REST API using Django framework

**Core Concepts of Django REST Framework**

1. **Serializers**:
   * **Purpose**: Convert complex data types such as Django models into JSON, XML, or other content types, and vice versa.
   * **Usage**: Define what data should be included in the response and how it should be validated.

# serializers.py

from rest\_framework import serializers

from .models import Book

class BookSerializer(serializers.ModelSerializer):

class Meta:

model = Book

fields = '\_\_all\_\_'

1. **Views**:
   * **Purpose**: Handle HTTP requests and return HTTP responses. In DRF, views provide the API endpoints.
   * **Types**:
     + **Function-based views (FBV)**: Simple functions to handle requests.
     + **Class-based views (CBV)**: Classes that inherit from DRF’s APIView or other generic views.
     + **Viewsets**: A high-level way of handling views that combine multiple actions (like list, create, retrieve, update, and delete).

# views.py

from rest\_framework import viewsets

from .models import Book

from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):

queryset = Book.objects.all()

serializer\_class = BookSerializer

1. **URLs and Routers**:
   * **Purpose**: Map URL patterns to views. Routers provide a simple way to automatically generate the URL patterns for viewsets.
   * **Usage**: Define URL patterns and include routers for viewsets.

# urls.py

from django.urls import path, include

from rest\_framework.routers import DefaultRouter

from .views import BookViewSet

router = DefaultRouter()

router.register(r'books', BookViewSet)

urlpatterns = [

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

]

1. **Authentication and Permissions**:
   * **Purpose**: Secure your API by specifying who can access it and what actions they can perform.
   * **Types**:
     + **Authentication**: Methods to authenticate the user (e.g., TokenAuthentication, SessionAuthentication).
     + **Permissions**: Determine what the authenticated user is allowed to do (e.g., IsAuthenticated, IsAdminUser, custom permissions).

# settings.py

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

'rest\_framework.authentication.TokenAuthentication',

],

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions.IsAuthenticated',

],

}

**Custom Permissions**:

# permissions.py

from rest\_framework.permissions import BasePermission

class IsOwner(BasePermission):

def has\_object\_permission(self, request, view, obj):

return obj.owner == request.user

1. **Pagination**:
   * **Purpose**: Split large querysets into pages, improving performance and usability.
   * **Usage**: DRF provides built-in pagination classes that can be configured globally or per-view.

# settings.py

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

'PAGE\_SIZE': 10,

}

1. **Throttling**:
   * **Purpose**: Control the rate of requests to your API to prevent abuse and manage resources.
   * **Usage**: DRF provides throttling classes like UserRateThrottle, AnonRateThrottle, and custom throttles.

# settings.py

REST\_FRAMEWORK = {

'DEFAULT\_THROTTLE\_CLASSES': [

'rest\_framework.throttling.AnonRateThrottle',

'rest\_framework.throttling.UserRateThrottle',

],

'DEFAULT\_THROTTLE\_RATES': {

'anon': '100/day',

'user': '1000/day',

}

}

**DRF Components**

1. **Generic Views**:
   * **Purpose**: Provide a set of built-in views for common patterns, reducing boilerplate code.
   * **Types**: ListCreateAPIView, RetrieveUpdateDestroyAPIView, etc.

from rest\_framework import generics

from .models import Book

from .serializers import BookSerializer

class BookListCreate(generics.ListCreateAPIView):

queryset = Book.objects.all()

serializer\_class = BookSerializer

1. **ViewSets and Routers**:
   * **ViewSets**: Group related views together (e.g., list, create, retrieve, update, delete).
   * **Routers**: Automatically generate URL patterns for viewsets.

from rest\_framework import routers

from .views import BookViewSet

router = routers.DefaultRouter()

router.register(r'books', BookViewSet)

1. **Serializers**:
   * **ModelSerializer**: Automatically generate a serializer class based on a Django model.
   * **HyperlinkedModelSerializer**: Similar to ModelSerializer, but uses hyperlinks for relationships instead of primary keys.

from rest\_framework import serializers

from .models import Book

class BookSerializer(serializers.HyperlinkedModelSerializer):

class Meta:

model = Book

fields = ['url', 'title', 'author', 'published\_date', 'isbn', 'pages', 'cover', 'language']

**Example Project Structure**

markdown

myproject/

manage.py

myproject/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

myapp/

\_\_init\_\_.py

admin.py

apps.py

models.py

serializers.py

views.py

urls.py

tests.py

**Example Workflow**

1. **Define Models**: Define your data models in models.py.
2. **Create Serializers**: Define serializers to convert model instances to JSON and vice versa.
3. **Create Views**: Create views using viewsets or generic views to handle CRUD operations.
4. **Configure URLs**: Use routers to map URL patterns to viewsets.
5. **Run Server**: Start the development server and test your API endpoints.

