

Django REST framework

Shad Sheikh

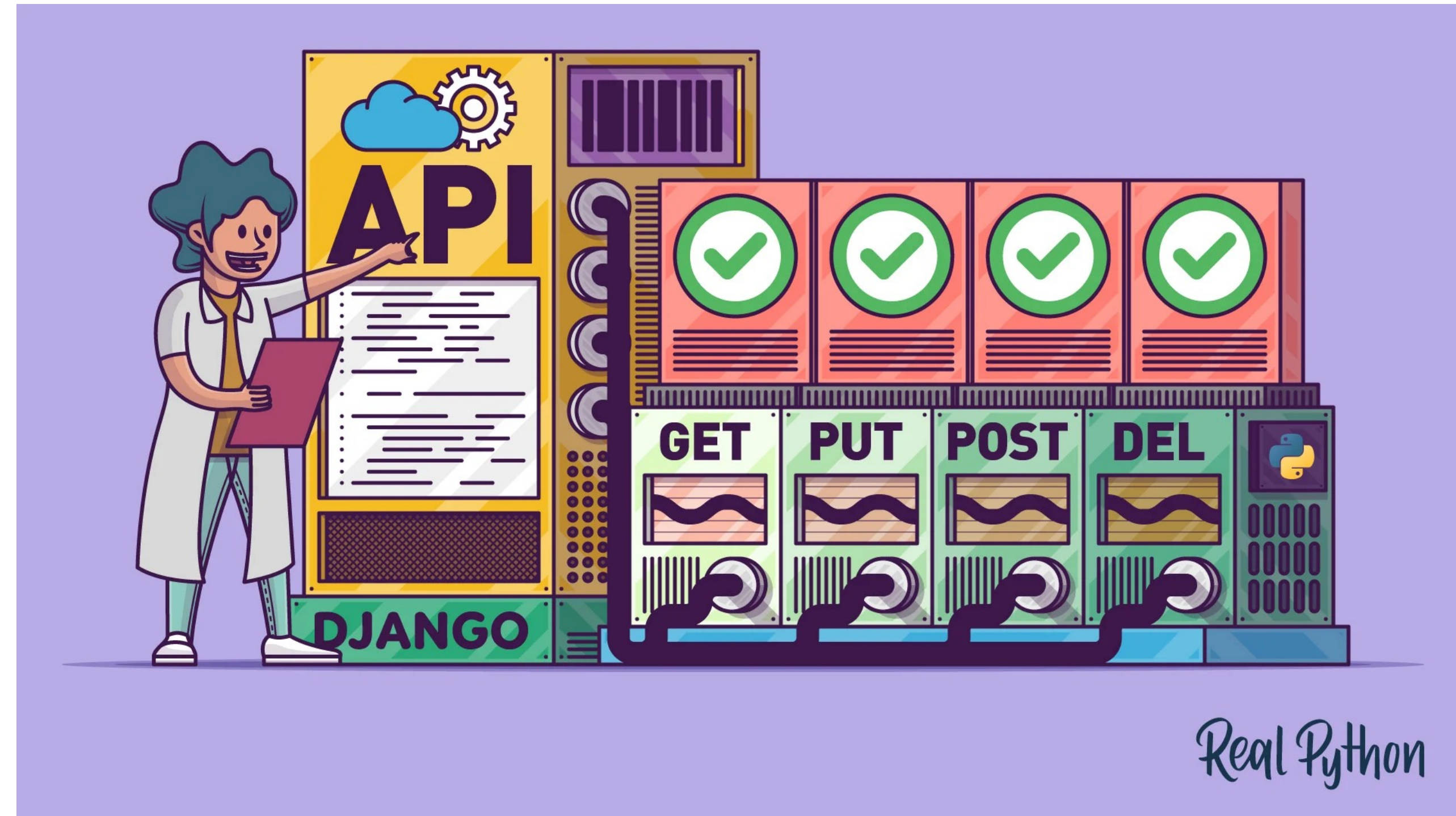
Introduction to REST Django Framework

- Django Rest Framework (DRF) is a powerful and flexible toolkit for building Web APIs (Application Programming Interfaces) using Django. It provides a set of tools and utilities that make it easier to create, test, and manage APIs in Django.




