Lab 9 Part 7
Unit Testing

Tests for the Accounts app

The Python programming language contains its own unit testing framework and Django's automated testing framework extends this with multiple additions into a web context. It is important to note that not everything needs to be tested. For example, any built-in Django features already contain tests in the source code. If we were using the default User model in our project, we would not need to test it. But since we have created a CustomUser model we should test it.

To write unit tests in Django we use **TestCase** which is, itself, an extension of Python's TestCase. Our **accounts** app already contains a **tests.py** file which is automatically added when the startapp command is used. Currently it is empty.

Each method must be prefaced with test in order to be run by the Django test suite. It is also recommended to be descriptive with your unit test names since Django projects can have hundreds if not thousands of tests.

Open **accounts/tests.py,** delete the top line of code and copy-paste the following code:

```
from django.test import TestCase
from django.contrib.auth import get_user_model

class CustomUserTestCase(TestCase):
    def setUp(self):
        # Create a custom user for testing
        self.user = get_user_model().objects.create_user(
            username='testuser',
            email='testuser@t.com',
            password='tpassword',
            age=23
        )

    def test_create_user(self):
        # Test if a user is created properly
        self.assertEqual(self.user.username, 'testuser')
        self.assertEqual(self.user.email, 'testuser@t.com')
        self.assertEqual(self.user.age,23)
```

```
        self.assertTrue(self.user.check_password('tpassword'))


    def test_superuser_creation(self):
        # Test creating a superuser
        self.superuser = get_user_model().objects.create_superuser(
            username='suser',
            email='suser@s.com',
            password='spassword'
        )
        self.assertEqual(self.superuser.username, 'suser')
        self.assertEqual(self.superuser.email, 'suser@s.com')
        self.assertTrue(self.superuser.check_password('spassword'))
        self.assertTrue(self.superuser.is_staff)
        self.assertTrue(self.superuser.is_superuser)
```

At the top we have imported both get_user_model and TestCase before creating a **CustomUserTests** class. Within it are two separate tests. **test_create_user** confirms that a new user can be created. First, we set our user model to the variable User and then create a user via the manager method **create_user** which does the actual work of creating a new user with the proper permissions. The **User** model has a custom manager that has a helper method **create_user** which creates, saves, and returns a User.

For **test_superuser_creation** we follow a similar pattern, but reference **create_superuser** instead of **create_user**. The difference between the two users is that a superuser should have both **is_staff** and **is_superuser** set to True.

All test methods must take **self** as an argument, where self is a reference to the **TestCase** object. **TestCase**, which we inherit to create our class, provides assertion methods to evaluate booleans. A self.assertSomething() call passes if the values passed as arguments are consistent with the assertion and fails otherwise. A test method passes only if every assertion in the method passes.

You can run this test by typing the following at the command line:

python manage.py test accounts

After you run the test, you will see the following output indicating that both the tests passed:

```
Found 2 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.364s

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

Test Profile Model

Next we will test the Profile model along with the __str__ and get_absolute_url functions. Add the following import at the top of accounts/tests.py:

```
from .models import Profile
```

Add the following code to test the Profile model and it's two functions:

```
class ProfileModelTest(TestCase):
    def setUp(self):
        self.user = get_user_model().objects.create_user(
            username='testuser',
            password='testpassword'
        )
        self.profile = Profile.objects.create(
            user=self.user,
            date_of_birth='2001-12-12',
            fav_author = 'Archer'
        )

    def test_profile_creation(self):
        self.assertEqual(self.profile.user, self.user)
        self.assertEqual(self.profile.date_of_birth, '2001-12-12')
        self.assertEqual(self.profile.fav_author, 'Archer')

    def test_profile_str_representation(self):
        expected_str = "testuser"
        self.assertEqual(str(self.profile), expected_str)

    def test_get_absolute_url_method(self):
        # Test the get_absolute_url method of Profile
        expected_url = '/accounts/profile/{}/'.format(self.profile.pk)
        self.assertEqual(self.profile.get_absolute_url(), expected_url)
```

The test_get_absolute_url_method method tests the get_absolute_url method of the Profile model. In this example, we supply the url pattern as specified in our urls.py files in our project.

After you run the test, you will see the following output indicating that the 5 tests have passed:

```
Found 5 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
----------------------------------------------------------------------
Ran 5 tests in 3.329s

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

Tests for the Pages app

For our homepage we can use Django's **SimpleTestCase** which is a special subset of Django's **TestCase** that is designed for webpages that **do not have** a model included.

