Pushing the Django Project to the GitHub:

1. Freeze the Dependencies

- Inside your project folder, run the following command to generate a `requirements.txt` file that lists all the installed Python packages:
- Inside your **project folder**, run:

```
pip freeze > requirements.txt
```

- This will save all installed Python packages into a `requirements.txt` file.

2. Create a .gitignore File

- Inside the **project folder**, create a file named `.gitignore` and add:

```
# Python
*.pyc
__pycache__/
# Django
db.sqlite3
/staticfiles/
/media/
.env
env/
venv/
# VS Code
```

```
.vscode/

# OS Files

.DS_Store

# Git

.git
```

- This tells Git which files and folders to **ignore** (not track or upload).

3. Create a New Repository on GitHub

- Go to https://github.com and **Log In**.
- Click ➡️ **"+"** ➡️ **New Repository**.
- Enter a **repository name** (e.g., `my-django-app`).
- (Optional) Add a **description**.
- **DO NOT** select options for README, .gitignore, or license.
- Click ➡️ **Create Repository**.

4. Clone the Repository to Your Local Machine

In your terminal, run:

```
git clone https://github.com/<your_username>/<your_repo_name>.git
```

- This creates an **empty folder** on your machine linked to GitHub.

5. Copy Your Django Project Into the Cloned Folder

- Open the cloned repo folder on your computer.

- **Copy-paste** all your Django project files into this cloned folder (like `manage.py`, your main project folder, your apps, `requirements.txt`, etc.).
- Now your project is **inside the Git-Linked Folder**.

6. Stage the Files:

- Inside the cloned repo folder in the terminal, run:

```
git add .
```

7. Commit Your Changes:

```
git commit -m "Added Django project files"
```

8. Push to GitHub

- Now push everything to your GitHub repo:

```
git branch -M main

git push origin main
```

- This uploads your Django project to GitHub!

9. Verify on GitHub

- Open your GitHub repository in your browser.
- You should see **all your Django project files** there!

Note: before pushing the Django project to GitHub, configure your name and email to the git.

```
git config --global user.name "Your Name"

git config --global user.email "youremail@example.com"
```

Introduction to Django REST Framework (DRF)

- Django REST Framework (DRF) is a **powerful and flexible toolkit** for building Web APIs using Django.
- It is the **most widely used framework** in the Python world for developing **RESTful APIs.**
- DRF is built on top of Django, so it supports:
    - Django **models**
    - Django **views**
    - Django **authentication system**
    - Django **ORM**

Website: https://www.django-rest-framework.org

What is an API?

- **API** stands for **Application Programming Interface**.
- It is an interface that allows **two different software applications** to communicate with each other.
- It is like building a remote control with a bunch of buttons, where each button provides specific functionality similarly our API is gonna have bunch of **endpoints** for different purpose, so we can have endpoints like:
    - **/products :** getting the list of products,creating, updating and deleting products
    - **/carts:** managing our shopping carts
    - **/orders:** managing the orders

- Client application can send the request to these endpoints to get or save products, orders, shopping carts and so on

API Provider vs API Consumer:

- **API Provider:** The backend application that **exposes the API** (usually Django)
- **API Consumer**: The application that **uses the API**, like a React app or mobile app

Sometimes, one backend can **consume APIs** of another — e.g., a Django app calling a 3rd-party weather API.

What is a RESTful API?

REST stands for **Representational State Transfer**.
It is a set of architectural rules and constraints for building scalable and reliable web services.

A **RESTful API** follows these REST principles and allows client applications (like mobile apps or frontend apps) to interact with backend resources via **HTTP methods**.

RESTful APIs are:

- Fast
- Scalable
- Stateless
- Reliable
- Easy to use
- Easy to evolve and maintain

An API that follows these REST principles is called a **RESTful API**.

Key Concepts of RESTful APIs

1. Resources

- A resource is any object or data in the application — such as **Product**, **User**, **Order**, etc. Each resource is identified by a **URL**.
- An **URL** is a way to locate a resource on the web, it is basically a web address

Example:

- `GET http://example.com/products/` – List all products
- `GET http://example.com/products/1/` – Retrieve product with ID = 1
- `GET http://example.com/products/1/reviews/` – List all reviews of product 1

2. Resource Representations

- When a client accesses a resource via URL, the server responds with a **representation** of that resource.
- This representation is usually in:
    - **JSON** ✅ (most common)
    - XML
    - HTML (in browser views)

- These are not the internal representation of a resource inside the server, the server uses objects (like Python classes) to manage these resources.

3. HTTP Methods in REST

- When building a REST API we expose one or more endpoints for clients, each endpoint may support various kinds of operations like some endpoints may allow reading data while other may allow modifying data, here HTTP methods come into the picture.
- REST APIs use HTTP methods to perform actions on resources:
- Using the HTTP methods, the client can tell the server what he wants to do with the resource.

