Learning Objectives: Preparing for Production

Learners will be able to...

- **Create a Django Project**
- **Add a form to post information to the database**
- **Test the form**

info

Make Sure You Know

You should be comfortable with Python, specifically classes and inheritance.

Limitations

This project adds a form to post information to the database. Features such as editing or deleting information with additional forms is not covered. This project does not use authentication; any user can add whatever they want to the database.

Create a Django Project

Capital Cities Project

In this project, we are going to create a simple website that has a list of capital cities by country. The big difference with this project is the inclusion of a form so you can add cities to the list. This project is simple in nature, and there are many things missing from the website (like authentication, etc.). However, we are going to use this project and learn how to prepare Django for a production environment.

Installing Django

Start by creating the `django` virtual environment.

```
conda create --name django -y
```

Then activate the newly created environment.

```
conda activate django
```

Install Django as well as Black. Black is a Python code formatter. The project describes itself as being uncompromising, which means it is going to make changes to the code that might seem unusual. For instance, Black defaults to double quotes over single quotes. You can read more about the Black code style here. Black is very popular in the Django community, so we will be using this tool to format our code. Both packages are installed from the Conda Forge channel to get their latest versions.

```
conda install -c conda-forge django black -y
```

Starting Our Django Project

Now that we have a virtual environment and Django installed, it is time to create our project. Run the following command to create our Django project. We are going to name it `capital_cities`.

```
django-admin startproject capital_cities .
```

Use the `tree` command to print a graphical representation of our current directory.

```
tree
```

What was once an empty directory is now contains the directory `capital_cities`, which itself contains five files. We also have a `manage.py` file. This is the benefit of using a framework. It knows that you need these directories and files for a proper Django project, so it creates them for you.

```
.
├── capital_cities
│   ├── asgi.py
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

Each file in the `capital_cities` directory serves a specific purpose:

- `asgi.py` - this is an optional file used with an asynchronous server gateway interface.
- `__init__.py`- this indicates the directory is part of a Python package.
- `settings.py` - this file controls the settings for our Django project.
- `urls.py` - this file tells Django which page to load when a user visits our site.
- `wsgi.py` - the web server gateway interface is used when we host our site on a production server.

The `manage.py` file is not technically part of the Django project since it is outside the `capital_cities` directory. However, this file is very important. We will use it when we need to make migrations to the database, run our development server, etc.

If we activated the `django` virtual environment at the beginning of a page, we want to get into the habit of deactivating it at the end of a page.

```
conda deactivate
```

Codio-Specific Settings

Running Django On Codio

info

Django and Codio

The changes on this page are done to allow Django to run on the Codio platform. By default, Django has strict security features that keep Codio from embedding Django.

The code samples below **only** apply for Codio. You **would not** make these changes when running Django outside of Codio. If this module is about production-ready code, why make these changes? We want to verify that the site is working as expected first. We will undo these changes later on.

Before setting up Django to run on Codio, you first need to import the `os` module.

```
import os
```

The two biggest obstacles in getting Django to run inside Codio are recognizing the unique host name for each Codio project and cookies. The changes below will tell Django the exact host name of the Codio project and alter how Django handles cookies. Make sure your settings match the ones below.

```
ALLOWED_HOSTS = ['*']
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'
```

Scroll down a bit and look for the definition of the variable `MIDDLEWARE`. Comment out the two lines that refer to `csrf`. CSRF stands for cross-site request forgery, which is a way for a malicious actor to force a user to perform an unintended action. Django normally has CSRF protections in place, but we need to disable them. There are serious security implications to doing so, however your project on Codio is not publicly available on the internet. Comment out the two lines in the `MIDDLEWARE` setting as shown below.

```
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',

]
```

Starting the Server

Once you finish making the changes to the `settings.py` file, we want to enforce the Black style guide. Start by opening the terminal with the link below. Then activate the `django` virtual environment.

```
conda activate django
```

Now run Black on the `settings.py` file. If you were to click on the tab for the settings file, you would see the newly formatted settings.

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

Use the `manage.py` file to run the development server. We are going to specify `0.0.0.0` for the address of localhost and port 8000.

