Learning Objectives: Testing

Learners will be able to...

- **Differentiate the database in the Django project and the testing database**
- **Identify what must be imported to test a database**
- **Create a testing database**
- **Properly name a unit test**
- **Identify when to combine tests into a single test case**

info

Make Sure You Know

You are comfortable with basic commands in the terminal and creating classes in Python.

Limitations

This assignment focuses on testing the database but also repeats the testing principles from the previous Django project.

Testing the Database

Testing vs Local Database

This project focuses on integrating a database into our Django project. We are going to do the same thing with our testing. In the previous project, we tested the web pages (name, pattern, template, etc.) and then tested content in the HTML pages. We going to do something similar, but with an important difference.

Django creates a separate database for testing purposes. No matter what we do to the testing database (add, remove, modify, etc.), we will not affect the important information stored in the local database. Once the testing is done, Django deletes the testing database.

Setting Up the Testing Database

Start by activating the `django` virtual environment.

```
conda activate django
```

Open the testing from the Django app.

You should already see the import statement for `TestCase`. We need this to test our database. In addition, we need to import our `Review` model. Create the `ReviewTests` class and inherit from `TestCase`.

```
from django.test import TestCase
from .models import Review

class ReviewTests(TestCase):
```

Above, we talked about Django's ability to create a disposable testing database. This is done with the `setUpTestData` method. This should be a class method, so be sure to include the `@classmethod` decorator.

```
class ReviewTests(TestCase):
    @classmethod
    def setUpTestData(cls):
```

Name the test database `cls.review` and assign it the value of a `Review` model. Because our model inherits from `model`, we can call the `objects.create()` method. Pass this method named parameters and the associated value for each field in the database. Django will create a testing database with the provided information. **Note**, we did not include a numerical value for `stars`. Remember, the user sees a number from 1 to 4 as the choice, but a string of 1 to 4 characters is stored in the database.

```
class ReviewTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.review = Review.objects.create(title='Citizen Kane',
        director='Orson Welles',
        actors='Orson Welles and Joseph Cotten',
        review='One of the greatest films of all time, a must
        see. A true, timeless masterpiece.',
        year=1941,
        stars='ssss')
```

▼ **Did you notice?**

> In the `setUpTestData` method, `review` is prefaced by `cls`. When we reference `review` in the testing methods we will use `self.review` instead of `cls.review`. This is a stylistic change set out in PEP-8, the official style guide for Python, which states that `cls` should be used in class methods and `self` should be used in instance methods. Functionally, there is no difference between the two.

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

```
black reviews/tests.py
```

It is important to note that no testing should happen with the `setUpTestData` method. In fact, if we were to run our tests right now, Django would say that zero tests ran. The `manage.py test`command only executes methods that start with the word `test`.

Deactivate the virtual environment.

```
conda deactivate
```

Unit Tests

Testing the Database

Start by activating the `django` virtual environment.

```
conda activate django
```

In the last project, we used a the `get()` method to get a response from the client. When testing the database you need to querying the one we created in the `setUpTestData()` method. So the title field would be referenced as `self.review.title`. Just as before, we will use `assertEqual` to check the value in the database against the expected result. The unit test below checks to see if the title in the database is `"Citizen Kane"`.

```
    def test_example_title(self):
        self.assertEqual(self.review.title, 'Citizen Kane')
```

Follow the same pattern as the test above to create tests for the `director`, `actors`, `review`, and `stars` fields.

```
    def test_review_director(self):
        self.assertEqual(self.review.director, 'Orson Welles')


    def test_review_actors(self):
        self.assertEqual(self.review.actors, 'Orson Welles and
        Joseph Cotten')


    def test_review_review(self):
        self.assertEqual(self.review.review, 'One of the
        greatest films of all time, a must see. A true, timeless
        masterpiece.')


    def test_review_year(self):
        self.assertEqual(self.review.year, 1941)


    def test_review_stars(self):
        self.assertEqual(self.review.stars, 'ssss')
```

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

```
black reviews/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 6 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
......
----------------------------------------------------------------------
------
Ran 6 tests in 0.004s

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

Testing the Website

We still need to test that the homepage for our website is functioning as expected. These tests are the same from the previous example, though we only have one webpage to test. But before we can do that, we need to import the `reverse` function from `django.urls`.

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

Add the following tests to the `ReviewTests` class. They check that a URL exists at `/`, that a URL is available for the name `"home"`, that `home.html` is the template, and that specific text appears on the page.

