Initial Setup

1. Create a folder with the name of your project and open it with VS Code.
2. Open git bash terminal to Setup virtual environment and install django:
   1. Create virtual environment:

python -m venv env

* 1. Activate the virtual environment:

source env/Scripts/activate

* 1. Install Django inside the virtual environment:

pip install django

* 1. Check the django installation package:

pip freeze

1. Create the project.

django-admin startproject .

1. Run the server to check if it is working:

python manage.py runserver

Digging deep into the project

Setting up the home page of the project:

1. Create a folder named ‘templates’ at the root directory of your project which will contain all the html files of your project.
2. Copy the html file that your need as your home page or landing page or your website inside the ‘templates’ folder.
3. Register the ‘templates’ folder in the project. Go to your project’s ‘settings.py’ file and find ‘TEMPLATES’ list. Inside this list find ‘DIRS’. Inside ‘DIRS’ add the name of the folder that contains all html files. Here, ‘templates’.

'DIRS': ['templates'],

1. Now create a folder named ‘views.py’ inside your project folder that will be responsible for all the views of our project.
2. Edit ‘views.py’ file for our home page as follows:

from django.shortcuts import render

def home(request):

return render(request, 'index.html')

1. Edit your project’s ‘urls.py’ to assign the newly created view to an url. Code as follows:

from django.contrib import admin

from django.urls import path

from . import views

urlpatterns = [

path('admin/', admin.site.urls),

path("", views.home, name='home'),

]

1. Now run the server again and see if it is working.

Configuring static files. Make CSS, JS, and Images file to work.

1. Create a new folder named ‘static’ inside your project folder. Copy all the static files for your project inside this directory.
2. Register the static directory inside the project’s ‘settings.py’ file. Code as follows:

STATIC\_URL = 'static/'

STATIC\_ROOT = BASE\_DIR /'static'

STATICFILES\_DIRS = [

'greatkart/static',

]

1. Now from the terminal run the following command to register all the changes that we made for static files in the ‘settings.py’ file. Run the below command:

python manage.py collectstatic

1. The above command will collect the static folder from our project directory and make a copy of it to the root directory of our project and register all the files.
2. Now go to your html file and add the line below at the very beginning to load all the static files inside that particular html file so that we can make use of them.

{% load static %}

1. Now in the html file replace all the links with

{% static ‘link\_to\_that\_file’ %}

For example, In my html file favicon link was:

<link href="images/favicon.ico" rel="shortcut icon" type="image/x-icon" />

Now as we need to load it from our static folder the new link will be as follows:

<link href="{% static 'images/favicon.ico' %}" rel="shortcut icon" type="image/x-icon" />

1. Now run the server. Everything will work smoothly.

Refactor the html file. Break the main html file into segments, so that we can reuse it in other html files with out re writing it everywhere.

1. Create a subfolder under ‘templates’ folder. I’m naming it as ‘includes’. You can name it anything.
2. Separate the header code as ‘header.html’ save it inside ‘includes’ folder. And don’t forget to add

{% load static %}

at the very beginning in the ‘header.html’ file.

1. Separate the nav bar code as ‘nav\_bar.html’ save it inside ‘includes’ folder. And don’t forget to add

{% load static %}

at the very beginning in the ‘header.html’ file.

1. Separate the footer code as ‘footer.html’ save it inside ‘includes’ folder. And don’t forget to add

{% load static %}

at the very beginning in the ‘header.html’ file.

1. Now link the ‘header.html’ file, ‘nav\_bar.html’ file and ‘footer.html’ file inside the ‘index.html’. As:

<!-- Including header file -->

{% include 'includes/header.html' %}

<!-- Including nav bar file -->

{% include 'includes/nav\_bar.html' %}

<!-- Loading static files -->

{% load static %}

<!-- Your main html code here for index.html file -->

<!-- Including footer file -->

{% include 'includes/footer.html' %}

1. Now run the server again. It will work fine as pervious.

Creating new app for categories:

1. Goto terminal and type the following to create a new app. I’m naming my app as ‘category’ as we will do all the category stuff here.

python manage.py startapp category

1. Now firstly register your app to your project. Go to your project’s ‘settings.py’ file and there inside INSTALLED\_APPS add your app name.
2. Now we will create the model (database table) for our category. Go to your app’s ‘models.py’ file and configure your database table. Mine is as below:

class Category(models.Model):

category\_name = models.CharField(max\_length = 100, unique = True)

# A Slug is basically a short label for something, containing only

# letters, numbers, underscores or hyphens. They’re generally used in URLs.

category\_slug = models.SlugField(max\_length = 100, unique = True)

category\_description = models.TextField()

category\_image = models.ImageField(blank = True)

created\_at = models.DateTimeField(auto\_now = True)

updated\_at = models.DateTimeField(auto\_now\_add = True)

1. Now register your model to your app’s ‘admin.py’. Code as follows:

from .models import Category

# Register your models here.

admin.site.register(Category)

1. Now install the ‘pillow’ library as we are using image field. Otherwise, migrations will now be completed.

pip install pillow

1. Now make the migrations to the database. Run the below code to make the migration.

python manage.py makemigrations

1. Now migrate it to database. Code as follows:

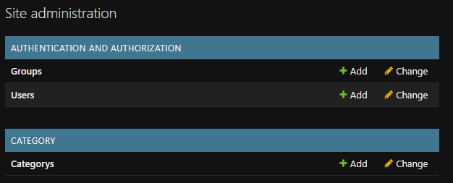
python manage.py migrate

1. Now create a superuser to log in into the Django admin panel. Run the code below:

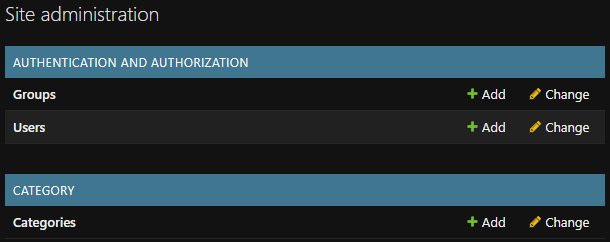
python manage.py createsuperuser

1. Run the server and log in into Django admin page.

python manage.py runserver

1. In the Django admin pannel you will see that our table name is showing as ‘categorys’ although we have named it as ‘category’.

The main reason is that Django is making the name as plural as there can be multiple records. We can make it better. For that go to your app’s ‘models.py’ and add the code below inside your existing class ‘Category’:

class Meta:

verbose\_name = 'category'

verbose\_name\_plural = 'categories'

Then, again run the makemigrations and migrate commands.

Creating Custom User model for user login instead of Django default user model.

1. Create a new app. I’m naming it as ‘accounts’. Run the code below to create this app:

python manage.py startapp accounts

1. Register your app to your project’s ‘settings.py’ file.
2. Now create the model. Edit your app’s ‘model.py’ file as below:

from django.db import models

from django.contrib.auth.models import AbstractBaseUser, BaseUserManager

# Create your models here.

class MyAccountManager(BaseUserManager):

def create\_user(self, first\_name, last\_name, username, email, password=None):

if not email:

raise ValueError('Email is mandatory')

if not username:

raise ValueError('User Name is mandatory')

user = self.model(

# this will convert any capitals in email to small

email = self.normalize\_email(email),

username = username,

first\_name = first\_name,

last\_name = last\_name,

)

# setting the password with the inbuilt function set\_password

user.set\_password(password)

# saving the user to database

user.save(using = self.\_db)

# returning user object

return user

def create\_superuser(self, first\_name, last\_name, username, email, password):

# creating super user with the help of create\_user method

user = self.create\_user(

email = self.normalize\_email(email),

username = username,

password = password,

first\_name = first\_name,

last\_name = last\_name,

)

# giving all the permissions to the super user

user.is\_admin = True

user.is\_active = True

user.is\_staff = True

user.is\_superadmin = True

# saving the super user with all the permissions

user.save(using = self.\_db)

# returning superuser

return user

class Account(AbstractBaseUser):

first\_name = models.CharField(max\_length = 50)

last\_name = models.CharField(max\_length = 50)

username = models.CharField(max\_length = 50, unique = True)

email = models.EmailField(max\_length = 100, unique = True)

phone\_number = models.CharField(max\_length = 10)

# Required fields

date\_joined = models.DateTimeField(auto\_now = True)

last\_login = models.DateTimeField(auto\_now\_add = True)

is\_admin = models.BooleanField(default = False)

is\_staff = models.BooleanField(default = False)

is\_active = models.BooleanField(default = False)

is\_superadmin = models.BooleanField(default = False)

# instead of username we are taking email as login credential

USERNAME\_FIELD = 'email'

# the below fields are required while creating an account

REQUIRED\_FIELDS = [

'username',

'first\_name',

'last\_name',

]

# specifying that we are using the MyAccountManager for all the operations

objects = MyAccountManager()

# in the django pannel we want to show emails instead of account object

def \_\_str\_\_(self):

return self.email

# checking if it is admin then has all the permissions

def has\_perm(self, perm, obj = None):

return self.is\_admin

def has\_module\_perms(self, add\_label):

return True

1. Now go to project’s ‘settings.py’ file to register the custom user model that we currently created. Add the below code there.

