**VIEWSETS AND ROUTERS**

Django REST framework's **ViewSets** provide an elegant way to define your API views. They combine the logic for a set of related views into a single class, reducing redundancy and making the code easier to manage. Here's a detailed explanation of **ViewSets** in Django REST framework (DRF):

**Overview of ViewSets**

A **ViewSet** is a class-based view that allows you to group together related views, typically those that operate on the same model or resource. Instead of defining separate views for list, create, retrieve, update, and delete operations, you can define a single **ViewSet** that handles all these actions.

**Types of ViewSets**

DRF provides several types of ViewSets, each with different levels of customization:

1. **ViewSet**: The most basic type. It provides the foundational structure but doesn't implement any default actions.
2. **GenericViewSet**: Extends **ViewSet** and integrates DRF's generic views, allowing you to use mixins to add actions.
3. **ModelViewSet**: Extends **GenericViewSet** and includes default implementations for list, create, retrieve, update, and destroy actions, based on a model.

In Django REST Framework, generics and viewsets are both tools for building APIs, but they serve slightly different purposes and offer different levels of abstraction and functionality:

1. **Generics Views**:
   * Generics are classes provided by DRF that help in quickly creating common patterns in views. Examples include GenericAPIView, ListAPIView, CreateAPIView, RetrieveAPIView, DestroyAPIView, UpdateAPIView, etc.
   * They provide pre-defined behaviors for common operations like listing all instances, retrieving a single instance, creating an instance, updating an instance, or deleting an instance.
   * Generics are useful when you want to quickly create views for standard CRUD operations with minimal customization.
   * They are more explicit and require you to define each view separately, which gives you fine-grained control over each endpoint.
2. **Viewsets**:
   * Viewsets are a higher-level abstraction that groups the common actions of a set of views into a single class. They are implemented using the ViewSet class provided by DRF.
   * Viewsets typically provide actions such as list, create, retrieve, update, and destroy for a resource. They consolidate these actions into methods like list(), create(), retrieve(), etc.
   * They promote DRY (Don't Repeat Yourself) principles by reducing the amount of code duplication.
   * Viewsets also support additional actions beyond the standard CRUD operations through custom methods defined within the class.

**Key Differences**:

* **Abstraction Level**: Viewsets provide a higher level of abstraction by grouping related views together, while generics are more granular and explicit.
* **Code Reusability**: Viewsets promote code reusability by consolidating common actions, whereas generics require more explicit definition of each view.
* **Customization**: Generics allow for more customization per view, whereas viewsets may require additional customization through mixins or overriding methods.

**Choosing Between Them**:

* Use **generics** when you need fine-grained control over each endpoint's behavior and want to explicitly define each view.
* Use **viewsets** when you have a resource that requires standard CRUD operations and you want to reduce code duplication by consolidating common actions into a single class.

In many cases, you can combine both approaches by using generics for some views and viewsets for others within the same project, depending on your API design needs.

**Example of a ModelViewSet**

Here’s a step-by-step example using **ModelViewSet**.

1. **Define the Model**:

from django.db import models

class Book(models.Model):

    title = models.CharField(max\_length=100)

    author = models.CharField(max\_length=100)

    published\_date = models.DateField()

    def \_\_str\_\_(self):

        return self.title

1. **Create a Serializer**:

from rest\_framework import serializers

from .models import Book

class BookSerializer(serializers.ModelSerializer):

    class Meta:

        model = Book

        fields = '\_\_all\_\_'

1. **Define the ViewSet**:

from rest\_framework import viewsets

from .models import Book

from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):

    queryset = Book.objects.all()

    serializer\_class = BookSerializer

**Customizing ViewSets**

You can customize the default behavior by overriding methods in the **ViewSet**. For example, to add custom behavior to the **create** method:

class BookViewSet(viewsets.ModelViewSet):

    queryset = Book.objects.all()

    serializer\_class = BookSerializer

    def create(self, request, \*args, \*\*kwargs):

        # Custom logic here

        response = super().create(request, \*args, \*\*kwargs)