Method	Meaning	Used For
GET	Retrieve data	List or fetch resource
POST	Create new resource	Add new product/order etc.
PUT	Replace entire resource	Full update
PATCH	Update part of resource	Partial update
DELETE	Delete resource	Remove product/order etc.

Example: Creating a Product

Endpoint: `POST /products`

Request Body:

```
{
  "title": "iPhone 15",
  "price": 90000
}
```

Server Action: Creates a new product and returns the saved object with its unique ID.

Note: For more information about the Web-services, REST API, please refer to the Flask REST API related notes.

Example:

- Develop a new Django Project called **DjangoRestProject**

  django-admin startproject DjangoRestProject

- Move inside the Project directory:

  cd DjangoRestProject

- Create a new application inside this project called **ProductApp**

  python manage.py startapp ProductApp

- Register this app inside the **settings.py** file
- Define the following model classes inside the **ProductApp/models.py** file

```
from django.db import models

class Product(models.Model):

    CATEGORY_CHOICES = [

        ("Electronics", "Electronics"),

        ("Stationary", "Stationary"),

         ("HomeAppliance", "HomeAppliance")

    ]

    name = models.CharField(max_length=20)

    price = models.IntegerField()

    quantity = models.IntegerField()
    category = models.CharField(max_length=20,
choices=CATEGORY_CHOICES)

    created_at = models.DateTimeField(auto_now_add=True)

    updated_at = models.DateTimeField(auto_now=True)


    def __str__(self):

        return self.name


class Review(models.Model):

    review_text = models.TextField()

    rating = models.DecimalField(max_digits=3, decimal_places=1)

    created_at = models.DateTimeField(auto_now_add=True)

    product = models.ForeignKey(Product, on_delete=models.CASCADE)


    def __str__(self):

        return f"Review for {self.product.name}"
```

- Register both classes inside the **ProductApp/admin.py** file to manage them from the admin interface.

```
from django.contrib import admin

from .models import Product, Review
```

```python
# Optional: Customize how Product appears in admin
class ProductAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'price', 'quantity',
                    'category', 'created_at', 'updated_at')
    list_filter = ('category',)


# Optional: Customize how Review appears in admin
class ReviewAdmin(admin.ModelAdmin):
    list_display = ('id', 'product', 'rating', 'created_at')
    list_filter = ('rating',)
# Registering both models with their custom admin classes
admin.site.register(Product, ProductAdmin)
admin.site.register(Review, ReviewAdmin)
```

- Perform the migrations:

    python manage.py makemigrations

    python manage.py migrate

- Create a super user and run the server, after that add some products and their reviews from the admin interface by login into it.

    python manage.py createsuperuser

    python manage.py runserver

http://127.0.0.1:8000/admin/

Installing Django Restframework: (DRF)

pip install djangorestframework

- Register it inside our project in the **settings.py** file right after the inbuilt django apps

```
INSTALLED_APPS = [

    ----

    'rest_framework',

    'ProductApp',

]
```

Creating API Views in Django REST Framework

- In Django, to handle requests and responses, we typically use:
  - `HttpRequest`
  - `HttpResponse`

- However, in **Django REST Framework (DRF)**, we use more powerful and flexible classes:
  - `Request`
  - `Response`

- In Django we can create API views using 2 ways:
  1. Function based views (FBV)
  2. Class based views (CBV)

Example: FBV

- Define the following function based views inside the **ProductApp/views.py** file

```
from rest_framework.response import Response
from rest_framework.decorators import api_view
# Create your views here.
@api_view()
def product_list_view(request):
    return Response('OK')


# API with parameter
@api_view()
def product_detail_view(request, id):
    return Response(id)
```

- Specify the url patterns for the above views inside the **ProductApp/urls.py** file

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.product_list_view, name='products'),
    path('<int:id>/', views.product_detail_view, name='product')
]
```

- Include the above **urls.py** file to **project level urls.py** file.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('products/', include('ProductApp.urls'))
]
```

- Run the server and access the following endpoints:

  http://127.0.0.1:8000/products/

  http://127.0.0.1:8000/products/10/

- The `@api_view()` decorator transforms the Django `HttpRequest` into a DRF `Request` object.

- It also enables the Browsable **API interface**, which allows easy testing via a web browser.

- When a **client app** (e.g., mobile app or frontend app) consumes this API, it will receive only the **JSON response** — not the browsable interface.

Creating Serializers in DRF:

- **Serialization**: Converting python objects into JSON data
- **Deserialization**: Converting JSON data into Python objects.

- Serializers in DRF help **convert Django model instances (Python objects) into JSON** and vice versa.

- They play the same role as forms in regular Django, but instead of HTML form handling, serializers handle **JSON data**.

- Create a new file called **serializers.py** inside the **ProductApp** folder. Inside this file define a serializer class to serialize and deserialize the Product model class object.

```
from rest_framework import serializers

class ProductSerializer(serializers.Serializer):

    # Specify the fields to serialize from the Product model

    id = serializers.IntegerField()

    name = serializers.CharField(max_length=20)

    price = serializers.IntegerField()

    quantity = serializers.IntegerField()

    category = serializers.CharField()
```

Here we can exclude some fields also from the above ProductSerializer class.

🧠 Think of serializers as the **external representation** of your model objects.

The model defines how data is stored in the database (internal), while the serializer defines how it is sent/received as JSON (external).

🔍 Check DRF documentation → **API Guide** → **Serializer Fields**

These fields closely resemble Django model fields but are for serialization purposes.

Converting a Model Object to JSON:

- Now that we have created the `ProductSerializer` class, we can use it to convert a `Product` model instance into a JSON object.
- Modify the **product_detail_view** function inside the **ProductApp/views.py** file.

```
from .models import Product
from .serializers import ProductSerializer
@api_view()
def product_detail_view(request, id):
    product = Product.objects.get(pk=id)
    serializer = ProductSerializer(product)
    return Response(serializer.data)
```

Run the server and try to access the following endpoint with a specific product id.

http://127.0.0.1:8000/products/1

❌ Handling Invalid Product IDs

If a product ID does not exist, Django will raise a `Product.DoesNotExist` exception. Let's handle this:

```
from rest_framework import status
@api_view()
def product_detail_view(request, id):
    try:
        product = Product.objects.get(pk=id)
        serializer = ProductSerializer(product)
        return Response(serializer.data)
    except Product.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)
```

- Instead of repeating this pattern, we can use a shortcut:

```
from django.shortcuts import get_object_or_404
@api_view()
def product_detail_view(request, id):
    product = get_object_or_404(Product, pk=id)
    serializer = ProductSerializer(product)
    return Response(serializer.data)
```

📋 Serializing a List of Products

To return a list of products:

```
@api_view()
def product_list_view(request):
    products = Product.objects.all()
    serializer = ProductSerializer(products, many=True)
    return Response(serializer.data)
```

🧠 Note: many=True tells DRF that we are serializing a **queryset** instead of a single object.

- Test the following endpoint to access all the products:

http://127.0.0.1:8000/products/

Creating Custom Serializer Fields:

- In DRF, your **API model (external representation)** doesn't have to match your **data model (internal representation)**.
- This allows you to:
  - Rename fields in the API.
  - Add computed or derived fields.
  - Hide internal fields.

Example: Changing Field Name + Adding Custom Field

```
from rest_framework import serializers
from .models import Product
class ProductSerializer(serializers.Serializer):
    # Specify the fields to serialize from the Product model
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=20)
    quantity = serializers.IntegerField()
    category = serializers.CharField()

    # Renaming `price` to `unit_price` in the API
    unit_price = serializers.IntegerField(source='price')

    # Adding a computed field
    price_with_tax = serializers.SerializerMethodField(
        method_name='calculate_tax')


    def calculate_tax(self, product: Product):

        return product.price * 2
```

🧠 **Explanation:**

- `source='price'`: Maps `unit_price` to the actual `price` field in the model.
- `SerializerMethodField`: Adds a **custom computed field** using a method `calculate_tax`.

Serializing Relationships in DRF or Nested Serialization:

- When serializing a model like `Product`, you may want to include related models (e.g. `Review`) in the output.

Step1: Create a serializer class for **Review** model class inside the **ProductApp/serializers.py** file.

```
class ReviewSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    review_text = serializers.CharField()

    rating = serializers.DecimalField(max_digits=3, decimal_places=1)

    created_at = serializers.DateTimeField()
```

Step2: Include this inside the **ProductSerializer** class as follows:

```
class ProductSerializer(serializers.Serializer):

    # Specify the fields to serialize from the Product model
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=20)
    quantity = serializers.IntegerField()
    category = serializers.CharField()

    # Renaming `price` to `unit_price` in the API
    unit_price = serializers.IntegerField(source='price')

    # Adding a computed field
    price_with_tax = serializers.SerializerMethodField(
        method_name='calculate_tax')
    reviews = ReviewSerializer(source='review_set', many=True,
read_only=True)

    def calculate_tax(self, product: Product):
        return product.price * 2
```

- DRF will look for a **reverse relation** called `review_set` on the `Product` model (auto-generated by Django since `Review` has `product = ForeignKey(...)`).
- By default, Django creates the reverse relation with the lowercase model name + `_set`.
- So, in your `Review` model:

product = models.ForeignKey(Product, on_delete=models.CASCADE)

- This creates an implicit reverse relation called `review_set` on the `Product` model.
- So DRF will look for:

product.review_set.all()

- read_only = True:
  - **The `reviews` field is only used when serializing data** (i.e., when returning a `Product` to the client).
  - **It will be ignored during deserialization** (i.e., when creating or updating a `Product` from client input).