# Registering custom user model

AUTH\_USER\_MODEL = 'accounts.Account' # accounts is app name. Accounts is the model name

1. Now register your ‘**Account**’ model to your ‘accounts’ app’s ‘admin.py’ file. Code as follows:

from django.contrib import admin

from .models import Account

# Register your models here.

admin.site.register(Account)

1. Now we need to delete the old database and all old migration files from the project and from both the apps. It will also delete our previous super user. So, we can add new super user with the new custom model.
   1. Delete the ‘db.sqlite3’ file from the root folder
   2. Delete all the files from ‘migrations’ folder except ‘\_\_init\_\_.py’ file from ‘category’ app.
   3. Delete all the files from ‘migrations’ folder except ‘\_\_init\_\_.py’ file from ‘accounts’ app.
2. Run the migration code.

python manage.py makemigrations

python manage.py migrate

1. Now create superuser. It will be created with the new custom use model that we have created.

python manage.py createsuperuser

Output:

$ python manage.py createsuperuser

Email: test@gmail.com

Username: test

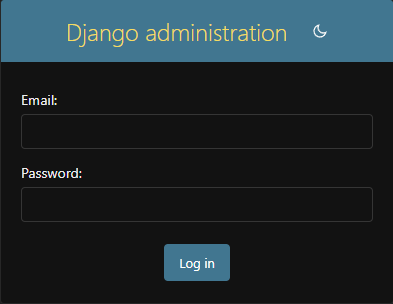
First name: test

Last name: user

Password:

Password (again):

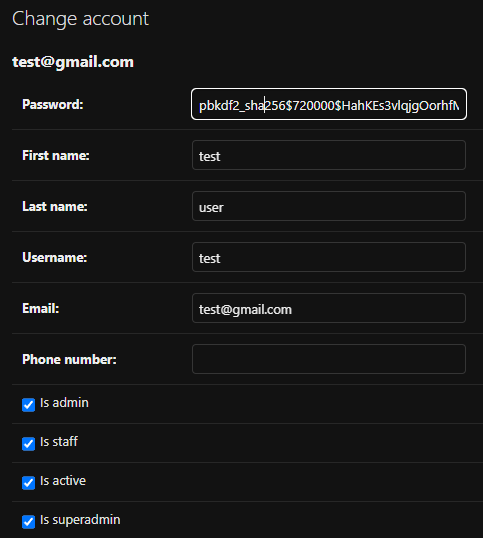
Superuser created successfully.

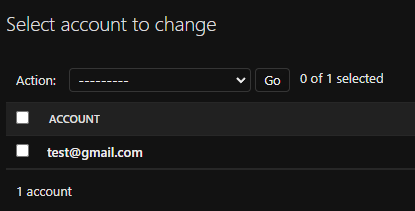
1. Now run server and go to the admin page.

python manage.py runserver

You can see there instead of asking username and password as before now it is asking for email and password for logging in into the Django admin.

1. Now if you login into the Django admin you can see the Account object is showing as only the email and if we open if we can see that the password is editable.





So, we want to change the view of the accounts and also, we want that the passwords can’t be editable.

1. For that we need to change our ‘accounts’ app’s ‘admin.py’ file.

from django.contrib import admin

from .models import Account

from django.contrib.auth.admin import UserAdmin

# Register your models here.

class AccountAdmin(UserAdmin):

list\_display = (

'email',

'first\_name',

'last\_name',

'username',

'date\_joined',

'last\_login',

'is\_staff',

)

# as we are using custom model,

# we need the below code to modify the view

filter\_horizontal = ()

list\_filter = ()

# we want to click on first\_name and last\_name to open the object

list\_display\_links = ('email', 'first\_name', 'last\_name')

# making date fields as readonly

readonly\_fields = ('date\_joined', 'last\_login')

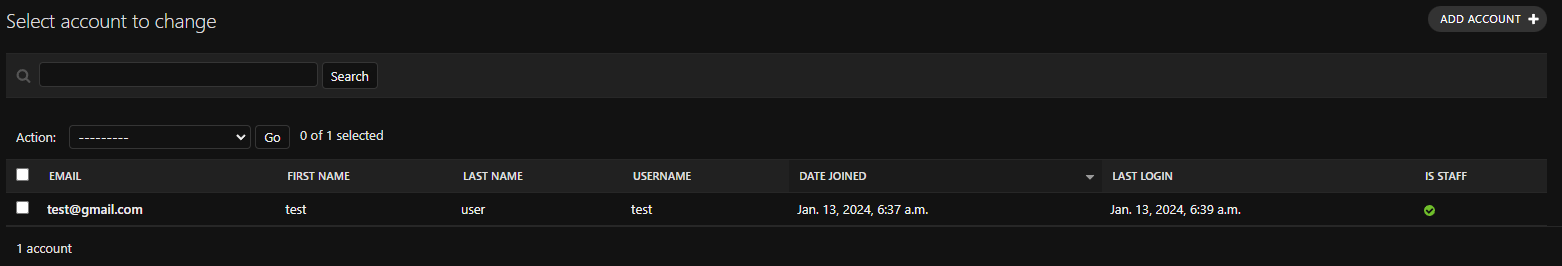
# sort the display list by date\_joined in descending order

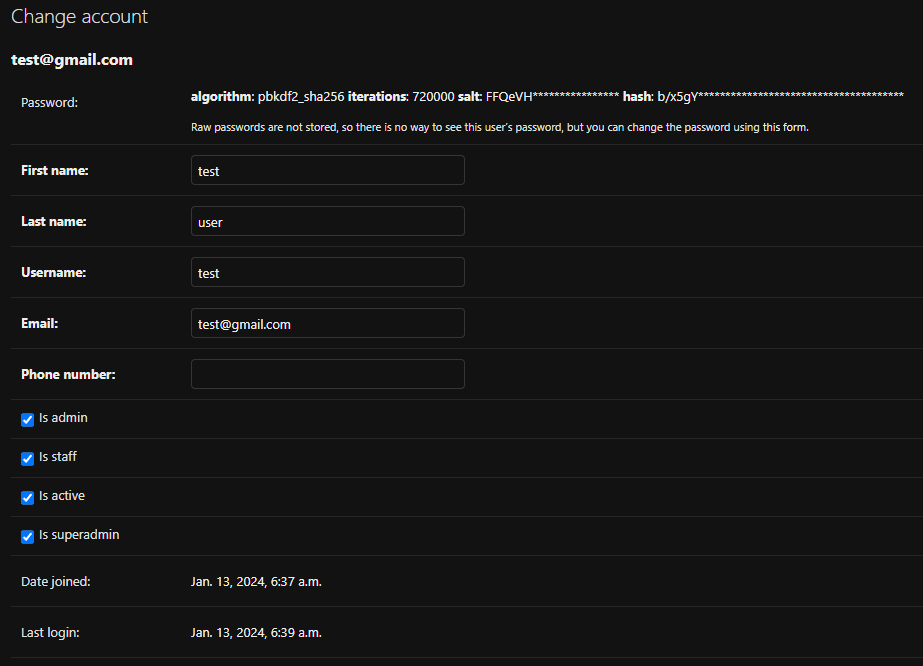
ordering = ('-date\_joined',)

# making password as non-editable

fieldsets = ()

admin.site.register(Account, AccountAdmin)





Configuring Media files:

1. Open project’s ‘settings.py’ file and add the media configuration code as follows:

# Media configuration

MEDIA\_URL = 'media/'

MEDIA\_ROOT = BASE\_DIR /'media'

1. Open project’s ‘urls.py’ file to set media configuration for URLs as well.

from django.contrib import admin

from django.urls import path

from . import views

from django.conf.urls.static import static

from django.conf import settings

urlpatterns = [

path('admin/', admin.site.urls),

path("", views.home, name='home'),

] + static(settings.MEDIA\_URL, document\_root = settings.MEDIA\_ROOT)

1. Now if we add any category with image the image will work fine.

Configuring slug field pre-populated.

