

Exposing the Backend Functionality through a REST API

Now that we've implemented the persistence layer with the models that best resemble the business for our application, it is time to use another cool Django feature that allows us to create a **REST API** for our models and live documentation for using it with little work and effort: the **Django REST Framework** or **DRF**.

The DRF, which is one Django extension as it does not come preinstalled by default, allows us to create fully operational REST endpoints for our application by typing only a minimal amount of code and thus following the general philosophy that less code means more, as it makes the code easier to maintain.

CoreAPI is an extension of DRF for automatically documenting your API in a visual and interactive way, allowing you to manage instances manually, showing in a single place all the generated REST endpoints available in our API.

You can find additional documentation about the DRF for your reference at the following URLs:

<https://www.django-rest-framework.org/>.

<https://www.coreapi.org/>

In this chapter, we're going to cover the following topics:

- Generating a REST API with Django Rest Framework
- Generating the API documentation with CoreAPI

As you will find later in **Chapter 8** and in the git repository for this project, the REST API layer has been extensively tested beforehand. As a good practice, I have not waited until then to write the tests, but instead used a test-driven development approach, as it is the

only way to demonstrate that the API behaves as expected and that the code produced is testable. And the last chapters of this book will just concentrate on explaining the importance of this subject in more detail.

Generating a REST API with Django Rest Framework

It is time to enable consumption of our application from an external system with one of the most popular ways of communication between distributed applications a.k.a. REST

A possible and typical consumer for an application can be a frontend like the one we will create later in [Chapter 5](#) or another application that uses a client to communicate with the API.

To this end, we will use the DRF. The DRF is a Django extension that allows developers to generate a REST API with minimal effort. The DRF will scaffold, that is, automatically generate the rest endpoints for our API for our models in the persistence layer with minimal code.

Installing DRF

To add the DRF to our application, type in the following command:

```
$ python -m pip install djangorestframework
```

This is almost all you need to do get all the required libraries for the DRF to work its magic.

Only one thing is missing: like we did in [Chapter 2](#), we need to add this to the **INSTALLED APPS** block in the `settings.py` file of our application.

Add a new line containing `'rest-framework'`, as shown in [Figure 3.1](#):

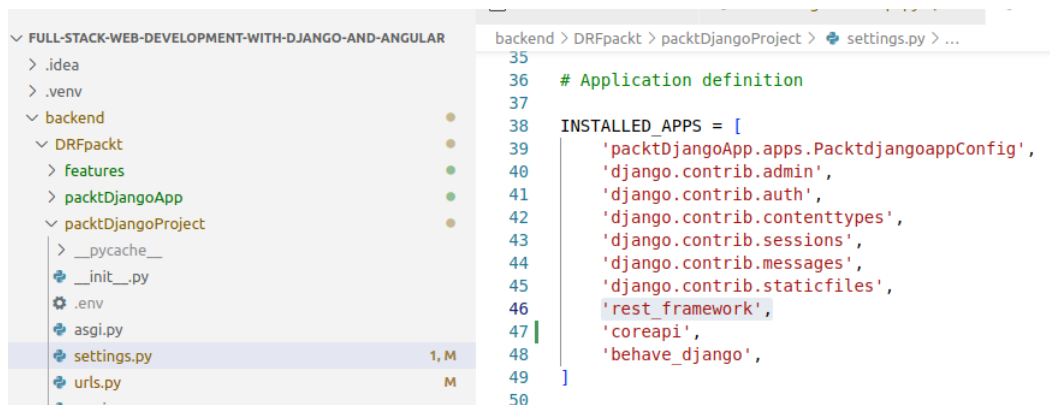


Figure 3.1 – Adding the DRF application to the Django project

Scaffolding the REST endpoint of a model

Now there are several steps you should follow to generate the API for each of the models in the persistence layer we created in the previous chapter:

- Create a **router** and wire up our API using automatic URL routing in the `urls.py` file for the application (only once).
- Add a suffix for the REST API endpoints in the `urls.py` file for the project (only once).
- Create a **Serializer** for each model in a new `serializers.py` file for the application (once for each model).
- Create a **ViewSet** that defines the view behavior in the `views.py` file for the application (once for each model).
- Registering the **ViewSet** in the router in the `urls.py` file for the application (once for each model).