Optional: Customize the Reverse Name

- If you want a cleaner reverse accessor (instead of `review_set`), you can explicitly set it in the model:

  ```
  product = models.ForeignKey(Product, on_delete=models.CASCADE, related_name='reviews')
  ```

- Then in the serializer:

  ```
  reviews = ReviewSerializer(many=True, read_only=True)  # source='reviews' is now automatic.
  ```

This is cleaner and more readable in both code and the API output.

ModelSerializer:

- Instead of manually defining each field in a serializer, we can use the `ModelSerializer` class to automatically generate fields based on the model.

Ways to Specify Fields in `ModelSerializer`

- We can specify fields inside the `Meta` class in **3 ways**:

1. **Include All Fields**
   - Use `fields = '__all__'` to include every field from the `Product` model.
   - Use this when you want to expose all fields from the model.(not recommended)

Example:

```
class ProductSerializer(serializers.ModelSerializer):
```

```
    class Meta:

        model = Product

        fields = '__all__'
```

2. **Include Only Specific Fields**
   - Use `fields = (...)` when you want to include a limited set of fields.
   - Best when you want to include just a few fields (minority of the total fields).

Example:

```
class ProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        fields = ('id', 'name', 'price', 'quantity')
```

3. **Exclude Some Fields:**
   - Use `exclude = (...)` when you want to include most fields but leave out a few.
   - Best when you want to include the majority of fields.

Example:

```
class ProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        exclude = ('price',)  # All fields except 'price' will be included
```

Example with Custom Fields and Field Renaming:

- You can also rename fields and add calculated/custom fields:

Example:

```
from rest_framework import serializers
from .models import Product
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = ['id', 'name', 'unit_price',
                  'quantity', 'price_with_tax', 'reviews']
    unit_price = serializers.IntegerField(
        source='price')  # Renaming price to unit_price
    price_with_tax = serializers.SerializerMethodField(
        method_name='calculate_tax')
    reviews = ReviewSerializer(many=True, read_only=True)
    def calculate_tax(self, product):
        return product.price * 1.2
```

Deserialization in Django REST Framework

- **Deserialization** is the process of converting incoming JSON data (from a client) into a Django model instance.

  For Example if the API client will send the product data:

POST /products

And send the product related json data inside the body of the request

```
{
    "name": "Speaker",
    "price": 8000,
    "category": "Electronics"
}
```

We need to read the data from the request body and Deserialize this data in the **Product** object.

Example: Handling GET and POST requests

- Modify the `product_list_view` function to support the POST method also.

```
@api_view(['GET', 'POST'])
def product_list_view(request):
    if request.method == 'GET':
        products = Product.objects.all()
        serializer = ProductSerializer(products, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = ProductSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        print(serializer.validated_data)  # To get the validated
data

        serializer.save()  # saves the validated data into the DB

        print(serializer.data)  # gives the clean data in the form
of JSON

        return Response(serializer.data,
status=status.HTTP_201_CREATED)
```

- Once `POST` is added to `@api_view`, the **Browsable API** will show an interactive form to send data.

  Sample POST request body:

  ```
  {
    "name": "Printer",
    "unit_price": 7000,
    "quantity": 12,
    "category": "Electronics"
  }
  ```

- serializer.save() # Automatically creates and saves the Product object

Data Validation:

- Before accessing `serializer.validated_data`, you must validate the input data.

  Automatic Validation (based on model fields)

  ```
  serializer = ProductSerializer(data=request.data)
  serializer.is_valid(raise_exception=True)
  ```

- Using `raise_exception=True` automatically returns a 400 response with error details if validation fails.

Custom Validations in Serializers:

- Custom validations are used when built-in validation isn't enough. In DRF, we can validate incoming data using **three main approaches**:

1. **Field-Level Validation:**
    - Use this when you want to apply custom rules on **individual fields**.
    - Method name must be `validate_<field_name>()`

Example: Quantity should not exceed 1000

```
class ProductSerializer(serializers.ModelSerializer):

   ---

    def validate_quantity(self, value):

        if value > 1000:

            raise serializers.ValidationError("Quantity cannot
exceed 1000")

        return value
```

2. Object-Level Validation:

- Use this when you need to validate multiple fields together, like comparing them.
- For example, comparing **password** with **confirm_password.**
- Use the `validate(self, data)` method for object-level validations.

Example: Ensure `price` matches `re_enter_price`

```
class ProductSerializer(serializers.ModelSerializer):

    re_enter_price = serializers.IntegerField(write_only=True)

    def validate(self, data):
        if data['price'] != data['re_enter_price']:
            raise serializers.ValidationError("Price and Re-entered
price must match.")
        return data
    class Meta:
        model = Product
        fields = ['id', 'name', 'price', 're_enter_price',
'quantity']
```

- The field `re_enter_price` is `write_only`, meaning it won't be returned in the response but is required for validation during creation.

3. **Using Custom Validators:**

- You can write external functions and attach them to specific fields using the `validators=[]` argument.

Example: Product name must not start with a digit

```
def validate_name_starts_with_letter(value):
    if value[0].isdigit():
        raise serializers.ValidationError("Product name should not start
with a number")
    return value


class ProductSerializer(serializers.ModelSerializer):
    name =
serializers.CharField(validators=[validate_name_starts_with_letter])
```

```
    class Meta:
        model = Product
        fields = ['id', 'name', 'price', 'quantity']
```

Updating a Product (PUT):

- We use the PUT method to update an existing product.
- Modify the product_detail_view function to support the update functionality

```
@api_view(['GET', 'PUT'])
def product_detail_view(request, id):
    product = get_object_or_404(Product, pk=id)

    if request.method == 'GET':
        serializer = ProductSerializer(product)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = ProductSerializer(product, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_200_OK)
```

Test it with a PUT request:

URL: http://127.0.0.1:8000/products/3/

(replace 3 with any product id)

Sample JSON:

```
{
  "name": "Updated Speaker",
  "price": 8500,
  "quantity": 50
}
```

- This will update the product with ID 3.

Deleting a Product (DELETE):

- We use the DELETE method to delete an existing product.
- Modify the product_detail_view function to support the delete functionality

```
@api_view(['GET', 'PUT', 'DELETE'])
def product_detail_view(request, id):
    product = get_object_or_404(Product, pk=id)

    if request.method == 'GET':
        serializer = ProductSerializer(product)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = ProductSerializer(product, data=request.data)
        serializer.is_valid(raise_exception=True)

        serializer.save()

        return Response(serializer.data)


    elif request.method == 'DELETE':

        product.delete()

        return Response(status=status.HTTP_204_NO_CONTENT)
```

Test it with a DELETE request:

URL: http://127.0.0.1:8000/products/3/

- This will delete the product with ID 3.

Final URLs:

Method	Endpoint	Description
GET	/products/	List all products
POST	/products/	Create a new product
GET	/products/<int:id>/	Retrieve a product by id
PUT	/products/<int:id>/	Update a product by id
DELETE	/products/<int:id>/	Delete a product by id

Django REST Framework: Class-Based Views (CBVs) for REST API

Introduction:

- Till now, we developed **Function-Based Views (FBVs)** to handle client requests.

- But Django REST Framework (DRF) **also supports Class-Based Views (CBVs)**
- Class-Based Views (CBVs) in Django REST Framework (DRF) provide a more organized and reusable way to build RESTful APIs compared to Function-Based Views (FBVs).

Advantages of CBVs over FBVs:

- Cleaner and more structured code.
- Easier to reuse and extend with mixins and generics.
- Provide **separation of logic**: `GET`, `POST`, `PUT`, `DELETE` methods are defined separately inside a class..

DRF provides two types of CBVs:

1. **APIView class** (low-level, full control)
2. **Generic views** and **mixins** (high-level, less code)

Example1: Using APIView (Low-level CBV)

- In DRF, all CBVs are based on the **`APIView`** class:
- This gives you full control over the logic in each HTTP method.

```
# views.py

from rest_framework.views import APIView

from rest_framework.response import Response

from rest_framework import status

from .models import Product

from .serializers import ProductSerializer

from django.shortcuts import get_object_or_404



class ProductListCreateView(APIView):

    def get(self, request):
        products = Product.objects.all()
        serializer = ProductSerializer(products, many=True)
        return Response(serializer.data)

    def post(self, request):
        serializer = ProductSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)


class ProductDetailView(APIView):
    def get_object(self, id):
        return get_object_or_404(Product, pk=id)

    def get(self, request, id):
        product = self.get_object(id)
        serializer = ProductSerializer(product)
        return Response(serializer.data)

    def put(self, request, id):
        product = self.get_object(id)
        serializer = ProductSerializer(product, data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data)

    def delete(self, request, id):
        product = self.get_object(id)
        product.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

- Here, **no need** to use `if request.method == "GET"` like FBVs.
- Each HTTP method has its **separate function**.

URL Patterns

```
# urls.py
from django.urls import path
from .views import ProductListCreateView, ProductDetailView

urlpatterns = [
    path('', ProductListCreateView.as_view(), name='product-list-create'),
    path('<int:id>/', ProductDetailView.as_view(), name='product-detail'),
]
```

- The **`.as_view()`** method **converts the class into a view function** that Django can call when a request comes.

2. Using GenericAPIView + Mixins (Mid-level abstraction)

- This approach allows combining modular mixins with reusable generic base classes.

Mixin Classes:

- A **Mixin** is a class that encapsulates **reusable code patterns**.
- In DRF we have various mixin classes to perform different kinds of operations on the resources.
- In DRF, mixins are available for common operations like:

    - ListModelMixin (GET multiple objects)
    - CreateModelMixin (POST new object)
    - RetrieveModelMixin (GET single object)
    - UpdateModelMixin (PUT/PATCH)
    - DestroyModelMixin (DELETE)

Refer the documentation: https://www.django-rest-framework.org/api-guide/generic-views/

```
from rest_framework.generics import GenericAPIView

from rest_framework.mixins import ListModelMixin, CreateModelMixin,
RetrieveModelMixin, UpdateModelMixin, DestroyModelMixin
```

```
class ProductListCreateView(GenericAPIView, ListModelMixin,
CreateModelMixin):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

class ProductDetailView(GenericAPIView, RetrieveModelMixin,
UpdateModelMixin, DestroyModelMixin):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    lookup_field = 'id'

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

3. Using Generic Views (High-level abstraction)

- Most of the time we don't use these mixin classes directly, instead we use some concrete classes which combine one or more mixin, we call these classes as **Generic Views.**
- For Example:

  ListCreateApiView: ListModelMixin + CreateModelMixin

  RetrieveUpdateDestroyAPIView: RetrieveModelMixin + UpdateModelMixin + DestroyModelMixin

Advantage:

No need to define `get`, `post`, `put`, `delete` methods explicitly unless you want to customize them.

Example:

```
from rest_framework.generics import ListCreateAPIView,
RetrieveUpdateDestroyAPIView

class ProductListCreateView(ListCreateAPIView):

    queryset = Product.objects.all()

    serializer_class = ProductSerializer


class ProductDetailView(RetrieveUpdateDestroyAPIView):

    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    lookup_field = 'id'
```

Customizing the Generic Views:

- Sometimes, you may want to **customize the behavior**.
- Example (override `delete` method):

```
class ProductDetailView(RetrieveUpdateDestroyAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    lookup_field = 'id'

    def delete(self, request, id):
        product = get_object_or_404(Product, pk=id)
       product.review_set.count() > 0:  # 👈 check if there are any reviews
            return Response({'error': 'Product cannot be deleted as it has reviews.'}, status=status.HTTP_400_BAD_REQUEST)
        product.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

- Here, before deleting, we check if the product is linked to any `orderitems`.

ViewSets:

- **ViewSet** groups **related views** (list, retrieve, create, update, delete) into a **single class**.
- It is a set of related views

Example:

```
from rest_framework.viewsets import ModelViewSet


class ProductViewSet(ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

    def destroy(self, request, id=None):
        product = get_object_or_404(Product, pk=id)
        product.review_set.count() > 0:
            return Response({'error': 'Product cannot be deleted as it has reviews.'}, status=status.HTTP_400_BAD_REQUEST)

        product.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

- Here we don't define separate views (`ProductList`, `ProductDetail`), everything is managed inside `ProductViewSet`.

Routers:

- When using **ViewSets**, you don't define URLs manually.
- Instead, you use a **Router** to automatically generate URLs.

urls.py:

```
from rest_framework.routers import SimpleRouter
from .views import ProductViewSet

router = SimpleRouter()
router.register('products', ProductViewSet)

urlpatterns = [
    path('', include(router.urls)),
]
```

Calling backend API from python application:

- In Python, to interact with **Backend APIs** (like REST APIs), we commonly use the **`requests`** library.

What is an API?

- API (Application Programming Interface) allows different software systems to communicate.

- A **Backend API** typically handles operations like **Create**, **Read**, **Update**, and **Delete** (CRUD).

How to test APIs?

- We can call and test APIs using **Postman** or any kind of **REST client tools** for testing purposes.

- However, if we want to **use the API-related data inside our Python application** — for example, to display it, process it, or store it — we **need to use the `requests` library** to call APIs programmatically.

- This allows us to **fetch, manage, and incorporate** the API data **directly into our Python code** and build powerful applications.

Why use the requests library in Python?

- Simple and human-friendly syntax.
- Supports all HTTP methods.
- Allows adding **headers**, **body**, **params**, **authentication**, etc.
- Essential for **connecting backend data** to **Python applications**.

Installing requests

pip install requests

Example: Using JSONPlaceholder

- https://jsonplaceholder.typicode.com
- `JSONPlaceholder` is a free online REST API for testing and prototyping.
- JSONPlaceholder is a safe playground to practice APIs.

