**Lab 10 Part 12 Unit Testing**

To avoid errors with images, add the following code to the **product.html** and **category.html** templates. This code will avoid errors in the project if there is no image added for a product in the database.

**category.html**

A computer code on a black background

Description automatically generated

Text

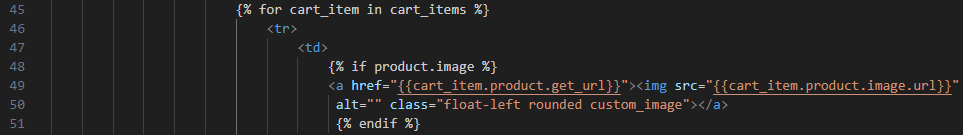
Description automatically generated

**product.html**

Text

Description automatically generated

**cart.html**



**Shop App**

Open **shop/tests.py** and copy-paste the following code to test the models:

from django.test import TestCase

reverse is a function from django.urls used for generating URLs for views based on their names.

from django.urls import reverse

from .models import Category, Product

class CategoryModelTest(TestCase):

    def setUp(self):

        self.category = Category.objects.create(

            name='Outdoor Cushions',

            description='Outdoor cushions for your summer living space'

        )

    def test\_category\_creation(self):

        self.assertEqual(self.category.name, 'Outdoor Cushions')

        self.assertEqual(self.category.description, 'Outdoor cushions for your summer living space')

    def test\_category\_str\_method(self):

This test checks whether the string representation (\_\_str\_\_ method) of the Category object is as expected.

        self.assertEqual(str(self.category), 'Outdoor Cushions')

    def test\_category\_get\_absolute\_url(self):

        expected\_url = reverse('shop:products\_by\_category', args=[str(self.category.id)])

        self.assertEqual(self.category.get\_absolute\_url(), expected\_url)

This test checks whether the get\_absolute\_url method of the Category model returns the expected URL. It uses the reverse function to generate the expected URL based on the name 'shop:products\_by\_category' and the category's ID.

class ProductModelTest(TestCase):

    def setUp(self):

        self.category = Category.objects.create(

            name='Clothing',

            description='Fabulous clothing items'

        )

        self.product = Product.objects.create(

            name='Blouse',

            description='Silk long sleeve blouse',

            category=self.category,

            price=99.99,

            stock=20,

            available=True

        )

    def test\_product\_creation(self):

        self.assertEqual(self.product.name, 'Blouse')

        self.assertEqual(self.product.description, 'Silk long sleeve blouse')

        self.assertEqual(self.product.category, self.category)

        self.assertEqual(self.product.price, 99.99)

        self.assertEqual(self.product.stock, 20)

        self.assertTrue(self.product.available)

    def test\_product\_str\_method(self):

        self.assertEqual(str(self.product), 'Blouse')

    def test\_product\_get\_absolute\_url(self):

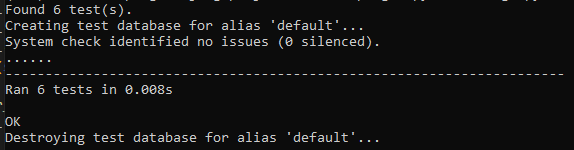
        expected\_url = reverse('shop:product\_detail', args=[str(self.category.id), str(self.product.id)])

        self.assertEqual(self.product.get\_absolute\_url(), expected\_url)

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

**python manage.py test shop**

You should see output similar to the following indicating that the tests ran ok:



Add the following code to test the **views** in the shop app:

class ShopViewsTest(TestCase):

    def setUp(self):

        # Create some test data (categories and products) for your views

        self.category = Category.objects.create(name='Test Category')

        self.product = Product.objects.create(

The setUp method is created here in order to create a Category object and a Product object. The product is part of a specific category.

            name='Test Product',

            category=self.category,

            description='Test Description',

            price=10.99,

This first test checks the behaviour of the view associated with the URL named 'shop:all\_products'. It makes a GET request to the URL and then asserts that the response status code is 200 (OK), that it contains the text 'Test Category', and that the template used is 'shop/category.html'.