The last three steps can be considered as implementation steps that are repeated once for each of the models, while the first two steps can be considered as configuration steps that need to be created just once independent of the number of models.

In the following image you can see the location of the files you have to change in the first two steps:

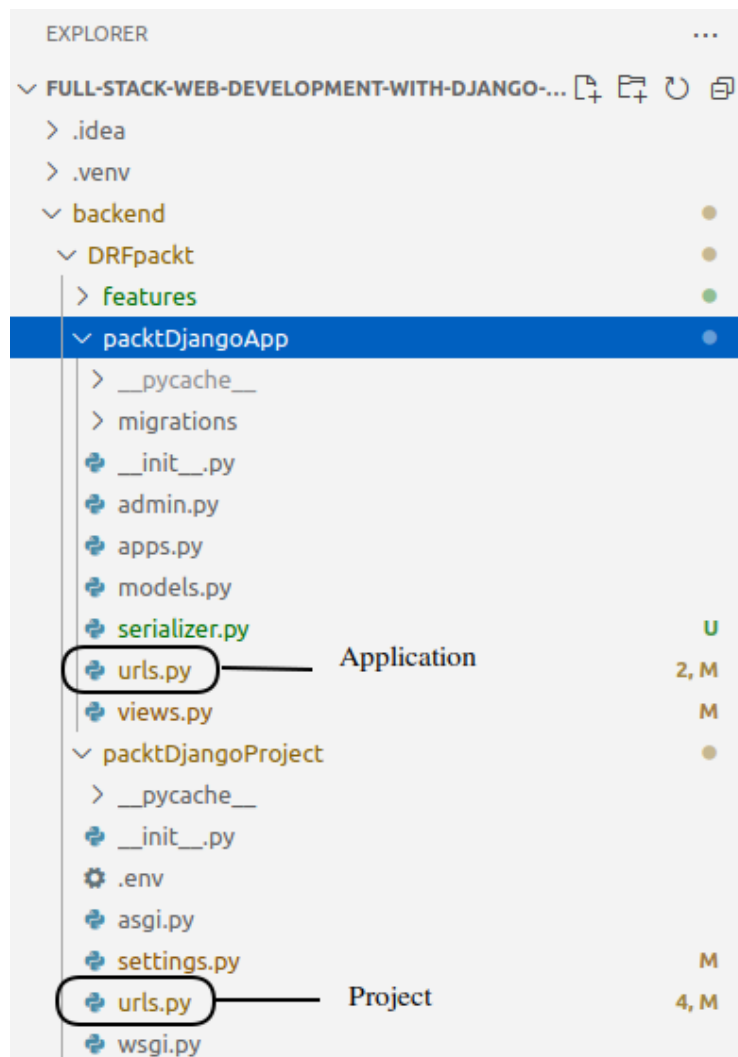


Figure 3.2 – Location of urls.py files

Let's now implement each of the steps listed above, starting with the first two configuration steps.

Creating a router and wiring up our API using automatic URL routing

A **router** is a Django component responsible for the configuration of the URLs or routes for the new API.

Add the following lines to the `urls.py` file in the application:

```
from rest_framework import routers

router = routers.DefaultRouter()

urlpatterns = [
    path('', include(router.urls))
]
```

Adding a suffix for the REST API endpoints

The following code is responsible for adding a suffix to the URLs of our new API.

The suffix is needed to define a starting point for the URL of our DRF API and to differentiate it from other endpoints that are not generated by the DRF. This means that if we select `api/v0/` as a suffix for our DRF API, it will be accessible from the following URL: `http://localhost:8000/api/v0/`.

It can also be used for API versioning when the models change, but we still want to keep backwards compatibility with the clients of our API.

Add the following lines to the `urls.py` file in the project:

```
from rest_framework.documentation import include_docs_urls

urlpatterns = [
    path('api/v0/', include("packtDjangoApp.urls"))
]
```

Creating a Serializer for the model

Serializers are responsible for converting the information from the database contained in a model to **JSON** format (the most widely used in the payload for communication in REST APIs). This conversion is bidirectional: from model to JSON format and vice versa.