Create and Run Project

- Installation: To use Django Rest Framework, you need to install it into your Django project. You can do this by running the following command:
`pip install djangorestframework`
- Create Project
`django-admin startproject project_name`
`python3 manage.py startapp app_name`
- Run Project
`Python3 manage.py runserver`



Serialization

- Serialization in Django Rest Framework (DRF) refers to the process of converting complex Python objects (such as model instances) into a format that can be easily transmitted over the network, typically in JSON or XML format. It allows you to represent your data in a structured manner, making it easier to transmit, consume, and manipulate.
- In DRF, serializers are the key components for serialization. Serializers allow you to control how your data is represented and deserialized. They define the fields and behavior for converting complex data types to primitive data types (such as strings, numbers, lists, etc.) and vice versa.
- Serialization is a fundamental concept in DRF as it allows you to transform your data into a format that can be easily consumed by other applications or clients. DRF's serializers provide a flexible and powerful way to control the serialization and deserialization process, making it easier to build APIs that transmit and receive data efficiently.

 serializers.py U ×

Django REST framework > serialization > api >  serializers.py >  StudentSerializer

```
1  from rest_framework import serializers
2
3  class StudentSerializer(serializers.Serializer):
4      # id = serializers.IntegerField()
5      name = serializers.CharField(max_length=100)
6      roll = serializers.IntegerField()
7      city = serializers.CharField(max_length=100)
```

Request and Response


- In Django Rest Framework (DRF), the request and response objects are essential components for handling incoming requests and generating appropriate responses in your API views.



- Request Object:

- The request object represents the incoming HTTP request made by a client. It provides access to various details and data associated with the request.
- The request object includes information such as the request method (GET, POST, PUT, DELETE, etc.), headers, URL parameters, query parameters, and the body of the request (data sent in the request payload).
- DRF enhances the request object provided by Django by adding additional functionality and methods specific to API development.
- You can access the request object in your API views or viewsets by using the `request` parameter passed to your view function or by accessing it from the `self.request` attribute within a class-based view.

- Response Object:

- The response object represents the HTTP response sent back to the client after processing a request.
- DRF provides a `Response` class that extends Django's `HttpResponse` class and adds additional features and convenience methods for API responses.
- The `Response` class allows you to easily return structured data in various formats like JSON, XML, or HTML.
- You can create a response object by instantiating the `Response` class and passing the data you want to include in the response.
- DRF also provides helper methods like `Response()` and `Response.status_code` to create responses with predefined HTTP status codes, such as `Response(status=status.HTTP_200_OK)` OR `Response(data, status=status.HTTP_201_CREATED)`.

 views.py U X

```
Django REST framework > serialization > api >  views.py >  student_list
1  from django.shortcuts import render
2  from .models import Student
3  from .serializers import StudentSerializer
4  from rest_framework.renderers import JSONRenderer
5  from django.http import HttpResponse, JsonResponse
6  # Create your views here.
7
8  def student_detail(request, pk):
9      stu = Student.objects.get(id=pk)
10     serializer = StudentSerializer(stu)
11     return JsonResponse(serializer.data)
12
13  def student_list(request):
14     stu = Student.objects.all()
15     serializer = StudentSerializer(stu, many=True)
16     return JsonResponse(serializer.data, safe=False)
```


Class base view

- In Django Rest Framework (DRF), class-based views (CBVs) are a powerful and flexible way to define API views. CBVs allow you to organize your code into reusable and structured classes, providing a clear separation of concerns.
- In DRF, you typically define class-based views by subclassing one of the provided generic class-based views such as `APIView`, `ViewSet`, or specific mixin classes. You define the class attributes and methods to handle different HTTP methods and other functionalities required by your API. For example, you can define a class-based view that inherits from `APIView` and implement the `get()` and `post()` methods to handle GET and POST requests, respectively.
- Common Methods in Class-Based Views:
 - `get()`: Handles HTTP GET requests and returns the requested data.
 - `post()`: Handles HTTP POST requests and creates new resources.
 - `put()`: Handles HTTP PUT requests and updates existing resources.
 - `delete()`: Handles HTTP DELETE requests and deletes resources.
 - `patch()`: Handles partial updates to existing resources.

```
14 @method_decorator(csrf_exempt, name='dispatch')
15 class StudentAPI(View):
16     def get(self, request, *args, **kwargs):
17         json_data = request.body
18         stream = io.BytesIO(json_data)
19         pythondata = JSONParser().parse(stream)
20         id = pythondata.get('id', None)
21         if id is not None:
22             stu = Student.objects.get(id=id)
23             serializer = StudentSerializer(stu)
24             json_data = JSONRenderer().render(serializer.data)
25             return HttpResponse(json_data, content_type='application/json')
26         stu = Student.objects.all()
27         serializer = StudentSerializer(stu, many=True)
28         json_data = JSONRenderer().render(serializer.data)
29         return HttpResponse(json_data, content_type='application/json')
30
```