By understanding these concepts and components, you can effectively use Django REST Framework to build robust and scalable APIs.

**1. Set Up Your Django Project**

First, ensure you have Django installed:

pip install django

Create a new Django project:

django-admin startproject myproject

cd myproject

Create a new app within the project:

manage.py startapp myapp

Add the new app to your project's settings.py:

# settings.py

INSTALLED\_APPS = [

...

'myapp',

'rest\_framework', # Add this line for DRF

]

**2. Install Django REST Framework**

Install Django REST Framework:

pip install djangorestframework

Add 'rest\_framework' to your INSTALLED\_APPS in settings.py as shown above.

**3. Define Your Models**

Create models for your application. For example, let's create a simple Book model.

# myapp/models.py

from django.db import models

class Book(models.Model):

title = models.CharField(max\_length=255)

author = models.CharField(max\_length=255)

published\_date = models.DateField()

isbn = models.CharField(max\_length=13, unique=True)

pages = models.IntegerField()

cover = models.CharField(max\_length=255)

language = models.CharField(max\_length=255)

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

return self.title

Run the migrations to create the database tables:

manage.py makemigrations

manage.py migrate

**4. Create Serializers**

Serializers are used to convert complex data types like querysets and model instances to native datatypes that can then be easily rendered into JSON, XML, or other content types. They also handle deserialization, allowing parsed data to be converted back into complex types.

# myapp/serializers.py

from rest\_framework import serializers

from .models import Book

class BookSerializer(serializers.ModelSerializer):

class Meta:

model = Book

fields = '\_\_all\_\_'

**5. Create Views**

DRF provides several classes to help you handle requests. The most commonly used are APIView, GenericAPIView, and viewsets. For simplicity, we'll use viewsets and routers in this example.

# myapp/views.py

from rest\_framework import viewsets

from .models import Book

from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):

queryset = Book.objects.all()

serializer\_class = BookSerializer

**6. Define URL Patterns**

To connect the viewsets to URLs, we'll use DRF's routers.

# myapp/urls.py

from django.urls import path, include

from rest\_framework.routers import DefaultRouter

from .views import BookViewSet

router = DefaultRouter()

router.register(r'books', BookViewSet)

urlpatterns = [

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

]

Include these URLs in your project's main URL configuration.

# myproject/urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

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

path('api/', include('myapp.urls')),

]

**7. Test the API**

Run the development server:

manage.py runserver

Navigate to http://127.0.0.1:8000/api/books/ to see the list of books and perform CRUD operations.

**8. Adding Authentication (Optional)**

To secure your API, you might want to add authentication. DRF provides several authentication classes, including TokenAuthentication and SessionAuthentication.

**Install Django REST Framework Auth Token:**

pip install djangorestframework-authtoken

**Add rest\_framework.authtoken to INSTALLED\_APPS:**

# settings.py

INSTALLED\_APPS = [

...

'rest\_framework.authtoken',

]

**Add Token Authentication to your DRF settings:**

# settings.py

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

'rest\_framework.authentication.TokenAuthentication',

],

}

**Create a URL for obtaining auth tokens:**

# myapp/urls.py

from rest\_framework.authtoken.views import obtain\_auth\_token

urlpatterns = [

...

path('api-token-auth/', obtain\_auth\_token),

]

**Migrate the database:**

manage.py migrate

Now, users can obtain a token by posting their username and password to http://127.0.0.1:8000/api/api-token-auth/.

**9. Use Browsable API**

DRF comes with a browsable API feature that helps you to easily navigate through the API using your browser. You can access it by navigating to your API endpoints.

By following these steps, you should have a fully functional REST API using Django and Django REST Framework. This API allows for CRUD operations on a Book model, with the ability to easily extend and customize it as needed.