Available endpoints (example):

- `/posts`
- `/users`
- `/comments`

Example: Getting All the posts

- Create a new file called **APITestApp.py** file.

APITestApp.py

```
import requests

BASE_URL = "https://jsonplaceholder.typicode.com/posts"

# GET request (fetch all the posts)

def get_posts():

    response = requests.get(BASE_URL)

    # print(f"The actual response is:{response} ")

    # print(f"The response headers are:{response.headers} ")

    # print(f"The response contents are:{response.content} ")
    if response.status_code == 200:
        posts = response.json()
        print("Fetched Posts:")
        for post in posts[:3]:  # Display only first 3 post
            print(post)
    else:
        print(f"Failed to fetch posts, Status code {response.status_code}")
# Calling the function
get_posts()
```

- **To retrieve the response body in its raw binary format, we use `response.content`, whereas to obtain the response body as a parsed JSON object (converted into a Python dictionary or list), we use the `response.json()` method.**

Assignment: Write a function to get a specific post based on the post id.

```
import requests
BASE_URL = "https://jsonplaceholder.typicode.com/posts"
def get_single_post(post_id):
    response = requests.get(f"{BASE_URL}/{post_id}")
    if response.status_code == 200:
        print(f"The Post is:{response.json()}")
    else:
        print(
            f"Failed to fetch the post with Id {post_id}, Status
code: {response.status_code}")


# Calling the function

get_single_post(2)
```

Example: Creating a new Post:

```
import requests

BASE_URL = "https://jsonplaceholder.typicode.com/posts"

def create_post():

    new_post = {

        "title": "My New Post",

        "body": "This is the body of my new post"

    }


   response = requests.post(BASE_URL, json=new_post)

    if response.status_code == 201:

        created_post = response.json()

        print(f"Created Post is: {created_post}")

    else:

        print(
            f"Failed to create a new post Status code:
{response.status_code}")
# Calling the above function
create_post()
```

Example: Update an existing Post:

```
import requests
BASE_URL = "https://jsonplaceholder.typicode.com/posts"
def update_post(post_id):
    updated_post = {
        "id": post_id,
        "title": "Updated Title",
        "body": "Updated Body Content",
        "userId": 1
    }
    response = requests.put(f"{BASE_URL}/{post_id}", json=updated_post)
    if response.status_code == 200:
        updated = response.json()
        print("Updated Post:")
        print(updated)
    else:
        print(f"Failed to update post. Status code:
{response.status_code}")
# Calling the above function
```

```
update_post(1)
```

Example: Deleting an existing Post:

```
import requests

BASE_URL = "https://jsonplaceholder.typicode.com/posts"

def delete_post(post_id):
    response = requests.delete(f"{BASE_URL}/{post_id}")
    if response.status_code == 200:
        print(f"Post with ID {post_id} deleted successfully.")
    else:
        print(f"Failed to delete post. Status code: {response.status_code}")

# Calling the above function

delete_post(1)
```

Example: Django Application to Display 5 Posts on the HTML template

- Create a Django project called: **DjangoPostApiProject**

```
django-admin startproject DjangoPostApiProject
```

- Move inside the project

```
cd DjangoPostApiProject
```

- Create a new app called : **PostApp**

  python manage.py startapp **PostApp**

- Register the **PostApp** inside the settings.py file

- Define the following view function inside the **PostApp/views.py** file

```
from django.shortcuts import render, redirect
import requests
API_URL = "https://jsonplaceholder.typicode.com/posts"
def list_posts_view(request):
    response = requests.get(API_URL)
    posts = response.json()
    return render(request, 'posts.html', context={'posts': posts[:5]})
```

- Define the url pattern for the above view function inside the **PostApp/urls.py** file.

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.list_posts_view, name='list_posts'),
]
```

- Register the above **urls.py** file inside the **project level urls.py** file

```
from django.contrib import admin

from django.urls import path, include

urlpatterns = [

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

    path('', include('PostApp.urls'))

]
```

- Create the following **posts.html** file inside the **PostApp/template** folder.

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width,
initial-scale=1.0">

    <title>Document</title>

</head>

<body bgcolor="cyan">

    <h1 style="text-align: center;">Post APP</h1>

    <ol>

        {% for post in posts %}
```

```
            <li>

                <strong>Title: </strong>

                {{post.title}}

                <br>

                <strong>Body: </strong>

                {{post.body}}

                <br><br>

            </li>

            {% endfor %}

        </ol>

    </body>

    </html>
```

- Run the server and access the application:

    python manage.py runserver

    http://127.0.0.1:8000/

Accessing the Protected API:

- In many web applications, REST APIs are protected to ensure that only authorized users can access sensitive resources. One common way to protect an API is by using JWT (JSON Web Token). With JWT, users first authenticate (usually via login), and then use a token to access protected endpoints.