        # Custom response modification here

        return response

**Adding Additional Actions**

You can add custom actions to a **ViewSet** using the **@action** decorator from **rest\_framework.decorators**:

from rest\_framework.decorators import action

from rest\_framework.response import Response

class BookViewSet(viewsets.ModelViewSet):

    queryset = Book.objects.all()

    serializer\_class = BookSerializer

    @action(detail=False, methods=['get'])

    def recent\_books(self, request):

        recent\_books = Book.objects.filter(published\_date\_\_year\_\_gte=2023)

        serializer = self.get\_serializer(recent\_books, many=True)

        return Response(serializer.data)

In this example, a custom action **recent\_books** is added, accessible via **GET /books/recent\_books/**.

**Conclusion**

**ViewSets** in Django REST framework streamline the development of RESTful APIs by encapsulating common CRUD operations into a single class. This approach reduces boilerplate code and enhances maintainability. By leveraging **ModelViewSet** and the built-in routing capabilities, you can rapidly develop and expose your models through RESTful APIs with minimal configuration.

**Overriding Default Methods**

You might need to override default methods such as **list**, **create**, **retrieve**, **update**, and **destroy** to add custom logic.

Example: Overriding **list** method to filter results:

class BookViewSet(viewsets.ModelViewSet):

    queryset = Book.objects.all()

    serializer\_class = BookSerializer

    def list(self, request, \*args, \*\*kwargs):

        queryset = self.get\_queryset().filter(published\_date\_\_year=2023)

        serializer = self.get\_serializer(queryset, many=True)

        return Response(serializer.data)

**Using Mixins**

**GenericViewSet** allows you to create a custom combination of CRUD operations using mixins like **CreateModelMixin**, **RetrieveModelMixin**, **UpdateModelMixin**, **DestroyModelMixin**, and **ListModelMixin**.

Example: Custom ViewSet using mixins:

from rest\_framework import mixins, viewsets

class BookViewSet(mixins.ListModelMixin,

                  mixins.RetrieveModelMixin,

                  viewsets.GenericViewSet):

    queryset = Book.objects.all()

    serializer\_class = BookSerializer

This example only provides the **list** and **retrieve** actions.

**Django Routers**

Django REST framework (DRF) routers handle the complex URL routing for ViewSets, streamlining the process of mapping URL patterns to the appropriate view logic. Here's a detailed look at how routers work in DRF:

**Overview of Routers**

Routers in DRF are responsible for automatically generating URL patterns for ViewSets, saving you from manually defining each URL configuration. They determine the mapping between HTTP verbs (GET, POST, PUT, DELETE, etc.) and the corresponding view actions (list, create, retrieve, update, destroy).

**Types of Routers**

DRF provides several types of routers, each catering to different needs:

1. **DefaultRouter**: The most commonly used router. It includes a default API root view and automatic URL routing for ViewSets.
2. **SimpleRouter**: Similar to **DefaultRouter** but does not include the default API root view.
3. **Route**: For defining custom routes within a router.

**Basic Usage of DefaultRouter**

1. **Configure the Router**:

from rest\_framework.routers import DefaultRouter

from django.urls import path, include

from .views import BookViewSet

router = DefaultRouter()

router.register(r'books', BookViewSet)

urlpatterns = [

    path('', include(router.urls)),

]

This configuration will automatically generate the following URL patterns:

* **GET /books/** to list all books (**list** action).
* **POST /books/** to create a new book (**create** action).
* **GET /books/{id}/** to retrieve a specific book (**retrieve** action).
* **PUT /books/{id}/** to update a specific book (**update** action).
* **PATCH /books/{id}/** to partially update a specific book (**partial\_update** action).
* **DELETE /books/{id}/** to delete a specific book (**destroy** action).

**Using SimpleRouter**

**SimpleRouter** works like **DefaultRouter** but does not include an API root view. This can be useful if you want to avoid exposing the API root view.

