4
Securing the Backend with Authentication and Authorization

APIs often serve as gateways to access other critical systems, and therefore any vulnerabilities can result in severe consequences. Therefore, securing backend APIs is crucial for protecting against unauthorized access to sensitive data and preventing the risk of **security breaches**.

In this chapter, we will discover how to secure the backend by restricting access to the REST endpoints we created in the previous chapter. In this case, restricting access means that we will only allow users registered in the backend or authentication server to access and use the exposed REST endpoints after successful authentication.

When choosing an authentication mechanism and implementing additional security measures for any application, we should carefully consider an API's specific requirements.

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

- Implementing security on top of the DRF
- Enhancing the DRF with a suitable security approach

Technical requirements

To proceed with this chapter, we will need to have the source code available in the git repository installed on our computer. As a reminder of what we installed in the previous chapters, that is also needed for this one:

- Python 3
- Django 5
- DRF framework

Here is the link for the Git repository that contains the source code for this chapter:

https://github.com/PacktPublishing/Full-Stack-Web-Development-with-Django-and-Angular.git

Implementing security on top of the DRF

Security for backend applications is a broad and tough field, comprising many different areas like attack prevention, authentication and authorization, encryption, certificates, single sign-on, multi-factor authentication, role-based access control (RBAC), etc.

As we saw in *Chapter 1* the Django framework implements built-in protection against common web application vulnerabilities, and this protection is activated by default.

The topic of adding security to applications would need a book on its own and going deep into each area is beyond the scope of this book.

We are just going to concentrate on the authentication and authorization part of security, the one that is implemented specifically for the DRF, and we are going to implement a simple but common approach for REST APIs.

Concerning authentication, which is the mechanism through which a user's identity is verified, Django comes preinstalled with several implementations:

- `BasicAuthentication`: This method of authentication relies on user credentials stored in a database, using the username and password in every request. It uses the Authorization header with the Basic type which is supported by most browsers. Basic authentication is typically a only appropriate for testing and not for production environments..
  If you use it in production, ensure your API is available only for the HTTPS protocol.
- `SessionAuthentication`: Django's built-in authentication system uses the `SessionAuthentication` class provided by the DRF. This approach is based on the mechanism of storing the user credentials in the application session after successful authentication. It relies on cookies, which most browsers easily support, and is recommended for interacting with APIs that are on the same domain as the front end.

- `TokenAuthentication`: This is a popular stateless authentication approach for securing APIs. A centralized server or database is not required to store the tokens. This approach is based on associating an access token to a user after successful authentication; once the user gets the access token, they can use it to authenticate any request to the API before the token expires. Once the token is expired, the user must refresh it and get a new one to get access.
- `JsonWebTokenAuthentication`: This is one of the implementations of the previous `Tokenauthentication` approach but based on a **JSON Web Token** (**JWT**). This is an extension for the DRF and needs to be installed manually (it doesn't come preinstalled). The advantage of using this type of token is that it is not opaque (can transfer non-sensitive user details and eliminates the need to query the database or authentication server for that information on every request). It does not provide security on its own. JWTs are cryptographically signed and cannot be modified by clients or attackers. JWT is suitable for stateless applications, as it allows the application to authenticate users and authorize access to resources without maintaining a session state on the server. JWTs are only stored on the client side, and not on the server. JWT authentication is the approach that will be used in this book because of its simplicity and popularity and because it is better suited to APIs. We will use the `djangorestframework-simplejwt` implementation.
- `Oauth2Authentication`: **OAuth2** is a popular stateful authentication and authorization protocol used by many APIs. It requires a connection to the authorization server to obtain and verify tokens. OAuth is suitable for delegating user authorization, accessing third-party applications, and session management. OAuth is used for authorization, while JWT is used for authentication. OAuth is often used to allow a user to log in to a third-party application using their account on a different site. To implement OAuth2 authentication in your Django REST API, you can use the `OAuth2Authentication` class provided by the DRF. OAuth2 may be too complex for simple applications with one frontend and one backend where JWT is more appropriate.

So far, we have learned about the different security implementations that are compatible with the DRF and what can be the best choice for our case. It is time now to get our hands dirty and implement the best approach for our application. As mentioned earlier, we are going to use the JWT approach because it is more suitable for securing APIs. We are going to extend the DRF with the `djangorestframework-simplejwt` implementation.