```
python manage.py runserver 0.0.0.0:8000
```

Finally, open the preview of Django running on the development server with the link below. You should see the default success message from Django. We see this message because all we did was start a Django project and change a few settings. There is no website to show, so we get the success message instead.

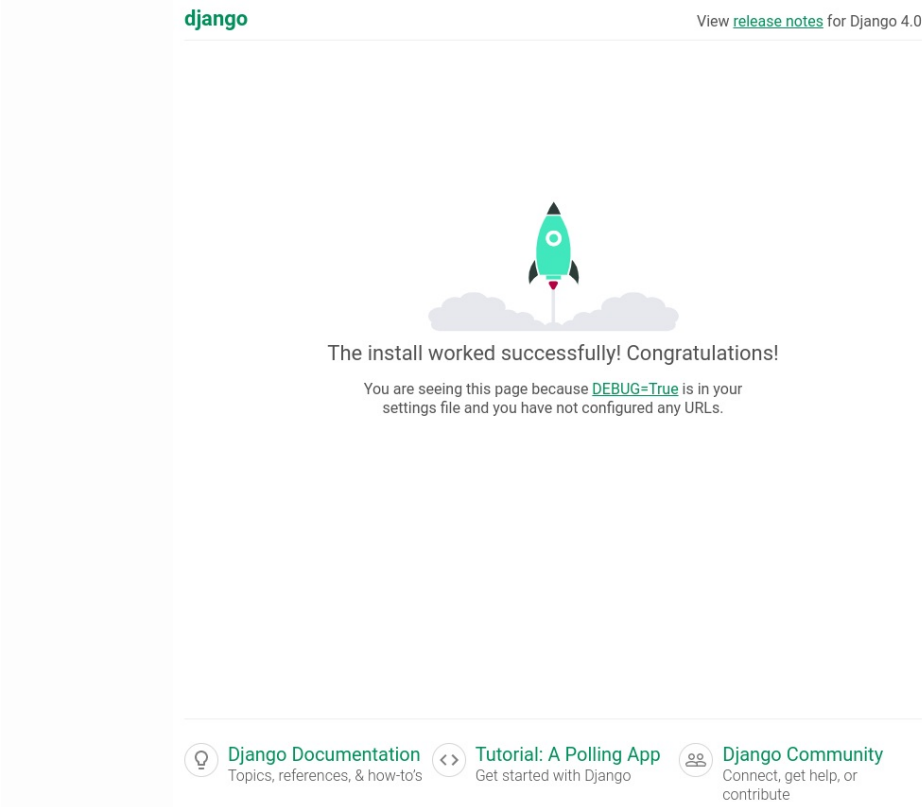


Image depicts the success message from Django that the install is working.

Reminder: these changes only apply to working with Django on Codio. **Do not** make these changes to a project you plan on making available on the internet.

Open the terminal and stop the Django server by pressing `Ctrl + C`. Then deactivate the virtual environment.

```
conda deactivate
```

Django Apps

Creating a Django App

We have the skeleton of a Django project. A project is comprised of several Django apps. Each app should perform a specific function of the overall project. Combine all the apps together, and you have a Django project.

Start by activating the `django` virtual environment.

```
conda activate django
```

To add an app to a Django project, use the `manage.py` file with the `startapp` argument. In addition, give this new app a name.

```
python manage.py startapp capitals
```

Next, let's create our initial database with the `migrate` command. We have not yet declared our model (which controls the database's structure), so the database will use Django's default settings.

```
python manage.py migrate
```

If you enter the `tree` command in the terminal, you will see the new structure of our Django project. The `db.sqlite3` is our database, which is new. In addition to the `capital_cities` directory, we now have the `capitals` directory. This new directory is our app.

```
.
├── capital_cities
│   ├── asgi.py
│   ├── __init__.py
│   ├── __pycache__
│   │   ├── __init__.cpython-310.pyc
│   │   ├── settings.cpython-310.pyc
│   │   ├── urls.cpython-310.pyc
│   │   └── wsgi.cpython-310.pyc
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── capitals
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── db.sqlite3
└── manage.py
```

Django automatically creates any files needed by the app. Each file has a specific purpose:

- `admin.py` - handles configuration of the built-in Django Admin feature
- `apps.py` - handles configuration for the app itself
- `migrations` - this directory keeps track of changes to models so the database stays in sync
- `models.py` - handles the definition of database models
- `tests.py` - handles any tests specific to the app
- `views.py` - handles any HTTP requests and responses for the app

Adding the App

Django makes it really easy to add an app to a project. The problem, however, is that the project "does not know" about the newly created app. We need to add our `capitals` app to the list of installed apps for the Django project. Open `settings.py` (the project settings) with the link below. Then add our languages app to the end of the list.

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    'capitals.apps.CapitalsConfig',
]
```

`CapitalsConfig` is a class in the `apps.py`file inside the `reviews` directory (our new Django app). Django created this file and its contents during the app creation process. We do not need to make any changes to this file. We just need to make our Django project aware of it.