            stock=45,

            available=True,

        )

    def test\_prod\_list\_view(self):

        response = self.client.get(reverse('shop:all\_products'))

        self.assertEqual(response.status\_code, 200)

        self.assertContains(response, 'Test Category')

        self.assertTemplateUsed(response, 'shop/category.html')

    def test\_prod\_list\_view\_with\_category(self):

        response = self.client.get(reverse('shop:products\_by\_category', args=[self.category.id]))

        self.assertEqual(response.status\_code, 200)

        self.assertContains(response, 'Test Category')

        self.assertTemplateUsed(response, 'shop/category.html')

    def test\_product\_detail\_view(self):

        response = self.client.get(reverse('shop:product\_detail', args=[self.category.id, self.product.id]))

        self.assertEqual(response.status\_code, 200)

        self.assertContains(response, 'Test Product')

        self.assertTemplateUsed(response, 'shop/product.html')

Note the use above of the following line of code in all three tests:

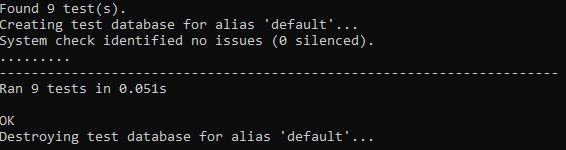
**self.client.get()**

In Django testing, self.client is an instance of the Django test client. The test client is a Python class that provides a simple way to make HTTP requests to your Django project during testing. It allows you to simulate a user interacting with your views without having to run the server. In these tests self.client.get(url) is making a GET request to the specified URL using the test client. This allows you to simulate a user visiting a particular URL in your Django project. The response variable will contain the HTTP response returned by the server for that request.

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

**python manage.py test shop**

You should see output similar to the following indicating that 9 tests ran ok:



Add the following code to test the **urls** in the **shop** app:

class ShopUrlsTestCase(TestCase):

    def setUp(self):

        # Create sample data for testing

        self.category = Category.objects.create(name='Test Category')

        self.product = Product.objects.create(name='Test Product', price='50.00',stock='4',category=self.category)

This test checks the behavior of the URL 'shop:all\_products'. It uses the reverse function to get the URL, then makes a GET request to that URL and asserts that the response status code is 200 (OK).

    def test\_all\_products\_url(self):

        url = reverse('shop:all\_products')

        response = self.client.get(url)

        self.assertEqual(response.status\_code, 200)

    def test\_products\_by\_category\_url(self):

        url = reverse('shop:products\_by\_category', args=[str(self.category.id)])

        response = self.client.get(url)

        self.assertEqual(response.status\_code, 200)

    def test\_product\_detail\_url(self):

        url = reverse('shop:product\_detail', args=[str(self.category.id), str(self.product.id)])

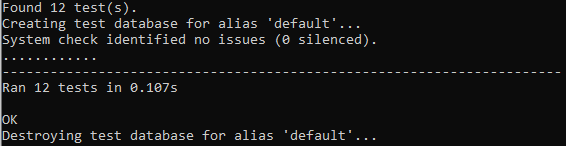
        response = self.client.get(url)

        self.assertEqual(response.status\_code, 200)

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

**python manage.py test shop**

You should see output similar to the following indicating that 12 tests ran ok:



**Cart App**

Open **cart/tests.py** and copy-paste the following code to test the models:

from django.test import TestCase

from django.utils import timezone

from shop.models import Product, Category

from cart.models import Cart, CartItem

class CartModelsTest(TestCase):

    def setUp(self):

        self.c = Category.objects.create(name='test category')

        self.product = Product.objects.create(

            name='Test Product',

            price=10.0,

            stock = 900,

            category=self.c

        )

        self.cart = Cart.objects.create(cart\_id='test\_cart', date\_added=timezone.now())

        self.cart\_item = CartItem.objects.create(

            product=self.product,

            cart=self.cart,

            quantity=2,

            active=True

        )

    def test\_cart\_str\_method(self):

        self.assertEqual(str(self.cart), 'test\_cart')

    def test\_cart\_item\_sub\_total\_method(self):

        expected\_sub\_total = self.product.price \* self.cart\_item.quantity

        self.assertEqual(self.cart\_item.sub\_total(), expected\_sub\_total)

    def test\_cart\_item\_str\_method(self):

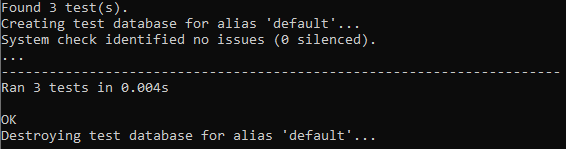
        expected\_str = str(self.product)

        self.assertEqual(str(self.cart\_item.product.name), expected\_str)

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

**python manage.py test cart**

You should see output similar to the following indicating that 3 tests ran ok:



**Cart App**

Open **cart/tests.py** and copy-paste the following code to test the **add\_cart** view:

from django.urls import reverse

class CartViewTests(TestCase):

    def setUp(self):

        self.c = Category.objects.create(name='test category')

        self.product = Product.objects.create(name='Test Product',

            price=10.0,

The first test Sends a GET request to the 'cart:add\_cart' URL with the ID of the test product. An assertion is used to check that the status code is 302, indicating a redirect. We then retrieve the Cart and CartItem instances associated with the current session and product. An assertion is used to check that the quantity of the CartItem is 1, indicating the product has been added to the cart.

            stock=2,

            category=self.c,

        )

    def test\_add\_cart(self):

        response = self.client.get(reverse('cart:add\_cart', args=[self.product.id]))

        self.assertEqual(response.status\_code, 302)

        cart = Cart.objects.get(cart\_id=self.client.session.session\_key)

        cart\_item = CartItem.objects.get(product=self.product, cart=cart)

        self.assertEqual(cart\_item.quantity, 1)

    def test\_add\_cart\_quantity\_limit(self):

        response = self.client.get(reverse('cart:add\_cart', args=[self.product.id]))

        self.assertEqual(response.status\_code, 302)

        cart = Cart.objects.get(cart\_id=self.client.session.session\_key)

        cart\_item = CartItem.objects.get(product=self.product, cart=cart)

        self.assertEqual(cart\_item.quantity, 1)

        # Attempt to add more items than stock

        response = self.client.get(reverse('cart:add\_cart', args=[self.product.id]))

        self.assertEqual(response.status\_code, 302)  # Expecting a redirect

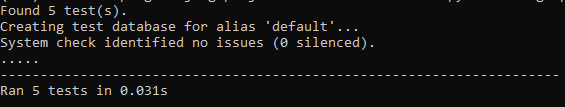
        cart\_item.refresh\_from\_db()  # Refresh the cart\_item instance

        self.assertEqual(cart\_item.quantity, self.product.stock)

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

**python manage.py test cart**

You should see output similar to the following indicating that 5 tests ran ok:



The 2nd test test\_add\_cart\_quantity\_limit is based on the expectation that trying to add more items than the stock would not be allowed, and the view would redirect to another page. This test also refreshes the CartItem instance from the database to ensure that it reflects the latest state. The last line in this test asserts that after the attempt, the cart item quantity is equal to the product's stock, indicating that the attempt did not increase the quantity beyond the stock limit.

Both tests use status code 303 which in HTTP, indicates a temporary redirect. When a client (e.g., a web browser) receives a response with a status code of 302, it means that the requested resource has been temporarily moved to a different location, and the client should issue a new request to that location. For example, in the test\_add\_cart method, after adding a product to the cart, the code expects a redirect to occur, as indicated by the 302 status code. The actual redirection target (i.e., the URL to which the client is redirected) is not explicitly checked in this particular assertion but in our view, we do redirect to the cart details page.

Open **cart/tests.py** and copy-paste the following code to test the **cart\_remove & full\_remove** views:

class CartRemoveViewTest(TestCase):

    def setUp(self):

        self.category = Category.objects.create(name='Test Category', description='Test description')

        # Create a product for testing

        self.product = Product.objects.create(

            name='Test Product',

            description='Test description',

            category=self.category,

            price=19.99,

            stock=10

        )

    def \_cart\_id(self, request):

        # Your implementation of the cart ID generation function using the request

        # This can be replaced with your actual implementation

        return request.session.session\_key

    def test\_cart\_remove\_view(self):

        # Set up a request with a session

        request = self.client.request().wsgi\_request

        request.session = self.client.session

        request.session.save()

        # Set the cart ID using the \_cart\_id function

        request.session['cart\_id'] = self.\_cart\_id(request)

        # Create a cart using the \_cart\_id function

        cart = Cart.objects.create(cart\_id=request.session['cart\_id'])

        cart\_item = CartItem.objects.create(product=self.product, quantity=2, cart=cart)

        # Make a request to the cart\_remove view

        response = self.client.post(reverse('cart:cart\_remove', args=[self.product.id]))

        # Assert that the response has a status code of 302 (redirect)

        self.assertEqual(response.status\_code, 302)

        # Assert that the cart\_item quantity is reduced by 1

        updated\_cart\_item = CartItem.objects.get(id=cart\_item.id)

        self.assertEqual(updated\_cart\_item.quantity, 1)

    def test\_cart\_full\_remove\_view(self):

        # Set up a request with a session

        request = self.client.request().wsgi\_request

        request.session = self.client.session

        request.session.save()