Open **pages/tests.py**, delete the top line of code and copy-paste the following code:

```python
from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase):
    def test_homepage_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)


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

At the top we import **SimpleTestCase** as well as **reverse** which is useful for testing our URLs. Then we create a class called **HomepageTests** that extends **SimpleTestCase** and within it add a method for each unit test.

The two tests here check that the HTTP status code for the homepage equals 200 which means that it exists. It does not yet tell us anything specific about the contents of the page. For **test_homepage_status_code** we are creating a variable called response that accesses the homepage (/) and then use Python's **assertEqual** to

check that the status code matches 200. A similar pattern exists for **test_homepage_url_name** except that we are calling the URL name of home via the **reverse** method. Recall that we added this to the **pages/urls.py** file as a best practice. Even if we change the actual route of this page in the future, we can still refer to it by the same home URL name.
You can run these by typing the following at the command line:

python manage.py test pages

After you run the test, you will see the following output indicating that both the tests passed:

```
Found 2 test(s).
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.024s

OK
```

Testing Templates

So far, we have tested that the homepage exists, but we should also confirm that it uses the correct template. **SimpleTestCase** comes with a method **assertTemplateUsed** just for this purpose.

Open **pages/tests.py** and copy-paste the following code just below the **test_homepage_url_name** test:

```
    def test_homepage_template(self):
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html')
```

In this test we have created a response variable again and then checked that the template **home.html** is used.
You can run these tests again by typing the following at the command line:

python manage.py test pages

After you run the test, you will see output indicating that the 3 tests passed:

setUp Method

In our last few tests, we load a response variable in each test which is not really adhering to Django's DRY (Don't Repeat Yourself) principle. Since the unit tests are executed top-to-bottom, we can add a **setUp** method that will be run before every test. It will set **self.response** to our homepage so we no longer need to define a response variable for each test. This also means we can remove the **test_homepage_url_name** test since we are using the reverse on home each time in **setUp**.

Replace all the code in **pages/tests.py** with the following:

```
from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase):
    def setUp(self):
        url = reverse('home')
        self.response = self.client.get(url)

    def test_homepage_status_code(self):
        self.assertEqual(self.response.status_code, 200)

    def test_homepage_template(self):
        self.assertTemplateUsed(self.response, 'home.html')
```

Now run the tests again. Because **setUp** is a helper method and does not start with test it will not be considered a unit test in the final tally. So only 2 tests will run.

python manage.py test pages

```
Found 2 test(s).
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.023s

OK
```

Resolve

A final check we can do is that our **HomePageView** "resolves" a given URL path. Django contains the utility function **resolve** for just this purpose. We will need to import it at the top of the file. Our actual test, **test_homepage_url_resolves_homepageview**, checks that the name of the view used to resolve / matches **HomePageView**.

Add the following code to **pages/tests.py**:

```
pages > tests.py > ...
1  from django.test import SimpleTestCase
2  from django.urls import reverse, resolve
3  from .views import HomePageView
4
5  class HomepageTests(SimpleTestCase):
6      def setUp(self):
7          url = reverse('home')
8          self.response = self.client.get(url)
9
10     def test_homepage_status_code(self):
11         self.assertEqual(self.response.status_code, 200)
12
13     def test_homepage_template(self):
14         self.assertTemplateUsed(self.response, 'home.html')
15
16     def test_homepage_url_resolves_homepageview(self):
17         view = resolve('/')
18         self.assertEqual(
19             view.func.__name__,
20             HomePageView.as_view().__name__
21         )
```

Now run the tests again and you should see that 3 tests have ran successfully:

```
Found 3 test(s).
System check identified no issues (0 silenced).
...
----------------------------------------------------------------------
Ran 3 tests in 0.015s

OK
```

Tests for the Books app

We want to ensure that the Books model works as expected, including its str representation. And we want to test both **ListView** and **DetailView**.

Open **books/tests.py**, delete the top line of code and copy-paste the following imports:

```
from django.test import Client, TestCase
from django.urls import reverse
from .models import Book
```

We import **TestCase** which we will subclass as this enables us to run each test inside a transaction to provide isolation. We also import **Client** which is used as a dummy Web browser for simulating GET and POST requests on a URL. In other words, whenever you are testing views, you should use Client().

Create a new class which is a subclass of **TestCase** and code the **setup** method. Copy-paste the following code just below the imports:

```
class BookTests(TestCase):
    def setUp(self):
        self.book = Book.objects.create(
            title='Django 4 By Example',
            author='Anthony Mele',
            price='25.00',
            date_publication='2022-08-01'
        )
```

In our setUp method above we add a sample book to test.

Copy-paste the following code for **test_book_listing** which checks that both its string representation and content are correct:

```
    def test_book_listing(self):
        self.assertEqual(f'{self.book.title}', 'Django 4 By Example')
        self.assertEqual(f'{self.book.author}', 'Anthony Mele')
        self.assertEqual(f'{self.book.price}', '25.00')
        self.assertEqual(f'{self.book.date_publication}', '2022-08-01')
```

You can run this test by typing the following command at the command line:

python manage.py test books

After you run the test, you will see the following output:

```
Found 1 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

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

The next test is called **test_book_list_view** which will confirm that our homepage returns a 200 HTTP status code, contains our body text, and uses the correct **books/book_list.html** template.

Copy-paste the following code for **test_book_list_view**:

```
    def test_book_list_view(self):
        response = self.client.get(reverse('book_list'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Django 4 By Example')
        self.assertTemplateUsed(response, 'books/book_list.html')
```

After you run the test, you will see the following output:

```
(env) C:\Users\pmagee\djangoprojects\lab-9-upload-pmagee>python manage.py test
Found 2 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.035s

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

The next test is called **test_book_detail_view** which tests that our detail page works as expected and that an incorrect page returns a 404.

Copy-paste the following code for test_book_list_view:

```
    def test_book_detail_view(self):
        response = self.client.get(self.book.get_absolute_url())
        no_response = self.client.get('/books/12345/')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(no_response.status_code, 404)
        self.assertContains(response, 'Django 4 By Example')
        self.assertTemplateUsed(response, 'books/book_detail.html')
```

Go ahead and run this test and it should also pass.

```
Found 3 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...
----------------------------------------------------------------------
Ran 3 tests in 0.023s

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

In our **books** app we also have a class based view for performing a simple search on the database.

Copy-paste the following code into **books/tests.py** for **test_search_results_view**:

```
    def test_search_results_view(self):
        response = self.client.get(reverse('search_results'), {'q': 'Django'})
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Django')
```

Here we are calling the URL name of search_results via the reverse method and passing in the search term via the Q object. We then check that this returns a 200 HTTP status code indicating a success and we use the **assertContains** method to check that our response contains our search query i.e., Django.

```
Found 4 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
----------------------------------------------------------------------
Ran 4 tests in 0.027s

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

The test_get_absolute_url_method method tests the get_absolute_url method of the Book model can also be tested. In this example, we supply the url pattern as specified in our urls.py files in our book project.

Copy-paste the following code at the end of tests.py:

```
    def test_book_str_representation(self):
        expected_str = "Django 4 By Example"
        self.assertEqual(str(self.book), expected_str)

    def test_get_absolute_url_method(self):
        # Test the get_absolute_url method of Profile
        expected_url = '/books/{}/'.format(self.book.pk)
        self.assertEqual(self.book.get_absolute_url(), expected_url)
```

After you run the test, you will see the output indicating that the 6 tests have passed:

Measuring Coverage

After you have written some tests for your Django project, and they have passed, it would be interesting to measure the coverage of your tests–that is, how thoroughly your tests exercise the application's code.

One of the most popular tools for measuring coverage in Python is simply called **coverage**. While your tests are running, it keeps track of which lines of the application code are executed, which ones are skipped (like comments), and which ones are never reached. At the end, it produces a report that indicates which lines of code were not executed which may point to gaps in your test coverage.

To install coverage into your virtual environment, run the following command:

pip install coverage

To run coverage, type the following command:

coverage run manage.py test -v 2

Use verbosity level 2, -v 2, for more detail. This command will run all the tests and coverage will generate statistics for us.

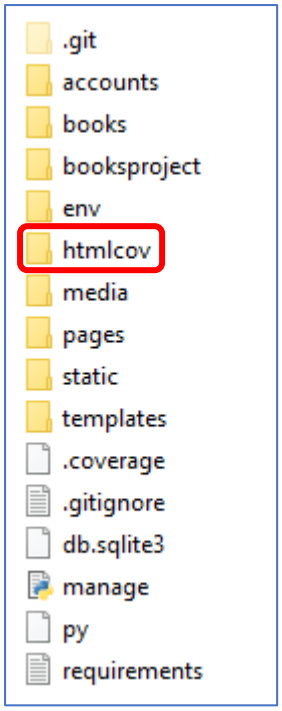
To generate the report run the following command:

coverage html

You will see the following output:

```
Wrote HTML report to htmlcov\index.html
```

In Windows Explorer navigate to your lab 9 project folder and you will see a new folder called htmlcov as shown below:



Open this folder and then open the **index.html** page which should open in your browser. You will see a report like the following showing that the coverage for this application is at 99%. Some tests are not required even though coverage highlights them. For example, the **manage.py** file below is at 83% but we do not need to test this as we have not written any of the code in this file.

Coverage report: 99%

coverage.py v7.3.2, created at 2023-10-25 17:41 +0100

Module	statements	missing	excluded	coverage
accounts__init__.py	0	0	0	100%
accounts\admin.py	11	0	0	100%
accounts\apps.py	4	0	0	100%
accounts\forms.py	11	0	0	100%
accounts\migrations\0001_initial.py	8	0	0	100%
accounts\migrations\0002_profile.py	6	0	0	100%
accounts\migrations__init__.py	0	0	0	100%
accounts\models.py	14	0	0	100%
accounts\tests.py	32	0	0	100%
accounts\urls.py	3	0	0	100%
accounts\views.py	15	0	0	100%
bookproject__init__.py	0	0	0	100%
bookproject\settings.py	27	0	0	100%
bookproject\urls.py	5	0	0	100%
books__init__.py	0	0	0	100%
books\admin.py	5	0	0	100%
books\apps.py	4	0	0	100%
books\migrations\0001_initial.py	6	0	0	100%
books\migrations\0002_book_cover.py	4	0	0	100%
books\migrations__init__.py	0	0	0	100%
books\models.py	14	0	0	100%
books\tests.py	33	0	0	100%
books\urls.py	3	0	0	100%
books\views.py	17	0	0	100%
manage.py	12	2	0	83%
pages__init__.py	0	0	0	100%
pages\admin.py	1	0	0	100%
pages\apps.py	4	0	0	100%
pages\migrations__init__.py	0	0	0	100%
pages\models.py	1	0	0	100%
pages\tests.py	14	0	0	100%
pages\urls.py	3	0	0	100%
pages\views.py	3	0	0	100%
Total	**260**	**2**	**0**	**99%**

coverage.py v7.3.2, created at 2023-10-25 17:41 +0100