Open the terminal and run the Black formatter to make sure our code has a consistent style.

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

Deactivate the virtual environment.

```
conda deactivate
```

Model

Movie Fields

Start by activating the `django` virtual environment.

```
conda activate django
```

Open the `models.py` file.

Our database needs a model, which represents the information stored. Since this is such a simple project, we only need two fields:

- City - short-form text with a max length of 100 characters
- Country - short-form text with a max length of 100 characters

Since each field is short-form text, we are going to use a `CharField`. We are also going to limit the length of the text to 100 characters.

```
from django.db import models

class Capital(models.Model):
    city = models.CharField(max_length=100)
    country = models.CharField(max_length=100)
```

We also want to create the `__str__` method to provide a human-readable version of the model in the Django Admin. Use an f-string to return the city and country values separated by a comma.

```
from django.db import models

class Capital(models.Model):
    city = models.CharField(max_length=100)
    country = models.CharField(max_length=100)

    def __str__(self):
        return f'{self.city}, {self.country}'
```

The final thing we need to do is create the canonical URL for our model. Basically, Django will redirect our website to the homepage after entering a new city and country. Start by importing the `reverse` function from

`django.urls`. Then create the `get_absolute_url` method that returns the homepage.

```
from django.db import models
from django.urls import reverse

class Capital(models.Model):
    city = models.CharField(max_length=100)
    country = models.CharField(max_length=100)

    def __str__(self):
        return f'{self.city}, {self.country}'

    def get_absolute_url(self):
        return reverse("home")
```

Open the terminal and run the Black formatter to make sure our code has a consistent style.

```
black capitals/models.py
```

Activating the Database

Because our website is so simple, this is all we need for our model. Before we use it, we need to activate the new model. This is a two-step process and should performed whenever the model changes. The first step is to create a migrations file. These files keep track of all of the changes made to the model over time.

```
python manage.py makemigrations capitals
```

▼ **Did you notice?**

> We specified the `capitals` app with the `makemigrations` command. This is optional. If you do not specify the app, Django will make migration files for all apps in the project.

Next, use the `migrate` command to sync the database to the model. We have already seen this command once before when we first activated the database with the Django defaults.

```
python manage.py migrate
```

Our database is ready to go. Next, we'll take a look at how to view and modify the database using the powerful, built-in tool called Django Admin.

Deactivate the virtual environment.

```
conda deactivate
```

Views

List View

Start by activating the `django` virtual environment.

```
conda activate django
```

This project will list out the capital cities and their countries from the database. That means we need to use a `ListView`. This view is a list of every record in the database. This makes it easy to iterate over the list and place the information in an HTML document.

Open the `views.py` file from the Django app. Import the `ListView` generic class-based view. We also need to import the `Capital` model used in our database. Name the view `HomePageView` and inherit from `ListView`.

```
from django.views.generic import ListView
from .models import Review

class HomePageView(ListView):
```

Create the variable `template_name` and assign it an HTML file. In this case, the file is called `home.html`. In addition, we need to create the variable `model` and assign it our model `Capital`.

```
from django.views.generic import ListView
from .models import Capital

class HomePageView(ListView):
    template_name = 'home.html'
    model = Capital
```

Create View

We also need to make a view for adding a new capital city to the list. This is going to use a form where a user can enter a new city and country into the database of capital cities. To do this, we need to import the generic class `CreateView`.

```
from django.views.generic import ListView
from django.views.generic.edit import CreateView
from .models import Capital
```

Next create the `NewCapitalView` class and inherit from `CreateView`. This view uses the `capital_new.html` template and it also uses the same `Capital` model.

```
class NewCapitalView(CreateView):
    template_name = 'capital_new.html'
    model = Capital
    fields = ['city', 'country']
```

The `fields` attribute exposes the database fields we want in the form. Note, you can expose as many or as few fields as you want. Our simple database only has two fields, and we want to expose them both.