        # Set the cart ID using the \_cart\_id function

        request.session['cart\_id'] = self.\_cart\_id(request)

        # Create a cart using the \_cart\_id function

        cart = Cart.objects.create(cart\_id=request.session['cart\_id'])

        cart\_item = CartItem.objects.create(product=self.product, quantity=1, cart=cart)

        # Make a request to the cart\_remove view for the last item

        response = self.client.post(reverse('cart:full\_remove', args=[self.product.id]))

        # Assert that the response has a status code of 302 (redirect)

        self.assertEqual(response.status\_code, 302)

        # Assert that the cart\_item is deleted

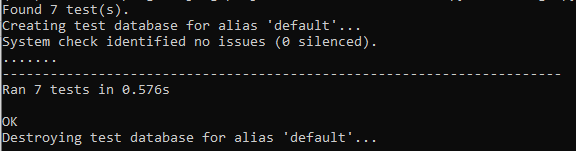
        with self.assertRaises(CartItem.DoesNotExist):

            CartItem.objects.get(id=cart\_item.id)

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

**python manage.py test cart**

The output should indicate that the 7 tests ran ok:



Copy-paste the following code into **cart/tests.py** to test the **stripe** payment functionality in the **cart\_detail** view:

from django.test import Client

from accounts.models import CustomUser

import stripe

class CartDetailViewTest(TestCase):

    def setUp(self):

        # Create a client

        self.client = Client()

        self.category = Category.objects.create(name='Test Category', description='Test description')

        # Create a product for testing with a UUID and associate it with the category

        self.product = Product.objects.create(

            name='Test Product',

            description='Test description',

            category=self.category,

            price=19.99,

            stock=10

        )

        # Create a user for the session

        self.user = CustomUser.objects.create\_user(username='testuser', password='testpassword')

        self.client.login(username='testuser', password='testpassword')

        # Add Stripe testing keys to your settings if not already present

        stripe.api\_key = ''

For stripe\_api\_key, you need to insert your private key in here.

    def \_cart\_id(self, request):

        # Your implementation of the cart ID generation function using the request

        # This can be replaced with your actual implementation

        return request.session.session\_key

    def test\_cart\_detail\_view(self):

        # Set up a request with a session

        request = self.client.request().wsgi\_request

        request.session = self.client.session

        request.session.save()

        # Set the cart ID using the \_cart\_id function

        request.session['cart\_id'] = self.\_cart\_id(request)

        # Create a cart using the \_cart\_id function

        cart = Cart.objects.create(cart\_id=request.session['cart\_id'])

        cart\_item = CartItem.objects.create(product=self.product, quantity=2, cart=cart)

        # Mock the Stripe API calls using the stripe.testing library

You need to insert your private and public keys in here.

        with self.settings(STRIPE\_SECRET\_KEY='', STRIPE\_PUBLISHABLE\_KEY=''):

            with self.assertLogs('stripe', level='INFO') as stripe\_logs:

                response = self.client.post(reverse('cart:cart\_detail'), {

                    'stripeToken': 'tok\_visa',

                    'stripeEmail': 'test@example.com',

                    'stripeBillingName': 'Test User',

                    'stripeBillingAddressLine1': '123 Main St',

                    'stripeBillingAddressCity': 'City',

                    'stripeBillingAddressCountryCode': 'US',

                    'stripeShippingName': 'Test User',

                    'stripeShippingAddressLine1': '123 Main St',

                    'stripeShippingAddressCity': 'City',

                    'stripeShippingAddressCountryCode': 'US',

                })

        # Assert that the response has a status code of 302 (redirect)

        self.assertEqual(response.status\_code, 302)

        # Assert that the expected Stripe logs are present in the logs

        self.assertIn('Request to Stripe api', stripe\_logs.output[0])

        self.assertIn('method=post path=https://api.stripe.com/v1/', stripe\_logs.output[0])

        # Print the content of the logs for debugging

        print('\n'.join(stripe\_logs.output))

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

**python manage.py test cart**

The output should indicate that the tests ran ok. Check your **Stripe** account and you will see that the test payment has gone through.

**Check Testing 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.

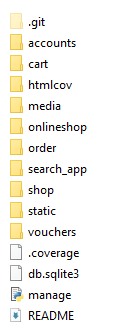
To generate the report run the following command:

**coverage html**

You will see the following output:



In Windows Explorer navigate to your lab 10 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 that the coverage for this project is at **90%**. This lab exercise does not cover all the tests but gives you an idea of how to write the unit tests.