Example: Consuming the Flask REST API application: FlaskDBAuthProductAppJWT

- Refer the **readme.txt** file of the **FlaskDBAuthProductAppJWT**application:

- Run the above Flask Application inside another VS-Code.
    - python app.py

      http://127.0.0.1:5000

- Create a new file called **APITestDemo.py** inside another workspace:

```
import requests
# API URLs
register_url = "http://localhost:5000/registerapi"
login_url = "http://localhost:5000/loginapi"
products_url = "http://localhost:5000/products"

# Function to Register a New User
def register_user(name, email, mobile, password):
    register_data = {
        "name": name,
        "email": email,
        "mobile": mobile,
        "password": password,
    }
    # Send POST request to register user
    register_response = requests.post(register_url, json=register_data)

    if register_response.status_code == 201:
        print("User registered successfully.")
        return True
    else:
        print(f"User registration failed: {register_response.status_code}")
        return False


# Function to Login and Get JWT Token



def login_user(username, password):
    login_data = {
        "username": username,
        "password": password
    }
    # Send POST request to login and get JWT token
    login_response = requests.post(login_url, json=login_data)

    if login_response.status_code == 200:
        jwt_token = login_response.json()['access_token']
        print(f"JWT Token: {jwt_token}")
        return jwt_token
    else:
        print(f"Login failed: {login_response.status_code}")
        return None

# Function to Get Product List using JWT Token

def get_product_list(jwt_token):
    headers = {
        "Authorization": f"Bearer {jwt_token}",
        "Content-Type": "application/json"
    }

    # Send GET request to fetch product list
    product_response = requests.get(products_url, headers=headers)

    if product_response.status_code == 200:
        response_data = product_response.json()
        # Debugging the response
        print("Response from Product API:", response_data)
        products = response_data.get('products', [])  # Get 'products' key
        print("Product List:")
        for product in products:
            print(
                f"- {product['name']} | {product['price']} |
{product['category']}")
    else:
        print(f"Failed to fetch products: {product_response.status_code}")

# Main Execution Flow

def main():
    # Step 1: Register a New User
    if register_user("Raj", "raj@gmail.com", "9876543210", "raj123"):

        # Step 2: Login to get JWT token
        jwt_token = login_user("raj@gmail.com", "raj123")

        if jwt_token:
            # Step 3: Use the JWT token to get the product list
            get_product_list(jwt_token)
# Run the main function
if __name__ == "__main__":
    main()
```

Assignment: Integrating Django with Flask API

Objective:

- Create a Django project that interacts with a Flask-based REST API to perform the following operations:

1. **User Registration and Login:**
    - Display a registration form in Django.
    - Allow the user to register by sending data to the Flask API (`POST /registerapi`).
    - Display a login form in Django.
    - Allow the user to log in by sending credentials to the Flask API (`POST /loginapi`) and receive a JWT token.
2. **Add New Product to Flask Application:**
    - Create a product registration page in Django (HTML form).
    - Allow users to add a new product by sending product data to the Flask API (`POST /products`).
3. **Display Products in Django:**
    - Retrieve all products from the Flask API (`GET /products`).
    - Display the list of products in an HTML table on the Django application.

Steps:

1. **Setup the Django Project:**
    - Create a new Django project.
    - Set up necessary Django apps (e.g., `authapp`, `productapp`).
2. **Integrate User Registration and Login with Flask API:**
    - Design the registration and login forms in Django using HTML templates.

- In the `register` view, send the registration data as a `POST` request to the Flask API (`/registerapi`).

- In the `login` view, send the login credentials to the Flask API (`/loginapi`) and retrieve the JWT token.

- Use the JWT_Token for subsequent authenticated requests.

3. **Create Product Registration Form in Django:**

   - Design an HTML page where users can input product details.

   - Send the product data as a `POST` request to the Flask API (`/products`) to add the product to the Flask app.

4. **Display Product List from Flask API in Django:**

   - Create a view in Django to display all products.

   - Send a `GET` request to the Flask API (`/products`) to retrieve the list of products.

   - Display the products in an HTML table within a Django template.

5. **Ensure Proper Authentication:**

   - Secure all product-related actions (adding a new product, fetching product list) by passing the JWT token in the request headers as a `Bearer` token.

Deliverables:

- Django project with integrated forms for user registration, login, and product management.

- Ability to perform CRUD operations for products by communicating with the Flask API.

- Clear HTML templates for displaying user registration/login forms and product data.

Expected Outcome:

- The project will display the user registration and login forms in Django, interact with the Flask API to perform registration and login, and securely add or view products via API calls.