▼ **Code**

> The code for the `views.py` page should look like this:
>
> ```
> from django.views.generic import ListView
> from django.views.generic.edit import CreateView
> from .models import Capital
>
> class HomePageView(ListView):
>     template_name = 'home.html'
>     model = Capital
>
> class NewCapitalView(CreateView):
>     template_name = 'capital_new.html'
>     model = Capital
>     fields = ['city', 'country']
> ```

Open the terminal and run the Black formatter to make sure our code has a consistent style.

```
black capitals/views.py
```

Deactivate the virtual environment.

```
conda deactivate
```

Templates

Templates

Start by activating the `django` virtual environment.

```
conda activate django
```

We are going to store all our templates in a single `templates` directory. In the terminal, enter the following command to create this directory.

```
mkdir templates
```

We need to update the `settings.py` file since we want to use our own directory for the templates. Look for the variable `TEMPLATES` which should be around line 65. Look for the key `"DIRS"`, whose value is an empty list. Update this value to the one below.

```
        "DIRS": [BASE_DIR / 'templates'],
```

Open the terminal and run the Black formatter to make sure our code has a consistent style.

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

We are going to start with a `base.html` template that will serve as a parent for the list of capital cities and the form for adding a new capital. Create the `base.html` file as well as the `home.html` and `capital_new.html` files inside of the `templates` directory with the command below.

```
touch templates/base.html templates/home.html
        templates/capital_new.html
```

Let's start with the `base.html` template. We are going to create a header to be used on all pages and then create a content block that the children templates can override. The header has the name of the site and a link to add a new capital to the list.

```
<html>
  <head>
    <title>Capital Cities</title>
  </head>
  <body>
    <div>
      <header>
        <div>
          <h1><a href="{% url 'home' %}">Capital Cities</a></h1>
        </div>
        <div>
          <a href="{% url 'capital_new' %}">+ New Capital</a>
        </div>
      </header>
      {% block content %}
      {% endblock content %}
    </div>
  </body>
</html>
```

Open `home.html`. Inherit from the parent template and override the block named `content`. We want to display a sentence that has the capital and country for each record in the database. Django has a `for` template tag that iterates over a list. We are going to use `capital_list` as the list. This is a list of every record in our database. The name of the list is derived from the name of the model (all lowercase) followed by `_list`. We are able to do this thanks to the `ListView` generic class-based view. The for loop function almost identically to a traditional for look in Python. When using the loop variable, be sure to surround with `{{ }}` to differentiate it from HTML. We also need to end the `for` template tag.

```
{% extends "base.html" %}
{% block content %}
{% for capital in capital_list %}
<h3>{{ capital.city }} is the capital of {{ capital.country }}
        </h3>
{% endfor %}
{% endblock content %}
```

Open `capital_new.html`. Inherit from the parent template and override the block named `content`. Create a form with `"post"` as the method. We do this because we are sending data to the website. We also include the `{% csrf_token %}` to prevent cross-site request forgeries. The form template tag (`{{ for.as_p }}`) means that each field in our form is rendered in its own paragraph tag. Finally, we have a save button.

```
{% extends "base.html" %}
{% block content %}
<h2>Add a New Capital to the List</h2>
<form action="" method="post">{% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="Save">
</form>
{% endblock content %}
```

Since we did not edit any Python code, we do not need to run the Black formatter. Unfortunately, we cannot yet look at our website until we address the URL patterns.

Deactivate the virtual environment.

```
conda deactivate
```

URL Patterns

Creating the Patterns

Start by activating the `django` virtual environment.

```
conda activate django
```

The first URL pattern will be for the Django project. We want to include the `urls.py` file in the `reviews` app. Add this to the list after the URL pattern for the Django Admin.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("", include('capitals.urls')),
]
```

Open the terminal and run the Black formatter to make sure our code has a consistent style.

```
black capital_cities/urls.py
```

Currently, the `urls.py` file for the `capitals` app does not exist. Enter the following command to create it. Then click the link to open the file.

```
touch capitals/urls.py
```

Import `path` from `django.urls` and import our `HomePageView` from the `views.py` file. Create a pattern for the root of our website which renders `HomePageView`. Because this is a class-based view, we need to use the `as_view()` method. Give this pattern the name `home`.

