**Add Swagger API Documentation To Django Rest API | GenericAPIView | part2**

[[Kazi Mushfiqur Rahman](https://medium.com/@KaziMushfiq1234?source=post_page-----5ca6294e66b9--------------------------------)](https://medium.com/@KaziMushfiq1234?source=post_page-----5ca6294e66b9--------------------------------)

[Kazi Mushfiqur Rahman](https://medium.com/@KaziMushfiq1234?source=post_page-----5ca6294e66b9--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F16ea6542ad64&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40KaziMushfiq1234%2Fadd-swagger-api-documentation-to-django-rest-api-genericapiview-part2-5ca6294e66b9&user=Kazi+Mushfiqur+Rahman&userId=16ea6542ad64&source=post_page-16ea6542ad64----5ca6294e66b9---------------------post_header-----------)

8 min read

·

Sep 12, 2023



Photo by [taha siddiqui](https://unsplash.com/@xxtahaxx?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

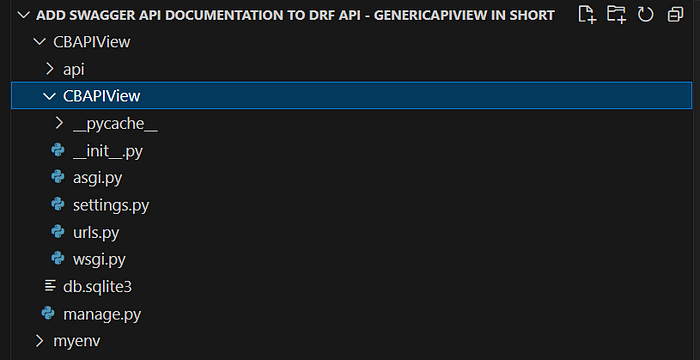
**Complete project in GitHub**

**[GitHub - mushfiq1998/add-swagger-api-documentation-to-drf-api-generic-apiview-part2](https://github.com/mushfiq1998/add-swagger-api-documentation-to-drf-api-generic-apiview-part2?source=post_page-----5ca6294e66b9--------------------------------" \t "_blank)**

[Contribute to mushfiq1998/add-swagger-api-documentation-to-drf-api-generic-apiview-part2 development by creating an…](https://github.com/mushfiq1998/add-swagger-api-documentation-to-drf-api-generic-apiview-part2?source=post_page-----5ca6294e66b9--------------------------------" \t "_blank)

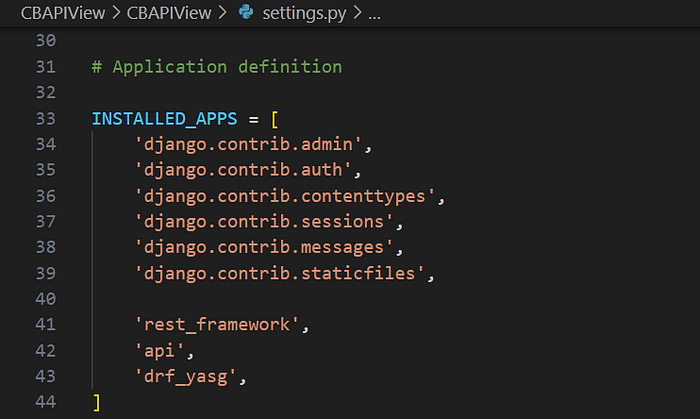
[github.com](https://github.com/mushfiq1998/add-swagger-api-documentation-to-drf-api-generic-apiview-part2?source=post_page-----5ca6294e66b9--------------------------------" \t "_blank)

**Project directory structure**



**Configure settings.py file in inner project folder**

In the following code we have listed three apps — rest\_framework, api and drf\_yasg. Here, api is our custom app and must be included in INSTALLED\_APPS.