Add a new file called `serializers.py` in the `man` folder of the application as shown in the next image:

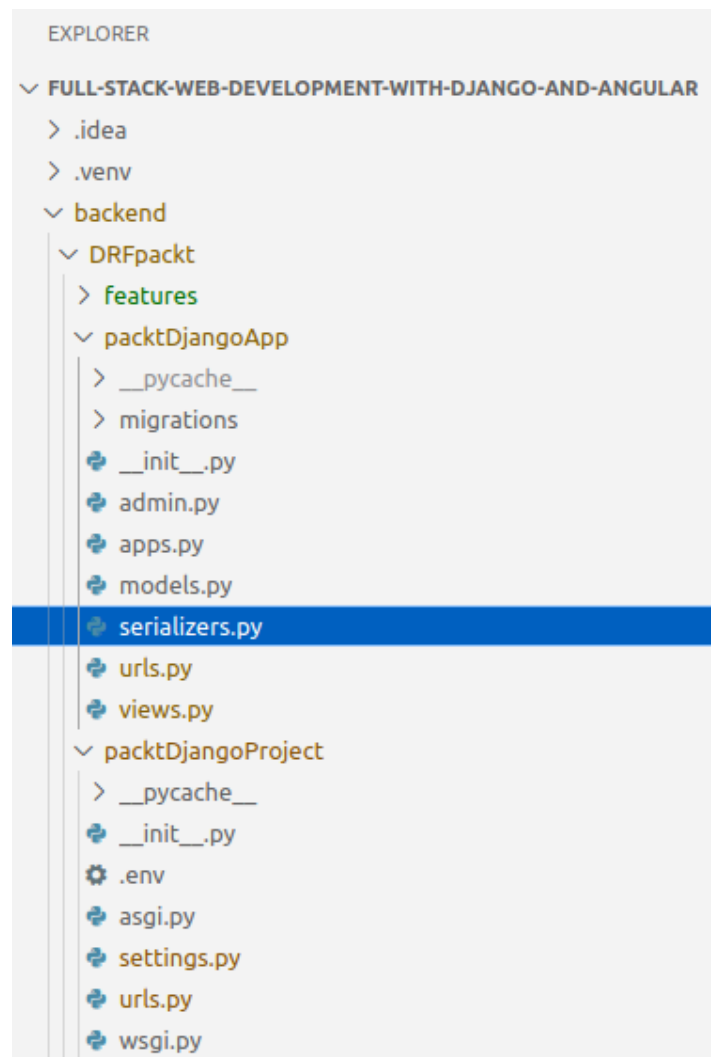


Figure 3.3 – Location of serializers.py file

For every model exposed through a REST API, we create one serializer.

Next, we show an example of a serializer for the Product model we created in **Chapter 2**:

```
from .models import Product
from rest_framework import serializers
```

```
# Serializers define the API representation.
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        # fields = ['name', 'price', 'currency']
        fields = '__all__'
```

Let's break down the code for this example of serializer:

- `class ProductSerializer(serializers.ModelSerializer):`: declares the class `ProductSerializer` as a serializer.
- `model = Product`: Declares the Product model as the source and target for the serialization.
- `fields = ['name', 'price', 'currency']`: Declares the fields to include in the serialization; other fields will be excluded.
- `fields = '__all__'`: Optionally If you want to include all fields in a model you are not forced to write them all.

Keep in mind that a new serializer is needed for every model in the persistence layer. You can find the code in the git repository for the backend in the following URL:

<https://github.com/PacktPublishing/Full-Stack-Web-Development-with-Django-and-Angular/blob/main/backend/DRFpackt/packtDjangoApp/serializers.py>

Creating a ViewSet that defines the view behavior

Rather than write multiple views, we'll group together all the common behavior into a single class called `ViewSet`.

We can easily break these down into individual views if we need to, but using Viewsets keeps the view logic nicely organized as well as being very concise.

Add the new ViewSets to the `views.py` file located in the main folder of the application as shown in the next image:

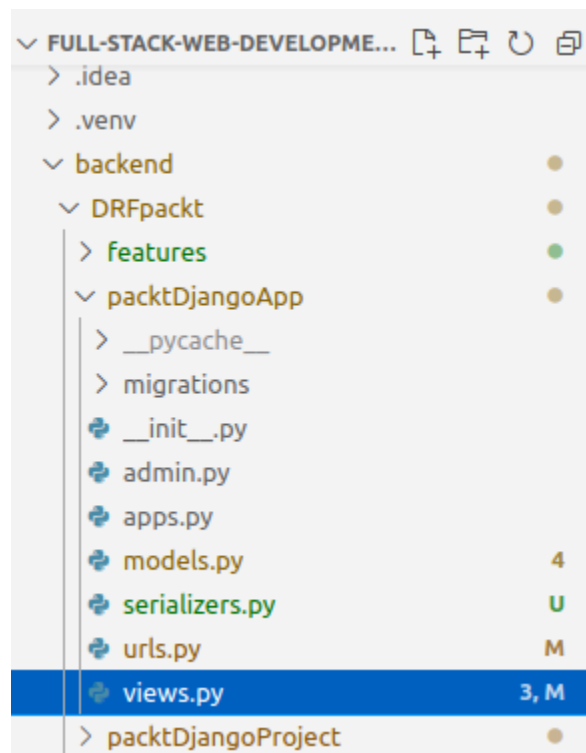


Figure 3.4 – Location of views.py file

Here is an example of the `ViewSet` associated to the Product model.

```
from rest_framework import viewsets
from .models import Product
from .serializers import ProductSerializer

# ViewSets define the view behavior.
class ProductViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows products to be viewed or
    edited.
    """
```



```
queryset = Product.objects.all().order_by('name')
serializer_class = ProductSerializer
```

Let's break down the code for this example of `ViewSet`:

- `class ProductViewSet(viewsets.ModelViewSet):`: Declares the new `ViewSet` for a model
- `queryset = Product.objects.all().order_by('name')`: Defines the behavior for the view, in this case ordered by one model field.
- `serializer_class = ProductSerializer`: Binds the serializer we created in the previous step to the new `ViewSet`.

And so far, we are done with the first REST endpoint for the Product model. Django under the hood will apply its magic and automatically generate the endpoints from every `ViewSet` for us without the need to type anything else, in a similar way as the models that we created in [Chapter 2](#) did to automatically generate the persistence layer. Django and DRF, almost codeless can generate the persistence layer as well as the REST API respectively with very little configuration code, demonstrating how Django shines for rapid application development.

Now if we start the application (see [Chapter 2](#)) and point to the API URL in our preferred browser:

```
http://localhost:8000/api/v0/products
```

Then in the browser you will see that Django will provide a form for listing all the products as well as manually creating new product records in the database, as shown in the next picture:

localhost:8000/api/v0/products/

Django REST framework admin

Api Root / Product List

Product List

OPTIONS GET

API endpoint that allows products to be viewed or edited.

GET /api/v0/products/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[
  {
    "id": 1,
    "name": "Product1",
    "price": "44.00",
    "currency": "€"
  }
]
```

Raw data HTML form

ProductName

Price

Currency

Eur

POST

Figure 3.5 – REST endpoint for products

Note

Notice the URL prefix used for the endpoint `/api/v0/` is the one we configured before in one of the steps.

If you want to perform operations on one **Product** instance, for example, the one with the id 1, just point your browser to the following URL:

<http://localhost:8000/api/v0/products/1/>

Then you will see the form for manually updating or deleting the selected product instance in the database as shown in the next picture:

Django REST framework admin

Api Root / Product List / Product Instance

Product Instance

DELETE OPTIONS GET

API endpoint that allows products to be viewed or edited.

GET /api/v0/products/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{
  "id": 1,
  "name": "Product1",
  "price": "44.00",
  "currency": "€"
}
```

Raw data HTML form

ProductName
Product1

Price
44.00

Currency
Eur

PUT

Figure 3.6 – REST endpoint for one product instance

Optionally instead of using the Forms provided by Django, you could also use the endpoints with the `curl` or `Postman` tools to manage `Product` instance records in the database.

Keep in mind that a new `ViewSet` is needed for every model in the persistence layer. You can find the code in the git repository for the backend in the following URL:

<https://github.com/PacktPublishing/Full-Stack-Web-Development-with-Django-and-Angular/blob/main/backend/DRFpackt/packtDjangoApp/views.py>

Registering the Viewset in the router

Add the following line in `urls.py` file in the application:

```
router.register(r'product', views.ProductViewSet)
```

This configuration will add the `/product` suffix to the API endpoint for products. This is necessary to expose the new endpoint for the product model; otherwise, the API will not be accessible.

We have finally generated a REST API for one of our models. Keep in mind that the last three steps (creating the Serializer, creating the ViewSet, and registering the ViewSet) have to be carried out again once for each model in our application.

You can find the code in the git repository for the backend in the following URL:

<https://github.com/PacktPublishing/Full-Stack-Web-Development-with-Django-and-Angular/blob/main/backend/DRFpackt/packtDjangoApp/urls.py>

In the next section, we will see how to add pagination; this will enable us to view large result sets in a more manageable manner.

Adding pagination

Whenever you want to return a big list containing records from any search, you must do it by returning those records in small groups instead of all at once. Returning all records at once may be not manageable for a huge number of records. When you set up pagination, you can define the page size (maximum number of records per page) this way you can navigate between pages with your REST API.

Adding pagination to your REST API is as simple as adding two lines in the `settings.py` file as you can see in the **Figure 3.7**:

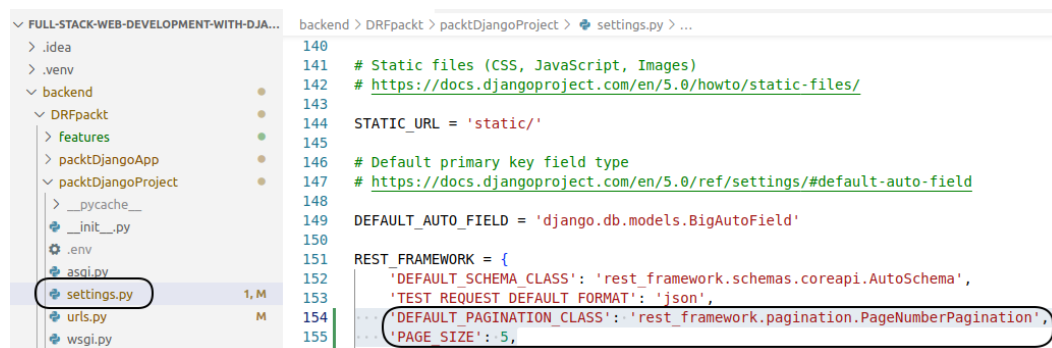


Figure 3.7 – Adding pagination

As you can see in the first line, we have set up the pagination type:

```
'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination'
```

If now you call the endpoint for listing all records you must add the 'page' number as parameter to the URL. For example, to navigate to page two in the listing of all products in your model you must use the following URL:

```
http://localhost:8000/api/v0/products/?page=2
```

In the second line, we have set up the page size to a small number of records. Five in this case. But you can customize it at your own will:

```
'PAGE_SIZE': 5
```

And in response to calling the URL mentioned before, you will get a list of products containing a maximum of five records, as configured in the page size.

In the next section, we are going to explore how to test and validate that the REST API is working as expected.

Testing whether your API is working

There are two main ways to demonstrate how the API works: one is manual testing, and the other is automated testing:

Manual testing is carried out as it's name says, to verify that the application behaves as expected by reproducing every step of the functionality by hand on the running application in the same way as a user would do.

Automated testing is carried out as it's name says, to verify that the application behaves as expected by testing the source code with a testing framework and running the test suite automatically in a CI/CD Pipeline.

We will use the test-driven development approach and write automated tests for the requests to the API endpoints.

Note

Test-driven development is a best practice commonly used in software engineering where the automated tests are written before the production code ensuring that the source code behaves as expected and is testable so that the right solution is implemented.

The automated approach has several advantages over the manual approach:

- You only need to write it once to verify that your API behaves as expected at any time when you run the test suite.
- It serves as living documentation for your API.
- You have covered your code with tests, so you know the source code for this application is testable and well designed. Also, you have completed all the effort needed to test your API in a CI/CD pipeline.
- In case you need to change the API or refactor it in the future, you can do it without risks.

Here is the link to the git repository that contains the tests:

<https://github.com/PacktPublishing/Full-Stack-Web-Development-with-Django-and-Angular/tree/main/backend/DRFpackt/tests>

We will be covering this topic in more detail in **Chapter 8**, where we will learn how to test all layers of the backend application.

In the next Topic we explain why just manually testing our API with tools like **Curl** or **Postman** is not a good practice.

Using the curl or postman tools to perform manual requests against the API endpoints.

I don't think it is necessary to say that the manual approach has no advantages over the automated approach. It is quite obvious. In addition, Apart from that, it has the disadvantage of being very time-consuming. Manually testing your application every time takes a huge amount of precious time that you could use for development or improving the quality. For these reasons, I am not even showing how to do it in this book.

Manual testing is a complement to, not a substitute for, automated testing. It can serve as additional verification when the automated tests are insufficient to catch bugs in a few parts of the source code, and should generally be performed only once or after significant changes have been made as a proof of concept. Developers, architects, and technical leads who believe they can save time and meet deadlines by skipping automated tests are mistaken. In reality, avoiding automated tests will definitely lead to missed deadlines and an unmaintainable, chaotic application, ultimately jeopardizing the project.

We have obviously followed the automated approach in this book, and you can find the automated tests for the API in the **tests** folder of the git repository. So, there is no excuse to write the tests for your project from the very first day because you have the best examples already written for you in this book. That is the main benefit you get when buying this book, that the return of investment is absolute and immediate, because you don't have to waste time and effort to figure out how to test your application.

In the next section we are going to explore a cool feature for generating a living documentation of our API with the **coreapi** extension.

Generating the API documentation with CoreAPI

CoreAPI is an extension of DRF for automatically documenting your API in a visual and interactive way, allowing you to manage instances manually, showing in a single place all the generated REST endpoints available in our API.

Adding the CoreAPI application to our Django project is simple, and we only need to execute the following three steps:

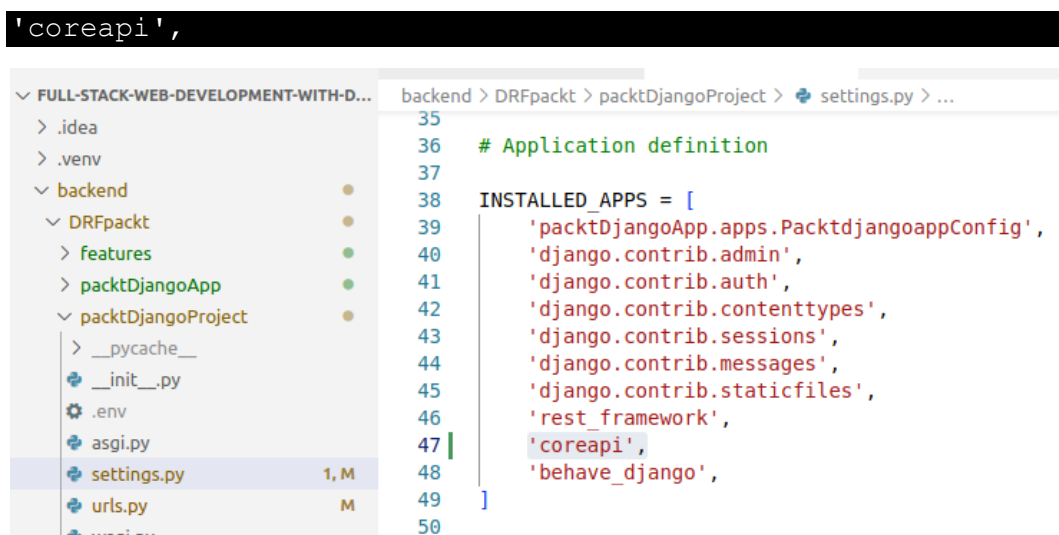
1. Install the `coreapi` library

```
$ python -m pip install coreapi
```