```
from django.urls import path
from .views import HomePageView, NewCapitalView

urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
    path('new/', NewCapitalView.as_view(), name='capital_new')
]
```

Open the terminal and run the Black formatter to make sure our code has a consistent style.

```
black capitals/urls.py
```

Testing our Site

We are finally ready to display information from the database to our Django website. Run the dev server and open the website. We should only see the title and a link to add a new captial. There is no information in the database, so there is no list of capitals.

```
python manage.py runserver 0.0.0.0:8000
```

This image depicts the homepage. There is the title “Capital Cities”. Below that is a link that says “+ New Capital”. There is nothing else on the page.

Click on the link to add a new capital. You should see a form that allows you to enter a city, a country, and the save this by clicking a button. Enter `Paris` and `France` for the respective fields and click the save button.

The image depicts the

page to add a new capital. There is a form with fields for “City” and “Country” as well as a save button.

Django automatically redirects you to the homepage. This time, however, you should see the text:

```
Paris is the capital of France
```

The image depicts the homepage with the sentence “Paris is the capital of France”.

challenge

Try this variation:

- Add more capitals to the database:
  - `Ottawa` and `Canada`
  - `Cairo` and `Egypt`
  - `Rome` and `Italy`

Switch back to the terminal and stop the dev server with `Ctrl+C` on the keyboard. Then deactivate the virtual environment.

```
conda deactivate
```

Bootstrap

Adding Bootstrap

Start by activating the `django` virtual environment.

```
conda activate django
```

Instead of using traditional CSS to style our Django project, we are going to use the Bootstrap CSS framework. Like Django, Bootstrap abstract away many of the details of styling. Before we can use it, we first need to install the framework in our `base.html` file.

In the `<head>` tag, add the link to content delivery network (CDN) that hosts the Bootstrap framework. Also add a `<meta>` tag to indicate that our site is using the UTF-8 character set.

```
<head>
  <title>Capital Cities</title>
  <meta charset="utf-8">
  <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap
      .min.css" rel="stylesheet" integrity="sha384-
      1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
      crossorigin="anonymous">


</head>
```

Bootstrap relies on JavaScript to properly work. Because of this, we need to add the Bootstrap `<script>` tag in the body after the block template tag.

```
<body>
  <div>
    <header>
      <div>
        <h1><a href="{% url 'home' %}">Capital Cities</a></h1>
      </div>
      <div>
        <a href="{% url 'capital_new' %}">+ New Capital</a>
      </div>
    </header>
    {% block content %}
    {% endblock content %}
    <script
      src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle
      js" integrity="sha384-
      ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p"
      crossorigin="anonymous"></script>


  </div>
</body>
```

Styling the Site

We are going to start by styling the `<div>` inside the body as a container for our content. The container will provide some margins on the left and right. We are also going to add some a margin to the top and bottom of the container. Give the `<header>` a bottom border. We also want to use flexbox to position the `<div>` elements in the center (vertically speaking) of the header.

```
    <div class="container my-3">
      <header class="border-bottom border-secondary pb-1 d-flex
        align-items-center">
```

Next we are going to style the two `<div>` tags in the header. The first `<div>` should take up 75% of the width (`col 9`) and use a display heading that is unique to Bootstrap. The second `<div>` should take the column class. The link should now be a blue button with an outline instead of a solid color.

```
        <div class="col-9 display-3">
          <a href="{% url 'home' %}">Capital Cities</a>
        </div>
        <div class="col">
            <a class="btn btn-outline-primary" href="{% url
        'capital_new' %}">+ New Capital</a>
        </div>
```

Next, we are going to style the `home.html` template. All we need to do is provide a bit of space between the list of capital cities and the border in the header and change the typography of the list. Add a `<div>` around name of the city and country. Add the `my-2` class, which adds a small margin at the top and bottom of the list. Use the `display-6` class to style the text.

```
<div class="my-2">
  <h3 class="display-6">{{ capital.city }} is the capital of {{
        capital.country }}</h3>
</div>
```

Update the `capital_new.html` template so that a `<div>` surrounds the title and form. Again, we are going to provide a small margin on the top and bottom. Change the title to use the `display-6` class. Finally, use a blue button with an outline to match the other button on the page.