When ever we’re adding a new category, we need to manually add the slug field. We need it to be auto generated from the ‘category\_name’. For that we need to do the following changes:

1. Firstly, we need to change the ‘category’ app’s ‘admin.py’ file.

from django.contrib import admin

from .models import Category

# Register your models here.

class CategoryAdmin(admin.ModelAdmin):

prepopulated\_fields = {

'category\_slug' : ('category\_name',)

}

# displaying category\_name and category\_slug

list\_display = ('category\_name', 'category\_slug')

admin.site.register(Category, CategoryAdmin)

Creating app for Store:

1. Go to git bash terminal and type the command to create a new app named ‘store’. Code as follows:

python manage.py startapp store

1. Register your app in the project’s ‘settings.py’ file’s INSTALLED\_APPS list.
2. Now edit your newly created ‘store’ app’s ‘model.py’ to create fields to store all the products in our database. Code as follows:

from django.db import models

from category.models import Category

# Create your models here.

class Product(models.Model):

product\_name = models.CharField(max\_length = 200, unique = True)

product\_slug = models.SlugField(max\_length = 200, unique = True)

description = models.TextField(blank = True)

price = models.IntegerField()

image = models.ImageField(upload\_to = 'photos/products')

stock = models.IntegerField()

is\_available = models.BooleanField(default = True)

# CASCADE means if the category is deleted then

# all products of that category will also be deleted

category = models.ForeignKey(Category, on\_delete = models.CASCADE)

created\_at = models.DateTimeField(auto\_now = True)

updated\_at = models.DateTimeField(auto\_now\_add = True)

def \_\_str\_\_(self):

return self.product\_name

1. Now edit ‘store’ app’s ‘admin.py’ file for :- i) registering our model, ii) ‘product\_slug’ as populated field and iii) custom items displaying list. Code as follows:

from django.contrib import admin

from .models import Product

# Register your models here.

class ProductAdmin(admin.ModelAdmin):

prepopulated\_fields = {

'product\_slug' : ('product\_name',)

}

list\_display = ('product\_name', 'price', 'stock', 'category', 'updated\_at', 'is\_available')

admin.site.register(Product, ProductAdmin)

1. Now make migrations to the database: Run the below code:

python manage.py makemigrations

1. Now do all the migrations to the database. Run the code as follows:

python manage.py migrate

1. Now run the server and add all the products from Django admin panel.

Displaying products from the database to our webpage.

1. Edit project’s ‘views.py’ file to get all products from ‘store’ app’s ‘models.py’ which are available.

from django.shortcuts import render

from store.views import Product

def home(request):

products = Product.objects.all().filter(is\_available = True)

context = {

'products' : products,

}

return render(request, 'index.html', context)

1. Now edit ‘index.html’ file to show all the products from database to webpage.
   1. Keep only one column from the row div and delete all others.
   2. Loop over that one particular row to show all the details from database. Code as follows:

{% for product in products %}

<div class="col-md-3">

<div class="card card-product-grid">

<a href="#" class="img-wrap">

<img src="{{ product.image.url }}" />

</a>

<figcaption class="info-wrap">

<a href="#" class="title">{{ product.product\_name }}</a>

<div class="price mt-1">${{ product.price }}</div>

<!-- price-wrap.// -->

</figcaption>

</div>

</div>

{% endfor %}

Making the store page

1. Redirect ‘store/’ related all urls to the ‘store’ app’s ‘urls.py’. Code as follows:

from django.contrib import admin

from django.urls import include, path

from . import views

from django.conf.urls.static import static

from django.conf import settings

urlpatterns = [

path('admin/', admin.site.urls),

path("", views.home, name='home'),

path("store/", include('store.urls')),

] + static(settings.MEDIA\_URL, document\_root = settings.MEDIA\_ROOT)

1. Go to ‘store’ app and create a new python file named ‘urls.py’. Add the code as follows:

from django.urls import path

from store import views

urlpattern = [

path("", views.store, name="store")

]

1. Create a new folder named ‘store’ inside the ‘templates’ folder and in that create a new html file named ‘store.html’. Copy the template code into the ‘store.html’ file.
2. Goto ‘store’ app’s ‘views.py’ file. Add the following code there.

from django.shortcuts import render

from .models import Product

# Create your views here.

def store(request):

products = Product.objects.all().filter(is\_available = True)

products\_count = products.count()

context = {

'products' : products,

'products\_count' : products\_count,

}

return render(request, 'store/store.html', context)

1. Now modify the ‘store.html’ file to show the product names and its details accordingly.

Making use of slug:

1. Edit your ‘store’ app’s ‘urls.py’ file. Code as follows:

from django.urls import path

from . import views

urlpatterns = [

path("", views.store, name="store"),

path("<slug:category\_slug>/", views.store, name="products\_by\_category"),

]

1. Edit your ‘store’ app’s ‘views.py’ file. Code as follows:

from django.shortcuts import get\_object\_or\_404, render

from category.models import Category

from .models import Product

# Create your views here.

def store(request, category\_slug = None):

categories = None

products = None

# if category\_slug is not None

if category\_slug != None:

# getting category by category\_slug

categories = get\_object\_or\_404(Category, category\_slug = category\_slug)

# getting products by category\_slug

products = Product.objects.filter(category = categories, is\_available = True)

products\_count = products.count()

# if category\_slug is None

else:

products = Product.objects.all().filter(is\_available = True)

products\_count = products.count()

context = {

'products' : products,

'products\_count' : products\_count,

}

return render(request, 'store/store.html', context)

Context\_Processor -> Getting slug links in the menu:

1. Create a new python file named ‘context\_processor.py’ under the ‘category’ app. Add the below code there:

from .models import Category

# context\_processors is a list of dotted Python paths to callables

# that are used to populate the context when a template is rendered

# with a request. These callables take a request object as argument

# and return a dict of items to be merged into the context.

def menu\_links(request):

all\_links = Category.objects.all()

return dict(all\_links = all\_links)

1. Register your context\_processor to the project’s ‘settings.py’ file. Go to project’s ‘settings.py’ file. There go to TEMPLATES -> OPTIONS -> context\_processors and add the function to it like below:

'category.context\_processors.menu\_links',

\*\* Now we can use the menu\_links to anywhere in our apps or in project itself. A context processor is a function that takes the current HttpRequest object as an argument and returns a dictionary of variables that can be made available to all templates.

1. Delete all the categories from the dropdown except one and modify is as follows:

<div class="dropdown-menu">

{% for link in all\_links %}

<a class="dropdown-item" href="#">{{ link.category\_name }}</a>

{% endfor %}

</div>

1. Now make the links to the category list.
2. Edit the href for categories as follows:

<div class="dropdown-menu">

<a class="dropdown-item" href="{% url 'store' %}">All Products</a>

{% for category in all\_links %}

<a class="dropdown-item" href="{{ category.get\_url }}"

>{{ category.category\_name }}</a

>

{% endfor %}

</div>

1. Now go to ‘models.py’ file of your ‘category’ app. And add a new function ‘get\_url’ in the class ‘Category’ which will be responsible for the category links. Code as follows:

def get\_url(self):

# The reverse function allows retrieving url details from

# the url’s.py file through the name value provided.

return reverse('products\_by\_category', args=[self.category\_slug])

Displaying categories in the store page.