Enhancing the DRF with a suitable security approach

As mentioned in the previous section, the security implementation based on a **JWT** is a perfect choice for securing a REST API. This mechanism has the advantages of being simpler to use than Oauth for applications with one backend and one frontend that only need authentication to restrict access.

This implementation is easy to integrate with the DRF as it does not require any additional change in the database schema, and securing the endpoints requires almost no additional code.

Next, we will see the steps involved in applying the `JsonWebTokenAuthentication` approach to the DRF.

Installing the Simple Jwt DRF extension

Install the extension with the following command:

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

After successful installation, you can verify it with the following command:

```
$ python -m pip list

djangorestframework-simplejwt 5.3.1
```

Adding the Simple Jwt extension to the installed applications

All you need to do in this step is add `rest_framework_simplejwt` to the list of installed applications.

In other words, add `rest_framework_simplejwt` inside the `INSTALLED_APPS` block in the `settings.py` file:

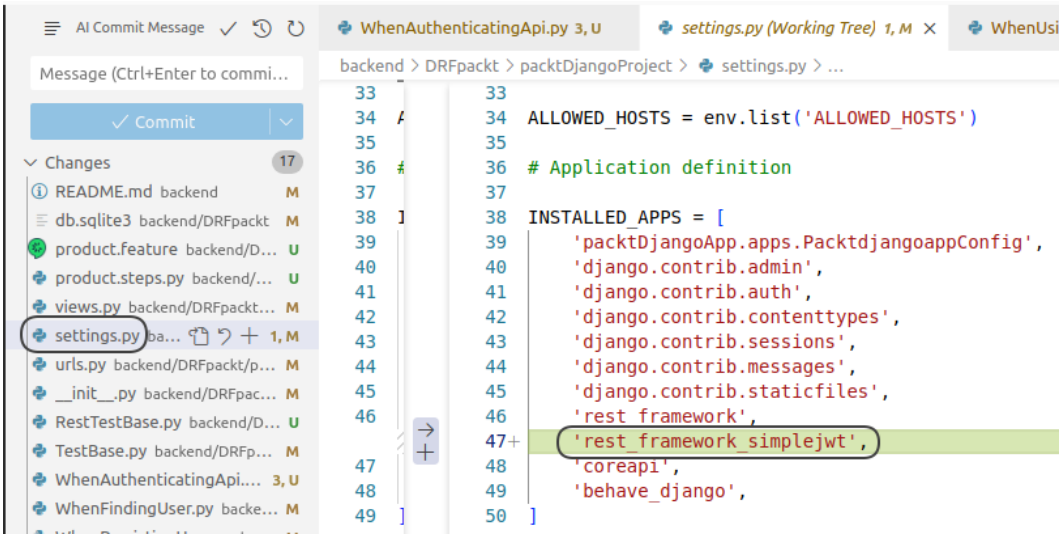

Figure 4.1 – Adding simplejwt application

Adding the Simple Jwt authentication as the default authentication class to the REST framework

All you need to do in this step is add `rest_framework_simplejwt.authentication.JWTAuthentication` to the list of default authentication classes for the REST framework. In other words, add `rest_framework_simplejwt.authentication.JWTAuthentication` inside the `REST_FRAMEWORK` block in the `settings.py` file:

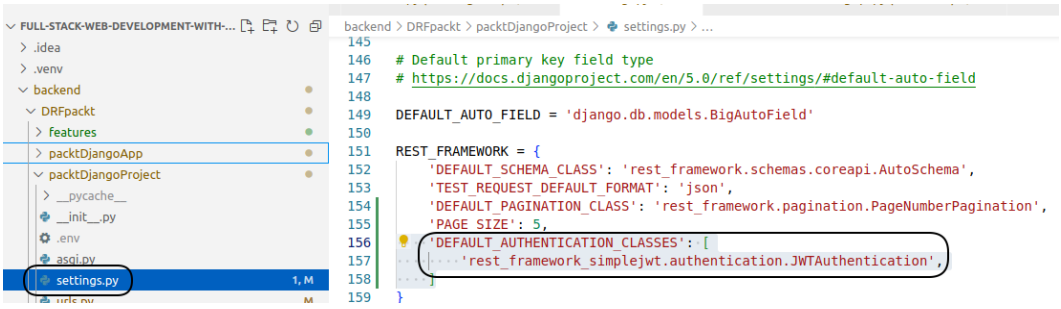

Figure 4.2 – Adding simplejwt authentication

Adding the Simple Jwt REST endpoints to our API

Add the following lines inside the `urls.py` file:

```
from rest_framework_simplejwt.views import (

TokenObtainPairView,

TokenRefreshView,

TokenVerifyView

)


urlpatterns = [

path('api/token/', TokenObtainPairView.as_view(),
name='token_obtain_pair'),

path('api/token/refresh/', TokenRefreshView.as_view(),
name='token_refresh'),

path('api/token/verify/', TokenVerifyView.as_view(),
name='token_verify'),

]
```

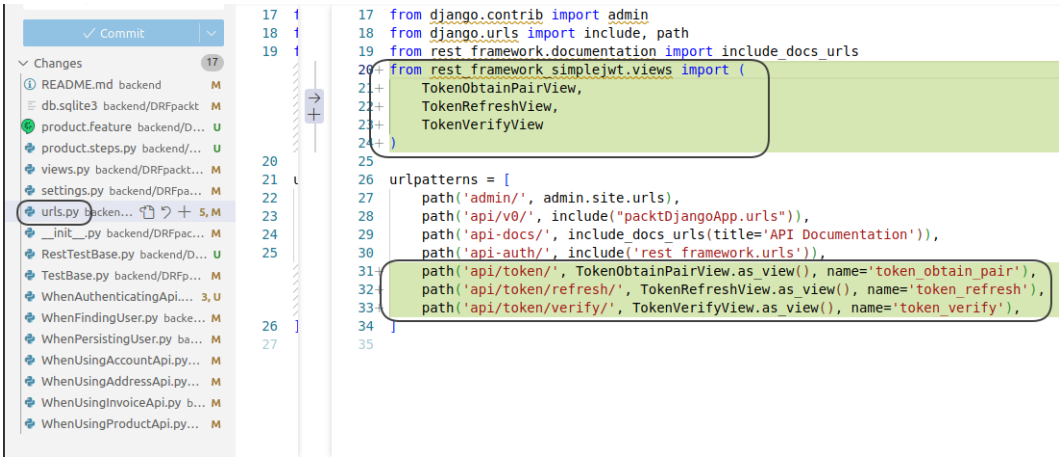

Figure 4.3 – Adding simplejwt JWT REST endpoints

We have now added three new endpoints that are provided by the `Simple Jwt` extension out-of-the-box to our application REST API that offers the ability to manage the JWTs.

The three provided endpoints are as follows:

- `api/token/`: Endpoint to get a short-lived (expires in a brief period) access token, and a long-lived (expires in a prolonged period) refresh token.
- `api/token/refresh/`: Endpoint to get a new short-lived access token after expiration of the previous token.
- `api/token/verify/`: Endpoint to verify that both the access and refresh tokens are valid and have not expired.

The subsections that follow will provide an explanation of each endpoint.

Short-lived access and long-lived refresh token endpoint

This endpoint is available without access restrictions in our application on the URL: `api/token/`

This endpoint is used to get a short-lived access token for any user that is registered and activated in the application database. The access token is needed by a user to authenticate every request to the application API and have access to it without restriction. Without adding this token to the request, access to any secured endpoints is denied. This token expires and is no longer valid a few minutes after creation, so it needs to be recreated or refreshed with a new one to be able to access the secured API again. The timeframe before expiration for both access and refresh tokens is configurable as will be explained later in this section.

The endpoint is also used to get a long-lived refresh token that is used for getting a new short-lived access token after expiration.

You can get a new access and refresh tokens with the following command:

```
$ curl -X POST -H "Content-Type: application/json" -d
'{"username": "[validUserName]", "password":
"[validPassword]"}' http://localhost:8000/api/token/
```

As is evident, you provide the user credentials stored in the database as parameters: `username` and `password`. The endpoint authenticates the user, and if the provided

credentials are valid, the endpoint returns a JSON response containing the two tokens: one for access and one for refresh.