```
<div class="my-2">
  <h2 class="display-6">Add a New Capital to the List</h2>
  <form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" class="btn btn-outline-
        primary">
  </form>
</div>
```

Viewing the Changes

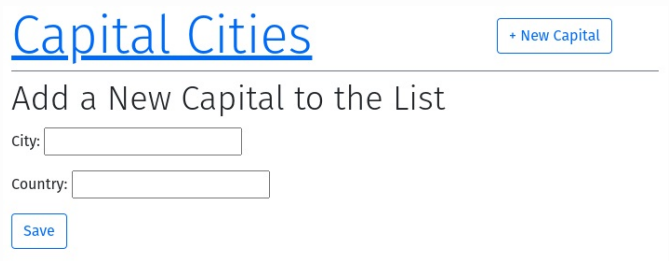
Open the terminal and run the dev server. Then open the website. You should see the styling added to our site.

```
python manage.py runserver 0.0.0.0:8000
```

The `home.html` template should now render like the image below.

The image depicts the style homepage. The title is blue and larger. There is a blue button to the right to add a new capital. The header has a border on the bottom. Finally, the text for the capitals is taller and skinnier.

Click on the button to add a new capital to the list. Your page should now look like this:



The image depicts the page with the form to add a new capital city. You have the same revised header as before. The text to add a new city and country is taller and skinnier.

warning

No Authentication

This project uses a form without any authorization. That means anyone who visits your site can enter whatever information they want in the form and it will appear on the site. This is not a safe. A better solution would be to grant access to the form to authorized users. This project is only for demonstration.

Switch back to the terminal and stop the dev server with `Ctrl+C` on the keyboard. Then deactivate the virtual environment.

```
conda deactivate
```

Testing

Testing the Homepage

Start by activating the `django` virtual environment.

```
conda activate django
```

The last step in our Django project is writing tests. We are going to test the homepage, the form, and the database. Open the test file for the `capitals` Django app.

Begin the test by importing the `Capital` model and the `reverse` function. In addition, create the `CapitalTests` class which inherits from `TestCase`. We inherit from `TestCase` since we are testing the database.

```
from django.test import TestCase
from .models import Capital
from django.urls import reverse

class CapitalTests(TestCase):
```

Create the `test_home_page` method. This method will fetch the template associated with the name `home`. It then checks that the status code is `200`, that the template used is `home.html`, and that the page contains the text `Capital Cities`.

```
    def test_home_page(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')
        self.assertContains(response, 'Capital Cities')
```

Testing the form

Many Django tests create a `response` variable that uses `self.client.get`. The `post` method is used by forms to send information to the database. We need to use `self.client.post` when testing the form. Start by creating the

`test_capital_createview` method. Create the `response` variable by posting a JSON object that has `New Delhi` as the city and `'India'` as the country. The JSON should be posted to the template with the name `'capital_new'`.

```
    def test_capital_createview(self):
        response = self.client.post(
            reverse('capital_new'),
            {
              'city' : 'New Delhi',
              'country' : 'India'
            }
        )
        self.assertEqual(response.status_code, 302)
        self.assertEqual(Capital.objects.last().city, 'New
        Delhi')
        self.assertEqual(Capital.objects.last().country,
        'India')
```

Assert that the page has a `302` redirect status code, that the city field for the last object is `'New Delhi'`, and that country field for the last object is `'India'`.

Testing the Database

Django creates a special testing database used for the tests. Afterwards, it is deleted. Create the `setUpTestData` as a class method. Create a `Capital` object that has `'Riga'` as the city and `'Latvia'` as the country.

```
    @classmethod
    def setUpTestData(cls):
        cls.capital = Capital.objects.create(city='Riga',
        country='Latvia')
```

Once the dummy database is complete, we can run tests against it. Assert that the value of the `city` field is `Riga` and that the value of the `country` field is `Latvia`.

```
    def test_capital_city(self):
        self.assertEqual(self.capital.city, 'Riga')

    def test_capital_country(self):
        self.assertEqual(self.capital.country, 'Latvia')
```

Running the Tests

Open the terminal and run the Black formatter to make sure our code has a consistent style.

```
black capitals/tests.py
```

Once all the tests are done, run the test file with the command below.

```
python manage.py test
```

If all tests pass, you should see the following output:

```
Found 4 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
----------------------------------------------------------------------
------
Ran 4 tests in 0.023s

OK
Destroying test database for alias 'default'...
```

Deactivate the virtual environment.

```
conda deactivate
```