

You have 2 free stories left this month. [Sign up and get an extra one for free.](#)

Learn the Django REST Framework in Minutes

Let's build our first API



Agustin Castro

Jan 24 · 4 min read ★



Writing REST APIs is really commonplace for developers these days. When faced with the choice of technology stack, I tend to think of Django as one of my first options.

“Why?” you should ask. The reasons are fairly simple:

- It's developed in Python (which I love).
- It is a battle-tested framework that lets you quickly deploy to production and test your API.
- The Django Rest Framework (DRF from now on) is really flexible and simple to understand.

So, with that said, let's write a REST API that manages food in restaurants. By the end of the project you should have a working API with the following endpoints to play with:

Endpoint	HTTP Verb	Description
/restaurants	GET	Get all restaurants
/restaurants	POST	Create a restaurant
/restaurants/<str:restaurant_id>/	GET	Get a restaurant
/restaurants/<str:restaurant_id>/	DELETE	Delete a restaurant
/restaurants/<str:restaurant_id>/recipes/	GET	Get all recipes in a restaurant
/restaurants/<str:restaurant_id>/recipes/	POST	Create a recipe in a restaurant
/restaurants/<str:restaurant_id>/recipes/<str:recipe_id>	GET	Get a recipe in a restaurant
/restaurants/<str:restaurant_id>/recipes/<str:recipe_id>	DELETE	Delete a recipe in a restaurant

• • •

Project Setup

1. Have Python installed (I'm using Python 3.6).
2. Initialize a virtual environment `python3 -m venv env` and activate it: `source env/bin/activate`.
3. Install Django, DRF, and psycopg2 (SQL adapter for our DB): `pip install django djangorestframework psycopg2`.
4. Create the Django project: `django-admin startproject restaurants`.
5. Create a Postgres database and replace the configurations in the `settings.py` file of your project.
6. Apply migrations: `python3 manage.py migrate`.
7. Create superuser: `python3 manage.py createsuperuser`.
8. Start the local server: `python3 manage.py runserver`.

Great! Now if you go to `http://127.0.0.1:8000/` in your browser you should see a functional starter Django site.

• • •

Django Rest Framework Setup

To add DRF to the project, go to `settings.py` and add `rest_framework` to your installed apps. Let's also create an app for the API: `python3 manage.py startapp api` and add it to your installed apps.

```
1  INSTALLED_APPS = [  
2      'django.contrib.admin',  
3      'django.contrib.auth',  
4      'django.contrib.contenttypes',  
5      'django.contrib.sessions',  
6      'django.contrib.messages',  
7      'django.contrib.staticfiles',  
8      'rest_framework',  
9      'api'  
10 ]
```

`settings.py` hosted with ❤ by GitHub

[view raw](#)

Now add the API project URLs to your main `url.py` file in the restaurants project:

```
1  from django.conf.urls import url  
2  from django.contrib import admin  
3  from django.urls import path, include  
4  
5  urlpatterns = [  
6      url(r'^admin/', admin.site.urls),  
7      path('', include('api.urls'))  
8  ]
```

`mainurls.py` hosted with ❤ by GitHub

[view raw](#)

. . .

Create the Models

We'll model our restaurants very simply. Each restaurant has many recipes and each recipe has many ingredients. Ingredients could belong to more than one recipe.



```
direction: varchar(120)
phone: Int
```

```
type: varchar(20)
thumbnail: varchar(100)
```

As the attribute `thumbnail` in the `Recipe` model is an image, we need to install Pillow with `pip install pillow`.

This is the code in `models.py`:

```
1  from django.db import models
2  import uuid
3
4
5  class Restaurant(models.Model):
6      id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
7      name = models.CharField(max_length=120, unique=True, verbose_name="Name")
8      direction = models.CharField(max_length=120, verbose_name="Direction")
9      phone = models.IntegerField()
10
11     def __str__(self):
12         return self.name
13
14
15     class Recipe(models.Model):
16         id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
17         restaurant = models.ForeignKey(Restaurant, on_delete=models.CASCADE)
18         name = models.CharField(max_length=120, unique=True, verbose_name="Name")
19         type = models.CharField(max_length=20,
20                                 choices=[('BREAKFAST', 'Breakfast'), ('LUNCH', 'Lunch'), ('COFFEE',
21                                             ('DINNER', 'Dinner'))])
22         thumbnail = models.ImageField(upload_to="recipe_thumbnails", default="recipe_thumbnails/default.png")
23
24     def __str__(self):
25         return self.name
26
27
28     class Ingredient(models.Model):
29         id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
30         recipe = models.ManyToManyField(Recipe)
31         name = models.CharField(max_length=120, unique=True, verbose_name="Name")
32
33     def __str__(self):
34         return self.name
```

To create our DB schema we need to make the proper migrations:

`python3 manage.py makemigrations` and `python3 manage.py migrate ...` and now your models should be tables in your DB.