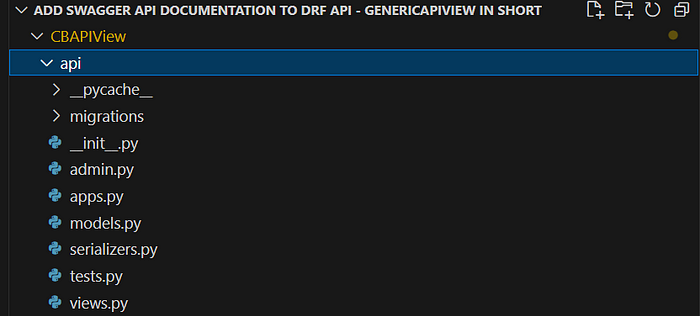


**Define urls**

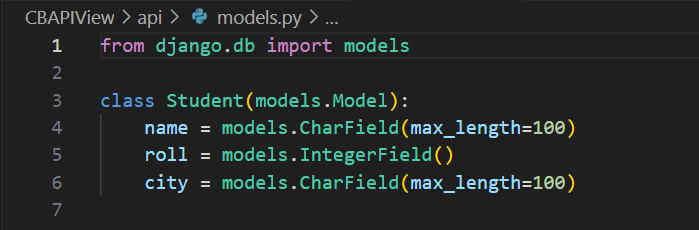




**App directory structure**



**Design database**



**Explanation of the above code**

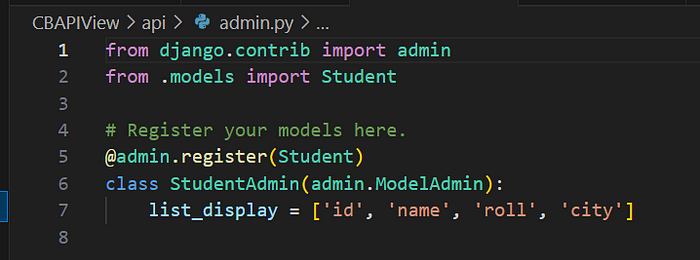
This is a simplified explanation of the Django code I provided:

1. from django.db import models: This line imports th e necessary module (models) from the Django framework for creating database models.
2. class Student(models.Model):: This line defines a Python class named "Student" that inherits from models.Model. In Django, models are used to define the structure and attributes of database tables. So, this class will represent a table in the database called "Student."
3. Inside the “Student” class, you define three fields:

* name = models.CharField(max\_length=100): This creates a field named "name" for the "Student" table, which will store character data (strings) with a maximum length of 100 characters. This field is used to store the name of a student.
* roll = models.IntegerField(): This creates a field named "roll" for the "Student" table, which will store integer data. It's used to store the roll number of a student.
* city = models.CharField(max\_length=100): This creates a field named "city" for the "Student" table, similar to the "name" field, but for storing the city where a student lives.

By defining these fields, you’re essentially specifying the structure of the “Student” table in the database, including the types of data it can store (name as text, roll as an integer, and city as text), as well as any constraints (such as the maximum length of text fields). This allows you to work with student data in your Django application and interact with the database easily.

**Register the designed model to show it in admin site**



**Explanation of the above code**

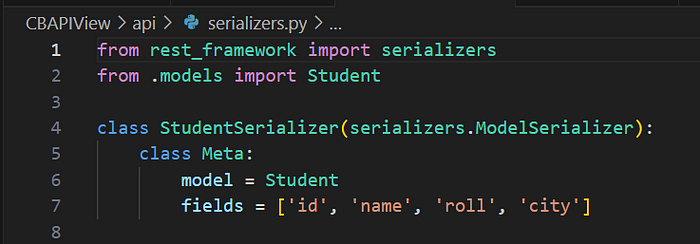
This Django code is for setting up the admin interface to manage your “Student” model in a web-based interface. Let’s break it down step by step:

1. from django.contrib import admin: This line imports the "admin" module from Django. The admin module provides a ready-made interface for managing your application's data models without writing custom views or templates.
2. from .models import Student: This line imports the "Student" model you defined earlier in your code. It allows you to make this model accessible in the admin interface.
3. @admin.register(Student): This is a Python decorator that registers the "Student" model with the admin interface. In simpler terms, it tells Django that you want to create a web-based interface to manage student records.
4. class StudentAdmin(admin.ModelAdmin):: This line defines a custom class named "StudentAdmin" that inherits from "admin.ModelAdmin." This class allows you to customize how the "Student" model is displayed and managed in the admin interface.
5. list\_display = ['id', 'name', 'roll', 'city']: Here, you specify which fields from the "Student" model should be displayed in the list view of the admin interface. In this case, it's the student's ID, name, roll number, and city.

Now, when you navigate to the admin interface of your Django project, you’ll see a section for managing “Student” records. In this section, you can view, add, edit, and delete student records, and the list view will display the ID, name, roll number, and city for each student for easy management and organization of your data.

**Define Serializer**

In a web application, you often need to exchange data between your server (backend) and the client (frontend or other applications). Serializers in Django REST Framework help you convert complex data types, such as Django model instances, into Python data types that can be easily rendered into JSON, XML, or other content types.