1. Go to ‘store/store.html’ file and delete all the list menus except one. And with that code as follows:

<ul class="list-menu">

<li>

<a href="{% url 'store' %}">All Products</a>

</li>

{% for category in all\_links %}

<li>

<a href="{{ category.get\_url }}">{{ category.category\_name }}</a>

</li>

{% endfor %}

</ul>

Single Product details page setup:

1. Goto ‘store’ app’s ‘urls.py’ file. Add the code for ‘category\_slug’. Code as follows:

from django.urls import path

from . import views

urlpatterns = [

path("", views.store, name="store"),

path("<slug:category\_slug>/", views.store, name="products\_by\_category"),

path("<slug:category\_slug>/<slug:product\_slug>/", views.product\_details, name="product\_details"),

]

1. Now go to ‘store’ app’s ‘views.py’ file. Create the function to show the product details. Code as follows:

def product\_details(request, category\_slug, product\_slug):

try:

# category\_\_category\_slug means we are refering cate\_slug of category model.

# double unserscore \_\_ means accessing models propoerty directly.

product = Product.objects.get(category\_\_category\_slug = category\_slug, product\_slug = product\_slug)

except Exception as e:

raise e

context = {

'product' : product,

}

return render(request, 'store/product\_details.html', context)

1. Now create a new html file named ‘product\_details.html’ under the ‘store’ folder inside the ‘templates’ folder.
2. Add the html code there for product details.

Get URL for products:

1. Firstly, go to ‘store’ app’s ‘models.py’ and add a function ‘get\_url’ as follows:

def get\_url(self):

return reverse('product\_details', args=[self.category.category\_slug, self.product\_slug])

1. Now go to ‘store.html’ file and make the links there as:

<figure class="card card-product-grid">

<div class="img-wrap">

<a href="{{ product.get\_url }}" class="title"

><img src="{{ product.image.url }}"

/></a>

</div>

<!-- img-wrap.// -->

<figcaption class="info-wrap">

<div class="fix-height">

<a href="{{ product.get\_url }}" class="title"

>{{ product.product\_name }}</a

>

<div class="price-wrap mt-2">

<span class="price">${{ product.price }}</span>

<del class="price-old">$1980</del>

</div>

<!-- price-wrap.// -->

</div>

<a href="#" class="btn btn-block btn-primary">Add to cart </a>

</figcaption>

</figure>

1. Now go to ‘index.html’ file and make the links as follows:

<div class="card card-product-grid">

<a href="{{ product.get\_url }}" class="img-wrap">

<img src="{{ product.image.url }}" />

</a>

<figcaption class="info-wrap">

<a href="{{ product.get\_url }}" class="title">{{ product.product\_name }}</a>

<div class="price mt-1">${{ product.price }}</div>

<!-- price-wrap.// -->

</figcaption>

</div>

Adding ‘carts’ functionality:

1. Create new app named ‘carts’ from the terminal. Type the code as follows:

python manage.py startapp carts

1. Register your app in your project’s ‘settings.py’ file. Go to INSTALLED\_APPS and add the name of your newly created app there. Here ‘carts’.
2. Now register the urlpatterns for the ‘carts’ app to the project’s ‘url.py’ file.
   1. Create a new python file named ‘urls.py’ inside the ‘carts’ app.
   2. Go to project’s ‘urls.py’ file and add the path for the ‘carts’ app’s ‘urls.py’ as follows:

urlpatterns = [

path('admin/', admin.site.urls),

path('', views.home, name='home'),

path('store/', include('store.urls')),

path('cart/', include('carts.urls')),

] + static(settings.MEDIA\_URL, document\_root = settings.MEDIA\_ROOT)

1. Now go to your app’s ‘urls.py’ file for configuring urlpatterns. Code as follows:

from django.urls import path

from . import views

urlpatterns = [

path("", views.cart, name="cart"),

]

1. Now go to your app’s ‘views.py’ file to create the view for the url. Code as follows:

from django.shortcuts import render

# Create your views here.

def cart (request):

return render(request, 'store/cart.html')

1. Now create a new html file named ‘cart.html’ under the ‘store’ folder inside the ‘templates’ folder. And code accordingly.

Adding ‘cart’ tables to the Data Base:

1. Go to ‘cart’ app’s ‘models.py’ file and add the code as follows:

from django.db import models

from store.views import Product

# Create your models here.

class Cart(models.Model):

cart\_id = models.CharField(max\_length = 100, blank = True)

date\_added = models.DateField(auto\_now\_add = True)

def \_\_str\_\_(self):

return self.cart\_id

class CartItem(models.Model):

product = models.ForeignKey(Product, on\_delete = models.CASCADE)

cart = models.ForeignKey(Cart, on\_delete = models.CASCADE)

quantity = models.IntegerField()

is\_active = models.BooleanField(default = True)

def \_\_str\_\_(self):

return self.product.product\_name

1. Now go to your ‘store’ app’s ‘admin.py’ file and register both the newly created tables as follows:

from django.contrib import admin

from .models import Cart, CartItem

# Register your models here.

admin.site.register(Cart)

admin.site.register(CartItem)

1. Now make the migrations to the Data Base. Code as follows:

python manage.py makemigrations

1. Now migrate all the changes to the database. Code as follows:

python manage.py migrate

Adding add to cart functionality and session key:

1. Go to your ‘cart’ app’s ‘views.py’ file and edit the code as follows:

from django.shortcuts import render, redirect

from store.models import Product

from .models import Cart, CartItem

# Create your views here.

# getting cart\_id by session key using this provate function

def \_cart\_id(request):

cart = request.session.session\_key

# if there is no cart\_id then creating a new one

if not cart:

cart = request.session.create()

# returning the cart

return cart

def add\_to\_cart(request, product\_id):

# getting the product by its id

product = Product.objects.get(id = product\_id)

# getting the cart by using the \_cart\_id() using the session key

try:

cart = Cart.objects.get(cart\_id = \_cart\_id(request))

# if cart does not exists then creating a new one

except Cart.DoesNotExists:

cart = Cart.objects.create(

cart\_id = \_cart\_id(request)

)

# saving the cart

cart.save()

# adding items to existing cart

try:

# getting the item with the which product and which cart

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

# incrementing the quantity

cart\_item.quantity += 1

# saving the cart\_item

cart\_item.save()

# if no cart is there then creating a new one

except CartItem.DoesNotExist:

cart\_item = CartItem.objects.create(

product = product,

cart = cart,

quantity = 1

)

# saving the cart item

cart\_item.save()

return redirect('cart')

def cart (request):

return render(request, 'store/cart.html')

1. Now go to ‘cart’ app’s ‘urls.py’ file and add the url path for add to cart as follows:

from django.urls import path

from . import views

urlpatterns = [

path("", views.cart, name="cart"),

path("add\_to\_cart/<int:product\_id>", views.add\_to\_cart, name='add\_to\_cart'),

]

1. Now go to the ‘product\_details.html’ page under the ‘store’ folder under the ‘templates’ folder. Now replace the link from the <a> tag of the ‘Add to cart’ link as follows:

<a href="{% url 'add\_to\_cart' product.id %}" class="btn btn-primary">

<span class="text">Add to cart</span>

<i class="fas fa-shopping-cart"></i>

</a>

Cart view for getting Items, Total & Quantity:

1. Go to ‘cart’ app’s ‘views.py’ file and modify the cart method as follows:

def cart (request, total = 0, quantity = 0, cart\_items = None):

try:

# getting the cart objects from the session key with the private function

cart = Cart.objects.get(cart\_id = \_cart\_id(request))

# getting all the cart items depending on the cart

cart\_items = CartItem.objects.filter(cart = cart, is\_active = True)