. . .

Create the Serializers

In the Django REST framework, serializers transform complex data such as querysets or model instances in JSON or XML into Python datatypes and vice-versa.

Serializers also provide extra functionality which allows you to encapsulate logic for CRUD operations when operating with resources.

We'll define the serializers for our models in a separate `serializers.py` file in our API project as follows:

```
1  from rest_framework import serializers
2  from . import models
3  import base64
4  from django.conf import settings
5  import os
6
7
8  class RestaurantSerializer(serializers.ModelSerializer):
9      # Serializer for the Restaurant model, in fields we specify the model attributes we want to
10     # deserialize and serialize
11     class Meta:
12         model = models.Restaurant
13         fields = ['id', 'name', 'direction', 'phone']
14
15
16     class IngredientSerializer(serializers.ModelSerializer):
17         class Meta:
18             model = models.Ingredient
19             fields = ['id', 'name']
20
21
22     class RecipeSerializer(serializers.ModelSerializer):
23
24         # As each recipe has an image thumbnail we deal with the serialization of the image in the
25         # 'encode_thumbnail' where the image is read from the media folder and encoded into base64
```

```

26     thumbnail = serializers.SerializerMethodField('encode_thumbnail')
27     # When getting a recipe I want an 'ingredients' field, the value of this field is the return
28     # function that serializes the ingredients for the recipe.
29     ingredients = serializers.SerializerMethodField('get_ingredients')
30
31     def encode_thumbnail(self, recipe):
32         with open(os.path.join(settings.MEDIA_ROOT, recipe.thumbnail.name), "rb") as image_file:
33             return base64.b64encode(image_file.read())
34
35     def get_ingredients(self, recipe):
36         try:
37             recipe_ingredients = models.Ingredient.objects.filter(recipe__id=recipe.id)
38             return IngredientSerializer(recipe_ingredients, many=True).data
39         except models.Ingredient.DoesNotExist:
40             return None
41
42     def create(self, validated_data):
43         """
44         Create function for recipes, a restaurant and a list of ingredients is associated. The restaurant
45         is taken from the corresponding path parameter and the ingredients can be added optionally.
46         """
47         ingredients_data = validated_data.pop("ingredients")
48
49         restaurant = models.Restaurant.objects.get(pk=validated_data["restaurant_id"])
50         validated_data["restaurant"] = restaurant
51         recipe = models.Recipe.objects.create(**validated_data)
52
53         # Assign ingredients if they are present in the body
54         if ingredients_data:
55             for ingredient_dict in ingredients_data:
56                 ingredient = models.Ingredient(name=ingredient_dict["name"])
57                 ingredient.save()
58                 ingredient.recipe.add(recipe)
59         return recipe
60
61     class Meta:
62         model = models.Recipe
63         fields = ['id', 'name', 'type', 'thumbnail', 'ingredients']

```

As you can see, we are defining a `serializer` class for each model. A serializer behaves somewhat like a form, it validates data, it controls the output of your response, and provides functions to create and update models.

Now our requests will be handled by our views which will use these serializers to validate and translate data to and from JSON.

. . .

Create the Views

Django provides different ways to create views: class-based views and function-based views.

In the Django REST framework, you can create API views by decorating function-based views with `@api_view` or subclassing `APIView` if you prefer class-based views.

In this tutorial, I will use class-based views but switching to function-based views is a trivial task. The Django REST framework also implements common actions like CRUD operations for class-based views via mixin classes.

The Django REST framework even provides a set of already mixed-in class-based views called *generic class-based views* like `ListCreateAPIView` or

`RetrieveUpdateDestroyAPIView`.