**Explanation of the above code**

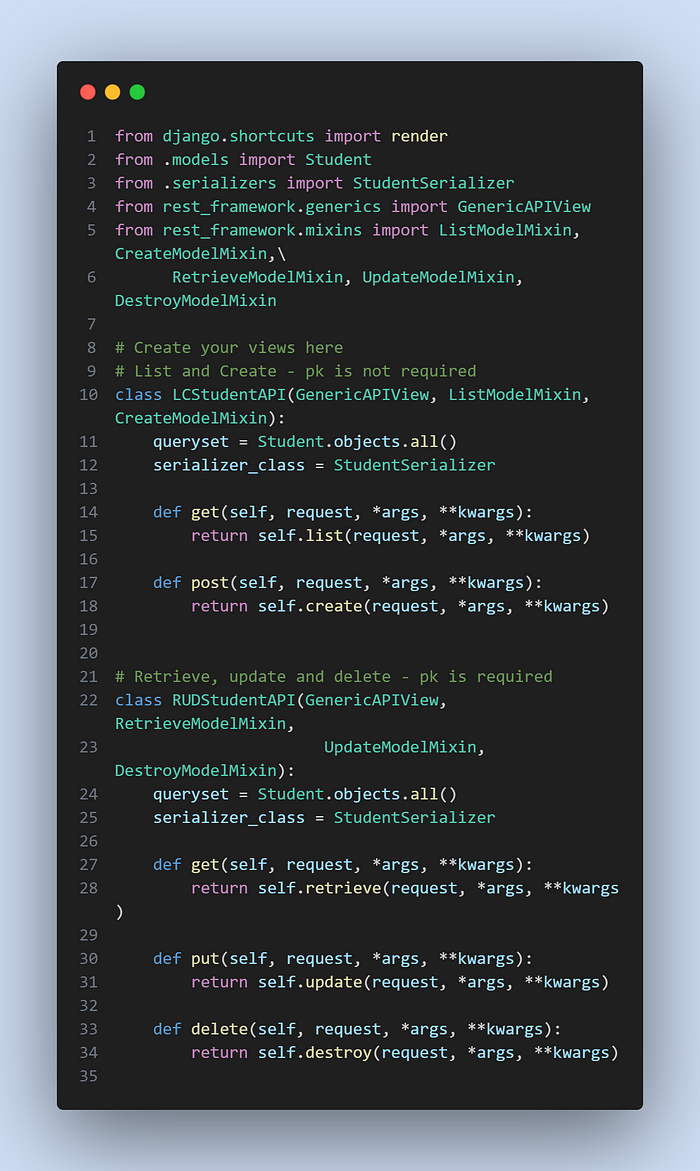
This Django code defines a serializer for the “Student” model using the Django REST framework. Let me explain it in simple terms:

1. from rest\_framework import serializers: This line imports the serializers module from the Django REST framework. Serializers are used to convert complex data types, such as Django model instances, into native Python data types like dictionaries, and vice versa.
2. from .models import Student: This line imports the "Student" model from your Django application. You want to create a serializer that can convert instances of this model into JSON data and vice versa.
3. class StudentSerializer(serializers.ModelSerializer):: Here, you define a Python class called "StudentSerializer" that inherits from "serializers.ModelSerializer." This class is used to define how your "Student" model should be serialized (converted into JSON) and deserialized (parsed from JSON).
4. class Meta:: Inside the "StudentSerializer" class, you define a nested class called "Meta." This is used to provide metadata about the serialization process.
5. model = Student: In the "Meta" class, you specify the model that this serializer is associated with, which is your "Student" model. This tells Django which model to use when serializing or deserializing data.
6. fields = ['id', 'name', 'roll', 'city']: You specify the fields from the "Student" model that you want to include in the serialized output. In this case, you've chosen to include the student's ID, name, roll number, and city.

So, with this “StudentSerializer,” you can easily convert instances of the “Student” model to JSON format, which is commonly used for API responses, and also parse JSON data back into Python objects when creating or updating student records through your API. This simplifies the process of working with your Django model in the context of a RESTful API.

**Rendering Student Information as JSON Response**





**Explanation of the above code**

The code is for defining two views in a Django REST framework (DRF) API for handling CRUD (Create, Read, Update, Delete) operations on a “Student” model. These views are implemented as Django class-based views using DRF mixins to handle the different HTTP methods for each operation.