# calculating the total price and quantity by looping over the cart items

for cart\_item in cart\_items:

total += (cart\_item.product.price \* cart\_item.quantity)

quantity += cart\_item.quantity

except Cart.DoesNotExist:

# simply do nothing

pass

context = {

'cart\_items' : cart\_items,

'total' : total,

'quantity' : quantity,

}

return render(request, 'store/cart.html', context)

1. Now edit ‘cart.html’ file under the ‘store’ folder inside the ‘templates’ folder. Loop over the item list to show the items and its details on the cart page.
2. For the calculating the sub-total price of each item depending on its price and quantity, create a new function named ‘sub\_total’ under the class ‘CartItems’ inside the ‘cart’ app’s ‘models.py’ file. The function code as follows:

def sub\_total(self):

return self.product.price \* self.quantity

1. Now go back to ‘cart.html’ file and for calculating the sub-total price of each item add the code as follows:

{{ cart\_item.sub\_total }}

Implementing the Total, Tax and Grand Total:

1. For calculating the Tax and Grand total go to ‘cart’ app’s ‘views.py’ file and edit the ‘cart’ function as follows:

def cart (request, total = 0, quantity = 0, cart\_items = None):

try:

# getting the cart objects from the session key with the private function

cart = Cart.objects.get(cart\_id = \_cart\_id(request))

# getting all the cart items depending on the cart

cart\_items = CartItem.objects.filter(cart = cart, is\_active = True)

# calculating the total price and quantity by looping over the cart items

for cart\_item in cart\_items:

total += (cart\_item.product.price \* cart\_item.quantity)

quantity += cart\_item.quantity

tax = total \* 0.02 # tax is 2%

grand\_total = total + tax

except Cart.DoesNotExist:

# simply do nothing

pass

context = {

'cart\_items' : cart\_items,

'total' : total,

'quantity' : quantity,

'tax' : tax,

'grand\_total' : grand\_total,

}

return render(request, 'store/cart.html', context)

1. Now go back to ‘cart.html’ file and display the Total, Tax and Grand total as follows:

<dl class="dlist-align">

<dt>Total price:</dt>

<dd class="text-right">${{ total }}</dd>

</dl>

<dl class="dlist-align">

<dt>Tax:</dt>

<dd class="text-right">${{ tax }}</dd>

</dl>

<dl class="dlist-align">

<dt>Grand Total:</dt>

<dd class="text-right text-dark b">

<strong>${{ grand\_total }}</strong>

</dd>

</dl>

Adding the increment, decrement and remove functionality to the cart page:

1. Increment =>

Go to ‘cart.html’ page and for the increment button convert the button tag to anchor tag and add href attribute as follows:

<a

href="{% url 'add\_to\_cart' cart\_item.product.id %}"

class="btn btn-light"

type="button"

id="button-plus"

></a>

1. Decrement =>

* Go to ‘cart’ app’s ‘views.py’ file and add a new function for this functionality as follows:

def remove\_from\_cart(request, product\_id):

cart = Cart.objects.get(cart\_id = \_cart\_id(request))

product = get\_object\_or\_404(Product, id = product\_id)

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

# if more than one items then decrease the count

if cart\_item.quantity > 1:

cart\_item.quantity -= 1

cart\_item.save()

# if only one item then delete

else:

cart\_item.delete()

return redirect('cart')

* Now go to ‘cart’ app’s ‘urls.py’ file to register the url. Code as follows:

urlpatterns = [

path("", views.cart, name="cart"),

path("add\_to\_cart/<int:product\_id>", views.add\_to\_cart, name='add\_to\_cart'),

path("remove\_from\_cart/<int:product\_id>", views.remove\_from\_cart, name='remove\_from\_cart'),

]

* Now go to ‘cart.html’ file and for the decrement button, remove the button tag with anchor tag and add href as follows:

<a

href="{% url 'remove\_from\_cart' cart\_item.product.id %}"

class="btn btn-light"

type="button"

id="button-minus"

></a>

1. Remove Button =>

* Go to ‘cart’ app’s ‘views.py’. Add a new function as follows:

# Remove whole item from cart

def remove\_cart\_item(request, product\_id):

cart = Cart.objects.get(cart\_id = \_cart\_id(request))

product = get\_object\_or\_404(Product, id = product\_id)

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

cart\_item.delete()

return redirect('cart')

* Now go to ‘cart’ app’s ‘urls.py’ add a new url pattern for removing the whole item as follows:

urlpatterns = [

path("", views.cart, name="cart"),

path("add\_to\_cart/<int:product\_id>", views.add\_to\_cart, name='add\_to\_cart'),

path("remove\_from\_cart/<int:product\_id>", views.remove\_from\_cart, name='remove\_from\_cart'),

path("remove\_cart\_item/<int:product\_id>", views.remove\_cart\_item, name='remove\_cart\_item'),

]

* Lastly, go to ‘cart.html’ file and add the url to the href of the remove button as follows:

<a

href="{% url 'remove\_cart\_item' cart\_item.product.id %}"

class="btn btn-danger"

>

Remove</a

>

Adding message for empty shopping cart:

1. Go to ‘cart.html’ and add if there is not cart item then show Cart is Empty message else wrap every line of code responsible for showing the cart items inside the else block. Code as follows:

{% if not cart\_items %}

<div class="text-center">

<h1>Your shopping cart is empty</h1>

<br />

<a href="{% url 'store' %}" class="btn btn-primary">Continue Shopping</a>

</div>

{% else %}

<!-- Code for displaying the items if cart is not empty -->

{% endif %}

Fixing Add to Cart links:

1. Fixing the ‘Add to cart’ link on the store page itself. Go to ‘store.html’ file and for Add to Cart add the link as follows:

<a href="{% url 'add\_to\_cart' product.id %}" class="btn btn-block btn-primary"

>Add to cart

</a>

1. Fixing the product name link in the cart page. Go to ‘cart.html’ and for the product name link add the link as follows:

<a href="{{ cart\_item.product.get\_url }}" class="title text-dark"

>{{ cart\_item.product.product\_name }}</a

>

1. Fixing the continue shopping button link on the cart page. Go to ‘cart.html’ file and for the Continue Shopping button add the link to the ‘store.html’ file as follows:

<a href="{% url 'store' %}" class="btn btn-light btn-block"

>Continue Shopping</a

>

1. Fixing the cart icon link on the navigation bar. Go to ‘nav-bar.html’ and find the cart icon. There code as follows:

<a href="{% url 'cart' %}" class="widget-header pl-3 ml-3">

<div class="icon icon-sm rounded-circle border">

<i class="fa fa-shopping-cart"></i>

</div>

<span class="badge badge-pill badge-danger notify">0</span>

</a>

Check if product is added to Cart, then show Added to Cart, else show Add to Cart:

1. Go to ‘store’ app’s views.py file and add some lines of code in the ‘product\_details’ function. Code as follows:

from carts.models import CartItem

from carts.views import \_cart\_id

def product\_details(request, category\_slug, product\_slug):

try:

# category\_\_category\_slug means we are refering cate\_slug of category model.

# double unserscore \_\_ means accessing models propoerty directly.

product = Product.objects.get(category\_\_category\_slug = category\_slug, product\_slug = product\_slug)

in\_cart = CartItem.objects.filter(cart\_\_cart\_id = \_cart\_id(request), product = product).exists()

except Exception as e:

raise e

context = {

'product' : product,

'in\_cart' : in\_cart,

}

return render(request, 'store/product\_details.html', context)

1. Now go to ‘product\_details.html’ page and modify the code where Add to Cart option is. Code as follows:

<!-- if not in stock -->

{% if product.stock <= 0 %}

<h3 class="text-danger">Out of stock</h3>

<!-- if in stock -->

{% else %}

<!-- -->

{% if in\_cart %}

<a href="#" class="btn btn-success">

<span class="text">Added to Cart</span>

<i class="fas fa-check"></i>

</a>

<a href="{% url 'cart' %}" class="btn btn-outline-primary">

<span class="text">View Cart</span>

<i class="fas fa-eye"></i>

</a>

{% else %}

<a href="{% url 'add\_to\_cart' product.id %}" class="btn btn-primary">

<span class="text">Add to Cart</span>

<i class="fas fa-shopping-cart"></i>

</a>

{% endif %}

<!-- end of nested if -->

{% endif %}

Showing the cart item count using context processor:

1. Go to ‘carts’ app and create a new python file named ‘context\_processor.py’. Code as follows:

from .models import Cart, CartItem

from .views import \_cart\_id

def counter(request):

cart\_count = 0

# if admin login then nothing

if 'admin' in request.path:

return {}

# if customer page then

else:

try:

# getting the cart

cart = Cart.objects.filter(cart = \_cart\_id(request))

# getting all the cart items

cart\_items = CartItem.objects.all()

# looping through the items in cart

for cart\_item in cart\_items:

# incrementing the counter

cart\_count += cart\_item.quantity

# if no cart is there

except Cart.DoesNotExist:

cart\_count = 0

return dict(cart\_count = cart\_count)

1. Now go to project’s ‘settings.py’ file to register the newly created context\_processor. Go to ‘settings.py’ there find the TEMPLATES -> OPTIONS -> context\_preprocessors add the context\_preprocessor of ‘carts’ app there as:

'carts.context\_processors.counter',

1. Now go to ‘nav-bar.html’ where out cart count is showing and replace the 0 with the cart\_count as follows:

<a href="{% url 'cart' %}" class="widget-header pl-3 ml-3">

<div class="icon icon-sm rounded-circle border">

<i class="fa fa-shopping-cart"></i>

</div>

<span class="badge badge-pill badge-danger notify">{{ cart\_count }}</span>

</a>

Pagination:

1. Go to ‘store’ app’s ‘views.py’ file and firstly import the Paginator as follows:

# imporing paginator

from django.core.paginator import EmptyPage, PageNotAnInteger, Paginator

1. Then in the ‘views.py’ file slightly modify the ‘store’ method as follows:

def store(request, category\_slug = None):

categories = None

products = None

# if category\_slug is not None

if category\_slug != None:

# getting category by category\_slug

categories = get\_object\_or\_404(Category, category\_slug = category\_slug)

# getting products by category\_slug

products = Product.objects.filter(category = categories, is\_available = True)

# we are showing 1 product in one page

paginator = Paginator(products, 1)

# getting the page from url using GET request, like, page\_url/?page=2

page = request.GET.get('page')

# with the page number showing the products

paged\_products = paginator.get\_page(page)

products\_count = products.count()

# if category\_slug is None

else:

products = Product.objects.all().filter(is\_available = True)

# we are showing 3 products in one page

paginator = Paginator(products, 3)

# getting the page from url using GET request, like, page\_url/?page=2

page = request.GET.get('page')

# with the page number showing the products

paged\_products = paginator.get\_page(page)

products\_count = products.count()

context = {

# 'products' : products,

# instead of all products we are only passing the specified number of products

'products' : paged\_products,

'products\_count' : products\_count,

}

return render(request, 'store/store.html', context)

1. Now implement this paginator functionality in the html template. Go to ‘store.html’ file. Change the nav (pagination) section as follows:

<nav class="mt-4" aria-label="Page navigation sample">

<!-- If we have more than one pages then showing the pagination -->

{% if products.has\_other\_pages %}

<ul class="pagination">

<!-- Previous page button logic -->

{% if products.has\_previous %}

<!-- if there is previous page then show button -->

<li class="page-item">

<a class="page-link" href="?page={{ products.previous\_page\_number }}"

>Previous</a

>

</li>

<!-- if there is no previous page then disable button -->

{% else %}

<li class="page-item disabled">

<a class="page-link" href="#">Previous</a>

</li>

{% endif %}

<!-- Page number buttons logic -->

{% for i in products.paginator.page\_range %}

<!-- if the current page then show as active -->

{% if products.number == i %}

<li class="page-item active">

<a class="page-link" href="#">{{ i }}</a>

</li>

<!-- if any other page then show page number -->

{% else %}

<li class="page-item">

<a class="page-link" href="?page={{i}}">{{ i }}</a>

</li>

{% endif %}

<!-- end of for loop -->

{% endfor %}

<!-- next page button logic -->

{% if products.has\_next %}

<!-- if there is next page then show button -->

<li class="page-item">

<a class="page-link" href="?page={{ products.next\_page\_number }}">Next</a>

</li>

{% else %}

<!-- if there is no next page then disable button -->

<li class="page-item disabled">

<a class="page-link" href="#">Next</a>

</li>

{% endif %}

</ul>

{% endif %}

</nav>

Implementing the search functionality

1. Firstly, go to the ‘nav-bar.html’ and find the search form. Edit the form as follows:

<form action="{% url 'search' %}" class="search" method="GET">

<div class="input-group w-100">

<input

type="text"

class="form-control"

style="width: 60%"

placeholder="Search"

name="keyword"

/>

<div class="input-group-append">

<button class="btn btn-primary" type="submit">

<i class="fa fa-search"></i>

</button>

</div>

</div>

</form>

1. Now, we need to create the urlpattern for the search functionality. For that go to ‘store’ app’s ‘urls.py’ file and add a new url pattern for search like below:

urlpatterns = [

path("", views.store, name="store"),

path("category/<slug:category\_slug>/", views.store, name="products\_by\_category"),

path("category/<slug:category\_slug>/<slug:product\_slug>/", views.product\_details, name="product\_details"),

path("search/", views.search, name="search"),

]

\*\* Here we added ‘category/’ before the ‘products\_by\_category’ and ‘product\_details’ urls because is we left it as normal the search url will be treated as one of the category slugs and will not find any category named search. So, for that we modified these two urls like this.

1. Now implement that search functionality in the ‘store’ app’s ‘views.py’ file. Code as follows:

# Importing Q

from django.db.models import Q

def search(request):

# if keyword is in the url

if 'keyword' in request.GET:

# storing the keyword

keyword = request.GET['keyword']

# if keyword is not blank

if keyword:

# here description\_\_icontains means it will find the keyword in the full description

# products = Product.objects.filter(description\_\_icontains = keyword)

# in django filter , means AND operation and for OR in we need to use QuerySet Q like below

products = Product.objects.filter(Q(description\_\_icontains = keyword) | Q(product\_name\_\_icontains = keyword))

products\_count = products.count()

context = {

'products' : products,

'products\_count' : products\_count,

}

return render(request, 'store/store.html', context)

1. Now change the ‘store.html’ page so that is any one is searching something then show message as ‘Search Result’ and if visiting the store then show message as ‘Our Store’. Find the area where ‘Our Store’ is written and then change it as follows:

<div class="container">

{% if 'search' in request.path %}

<h2 class="title-page">Search Result</h2>

{% else %}

<h2 class="title-page">Our Store</h2>

{% endif %}

</div>

1. Now lastly, if there is no result found then we want to show a message as ‘No result found’ in the store page. For that go to ‘store.html’ page, find where we are looping through the products and modify it as follows:

<!-- if any product is found then -->

{% if products %}

<!-- Looping through the products -->

{% for product in products %}

<div class="col-md-4">

<figure class="card card-product-grid">

<div class="img-wrap">

<a href="{{ product.get\_url }}" class="title"

><img src="{{ product.image.url }}"

/></a>

</div>

<!-- img-wrap.// -->

<figcaption class="info-wrap">

<div class="fix-height">

<a href="{{ product.get\_url }}" class="title"

>{{ product.product\_name }}</a

>

<div class="price-wrap mt-2">

<span class="price">${{ product.price }}</span>

<del class="price-old">$1980</del>

</div>

<!-- price-wrap.// -->

</div>

<a href="{{ product.get\_url }}" class="btn btn-block btn-primary"

>View Details

</a>

</figcaption>

</figure>

</div>

{% endfor %}

<!-- if no product is not found -->

{% else %}

<div>

<h2>No result found. Try again.</h2>

</div>

{% endif %}

Product Variations:

1. Go to ‘product-details.html’ page and wrap the article tag where the product details are showing, into a form tag, whose method will be ‘POST’ and action will be ‘{% url 'add\_to\_cart' product.id %}’. Also add csrf\_token as it is POST request. Code as follows:

<form action="{% url 'add\_to\_cart' product.id %}" method="GET">

{% csrf\_token %}

<!-- Code for showing product details -->

</form>

1. Now change the anchor tag for ‘Add to Cart’ to a button. Code as follows:

<button type="submit" class="btn btn-primary">

<span class="text">Add to Cart</span>

<i class="fas fa-shopping-cart"></i>

</button>

1. Now change the radio options for color and size for the product variation to dropdown options. Currently we are using static options. Later we will convert it into dynamic. Code as follows:

<div class="row">

<div class="item-option-select">

<h6>Choose Color</h6>

<select name="color" class="form-control">

<option selected disabled hidden>Select color</option>

<option value="red">Red</option>

<option value="yellow">Yellow</option>

<option value="green">Green</option>

</select>

</div>

</div>

<div class="row">

<div class="item-option-select">

<h6>Select Size</h6>

<select name="size" class="form-control">

<option selected disabled hidden>Select size</option>

<option value="small">Small</option>

<option value="medium">Medium</option>

<option value="large">Large</option>

</select>

</div>

</div>

Creating Product Variation Model:

1. Firstly, creating the model. Go to ‘store’ app’s ‘models.py’ file. Add the product variation class below the ‘Product’ class as follows:

# for dynamically fetching color and size

class VariationManager(models.Manager):

def colors(self):

return super(VariationManager, self).filter(variation\_category = 'color', is\_active = True)

def sizes(self):

return super(VariationManager, self).filter(variation\_category = 'size', is\_active = True)

variation\_category\_choice = (

('color', 'color'),

('size', 'size')

)

class Variation(models.Model):

product = models.ForeignKey(Product, on\_delete = models.CASCADE)

variation\_category = models.CharField(max\_length = 100, choices = variation\_category\_choice)

variation\_value = models.CharField(max\_length = 100)

is\_active = models.BooleanField(default = True)

created\_date = models.DateTimeField(auto\_now = True)

# registering VariationManager

objects = VariationManager()

1. Now make the migrations to the Database. Code as follows:

python manage.py makemigrations

python manage.py migrate

1. Register your model to the admin. Go to ‘store’ app’s ‘admin.py’ file and code as follows:

from django.contrib import admin

from .models import Product, Variation

# Register your models here.

class ProductAdmin(admin.ModelAdmin):

prepopulated\_fields = {

'product\_slug' : ('product\_name',)

}

list\_display = ('product\_name', 'price', 'stock', 'category', 'updated\_at', 'is\_available')

class VariationAdmin(admin.ModelAdmin):

list\_display = ('product', 'variation\_category', 'variation\_value', 'is\_active')

# making is\_active field editable directly from admin interface

list\_editable = ('is\_active',)

# Adding filer on the admin pannel

list\_filter = ('product', 'variation\_category', 'variation\_value')

admin.site.register(Product, ProductAdmin)

admin.site.register(Variation, VariationAdmin)

1. Now go to the ‘product\_details.html’ and convert the color and size variation section as follows:

<div class="row">

<div class="item-option-select">

<h6>Choose Color</h6>

<select name="color" class="form-control">

<option selected disabled hidden>Select color</option>

{% for i in product.variation\_set.colors %}

<option value="{{ i.variation\_value | lower }}">

{{ i.variation\_value | capfirst }}

</option>

{% endfor %}

</select>

</div>

</div>

<!-- row.// -->

<div class="row">

<div class="item-option-select">

<h6>Select Size</h6>

<select name="size" class="form-control">

<option selected disabled hidden>Select size</option>

{% for i in product.variation\_set.sizes %}

<option value="{{ i.variation\_value | lower }}">

{{ i.variation\_value | capfirst }}

</option>

{% endfor %}

</select>

</div>

</div>

Variation model for variation category:

1. Goto the ‘views.py’ file in the ‘carts’ app. And modify the add\_to\_cart function as follows:

def add\_to\_cart(request, product\_id):

# getting the product by its id

product = Product.objects.get(id = product\_id)

product\_variation = []

if request.method == 'POST':

for item in request.POST:

key = item

value = request.POST[key]

try:

variation = Variation.objects.get(product = product, variation\_category\_\_iexact = key, variation\_value\_\_iexact = value)

product\_variation.append(variation)

except:

pass

try:

# getting the cart by using the \_cart\_id() using the session key

cart = Cart.objects.get(cart\_id = \_cart\_id(request))

# if cart does not exists then creating a new one

except Cart.DoesNotExist:

cart = Cart.objects.create(

cart\_id = \_cart\_id(request)

)

# saving the cart

cart.save()

# adding items to existing cart

try:

# getting the item with the which product and which cart

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

# incrementing the quantity

cart\_item.quantity += 1

# saving the cart\_item

cart\_item.save()

# if no cart is there then creating a new one

except CartItem.DoesNotExist:

cart\_item = CartItem.objects.create(

product = product,

cart = cart,

quantity = 1

)

# saving the cart item

cart\_item.save()

return redirect('cart')

1. Goto ‘models.py’ file in the ‘store’ app. And modify the code as follows:

class Variation(models.Model):

product = models.ForeignKey(Product, on\_delete = models.CASCADE)

variation\_category = models.CharField(max\_length = 100, choices = variation\_category\_choice)

variation\_value = models.CharField(max\_length = 100)

is\_active = models.BooleanField(default = True)

created\_date = models.DateTimeField(auto\_now = True)

# registering VariationManager

objects = VariationManager()

def \_\_str\_\_(self):

return self.variation\_value

1. Now go to ‘models.py’ file in the ‘carts’ app. Modify the code as follows:

from store.models import Product, Variation

class CartItem(models.Model):

product = models.ForeignKey(Product, on\_delete = models.CASCADE)

# Many-To-Many Relationship : Multiple records in a table are associated with multiple records in another table.

variations = models.ManyToManyField(Variation, blank = True)

cart = models.ForeignKey(Cart, on\_delete = models.CASCADE)

quantity = models.IntegerField()

is\_active = models.BooleanField(default = True)

def sub\_total(self):

return self.product.price \* self.quantity

def \_\_unicode\_\_(self):

return self.product

1. Now migrate the changes in the database. As follows:

python manage.py makemigrations

python manage.py migrate

1. Now go to the ‘admin.py’ file of the ‘carts’ app. Code as follows:

from django.contrib import admin

from .models import Cart, CartItem

# Register your models here.

class CartAdmin(admin.ModelAdmin):

list\_display = ('cart\_id', 'date\_added')

class CartItemAdmin(admin.ModelAdmin):

list\_display = ('product', 'cart', 'quantity', 'is\_active')

admin.site.register(Cart, CartAdmin)

admin.site.register(CartItem, CartItemAdmin)

Adding Variation in Cart Item:

1. Firstly, go to the ‘views.py’ file of ‘carts’ app. Modify the getting cart item code as follows:

# GETTING THE CART-ITEM

# adding items to existing cart

try:

# getting the item with the which product and which cart

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

# incrementing the quantity

cart\_item.quantity += 1

# adding variation to the cart-item

if product\_variation:

for item in product\_variation:

cart\_item.variations.add(item)

# saving the cart\_item

cart\_item.save()

# if no cart is there then creating a new one

except CartItem.DoesNotExist:

cart\_item = CartItem.objects.create(

product = product,

cart = cart,

quantity = 1

)

# adding variation to the cart-item

if product\_variation:

for item in product\_variation:

cart\_item.variations.add(item)

# saving the cart item

cart\_item.save()

return redirect('cart')

1. Now go to the ‘product\_details.html’ and remove the code for if item is already in the cart. And only place the code for add\_to\_cart. Code as follows:

{% if product.stock <= 0 %}

<h3 class="text-danger">Out of stock</h3>

<!-- if in stock -->

{% else %}

<button type="submit" class="btn btn-primary">

<span class="text">Add to Cart</span>

<i class="fas fa-shopping-cart"></i>

</button>

{% endif %}

1. Now if we add new product with new variations, it will add the variation in the existing product not as a new. For that we need to modify the ‘views.py’ file of the ‘carts’ app for clearing the existing variation.

# GETTING THE CART-ITEM

# adding items to existing cart

try:

# getting the item with the which product and which cart

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

# incrementing the quantity

cart\_item.quantity += 1

# adding variation to the cart-item

if product\_variation:

cart\_item.variations.clear()

for item in product\_variation:

cart\_item.variations.add(item)

# saving the cart\_item

cart\_item.save()

# if no cart is there then creating a new one

except CartItem.DoesNotExist:

cart\_item = CartItem.objects.create(

product = product,

cart = cart,

quantity = 1

)

# adding variation to the cart-item

if product\_variation:

cart\_item.variations.clear()

for item in product\_variation:

cart\_item.variations.add(item)

# saving the cart item

cart\_item.save()

return redirect('cart')

1. Now to show products with its variations, modify the ‘cart.html’ file. Modify the section where we are showing the variation\_category (here, size & color) and variation\_value. Code as follows:

<p class="text-muted small">

<!-- Checking if item variation -->

{% if cart\_item.variations.all %}

<!-- iterating over the item variation -->

{% for item in cart\_item.variations.all %}

{{ item.variation\_category | capfirst }} :

{{ item.variation\_value }} <br />

<!-- end of for loop -->

{% endfor %}

<!-- end of if condition -->

{% endif %}

</p>

Grouping Cart Item Variations

1. Whenever we are adding new item variation for the same product, it will add an extra count but replace the existing variation with the new variation. But we need if existing variation the count will be incremented but if new variation then we add it as a new. For that we need to modify the ‘**GETTING THE CART-ITEM**’ section in the ‘**add\_to\_cart()**’ method in the ‘**views.py**’ file in the ‘**carts**’ app. Code as follows:

# GETTING THE CART-ITEM

# checking if cart item is already present in cart or not

is\_cart\_item\_exists = CartItem.objects.filter(product = product, cart = cart).exists()

# if already present

if is\_cart\_item\_exists:

cart\_item = CartItem.objects.filter(product = product, cart = cart)

existing\_variation\_list = []

id\_list = []

# getting all the existing variations

for item in cart\_item:

existing\_variation = item.variations.all()

existing\_variation\_list.append(list(existing\_variation))

id\_list.append(item.id)

# if variation is in the exsiting variation list

if product\_variation in existing\_variation\_list:

# getting current variation in existing list

curr\_item\_inex = existing\_variation\_list.index(product\_variation)

curr\_item\_id = id\_list[curr\_item\_inex]

# getting item

item = CartItem.objects.get(product = product, id = curr\_item\_id)

item.quantity += 1

item.save()

# if variation is not in the exsiting variation list

else:

item = CartItem.objects.create(product = product, quantity = 1, cart = cart)

if len(product\_variation) > 0:

item.variations.clear()

item.variations.add(\*product\_variation)

item.save()

# if no cart is there then creating a new one

else:

cart\_item = CartItem.objects.create(

product = product,

cart = cart,

quantity = 1

)

if len(product\_variation) > 0:

cart\_item.variations.clear()

cart\_item.variations.add(\*product\_variation)

cart\_item.save()

return redirect('cart')

Cart Increment/Decrement/Remove with variations:

1. Now, our add new and existing variations to the cart is working. But in the cart page when ever we are trying to increment the count, it is adding new item with no variations at all. And whenever we are trying to decrement the count or delete the item from cart it is showing an error **MultipleObjectsReturned**. Because it is unable to uniquely identify the cart items uniquely. To resolve this error, we need to modify our ‘cart.html’ file.

**Cart Increment with variation**:

1. As our ‘add\_to\_cart’ method is checking for ‘POST’ request, we are going to change the anchor tag, for plus button, with a button inside a form with POST method. Code as follows:

<div class="input-group-append">

<form action="{% url 'add\_to\_cart' cart\_item.product.id %}" method="post">

{% csrf\_token %}

<!-- looping over all the variations -->

{% for item in cart\_item.variations.all %}

<input

type="hidden"

name="{{ item.variation\_category | lower }}"

value="{{ item.variation\_value | capfirst }}"

/>

{% endfor %}

<button class="btn btn-light" type="submit" id="button-plus">

<i class="fa fa-plus"></i>

</button>

</form>

</div>

**Cart Decrement with variation:**

1. We need to pass an extra parameter to our ‘remove\_from\_cart’ method named ‘cart\_item\_id**’** which will be the id for the cart item. It will uniquely identify the cart item and this will resolve our **MultipleObjectsReturned** error. For that firstly we need to modify our ‘**cart.html**’ file for the minus button as follows:

<div class="input-group-prepend">

<a

href="{% url 'remove\_from\_cart' cart\_item.product.id cart\_item.id %}"

class="btn btn-light"

type="button"

id="button-minus"

>

<i class="fa fa-minus"></i>

</a>

</div>

1. Now we need to modify the url for accepting the new value. For that go to ‘urls.py’ file inside the ‘carts’ app. Code as follows:

urlpatterns = [

path("", views.cart, name="cart"),

path("add\_to\_cart/<int:product\_id>", views.add\_to\_cart, name='add\_to\_cart'),

path("remove\_from\_cart/<int:product\_id>/<int:cart\_item\_id>", views.remove\_from\_cart, name='remove\_from\_cart'),

path("remove\_cart\_item/<int:product\_id>", views.remove\_cart\_item, name='remove\_cart\_item'),

]

1. Now lastly, we need to modify the ‘remove\_from\_cart’ method in the ‘views.py’ file inside the ‘carts’ app to find the cart\_item with an additional property id. The we will wrap the code inside a try...except block. Code as follows:

def remove\_from\_cart(request, product\_id, cart\_item\_id):

try:

cart = Cart.objects.get(cart\_id = \_cart\_id(request))

product = get\_object\_or\_404(Product, id = product\_id)

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

# if more than one items then decrease the count

if cart\_item.quantity > 1:

cart\_item.quantity -= 1

cart\_item.save()

# if only one item then delete

else:

cart\_item.delete()

except:

pass

return redirect('cart')

**Cart Remove with variation:**

1. Pass the ‘cart\_item\_id’ to the url for remove button to uniquely identify. Go to the ‘cart.html’ file and modify the code for ‘Remove’ button. Code as follows:

<td class="text-right">

<a

href="{% url 'remove\_cart\_item' cart\_item.product.id cart\_item.id%}"

class="btn btn-danger"

>Remove</a>

</td>

1. Go to ‘urls.py’ file in the ‘carts’ app to modify the url pattern to accept the id. Code as follows:

urlpatterns = [

path("", views.cart, name="cart"),

path("add\_to\_cart/<int:product\_id>", views.add\_to\_cart, name='add\_to\_cart'),

path("remove\_from\_cart/<int:product\_id>/<int:cart\_item\_id>", views.remove\_from\_cart, name='remove\_from\_cart'),

path("remove\_cart\_item/<int:product\_id>/<int:cart\_item\_id>", views.remove\_cart\_item, name='remove\_cart\_item'),

]

1. Now lastly, go to the ‘view.py’ file of the ‘carts’ app to accept the id in the remove\_cart\_item method and with that uniquely identify the cart\_item. Code as follows:

# Remove whole item from cart

def remove\_cart\_item(request, product\_id, cart\_item\_id):

cart = Cart.objects.get(cart\_id = \_cart\_id(request))

product = get\_object\_or\_404(Product, id = product\_id)

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

cart\_item.delete()

return redirect('cart')

1. For an additional feature we are confirming the customer are they sure to remove the item from cart or not. For that we need to add a simple JavaScript code to the ‘cart.html’ file. Go to Remove button and add onclick JS code to it as follows:

<td class="text-right">

<a

href="{% url 'remove\_cart\_item' cart\_item.product.id cart\_item.id%}"

onclick="return confirm('Are you sure to delete this item?')"

class="btn btn-danger"

>

Remove</a

>

</td>

1. dsf
2. gfhfgh