We'll define our `APIViews` in the `views.py` file:

```
1  from rest_framework.response import Response
2  from rest_framework.views import APIView
3  from . import serializers
4  from .models import Restaurant, Recipe, Ingredient
5  from django.http import Http404
6  from rest_framework import status
7
8
9  class Restaurants(APIView):
10
11     def get(self, request):
12         restaurants = Restaurant.objects.all()
13         serializer = serializers.RestaurantSerializer(restaurants, many=True)
14         return Response(serializer.data)
15
16     def post(self, request):
17         serializer = serializers.RestaurantSerializer(data=request.data)
18         if serializer.is_valid():
19             serializer.save()
```

```

20         return Response(serializer.data, status=status.HTTP_201_CREATED)
21     return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
22
23
24     class RestaurantDetail(APIView):
25
26         def get(self, request, restaurant_id):
27             try:
28                 restaurant = Restaurant.objects.get(pk=restaurant_id)
29             except Restaurant.DoesNotExist:
30                 raise Http404
31             serializer = serializers.RestaurantSerializer(restaurant)
32             return Response(serializer.data)
33
34         def delete(self, request, restaurant_id):
35             try:
36                 restaurant = Restaurant.objects.get(pk=restaurant_id)
37             except Restaurant.DoesNotExist:
38                 raise Http404
39             restaurant.delete()
40             return Response(status=status.HTTP_204_NO_CONTENT)
41
42
43     class Recipes(APIView):
44
45         def get(self, request, restaurant_id):
46             recipes = Recipe.objects.filter(restaurant__id=restaurant_id)
47             serializer = serializers.RecipeSerializer(recipes, many=True)
48             return Response(serializer.data)
49
50         def post(self, request, restaurant_id):
51             try:
52                 Restaurant.objects.get(pk=restaurant_id)
53             except Restaurant.DoesNotExist:
54                 raise Http404
55
56             serializer = serializers.RecipeSerializer(data=request.data)
57             if serializer.is_valid():
58                 serializer.save(restaurant_id=restaurant_id, ingredients=request.data.get("ingredients"))
59                 return Response(serializer.data, status=status.HTTP_201_CREATED)
60             return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
61
62
63     class RecipeDetail(APIView):
64
65         def get(self, request, restaurant_id, recipe_id):
66             try:
67                 recipe = Recipe.objects.get(restaurant_id=restaurant_id, pk=recipe_id)

```



```

67         recipe = Recipe.objects.get(restaurant__id=restaurant_id, pk=recipe_id)
68     except Recipe.DoesNotExist:
69         raise Http404
70     serializer = serializers.RecipeSerializer(recipe)
71     return Response(serializer.data)
72
73     def delete(self, request, restaurant_id, recipe_id):
74         try:
75             recipe = Recipe.objects.get(restaurant__id=restaurant_id, pk=recipe_id)
76         except Recipe.DoesNotExist:
77             raise Http404
78         recipe.delete()
79         return Response(status=status.HTTP_204_NO_CONTENT)

```

Note how our views use the serializers we defined previously to validate and serialize data.

Now, the last piece of the puzzle is to define the URLs and point them to our views.

. . .

Define the URLs

When a Django server receives a request it matches the request URL with those described in `urls.py`, the first match found is handled by the corresponding view defined in the path.

Let's write the URLs for the restaurant API in the `urls.py` file:

```

1  from django.urls import path
2  from . import views
3
4  urlpatterns = [
5      path('restaurants/', views.Restaurants.as_view()),
6      path('restaurants/<str:restaurant_id>', views.RestaurantDetail.as_view()),
7      path('restaurants/<str:restaurant_id>/recipes/', views.Recipes.as_view()),
8      path('restaurants/<str:restaurant_id>/recipes/<str:recipe_id>', views.RecipeDetail.as_view()),
9  ]

```

urls.py hosted with ❤ by GitHub

[view raw](#)

Now, this should be the last piece of our API.

• • •

Conclusion

By now, if we launch the server and head to `http://127.0.0.1:8000/restaurants/` we should see the Django REST framework test site where you can play with the APIs you just created.

This API test site is one of the really cool things about the Django REST framework, you can test your APIs really easily from your browser.

There are lots of other goodies that DRF provides like different authorization and authentication models to protect your APIs.

The working code for this project can be found on GitHub (link below), feel free to make it yours.

agustincastro/DjangoRestAPI-Restaurant

API made with Django Rest API to manage recipes in restaurants -
agustincastro/DjangoRestAPI-Restaurant

[github.com](https://github.com/agustincastro/DjangoRestAPI-Restaurant)

[Software Development](#)

[Software Engineering](#)

[Python](#)

[Django](#)

[Programming](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app