Here is an example of the JSON response after a successful authentication (it contains the values for the access and refresh tokens):

```
{"refresh":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl
90eXBlIjoicmVmcmVzaCIsImV4cCI6MTcxODk3ODI3MSwiaWF0IjoxNzE4O
DkxODcxLCJqdGkiOiI4YjBiOWFhMjAwZjY0ZDdmYmY3OWQ3NWNlNDgyNGZh
OSIsInVzZXJfaWQiOjN9.z2VyUUnYgm2DWz7vTFAyRqxwwY3Jf4KBRDw-
M7v-
emE","access":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2t
lbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzE4ODkyMTcxLCJpYXQiOjE3MT
g4OTE4NzEsImp0aSI6IjU1MmZhMDk1NWUwZjRkMTA5MTNmYmQ2MzVhZjNkN
jYyIiwidXNlcl9pZCI6M30.CLxVaF-
MlHF6C6cK_XDY2xzDOtRJxaNLDv284SKjeQA"}
```

In this scenario, the response in the JSON format contains the values for both the access and the refresh tokens.

Here is an example of the JSON response after a failed authentication:

```
{"detail":"No active account found with the given
credentials"}
```

In this scenario, the response in JSON format contains the error produced by the failed authentication.

It is possible to customize the Simple JWT authentication behavior by adjusting settings such as the timeout expiration period for the access and refresh tokens. Add the following block for Simple JWT customization in the `settings.py` file mentioned in *Figure 4.2* and change it as per your requirements:

```
SIMPLE_JWT = {

"ACCESS_TOKEN_LIFETIME": timedelta(minutes=5),

"REFRESH_TOKEN_LIFETIME": timedelta(days=1),

}
```

In the preceding snippet, we have only changed the expiration period for the access and refresh tokens.

But you can find all the possible settings for Simple JWT in the following URL: https://django-rest-framework-simplejwt.readthedocs.io/en/latest/settings.html

Short-lived access token refresh endpoint

This endpoint is available without access restrictions in our application on the URL: `api/token/refresh/`

The endpoint is used to get a new short-lived access token for the same authenticated user after the current access token has expired.

You can get a new access token with the following command:

```
$ curl -X POST -H "Content-Type: application/json" -d
'{"refresh": "[refreshToken]"}'
http://localhost:8000/api/token/refresh/
```

As is evident, you provide the value of the refresh token obtained from the previous endpoint used as a unique parameter to obtain the first access token plus refresh token.

In a scenario when the refresh token is still valid and has not expired, the endpoint will return the following JSON response. It contains a new access token:

```
{"access":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl9
0eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzE4ODk2NTUwLCJpYXQiOjE3MTg4OT
E4NzEsImp0aSI6IjM5NzliY2ZkYWIyZDQ3MDQ5NzNlNGZhYTVhZjZiNGFmI
iwidXNlcl9pZCI6M30.SHqRpftpgF6rNyUPZ608jbKZofYP25kGrLyufKbO
gj4"}
```

In a scenario when the token has expired or is not valid, the endpoint will return the following JSON response:

```
{"detail":"Token is invalid or
expired","code":"token_not_valid"}
```

Token verification endpoint

This endpoint is available without access restrictions in our application on the URL: `api/token/verify/`

The endpoint is used to verify that the access and refresh tokens are valid or have not expired.

You can verify an access token with the following command:

```
$ curl -X POST -H "Content-Type: application/json" -d
'{"token": "[accessToken]"}'
http://localhost:8000/api/token/verify/
```

You can verify a refresh token with the following command:

```
$ curl -X POST -H "Content-Type: application/json" -d
'{"token": "[refreshToken]"}'
http://localhost:8000/api/token/verify/
```

In a scenario when a token is valid and has not expired, the endpoint will just return an empty response:

```
{}
```

In a scenario when a token has expired or is not valid, the endpoint will return the following JSON response:

```
{"detail":"Token is invalid or
expired","code":"token_not_valid"}
```

In the next section, we will configure the endpoints that we created with the DRF in *Chapter 3* with security. That is, we will restrict access to non-authenticated users using the DRF combined with Simple JWT

Securing the REST endpoints

Securing the API is as simple as adding one line in each ViewSet located in the `views.py` file of the application, as you can see in *Figure 4.4* below:

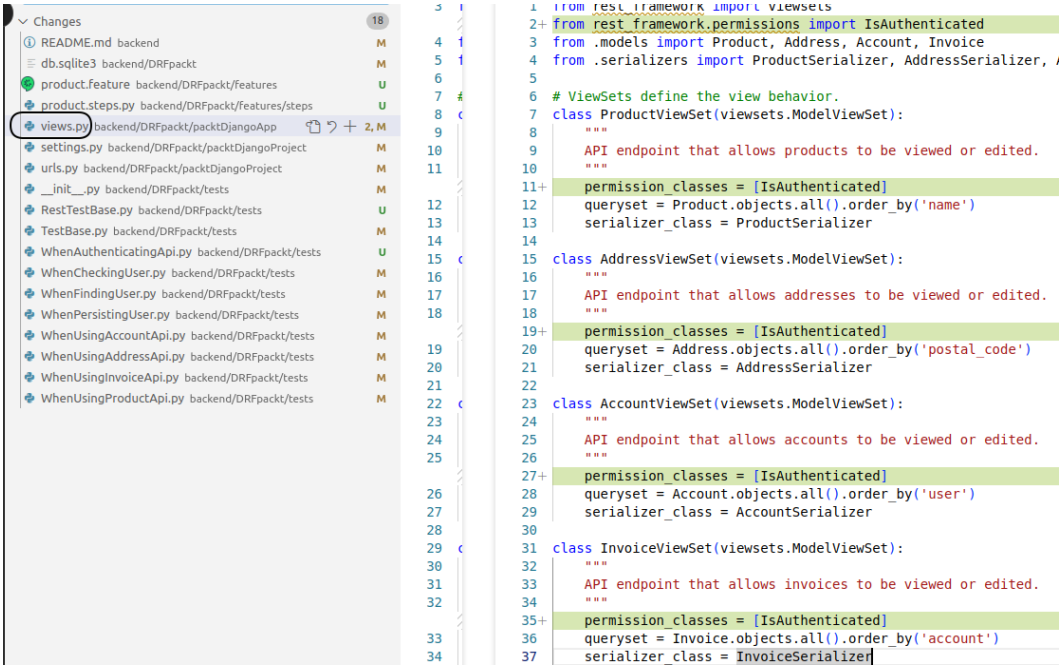

Figure 4.4 – Securing the REST endpoints

As you can see in the figure, we have just added one line for each ViewSet:

```
permission_classes = [IsAuthenticated]
```

This is enough to allow access only to authenticated users that provide a valid access token on each request to the secured API endpoints.

Once our API is secured, in the next section, we will verify that our API is effectively secured by testing it both automatically and manually.

Testing whether your API is secured

As mentioned in the section *Testing whether your API is working* in *Chapter 3*, manual testing is never a substitute for automated testing.

We have provided automated tests to verify that the API is secured. These tests are available in the Git repository for this book, and you can run them to verify that they work as expected.

Here again I mention 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

But also keep in mind that as a complement to those automated tests, we will show you how to test the secured endpoints manually for additional verification.

- **Scenario when you try to access an endpoint and you do not provide any access token**

  Validate that the endpoint is secured with the following command, for any secured endpoint (product list in this case):

```
$ curl -X GET http://localhost:8000/api/v0/products/
```

  And the endpoint will return the following response:

```
{"detail":"Authentication credentials were not provided."}
```

- **Scenario when you try to access an endpoint and you provide an invalid or expired access token**

  Validate that the endpoint is secured with the following command, for any secured endpoint (product list in this case):

```
$ curl -X GET -H "Authorization: Bearer [invalidToken]" http://localhost:8000/api/v0/products/
```

  And the endpoint will return the following response:

```
"detail":"Given token not valid for any token type","code":"token_not_valid","messages":[{"token_class":"AccessToken","token_type":"access","message":"Token is invalid or expired"}]}
```

- **Scenario when you try to access an endpoint and you provide a valid access token**

Validate that the endpoint is secured with the following command, for any secured endpoint (product list in this case):

```
$ curl -X GET -H "Authorization: Bearer [validToken]" http://localhost:8000/api/v0/products/
```

And the endpoint will return a list of records as response.

Summary

In this section we learned how to secure our REST API by allowing access only to authenticated users with a solution based on a JWT.

At this point of the book, we have everything we need to let clients consume this API safely.

In the next chapter, we will create one frontend application as a client or consumer of our API.

We will start the second part of the book in which we will use a different technology stack, that is the Angular framework plus the Typescript programming language.