Django project to GitHub.

```
SECRET_KEY = "django-insecure-a^i57h%uoa*v*c_$v5)7jeec7g!mgt%v^-
o!*0qfu#^z_30%hh"
```

Another reason to separate information is because we need one group of settings for development on our machine and another group of settings for production. For example, we run our Django project in debug mode when developing it. This mode provides useful feedback when encountering an error. Once the project goes on a public web server, we no longer want it to run in debug mode. The same is true for out database. For development purposes, the SQLite database works fine, but we want something different in a production environment.

Environment variables are variables that exist outside of the program. We can use environment variables as a solution to the problems stated above. Envinronment variables are stored in a separate file, which we can ignore when pushing our project to GitHub. This means we can keep our secret key an actual secret. In addition we can set the debug mode and database for local development to be different than a production environment.

Environment Variable Setup

There are many ways to setup environment variables on a computer. We are going to use the environs Python package as it works well with Django. Let's start by activating the `django` virtual environment.

```
conda activate django
```

Next, we need to install the `environs` package. This package allows us to work with environment variables. We also need to install a helper package for handling the database URL as an environment variable. In particular, we want a version of `environs` designed to work with Django. This package,

unfortunately, is not found in Conda or Conda Forge. This is not a problem because Conda lets us install packages with `pip`. Enter the command below to install `environs[django]` with `pip`.

```
conda install -c conda-forge environs dj-database-url -y
```

Open the `settings.py` file. We are going to import `Env` from the `evirons` package. We can add the following code at the top of the file. Create the `env` variable and instantiate it as a `Env` object. Finally call the `read_env()` method. This method tells Django to go and read the file of environment variables so they can be used in the project.

```
import os
from pathlib import Path
from environs import Env

env = Env()
env.read_env()
```

Go back to the terminal and run the Black code formatter to make sure our file has a consistent look and feel.

```
black capital_cities/settings.py
```

Environment variables are stored in the `.env` file. The `.` means that this is a hidden file. You will not see it in a file tree of if you use the `ls` command. The file is still there, however, which is something we will address when we talk about GitHub. Open the terminal once again, and create the `.env` file with the `touch` command.

```
touch .env
```

Deactivate the `django` virtual environment.

```
conda deactivate
```

Codio-Specific Settings and Debug Mode

Codio-Specific Settings

Start by activating the `django` virtual environment.

```
conda activate django
```

When creating Django projects that run on the Codio platform, there are a few changes that we need to make to the `settings.py` file. We got our project running in the previous assignment, so we no longer need these specific settings.

First, adjust the `ALLOWED_HOSTS` variable. Technically the `"*"` would work with anything, but we are going to allow this project to run on localhost (`127.0.0.1` is another name for localhost) and any domain from Heroku. Heroku is a cloud platform as service that has a free tier so we can run our Django project on the internet (more on them later).

We are also going to comment out the variables regarding `X_FRAME_OPTIONS`, CSRF, and cookies. There are security implications to these settings. In order to get Django to run on Codio, we had to weaken the security a bit. Putting a site on the internet with weakened security is a bad idea. You can remove these variables instead of commenting them out if you want.

```
ALLOWED_HOSTS = ['.herokuapp.com', 'localhost', '127.0.0.1']
#X_FRAME_OPTIONS = "ALLOW-FROM " +
        os.environ.get("CODIO_HOSTNAME") + "-8000.codio.io"
#CSRF_COOKIE_SAMESITE = None
#CSRF_TRUSTED_ORIGINS = [
#    "https://" + os.environ.get("CODIO_HOSTNAME") +
        "-8000.codio.io"
#]
#CSRF_COOKIE_SECURE = True
#SESSION_COOKIE_SECURE = True
#CSRF_COOKIE_SAMESITE = "None"
#SESSION_COOKIE_SAMESITE = "None"
```

Next, go the `MIDDLEWARE` section of the settings file and uncomment the two lines of code.

```
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
    'django.middleware.csrf.CsrfViewMiddleware',
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Open the terminal back up and then run Black on the settings file.

```
black capital_cities/settings.py
```

Debug Mode

As mentioned, our Django project is setup to run in debug mode, which provides helpful feedback to developers. We could hard code `False` in our settings file and be done with it. However, a common practice is to create a flexible environment that adapts to our situation. That is, we can create an environment variable such that it allows for debug mode on our machine and turns it off when running elsewhere.

Click the link below to open the `.env` file. Environment variables are comprised of a name and value separated by a `=`. Name our first variable `DEBUG` (all caps is a common convention for environment variables) and set its value to `True`.

```
DEBUG=True
```

This is not a Python file, so we are not going to use Black to format it. Now open the settings file. Find the line of code in the file that defines the `DEBUG` value. Replace the value with `env.bool("DEBUG")`. This might seem a bit odd. However, the computer does not automatically match the `DEBUG` variable in the `.env` file with the `DEBUG` in the settings file. You have to specificially tell Python to use `DEBUG` in the `.env` file and to interpret the value as a boolean.

