Web Shop App Testing

Django Testing Setup

A virtual environment was used for testing. This is a requirement as it does not alter the current database model. As postgres was unavailable for testing, the SQLite3 database was tested. A new folder within the project folder was created to hold the test environment with virtualenv package and a virtual environment created using the following commands:

```
pip install virtualenv
virtualenv ~\mushop_test_folder
```

Once created, the environment must be activated. On windows, this is done by calling the activate.bat file held in the “Scripts” folder of the test environment directory. The command below performs this operation.

```
~\mushop_test_folder\Scripts\activate.bat
```

This command will show the test environment name in parenthesis in the terminal once called. All requirements had to be installed using the command below. These are required as the project apps make use of one or more of these installed dependencies:

```
pip install -r requirements.txt
```

To get all models from the original database, a migrate must be performed to update the new test database

```
python manage.py migrate
```

To test the SQLite3 database, in the settings python, the database section must be as follows:

```
DATABASES = {
      'default': {
            'ENGINE': 'django.db.backends.sqlite3',
            'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
      }
}
```

SQLite3 database testing.

Testing the development database assures the production side runs smoothly. Changes in file paths and primarily the DATABASE_URL environment variable are some minor changes between development and production databases. The model and views will function the same. For the most part, testing was performed on views to verify all pages loaded the correct information without errors. Following this, model return methods were verified to be as designed.

Tests

Views

Product model views were tested to verify their output. This was done using the browser request codes, specifically the code for ok, 200. Each view url was tested against it's rendered html template. If a test resulted in a browser code equalling 200, the test passed.

```
def test_get_products_page(self):
       page = self.client.get("/products/")
       self.assertEqual(page.status_code, 200)
       self.assertTemplateUsed(page, "products.html")
```

For model field specific views, product_id for example, a test product is created and saved with all required fields including product_id. This id is then added to the end of the url via a format reference and tested as the other more general links were tested.

```
def test_get_one_product_page(self):
       category = Categorie(name="A name")
       category.save()
       product = Product(
                  id=4,
                  name="Create a Name",
                  category=category,
                  price=12
                  )
       product.save()
       page = self.client.get("/products/{0}/".format(product.id))
       self.assertEqual(page.status_code, 200)
       self.assertTemplateUsed(page, "single.html")
```

In this case, category was required as it is included in the product model as a foreign key.

To test reviews being posted were added to the admin panel correctly, a test review dictionary is created and tested against the required view. In this case, a redirect to the products template.

```
def test_create_review(self):
       review = Review(id=1)
       response = self.client.post("/products/")
       self.assertEqual(response.status_code, 200)
       self.assertTemplateUsed(response, "products.html")
```

Models

Testing models against their output is different for each app. In the products app, a string is returned, usually one field value. This required a test product creation referencing the required output field, and an equal assertion with a matching value in the assertEqual method.

```
class TestCategorieModel(TestCase):
    def test_category_name_as_a_string(self):
       category = Categorie()
       category.name="Create a Name"
       self.assertEqual("Create a Name", str(category))
```

For models with an additional function, this required a call to the model with an equal assertion method checking the function was being called.

```
class TestProductModel(TestCase):
    def test_product_has_average_rating(self):
       product = Product()
       product.name="Create a Name"
       self.assertTrue(product.average_rating())
       self.assertTrue("Create a Name", str(product))
```

In the orders model, a formatted string output is generated and sent to the admin panel. To test, a test model is created with the required fields provided with test values. The result is then referenced against an equal assertion with the expected output. Some models require more fields than others to be included, as the model field is set as required. The idea is to mimic the product output. This tests whether it outputs at all, and if the output is correct.

```python
class TestOrderModel(TestCase):
    def test_order_has_string_output(self):
       order = Order()
       order.full_name="Create a Name"
       order.id="An Id"
       order.date="A Date"
       result = "{0}-{1}-{2}".format(order.id, order.date, order.full_name)
       self.assertEqual("An Id-A Date-Create a Name", str(result))
```

Manual Testing

Browser

Making use of the browser inspector console to check functionality, and investigate errors was vital for a smooth running app. Errors in syntax and structural tags were revealed in the Microsoft Edge inspector tools. Status code tests were mostly done in Firefox, as this is a standard setting in the inspector. The materialize style library uses a lot of additional style declarations for different browsers. Firefox registers some of these as “Unknown pseudo class”. These are vendor prefixes for specific browsers. While registering as an error in the console, they cause no change to the style.

The navbar was a primary focus during visual testing. With use of the row and column system in materialize, the navbar was set to fit comfortably on a screen above 1200 pixels in width. The materialize collapsible menu then kicks. However while adding items to both cart and compare functions, the nav items would spill into the logo area. A custom class was used to hide the text just before the menu collapsed at 1366 pixels in width. This allowed items to fit on screen with no overlap. Icons were used to let the user know visually what the menu item represented.

Using the built in class names from materialize, grid flexibility optimisation was achieved on the following platforms:

- iPhone 5 - 8
- Samsung Galax7 S5, S7, S8
- ChromeBook 10"
- iPad Mini
- iPad Pro
- iPad 2.
- Kindle Fire HD Linux

Manual User Tests

My work colleagues and family provided front end feedback on functionality. Since the app is based on an online shop, user interactions from customers and retail employees was required.

Some of the key issues were:

- **Searching** - Initially the search function only covered product names. Additional fields were added to the search using the Django “Q Object”. This allows iteration over multiple model fields. These iterations can be joined together, or set as individual searches.
- **User Registration** - Separating login, logout and user registration function gave the site a need to register before payment. This is logical from a business perspective. User’s must register themselves to make a payment. In the login page, a suggestion was made to add a piece of text regarding registration. This is sound reasoning as it allows a new customer to register even if they click on the login menu item.
- **Pagination** - Making use of pagination for product listings was added, as the product database could grow in size. In addition, a list of new products added to the database is shown on the home page. Initially, this was not split into pages. This created a large area just above the footer. A work colleague suggested I split new products into groups of 4 to even out the page layout.
- **Stock Counter** - A stock control measure was set in place to stop items being unavailable for purchase once the stock level reached 0. There was no indication to the user of stock level however. A work colleague suggested a stock counter be added to each product telling the customer stock levels, and using colours to demonstrate low and high levels of stock.
- **Account Edit** - Most testers wanted to be able to change their account credentials. An adit profile function was created to meet this need. Django has a default profile editing template. This had to be over ridden with a new template to replace it. The django form for user profile has individually accessible elements. This allowed for a neater for to be created than the standard account form.