Authentication and Permission

- Authentication and permissions are essential components in Django Rest Framework (DRF) that help secure your API endpoints and control access to resources.
- Authentication:
 - Authentication refers to the process of verifying the identity of a user or client making a request to your API.
 - DRF provides various authentication mechanisms to handle user authentication, including session-based authentication, token-based authentication, OAuth, and more.
 - To enable authentication, you specify the authentication classes in your view or viewset. DRF will check the authentication classes in the order they are defined and use the first one that successfully authenticates the user.
 - Once a user is authenticated, DRF associates the user with the current request, allowing you to access the authenticated user in your views or serializers.
- Permissions:
 - Permissions determine whether a user is allowed to access a particular resource or perform a specific action.
 - DRF provides a variety of permission classes that you can use to control access to your API views.
 - Permissions can be set at the global level in your project settings or at the view or viewset level.
 - Common permission classes in DRF include:
 - `IsAuthenticated`: Requires that the user making the request is authenticated.
 - `IsAdminUser`: Requires that the user is an admin user.
 - `AllowAny`: Allows unrestricted access, meaning the permission check is skipped.
 - `IsAuthenticatedOrReadOnly`: Allows read access to unauthenticated users but requires authentication for write operations.
 - CustomPermission: You can also create your own custom permission classes by subclassing BasePermission and implementing the has_permission() or has_object_permission() methods.

```
1  from rest_framework import permissions
2
3
4  class IsOwnerOrReadOnly(permissions.BasePermission):
5      def has_object_permission(self, request, view, obj):
6
7          if request.method in permissions.SAFE_METHODS:
8              return True
9
10         return obj.owner == request.user
```

Relationship

- In Django Rest Framework (DRF), relationships are used to define the associations between different models in an API. Relationships allow you to represent connections and dependencies between resources.
- Relationships within our API are represented by using primary keys. In this part of the tutorial we'll improve the cohesion and discoverability of our API, by instead using hyperlinking for relationships.
- Representing Relationships: DRF provides serializers to represent relationships in your API. You can define the relationship field in your serializer class to specify how the related models should be represented. DRF provides different types of relationship fields like `PrimaryKeyRelatedField`, `StringRelatedField`, `HyperlinkedRelatedField`, and more.

```
1  from django.db import models
2
3  class Author(models.Model):
4      name = models.CharField(max_length=100)
5
6      def __str__(self):
7          return self.name
8
9
10 class Book(models.Model):
11     title = models.CharField(max_length=100)
12     author = models.ForeignKey(Author, on_delete=models.CASCADE, related_name='books')
13
14     def __str__(self):
15         return self.title
16
```


Viewsets and Router

- REST framework includes an abstraction for dealing with ViewSets, that allows the developer to concentrate on modeling the state and interactions of the API, and leave the URL construction to be handled automatically, based on common conventions.
- ViewSet classes are almost the same thing as View classes, except that they provide operations such as retrieve, or update, and not method handlers such as get or put.
- A ViewSet class is only bound to a set of method handlers at the last moment, when it is instantiated into a set of views, typically by using a Router class which handles the complexities of defining the URL conf for you.
- Because we're using ViewSet classes rather than View classes, we actually don't need to design the URL conf ourselves. The conventions for wiring up resources into views and urls can be handled automatically, using a Router class. All we need to do is register the appropriate view sets with a router, and let it do the rest.

```
4 snippet_list = SnippetViewSet.as_view({
5     'get': 'list',
6     'post': 'create'
7 })
8
9 snippet_detail = SnippetViewSet.as_view({
10     'get': 'retrieve',
11     'put': 'update',
12     'patch': 'partial_update',
13     'delete': 'destroy'
14 })
15
16 snippet_highlight = SnippetViewSet.as_view({
17     'get': 'highlight'
18 }, renderer_classes=[renderers.StaticHTMLRenderer])
19 user_list = UserViewSet.as_view({
20     'get': 'list'
21 })
22
23 user_detail = UserViewSet.as_view({
24     'get': 'retrieve'
25 })
```




Thank You

Shad Sheikh