Lab 10 Part 1

Online Shop

This lab contains multiple parts which you will complete to build an online shop. Our users will be able to browse through a product catalogue and add products to a shopping cart. Finally, they will be able to check out the cart and place an order. This lab sheet will cover the following functionalities of an online shop:

- Creating the product catalogue models, adding them to the administration site, and building the basic views to display the catalogue **(code provided for this part**).
- Building a shopping cart system using Django sessions to allow users to keep selected products while they browse the site.
- Creating the form and functionality to place orders on the site.
- Taking payments using Stripe.
- Creating & Viewing orders.
- Using voucher codes.

Initial Set Up

1. Go to www.github.com and log in to your GitHub account.
2. Go to Moodle and click on the link for Lab 10.
3. Once you have accepted the assignment you are asked to refresh the page, and you are then presented with a link to your repository for lab 10.
4. When you click on the repository link, you are taken to the repository where you see a starter project has already been added:
5. In Windows Command line make sure that you are in the djangoprojects folder and type the suitable command to activate the virtual environment:
6. Use the git clone command to clone the repo to your local computer:
7. Move into this lab-10-username folder using the cd command.
8. Open VS Code and open the project folder.

The rest of this lab sheet covers the code that has already been provided in the repo which you have cloned. Take the time to read through this document to familiarise yourself with the project. YOU DO NOT NEED TO WRITE ANY CODE FOR THIS PART OF THE LAB SHEET.

Accounts App

This project contains an **accounts** app with a **CustomUser** model already created in the database. You will use this app for authentication in a later part of the lab exercise.

Shop app

This project contains an app called **shop** which is registered in **settings.py** as shown below:

```
33    INSTALLED_APPS = [
34        'django.contrib.admin',
35        'django.contrib.auth',
36        'django.contrib.contenttypes',
37        'django.contrib.sessions',
38        'django.contrib.messages',
39        'django.contrib.staticfiles',
40        'accounts',
41        'shop',
42    ]
```

In VS Code, open the **shop/models.py** file and you will see the following code:

```
import uuid
from django.urls import reverse


class Category(models.Model):
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False)
    name = models.CharField(max_length=250, unique=True)
    description = models.TextField(blank = True)
    image = models.ImageField(upload_to = 'category', blank=True)


    class Meta:
        ordering = ('name',)
        verbose_name = 'category'
        verbose_name_plural = 'categories'
```

```
    def get_absolute_url(self):
        return reverse('shop:products_by_category', args=[self.id])

    def __str__(self):
        return self.name
```

The Category model consists of an id field, a name field, an optional description field and an optional image field. These last two fields are optional because we have set blank to True.

The Meta class inside the model contains metadata. On the first line we tell Django to sort results by the name field in ascending order (default) when we query the database. The code on the 2nd line assigns the value 'category' to verbose_name. In Django, we use meta data about a database table to tell Django things about the database table. In this example, we specify that the singular form of the object of the database table should be category and the plural form of the database table should be categories. This attribute changes the field name in admin interface. For example, if we don't specify verbose_name_plural then the plural name for categories in Django-admin is Categorys which is a misspelling.

To correct this, we supply the value for verbose_name_plural i.e. multiple category objects and then our Django Admin looks as follows:

Go back to the **models.py** file and take a look at the code for the **Product** class:

```
class Product(models.Model):
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False)
    name = models.CharField(max_length=250, unique=True)
    description = models.TextField(blank = True)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
    price = models.DecimalField(max_digits=10, decimal_places=2)
```

```
    image = models.ImageField(upload_to = 'product', blank=True)
    stock = models.IntegerField()
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True, blank = True, null= True
)
    updated = models.DateTimeField(auto_now=True, blank = True, null= True)

    class Meta:
        ordering = ('name',)
        verbose_name = 'product'
        verbose_name_plural = 'products'


    def get_absolute_url(self):
        return reverse('shop:product_detail', args=[self.category.id, self.id]
)

    def __str__(self):
        return self.name
```

The **Product** model fields are as follows:

- id: unique product id using uuid4
- name: The name of the product.
- description: An optional description of the product.
- category: ForeignKey to the Category model. This is a many-to-one relationship: a product belongs to one category and a category contains multiple products.
- price: This field uses Python's decimal.Decimal type to store a fixed-precision decimal number. The maximum number of digits (including the decimal places) is set using the max_digits attribute and decimal places with the decimal_places attribute.
- image: An optional product image.
- stock: Quantity in stock
- available: A boolean value that indicates whether the product is available or not. It will be used to enable/disable the product in the catalog.
- created: This field stores the date & time when the object was created.
- updated: This field stores the date & time when the object was last updated.

Static & Media Files

The following code is already added at the bottom of **settings.py** just after the `STATIC_URL = '/static/'` entry so that Django knows where to look for our static and media files. This topic was covered in lab 9.

```
STATICFILES_DIRS = [str(BASE_DIR.joinpath('static'))]
STATIC_ROOT = str(BASE_DIR.joinpath('staticfiles'))
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
]
MEDIA_URL = '/media/'
MEDIA_ROOT = str(BASE_DIR.joinpath('media'))
```

Both the Category and the product models have an **ImageField** which requires another configuration to our **settings.py** file to ensure that our images are uploaded to the right location. Since user-uploaded content such as images is assumed to exist in a production context, to see media items locally the **shop_project/urls.py** has been updated to show these files locally.

```
16  from django.contrib import admin
17  from django.urls import path
18  from django.conf import settings
19  from django.conf.urls.static import static
20
21  urlpatterns = [
22      path('admin/', admin.site.urls),
23  ] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

In VS Code, we have a **media** folder at the project level and inside the **media** folder there is a **product** folder and a **category** folder. We will create the **static** folder in lab 10 Part 2.

Image Processing

Since we are dealing with images in our models, make sure that the **Pillow** module is installed into your virtual environment.

Register models on the admin site

Open **shop/admin.py** and you will see the code is provided to register our models.

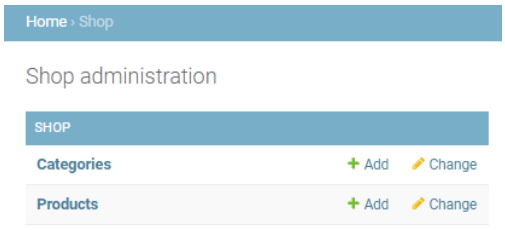
```
from .models import Category, Product


class CategoryAdmin(admin.ModelAdmin):
    list_display = ['name']

admin.site.register(Category, CategoryAdmin)

class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'price', 'description', 'category', 'stock',
'available', 'created', 'updated']
    list_editable = ['price', 'stock', 'available']
    list_per_page = 20