Here’s an explanation of the code:

1. Importing necessary modules:

* render: This is used for rendering HTML templates, but it's not used in the provided code.
* Student: This is assumed to be a Django model representing student data.
* StudentSerializer: This is assumed to be a serializer class for the "Student" model.
* Various mixins and GenericAPIView classes from DRF are imported for creating the views.

2. Defining the LCStudentAPI class:

* This class inherits from GenericAPIView and uses the mixins ListModelMixin and CreateModelMixin.
* queryset is set to retrieve all objects from the "Student" model.
* serializer\_class is set to use the "StudentSerializer" for serialization.

3. The purpose of this class is to handle two HTTP methods:

* GET: It uses the list method from ListModelMixin to retrieve a list of all students.
* POST: It uses the create method from CreateModelMixin to create a new student record.

4. Defining the RUDStudentAPI class:

* This class inherits from GenericAPIView and uses the mixins RetrieveModelMixin, UpdateModelMixin, and DestroyModelMixin.
* Like the previous class, queryset is set to retrieve all objects from the "Student" model, and serializer\_class is set to use the "StudentSerializer" for serialization.

5. The purpose of this class is to handle three HTTP methods:

* GET: It uses the retrieve method from RetrieveModelMixin to retrieve the details of a specific student by their primary key (pk).
* PUT: It uses the update method from UpdateModelMixin to update the details of a specific student by their pk.
* DELETE: It uses the destroy method from DestroyModelMixin to delete a specific student by their pk.

These two classes provide a set of views to perform CRUD operations on the “Student” model in your Django application. Depending on the HTTP method used (GET, POST, PUT, DELETE), the appropriate mixin methods are invoked to perform the corresponding operation on the database using the provided serializer. This is a common pattern in DRF for creating RESTful APIs for Django models.

**To run the project complete the following tasks**

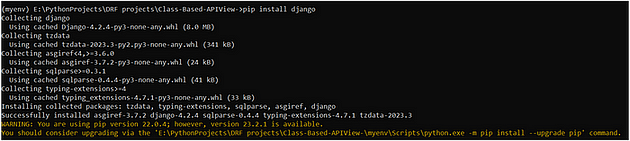
Create a virtual environment using the following command



Activate the virtual environment using the following command



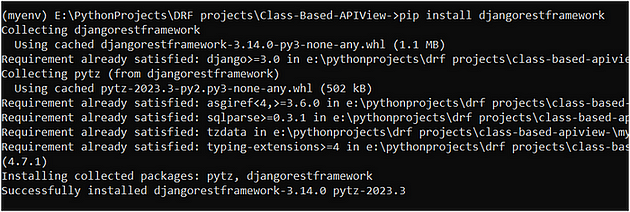
Install django using the following command



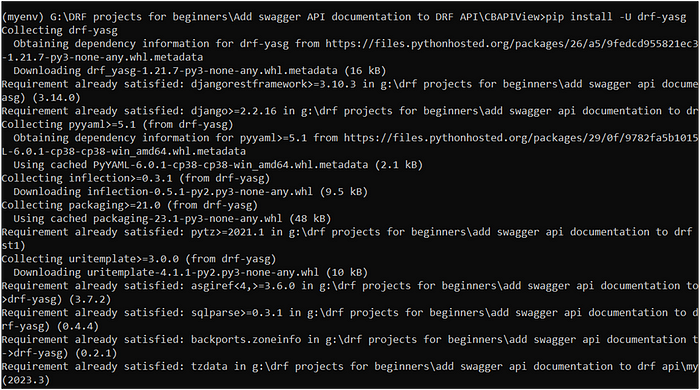
Upgrade pip using the following command



Install django rest framework using the following command

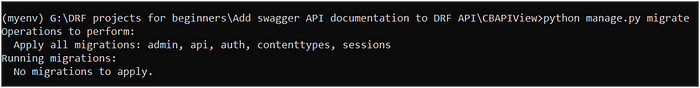


**Install drf\_yasg**

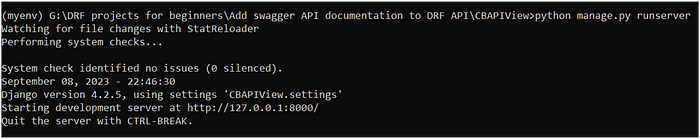


**Run the migration command**





**Run the server**



**Now, hit the following url to show swagger API documentation**