```
DEBUG = env.bool("DEBUG")
```

However, we set `DEBUG` to be `True` in the `.env` file. That's great for local development, but we need the value to be `False` in a production environment. Modify the settings file so that the default value for `DEBUG` is `False`. This means when Django is running on our machine (with the `.env`

file) the value of `DEBUG` is `True`. Move the project to another machine without the environment variables and Django uses `False` as the value. This way our Django project adapts to our situation.

```
DEBUG = env.bool('DEBUG', default=False)
```

Once more, open the terminal and format our file with Black.

```
black capital_cities/settings.py
```

Deactivate the `django` virtual environment.

```
conda deactivate
```

Secret Key

Reading the Secret Key

Start by activating the `django` virtual environment.

```
conda activate django
```

The Django settings file contains the variable `SECRET_KEY`. We want to remove this information from our code base so that when it ends up on GitHub, people will not see it. Open the settings file. Find the line where the `SECRET_KEY` variable is defined. Copy the value currently in the file. We need this for our `.env` file. Replace the value by reading the from the `.env` file. Use `env.str` so we can read the value as a string.

```
SECRET_KEY = env.str("SECRET_KEY")
```

Open the terminal back up and then run Black on the settings file.

```
black capital_cities/settings.py
```

Now open the `.env` file. Create the variable `SECRET_KEY` and set its value from the one you copied from the settings file. You do not need double quotes for the secret. This is handled when Python reads from the file. **Note**, your key will be different from the one listed below.

```
DEBUG=True
SECRET_KEY=django-insecure-a^i57h%uoa*v*c_$v5)7jeec7g!mgt%v^-
        o!*0qfu#^z_30%hh
```

Generating a New Key

You may have noticed that the secret key starts with `django-insecure-`. This is a newer feature in Django. It is a reminder that we should use a different key in production.

Python has the built-in `secrets` module that can generate a new secret for us. To simplify matters, we are going to write a few lines of Python code in the terminal. We can do this using the `-c` flag. The `-c` flag for Python is the

command flag. This means we can give commands to Python to execute in the terminal. Import the `secrets` module and print a new secret to use.

```
python -c "import secrets; print(secrets.token_urlsafe())"
```

Python should print output that looks something like string below. Again, your value will be different.

```
ljUD7aZJh5SQekDI8hL0qChr_oq-AYHJS88kvc2-ZBA
```

Copy this value and paste it into the `.env` file as the new value for `SECRET_KEY`.

```
DEBUG=True
SECRET_KEY=ljUD7aZJh5SQekDI8hL0qChr_oq-AYHJS88kvc2-ZBA
```

Go back to the terminal and deactivate the `django` virtual environment.

```
conda deactivate
```

Database

PostgreSQL

Start by activating the `django` virtual environment.

```
conda activate django
```

We are currently using a SQLite database. This is a convenient solution as it is the Django default and easy to use. However, it is not a production-grade database. For that, we are going to use PostgreSQL. We are not going to install PostgreSQL locally. Instead we are going to install that on Heroku, the service hosting our website. We do need to prepare our Django project to use SQLite locally and PostgreSQL in production.

Just as before, this calls for an environment variable. We are going to use `dj_db_url` method, which is a Django-specific method from the `environs` package. This is why we installed `environs[django]` instead of just `environs`. Our current database setting looks like this:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

The `"default"` key has another dictionary as its value. We are going to pass a single environment variable as the value for `"default"`. Open the `settings.py` file and change the value of `DATABASES` so it looks like this:

```
DATABASES = {
    'default': env.dj_db_url('DATABASE_URL')
}
```

The `dj_db_url` method will parse the environment variable such that everything works as expected. We do not have to worry about having one environment variable for `"ENGINE"` and another for `"NAME"`. All of this is taken care of by a single environment variable and the `dj_db_url` method.

Open the terminal and then run Black on the settings file.

```
black capital_cities/settings.py
```

Now open the `.env` file so we can add the `DATABASE_URL` variable to the list. Notice that we are setting `SQLite` as the database, PostgreSQL is not mentioned. When we migrate our project to Heroku, they automatically create an environment variable named `DATABASE_URL`. Since we will configure Heroku to run PostgreSQL, the environment variable will contain the information our project needs to use PostgreSQL. **Remember**, your secret key is different from the one below.

```
DEBUG=True
SECRET_KEY=imDnfLXy-8Y-YozfJmP2Rw_81YA_qx1XKl5FeY0mXyY
DATABASE_URL=sqlite:///db.sqlite3
```

Psycopg

Django has an Object Relational Mapper (ORM) that translates the code for our models into the database backend specified in our settings file. While the ORM works most of the time, there are a few edge cases where problems can arise. To avoid this, we are going to install Psycopg, a PostgreSQL adapter for Python.

Open the terminal and install `psycopg2` from the Conda Forge channel.

```
conda install -c conda-forge psycopg2 -y
```

That is all we need to do regarding Psycopg. One thing to keep in mind that it is a good practice to run PostgreSQL in your local production environment to help you catch any edge cases with the database. Our simple project should work just fine however.

Deactivate the `django` virtual environment.

```
conda deactivate
```