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:

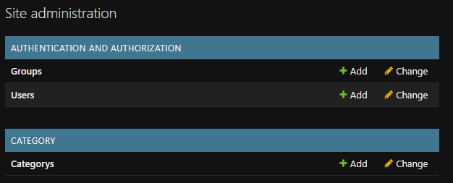
python manage.py migrate

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

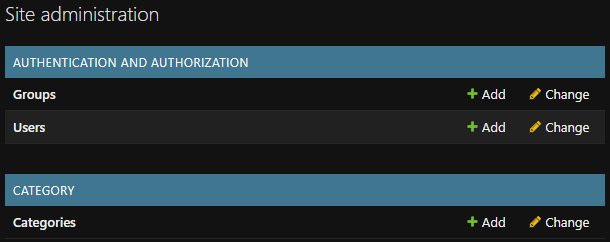
python manage.py createsuperuser

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

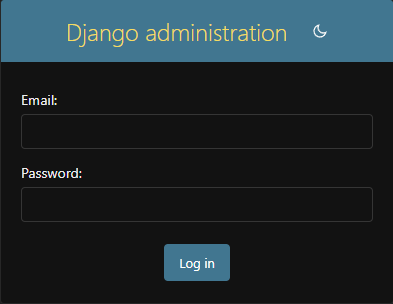
First name: test

Last name: user

Password:

Password (again):

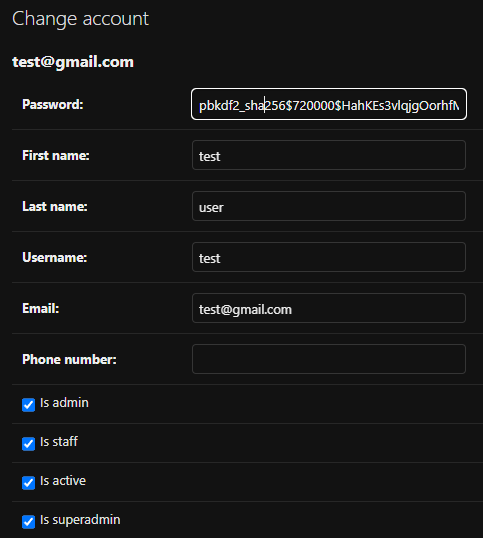
Superuser created successfully.

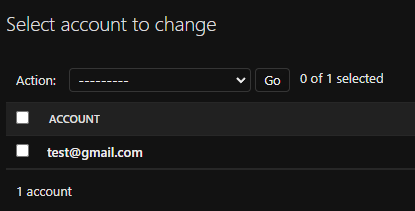
1. Now run server and go to the admin page.

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')

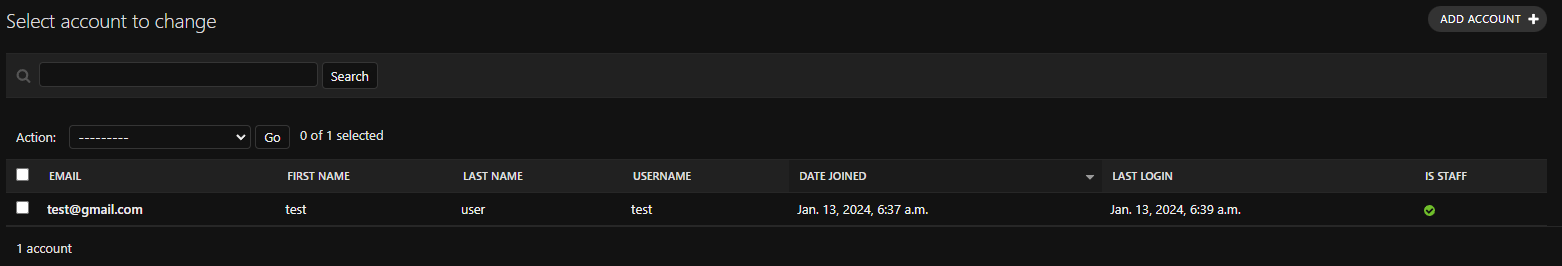
# sort the display list by date\_joined in descending order

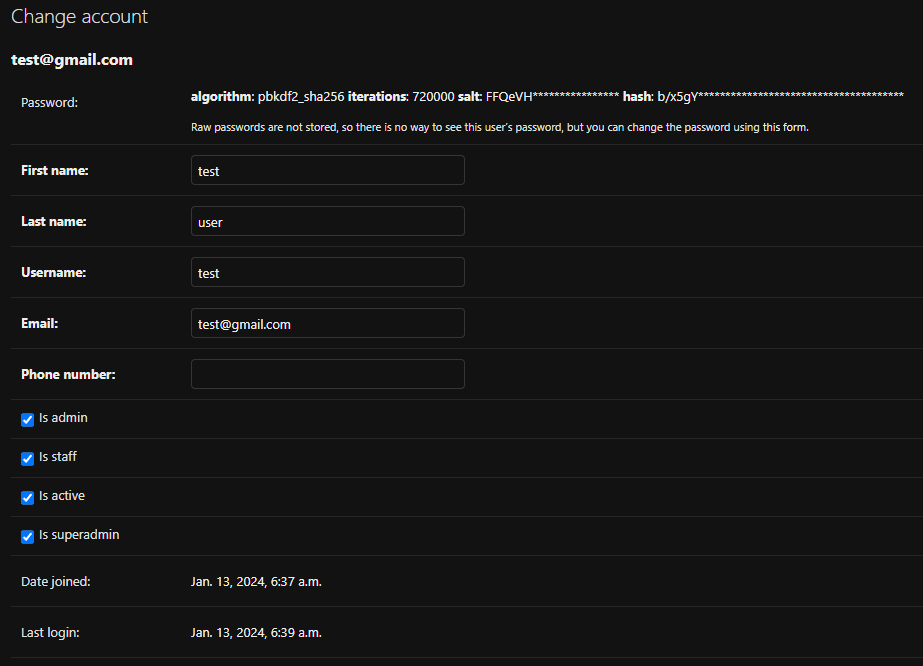
ordering = ('-date\_joined',)

# 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.models 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>

1. dsf
2. gfhfgh