Example:

from rest\_framework.routers import SimpleRouter

router = SimpleRouter()

router.register(r'books', BookViewSet)

urlpatterns = [

    path('', include(router.urls)),

]

**Nested Routers**

DRF does not natively support nested routing out of the box, but you can achieve it using third-party packages like **drf-nested-routers**.

Example:

1. **Install drf-nested-routers**:

pip install drf-nested-routers

1. **Configure Nested Routers**:

from rest\_framework\_nested.routers import NestedDefaultRouter

from .views import AuthorViewSet, BookViewSet

router = DefaultRouter()

router.register(r'authors', AuthorViewSet)

authors\_router = NestedDefaultRouter(router, r'authors', lookup='author')

authors\_router.register(r'books', BookViewSet, basename='author-books')

urlpatterns = [

    path('', include(router.urls)),

    path('', include(authors\_router.urls)),

]

This configuration creates nested routes like:

* **GET /authors/1/books/** to list books for a specific author.
* **POST /authors/1/books/** to create a book for a specific author.

**Custom Routes**

Sometimes you may need to add custom routes beyond the standard CRUD operations. You can achieve this using the **@action** decorator in your ViewSet.

Example:

from rest\_framework.decorators import action

from rest\_framework.response import Response

class BookViewSet(viewsets.ModelViewSet):

    queryset = Book.objects.all()

    serializer\_class = BookSerializer

    @action(detail=False, methods=['get'])

    def recent\_books(self, request):

        recent\_books = Book.objects.filter(published\_date\_\_year=timezone.now().year)

        serializer = self.get\_serializer(recent\_books, many=True)

        return Response(serializer.data)

This custom action will be accessible via **GET /books/recent\_books/**.

**Customizing Route URL Patterns**

You can customize the URL patterns by using the **urls** property of the router and manually adjusting them.

Example:

urlpatterns = [

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

    path('custom-books/<int:id>/', BookDetailView.as\_view(), name='custom-book-detail'),

]

**Conclusion**

Django REST framework routers simplify the process of creating and managing URL routes for your ViewSets, providing a declarative and automated way to handle routing. By using **DefaultRouter**, **SimpleRouter**, and third-party solutions for nested routing, you can efficiently map your API endpoints. Custom actions and custom routes further extend the flexibility of routers, allowing you to tailor the routing logic to fit your specific requirements.

Of course! Let's explore more advanced features and customization options of Django REST framework routers.

**Handling Nested Resources**

Nested resources represent relationships between different models, and sometimes you need to expose these relationships in your API. While DRF does not directly support nested routing, you can achieve it using custom URL patterns and view logic.

**Manual Nested Routing**

Example:

from django.urls import path

from .views import AuthorBooksAPIView

urlpatterns = [

    path('authors/<int:author\_id>/books/', AuthorBooksAPIView.as\_view(), name='author-books'),

]

Here, **AuthorBooksAPIView** is a custom view that retrieves books associated with a specific author.

**Using Third-Party Packages**

Alternatively, you can use third-party packages like **drf-nested-routers** for more automated nested routing.

Example:

from rest\_framework\_nested.routers import NestedDefaultRouter

from .views import AuthorViewSet, BookViewSet

router = DefaultRouter()

router.register(r'authors', AuthorViewSet)

nested\_router = NestedDefaultRouter(router, r'authors', lookup='author')

nested\_router.register(r'books', BookViewSet, basename='author-books')

urlpatterns = [

    path('', include(router.urls)),

    path('', include(nested\_router.urls)),

]

This configuration creates nested routes like **/authors/{author\_id}/books/**.

**Customizing Router Behavior**

**Overriding Default Router Behavior**

You can subclass the default router classes to customize their behavior according to your requirements.

Example:

from rest\_framework.routers import DefaultRouter

class CustomRouter(DefaultRouter):

    def \_\_init\_\_(self, \*args, \*\*kwargs):

        super().\_\_init\_\_(\*args, \*\*kwargs)

        # Custom initialization code

router = CustomRouter()

router.register(r'books', BookViewSet)

**Using Custom Route Classes**

DRF allows you to define custom route classes if you need more control over URL routing.

Example:

from rest\_framework.routers import Route, DynamicDetailRoute, SimpleRouter

class CustomRouter(SimpleRouter):

    routes = [

        Route(

            url=r'^{prefix}{trailing\_slash}$',

            mapping={'get': 'list', 'post': 'create'},

            name='{basename}-list',

            initkwargs={'suffix': 'List'}

        ),

        DynamicDetailRoute(

            url=r'^{prefix}/{url\_path}{trailing\_slash}$',

            name='{basename}-{url\_name}',

            initkwargs={}

        ),

    ]

router = CustomRouter()

router.register(r'books', BookViewSet)

**URL Namespaces and Reverse URL Resolution**

Namespaces allow you to organize URLs into distinct groups, which is useful for avoiding naming conflicts and improving code readability.

Example:

app\_name = 'library'

urlpatterns = [

    path('api/', include((router.urls, 'api'), namespace='api')),

]

With namespaces, you can use **reverse()** to generate URLs within your application.

Example:

from django.urls import reverse

url = reverse('api:book-list')

**Excluding Routes**

Sometimes you may need to exclude certain routes from the router. You can achieve this by overriding the **get\_default\_base\_name** method in your ViewSet and returning **None** for the routes you want to exclude.

Example:

class CustomViewSet(viewsets.ModelViewSet):

    queryset = Book.objects.all()

    serializer\_class = BookSerializer

    def get\_default\_base\_name(self):

        if self.action == 'custom\_action':

            return None

        return super().get\_default\_base\_name()

    @action(detail=False, methods=['get'])

    def custom\_action(self, request):

        # Custom action logic

        pass

**PROJECT**

**django-admin startproject food\_delivery\_api**

**cd food\_delivery\_api**

**python manage.py delivery**

**pip install dajngorestframework**

settinngs.py

'rest\_framework',

    'delivery',

Models.py

from django.db import models

class Restaurant(models.Model):

    name = models.CharField(max\_length=100)

    address = models.CharField(max\_length=200)

    def \_\_str\_\_(self):

        return self.name

class MenuItem(models.Model):

    name = models.CharField(max\_length=100)

    description = models.TextField()

    price = models.DecimalField(max\_digits=8, decimal\_places=2)

    restaurant = models.ForeignKey(Restaurant, related\_name='menu\_items', on\_delete=models.CASCADE)

    def \_\_str\_\_(self):

        return self.name

class Customer(models.Model):

    name = models.CharField(max\_length=100)

    address = models.CharField(max\_length=200)

    def \_\_str\_\_(self):

        return self.name

class Order(models.Model):

    customer = models.ForeignKey(Customer, related\_name='orders', on\_delete=models.CASCADE)

    items = models.ManyToManyField(MenuItem, related\_name='orders')

    total\_price = models.DecimalField(max\_digits=8, decimal\_places=2)

    created\_at = models.DateTimeField(auto\_now\_add=True)

    def \_\_str\_\_(self):

        return f"Order #{self.pk} by {self.customer.name}"

serializers.py

from rest\_framework import serializers

from .models import Restaurant, MenuItem, Customer, Order

class RestaurantSerializer(serializers.ModelSerializer):

    class Meta:

        model = Restaurant

        fields = '\_\_all\_\_'

class MenuItemSerializer(serializers.ModelSerializer):

    class Meta:

        model = MenuItem

        fields = '\_\_all\_\_'

class CustomerSerializer(serializers.ModelSerializer):

    class Meta:

        model = Customer

        fields = '\_\_all\_\_'

class OrderSerializer(serializers.ModelSerializer):

    class Meta:

        model = Order

        fields = '\_\_all\_\_'

views.py

from rest\_framework import viewsets

from .models import Restaurant, MenuItem, Customer, Order

from .serializers import RestaurantSerializer, MenuItemSerializer, CustomerSerializer, OrderSerializer

class RestaurantViewSet(viewsets.ModelViewSet):

    queryset = Restaurant.objects.all()

    serializer\_class = RestaurantSerializer

class MenuItemViewSet(viewsets.ModelViewSet):

    queryset = MenuItem.objects.all()

    serializer\_class = MenuItemSerializer

class CustomerViewSet(viewsets.ModelViewSet):

    queryset = Customer.objects.all()

    serializer\_class = CustomerSerializer

class OrderViewSet(viewsets.ModelViewSet):

    queryset = Order.objects.all()

    serializer\_class = OrderSerializer

project/urls.py

from rest\_framework.routers import DefaultRouter

from django.urls import path, include

from django.contrib import admin

from delivery.views import RestaurantViewSet, MenuItemViewSet, CustomerViewSet, OrderViewSet

# Set up the DefaultRouter and register viewsets

router = DefaultRouter()

router.register(r'restaurants', RestaurantViewSet)

router.register(r'menu-items', MenuItemViewSet)

router.register(r'customers', CustomerViewSet)

router.register(r'orders', OrderViewSet)

# Define the URL patterns for both the API and the admin interface

urlpatterns = [

    path('admin/', admin.site.urls),  # Admin interface URL

    path('api/', include(router.urls)),  # API endpoints for the app

]

To expand your Django REST API with additional models and incorporate a third-party package for enhanced routing, let's go through some steps. I'll add some additional models and suggest using **drf-spectacular** for automated API documentation as well.

### 1. Update models.py with Additional Models

Let's add some extra models like Category and DeliveryDriver to the existing ones for better organization and data representation.

# models.py

from django.db import models

class Category(models.Model):

name = models.CharField(max\_length=50)

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

return self.name

class DeliveryDriver(models.Model):

name = models.CharField(max\_length=100)

vehicle\_type = models.CharField(max\_length=50)

is\_available = models.BooleanField(default=True)

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

return self.name

### 2. Add Serializers for the New Models

Create serializers for Category and DeliveryDriver to handle serialization and deserialization.

python

Copy code

# serializers.py

from rest\_framework import serializers

from .models import Restaurant, MenuItem, Customer, Order, Category, DeliveryDriver

class CategorySerializer(serializers.ModelSerializer):

class Meta:

model = Category

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

class DeliveryDriverSerializer(serializers.ModelSerializer):

class Meta:

model = DeliveryDriver

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

### 3. Update Views with ViewSets for the New Models

Now, add viewsets for Category and DeliveryDriver in views.py.

python

Copy code

# views.py

from rest\_framework import viewsets

from .models import Restaurant, MenuItem, Customer, Order, Category, DeliveryDriver

from .serializers import (

RestaurantSerializer, MenuItemSerializer, CustomerSerializer, OrderSerializer,

CategorySerializer, DeliveryDriverSerializer

)

class CategoryViewSet(viewsets.ModelViewSet):

queryset = Category.objects.all()

serializer\_class = CategorySerializer

class DeliveryDriverViewSet(viewsets.ModelViewSet):

queryset = DeliveryDriver.objects.all()

serializer\_class = DeliveryDriverSerializer

### 4. Add Third-Party Package for API Documentation (e.g., drf-spectacular)

Install drf-spectacular for generating a schema and API documentation:

bash

Copy code

pip install drf-spectacular

Add drf\_spectacular to INSTALLED\_APPS in settings.py:

python

Copy code

# settings.py

INSTALLED\_APPS = [

...

'rest\_framework',

'drf\_spectacular',

'delivery',

]

Configure drf-spectacular in settings.py:

python

Copy code

REST\_FRAMEWORK = {

'DEFAULT\_SCHEMA\_CLASS': 'drf\_spectacular.openapi.AutoSchema',

}

SPECTACULAR\_SETTINGS = {

'TITLE': 'Food Delivery API',

'DESCRIPTION': 'API documentation for the food delivery system',

'VERSION': '1.0.0',

}

### 5. Update urls.py for Router and Schema Endpoints

In urls.py, register the new viewsets and add endpoints for the API schema and documentation.

python

Copy code

# urls.py

from rest\_framework.routers import DefaultRouter

from django.urls import path, include

from django.contrib import admin

from delivery.views import (

RestaurantViewSet, MenuItemViewSet, CustomerViewSet, OrderViewSet,

CategoryViewSet, DeliveryDriverViewSet

)

from drf\_spectacular.views import SpectacularAPIView, SpectacularSwaggerView

router = DefaultRouter()

router.register(r'restaurants', RestaurantViewSet)

router.register(r'menu-items', MenuItemViewSet)

router.register(r'customers', CustomerViewSet)

router.register(r'orders', OrderViewSet)

router.register(r'categories', CategoryViewSet)

router.register(r'drivers', DeliveryDriverViewSet)

urlpatterns = [

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

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

path('api/schema/', SpectacularAPIView.as\_view(), name='schema'),

path('api/docs/', SpectacularSwaggerView.as\_view(url\_name='schema'), name='swagger-ui'),

]

With drf-spectacular, you’ll now have automatically generated documentation available at /api/docs/. The schema is accessible at /api/schema/, which can also support OpenAPI-compatible clients and tools.

This setup provides structured routing for the new models and easy-to-navigate API documentation.

In Django REST framework (DRF), the @action decorator, imported from rest\_framework.decorators, is used to add custom actions to viewsets. These custom actions can be used to create additional endpoints for specific actions that do not fit into the standard CRUD operations (list, retrieve, create, update, and destroy). This is especially useful for operations that need custom functionality within a viewset.

**Usage of @action**

The @action decorator can be applied to any method within a ViewSet class. Here are the main features:

* **Route Creation**: The decorator creates a new route for the specified method, making it accessible through a new URL.
* **Methods**: You can specify HTTP methods for the custom action, such as GET, POST, PUT, DELETE, etc., using the methods argument.
* **Detail Argument**: The detail argument indicates whether the action applies to a single instance (detail=True) or a collection (detail=False).
* **URL Suffix**: By default, the name of the method becomes the suffix of the URL endpoint. You can change it by providing a different name.

**Example**

Suppose you have a BookViewSet and want to add a custom action that marks a specific book as "favorite." Here’s how it can be done using @action:

from rest\_framework import viewsets

from rest\_framework.response import Response

from rest\_framework.decorators import action

from .models import Book

from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):

queryset = Book.objects.all()

serializer\_class = BookSerializer

# Custom action to mark a book as favorite

@action(detail=True, methods=['post'])

def favorite(self, request, pk=None):

# Get the book instance

book = self.get\_object()

# Custom logic (e.g., mark the book as favorite)

book.is\_favorite = True

book.save()

# Return a response

return Response({'status': 'book marked as favorite'})

# Another custom action that returns the list of favorite books

@action(detail=False, methods=['get'])

def favorites(self, request):

favorite\_books = Book.objects.filter(is\_favorite=True)

serializer = self.get\_serializer(favorite\_books, many=True)

return Response(serializer.data)

**Explanation of the Example**

1. **@action(detail=True, methods=['post'])**:
   * This creates a new route for marking a specific book as favorite.
   * detail=True means this action operates on a single instance (requires a primary key, pk).
   * The endpoint can be accessed with a URL like /books/<pk>/favorite/ via a POST request.
2. **@action(detail=False, methods=['get'])**:
   * This creates a route for listing all favorite books.
   * detail=False indicates that this action applies to the collection of books rather than a single instance.
   * The endpoint can be accessed with a URL like /books/favorites/ via a GET request.

**URL Routing with @action**

When using @action, DRF automatically appends the name of the method to the base URL of the viewset, so there’s no need to define it separately in urls.py. The URLs in the example above would automatically be generated as:

* **For marking a book as favorite**: /books/<pk>/favorite/
* **For listing favorite books**: /books/favorites/

**Summary of Key Points**

* @action is used to define custom routes within viewsets.
* detail=True or detail=False controls whether the route operates on a single item or a collection.
* You can specify HTTP methods (methods=['get', 'post']) to restrict the action to specific request types.
* The method name becomes part of the URL endpoint by default.