```
    def test_url_pattern(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

    def test_url_name(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)

    def test_template_name(self):
        response = self.client.get(reverse('home'))
        self.assertTemplateUsed(response, 'home.html')

    def test_homepage_content(self):
        response = self.client.get(reverse('home'))
        self.assertContains(response, 'One of the greatest films
        of all time, a must see. A true, timeless masterpiece.')
```

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

```
black reviews/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 10 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..........
----------------------------------------------------------------------
------
Ran 10 tests in 0.026s

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

▼ **Code**

> Your code should look like this:
>
> ```
> from django.test import TestCase
> ```

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


class ReviewTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.review = Review.objects.create(
            title="Citizen Kane",
            director="Orson Welles",
            actors="Orson Welles and Joseph Cotten",
            review="One of the greatest films of all time,
        a must see. A true, timeless masterpiece.",
            year=1941,
            stars="ssss",
        )

    def test_example_title(self):
        self.assertEqual(self.review.title, "Citizen Kane")

    def test_example_director(self):
        self.assertEqual(self.review.director, "Orson
        Welles")

    def test_example_actors(self):
        self.assertEqual(self.review.actors, "Orson Welles
        and Joseph Cotten")

    def test_example_review(self):
        self.assertEqual(
            self.review.review,
            "One of the greatest films of all time, a must
        see. A true, timeless masterpiece.",
        )

    def test_example_year(self):
        self.assertEqual(self.review.year, 1941)

    def test_example_stars(self):
        self.assertEqual(self.review.stars, "ssss")

    def test_url_pattern(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_url_name(self):
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)

    def test_template_name(self):
```

```
        response = self.client.get(reverse("home"))
        self.assertTemplateUsed(response, "home.html")

    def test_homepage_content(self):
        response = self.client.get(reverse("home"))
        self.assertContains(
            response,
            "One of the greatest films of all time, a must
        see. A true, timeless masterpiece.",
        )
```

Combining Tests

Combining Homepage Tests

Start by activating the `django` virtual environment.

```
conda activate django
```

Let's take a look at the last three tests of our test file.

```
    def test_url_name(self):
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)

    def test_template_name(self):
        response = self.client.get(reverse("home"))
        self.assertTemplateUsed(response, "home.html")

    def test_homepage_content(self):
        response = self.client.get(reverse("home"))
        self.assertContains(
            response,
            "One of the greatest films of all time, a must see.
A true, timeless masterpiece.",
        )
```

You will notice that each test starts with the same line of code:

```
response = self.client.get(reverse("home"))
```

A general premise of programming is that you should not repeat yourself. With that in mind, let's create a single code test with all three assert statements. This way we only have to make a single `get` request as opposed to three. Make the following changes to your `tests.py` file.

```
    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,
            'One of the greatest films of all time, a must see.
        A true, timeless masterpiece.',
        )
```

▼ **Did you notice?**

> We did not include the `test_url_pattern` test in our combined test. One, `test_rul_pattern` uses a different kind of `get` request. Two, this test is not really testing our homepage. It is testing that a page loads, but it can be any HTML page. The tests that we combined are all specific to `home.html`. It does not make sense to combine `test_url_pattern` with the other tests.

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

```
black reviews/tests.py
```

Run the tests one more time to be sure that all of them still pass.

```
python manage.py test
```

▼ **Code**

> Your code should look like this:
>
> ```
> from django.test import TestCase
> from .models import Review
> from django.urls import reverse
>
>
> class ReviewTests(TestCase):
>     @classmethod
>     def setUpTestData(cls):
>         cls.review = Review.objects.create(
>             title="Citizen Kane",
>             director="Orson Welles",
>             actors="Orson Welles and Joseph Cotten",
> ```

```
            review="One of the greatest films of all time,
        a must see. A true, timeless masterpiece.",
            year=1941,
            stars="ssss",
        )

    def test_example_title(self):
        self.assertEqual(self.review.title, "Citizen Kane")

    def test_example_director(self):
        self.assertEqual(self.review.director, "Orson
        Welles")

    def test_example_actors(self):
        self.assertEqual(self.review.actors, "Orson Welles
        and Joseph Cotten")

    def test_example_review(self):
        self.assertEqual(
            self.review.review,
            "One of the greatest films of all time, a must
        see. A true, timeless masterpiece.",
        )

    def test_example_year(self):
        self.assertEqual(self.review.year, 1941)

    def test_example_stars(self):
        self.assertEqual(self.review.stars, "ssss")

    def test_url_pattern(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    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,
            "One of the greatest films of all time, a must
        see. A true, timeless masterpiece.",
        )
```

Combining Database Tests

Just as it makes sense to combine all of the homepage assert statements into a single test, it seemingly makes sense to combine all of the database assert statements into a single test as well. Assume we had a single test case

called `test_database`. Now assume there is a problem with the `title` field. We would see the error message that `test_database` failed. If there is an error with the `review` field, we get the same error message. Getting the same error message for two totally different problems does not help us more efficiently debug the problem.

It makes more sense to have the separate test cases because the name of the failed test helps us debug the problem. So why combine the tests for the homepage? What you lose in clarity, you gain in efficiency. Combining these tests reduces the number of `get` requests made. You can make a better argument for combining the homepage tests than the database tests. That said, if you prefer clarity above all else, you do not have to combine any tests. However, combining tests is common among developers, but you should not combine all of your tests for debugging reasons.