2. Add the `coreapi` application to the project

Add the line highlighted in grey in the **INSTALLED_APPS** module inside the `settings.py` file of the project, as shown in **Figure 3.7**:

```
'coreapi',
```



```
35
36 # Application definition
37
38 INSTALLED_APPS = [
39     'packtDjangoApp.apps.PacktdjangoappConfig',
40     'django.contrib.admin',
41     'django.contrib.auth',
42     'django.contrib.contenttypes',
43     'django.contrib.sessions',
44     'django.contrib.messages',
45     'django.contrib.staticfiles',
46     'rest_framework',
47     'coreapi',
48     'behave_django',
49 ]
50
```

Figure 3.8 – Adding the CoreAPI application to the Django Project

3. Add the coreapi URL

Add the line highlighted in grey in the **urlpatterns** module inside the `urls.py` file of the project as shown in **Figure 3.8**:

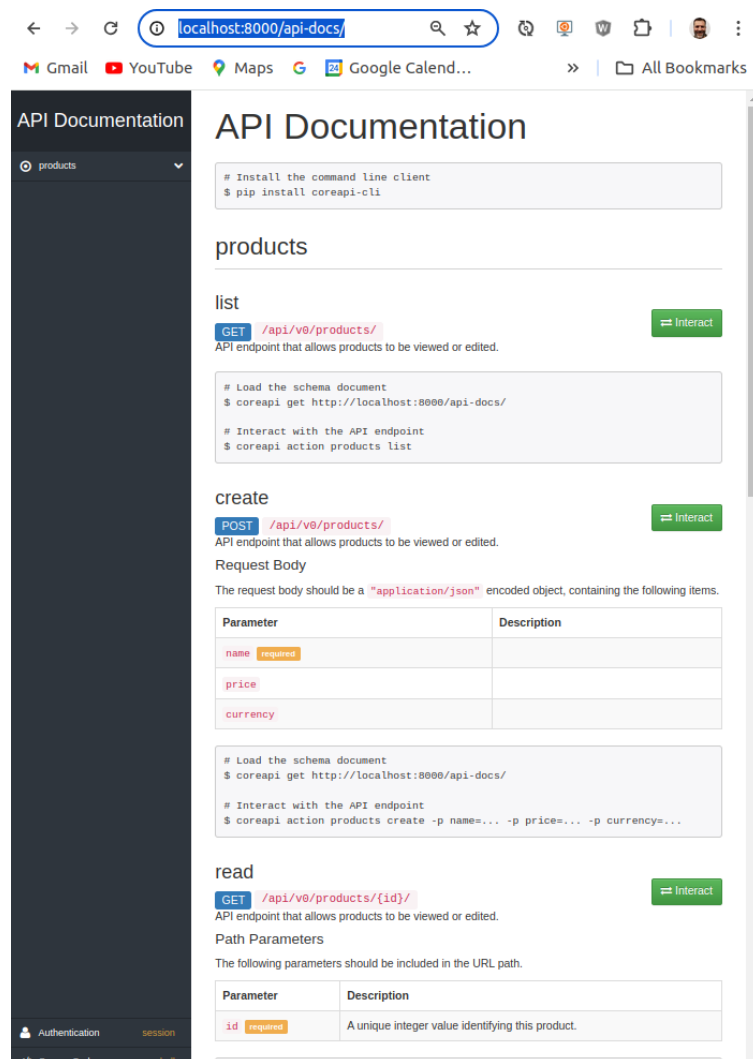


Figure 3.10 – CoreAPI documentation

Summary

In this chapter, we learned how to add a REST API to the models of our application. How the DRF framework Performs that magic for us with just very little configuration from our side. We also learned about the best approach to validate that the API is working as expected in an efficient way.

We also created online documentation for the API with CoreAPI. The CoreAPI library performs all the magic for us with little configuration from our side. It allows us to interact with it and manage instances manually in the database.

In the next chapter, we will learn how to secure the REST API to ensure only authenticated users have access to it.