admin.site.register(Product, ProductAdmin)
```

We use the list_editable attribute in the ProductAdmin class to set the fields that can be edited from the list display page of the administration site. This will allow you to edit multiple rows at once. Any field in list_editable must also be listed in the list_display attribute since only the fields displayed can be edited. The list_per_page attribute is used to set the number of products that can be displayed per page.

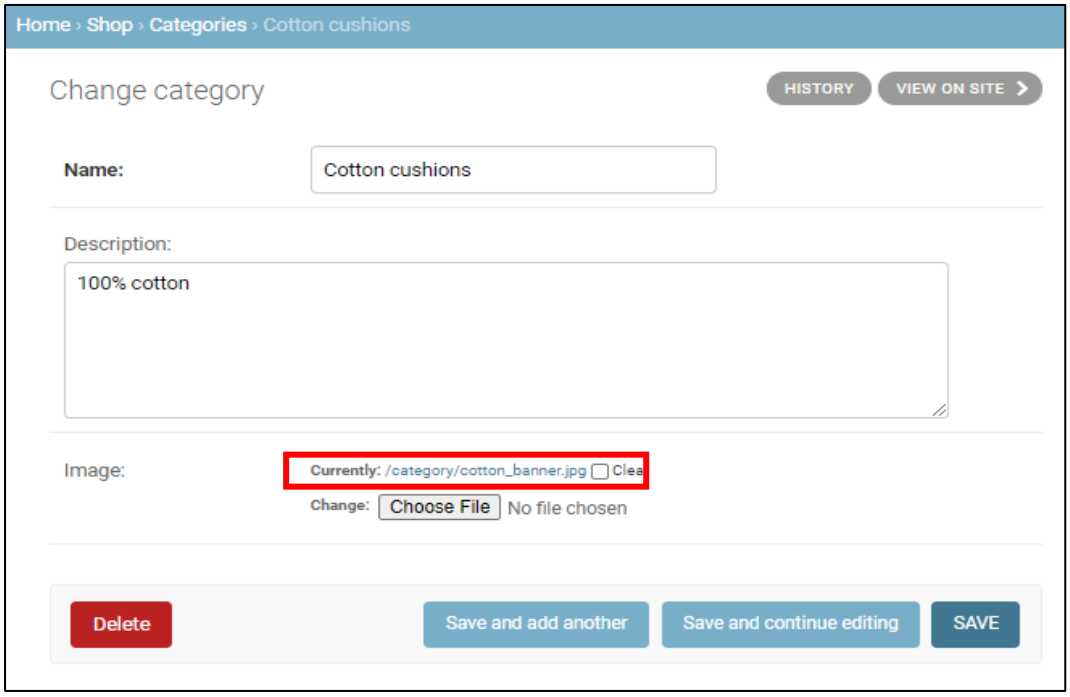
Category & Product data

In **VS Code** you will see a folder inside the **shop** app called **fixtures**. Inside this folder there is a file called shop.json which has already been used to load the data into the **Product** and **Category** tables:

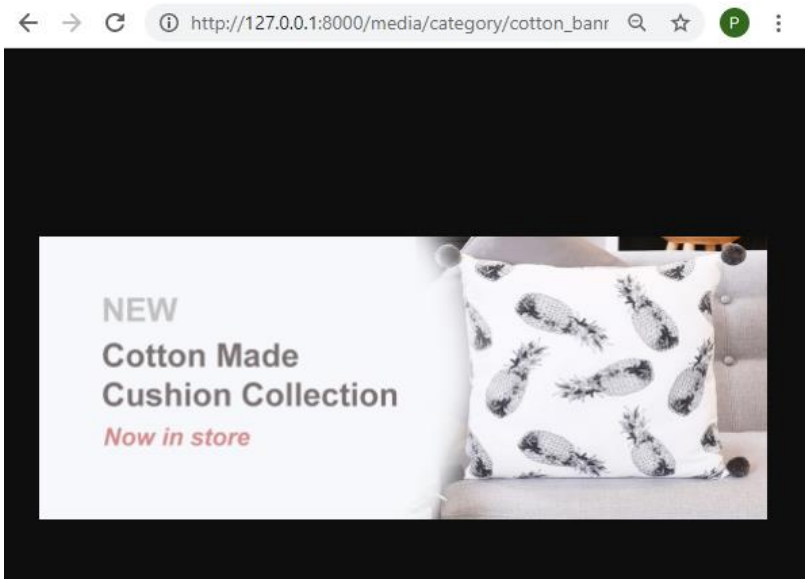
Create a superuser account, run the server, log into Django Admin and you should see the data for the category and product objects has been added.



Click on the Cotton cushions category to view the details of this category as shown below:



If you click on the **category/cotton_banner.jpg**, this image should display which means that Django is able to find where it is stored locally in our project.



Recap: The reason that Django can find the image is because earlier, we added the following setting below into **settings.py**:

```
16  from django.contrib import admin
17  from django.urls import path
18  from django.conf import settings
19  from django.conf.urls.static import static
20
21  urlpatterns = [
22      path('admin/', admin.site.urls),
23  ] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Images for the Product objects

In Django admin, click on a Product object and you should see an image for the product. In VS Code note that the product images are in the **media/products** folder as shown here: