DJANGO REST FRAMEWORK WITH JSON WEB TOKEN (JWT) PROJECT

How to Implement Token Authentication using Django REST Framework

In this tutorial you are going to learn how to implement Token-based authentication using Django REST Framework (DRF). The token authentication works by exchanging username and password for a token that will be used in all subsequent requests so to identify the user on the server side.

The specifics of how the authentication is handled on the client side vary **a lot** depending on the technology/language/framework you are working with. The client could be a mobile application using iOS or Android. It could be a desktop application using Python or C++. It could be a Web application using PHP or Ruby.

But once you understand the overall process, it’s easier to find the necessary resources and documentation for your specific use case.

Token authentication is suitable for client-server applications, where the token is **safely stored**. You should never expose your token, as it would be (sort of) equivalent of a handing out your username and password.

**Table of Contents**

* [Setting Up The REST API Project](https://simpleisbetterthancomplex.com/tutorial/2018/11/22/how-to-implement-token-authentication-using-django-rest-framework.html#setting-up-the-rest-api-project) (If you already know how to start a DRF project you can skip this)
* [Implementing the Token Authentication](https://simpleisbetterthancomplex.com/tutorial/2018/11/22/how-to-implement-token-authentication-using-django-rest-framework.html#implementing-the-token-authentication)
* [User Requesting a Token](https://simpleisbetterthancomplex.com/tutorial/2018/11/22/how-to-implement-token-authentication-using-django-rest-framework.html#user-requesting-a-token)
* [Conclusions](https://simpleisbetterthancomplex.com/tutorial/2018/11/22/how-to-implement-token-authentication-using-django-rest-framework.html#conclusions)

#### **Setting Up The REST API Project**

So let’s start from the very beginning. Install Django and DRF:

pip install django

pip install djangorestframework

Create a new Django project:

django-admin.py startproject myapi .

Navigate to the **myapi** folder:

cd myapi

Start a new app. I will call my app **core**:

django-admin.py startapp core

Here is what your project structure should look like:

myapi/

|-- core/

| |-- migrations/

| |-- \_\_init\_\_.py

| |-- admin.py

| |-- apps.py

| |-- models.py

| |-- tests.py

| +-- views.py

|-- \_\_init\_\_.py

|-- settings.py

|-- urls.py

+-- wsgi.py

manage.py

Add the **core** app (you created) and the **rest\_framework** app (you installed) to the INSTALLED\_APPS, inside the **settings.py** module:

**myapi/settings.py**

INSTALLED\_APPS **=** [

*# Django Apps*

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

*# Third-Party Apps*

'rest\_framework',

*# Local Apps (Your project's apps)*

'myapi.core',

]

Return to the project root (the folder where the **manage.py** script is), and migrate the database:

python manage.py migrate

Let’s create our first API view just to test things out:

**myapi/core/views.py**

from rest\_framework.views import APIView

from rest\_framework.response import Response

**class** **HelloView**(APIView):

**def** **get**(self, request):

content **=** {'message': 'Hello, World!'}

**return** Response(content)

Now register a path in the **urls.py** module:

**myapi/urls.py**

from django.urls import path

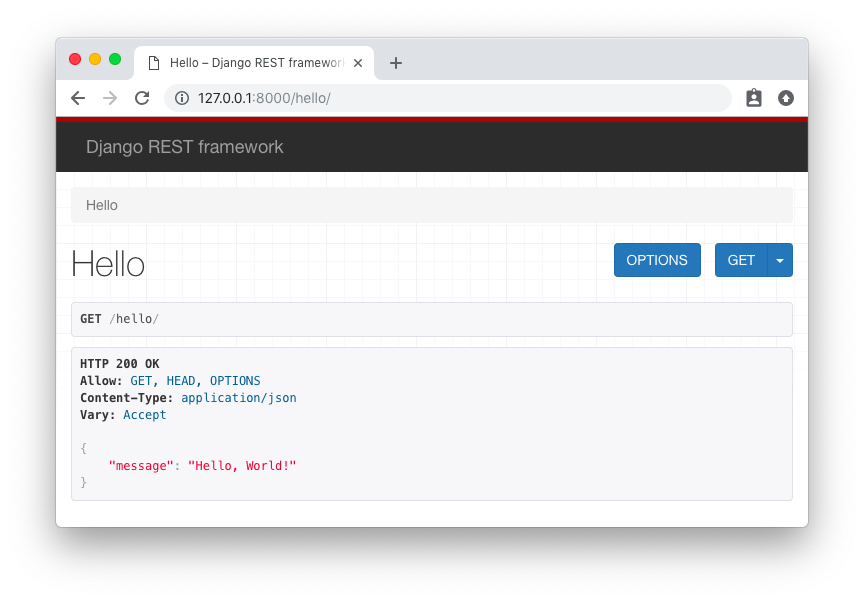
from myapi.core import views

urlpatterns **=** [

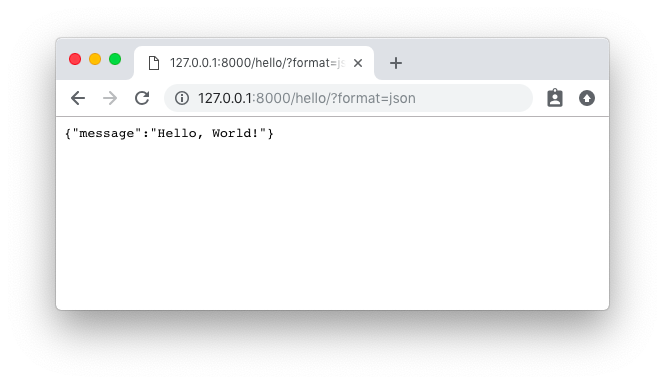
path('hello/', views**.**HelloView**.**as\_view(), name**=**'hello'),

]

So now we have an API with just one endpoint /hello/ that we can perform GET requests. We can use the browser to consume this endpoint, just by accessing the URL http://127.0.0.1:8000/hello/:

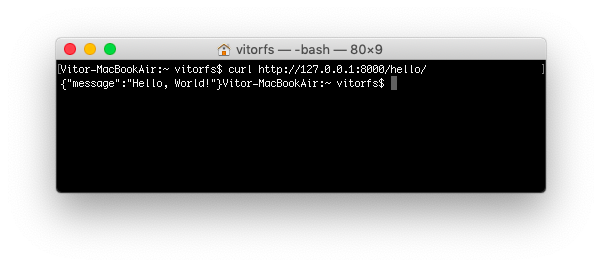


We can also ask to receive the response as plain JSON data by passing the format parameter in the querystring like http://127.0.0.1:8000/hello/?format=json:



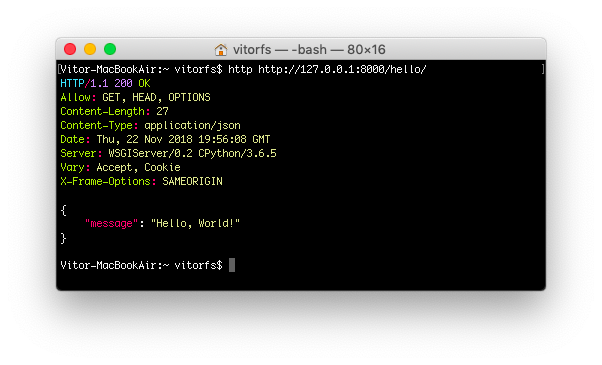
Both methods are fine to try out a DRF API, but sometimes a command line tool is more handy as we can play more easily with the requests headers. You can use [cURL](https://curl.haxx.se/), which is widely available on all major Linux/macOS distributions:

curl http://127.0.0.1:8000/hello/



But usually I prefer to use [HTTPie](https://httpie.org/), which is a pretty awesome Python command line tool:

http http://127.0.0.1:8000/hello/



Now let’s protect this API endpoint so we can implement the token authentication:

**myapi/core/views.py**

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework.permissions import IsAuthenticated *# <-- Here*

**class** **HelloView**(APIView):

permission\_classes **=** (IsAuthenticated,) *# <-- And here*

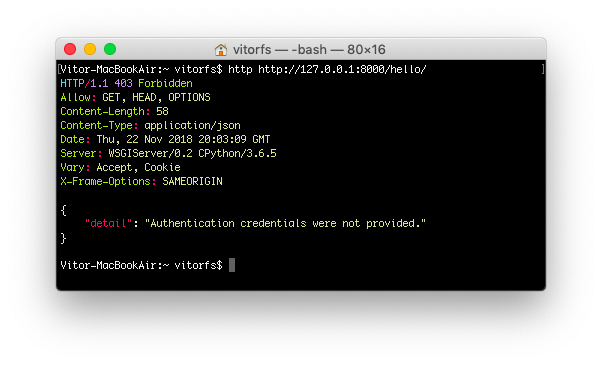
**def** **get**(self, request):

content **=** {'message': 'Hello, World!'}

**return** Response(content)

Try again to access the API endpoint:

http http://127.0.0.1:8000/hello/



And now we get an HTTP 403 Forbidden error. Now let’s implement the token authentication so we can access this endpoint.

#### **Implementing the Token Authentication**

We need to add two pieces of information in our **settings.py** module. First include **rest\_framework.authtoken** to your INSTALLED\_APPS and include the TokenAuthentication to REST\_FRAMEWORK:

**myapi/settings.py**

INSTALLED\_APPS **=** [

*# Django Apps*

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

*# Third-Party Apps*

'rest\_framework',

'rest\_framework.authtoken', *# <-- Here*

*# Local Apps (Your project's apps)*

'myapi.core',

]

REST\_FRAMEWORK **=** {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

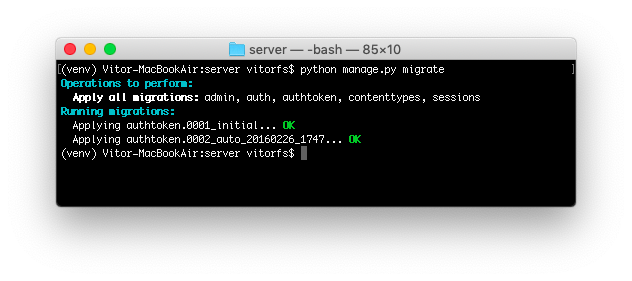
'rest\_framework.authentication.TokenAuthentication', *# <-- And here*

],

}

Migrate the database to create the table that will store the authentication tokens:

python manage.py migrate

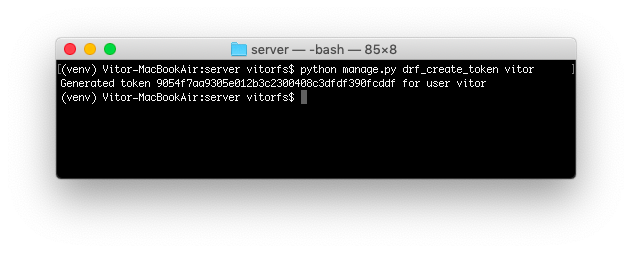


Now we need a user account. Let’s just create one using the manage.py command line utility:

python manage.py createsuperuser --username vitor --email vitor@example.com

The easiest way to generate a token, just for testing purpose, is using the command line utility again:

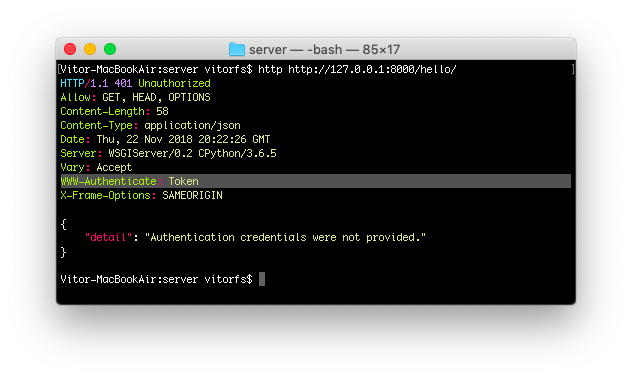
python manage.py drf\_create\_token vitor



This piece of information, the random string 9054f7aa9305e012b3c2300408c3dfdf390fcddf is what we are going to use next to authenticate.

But now that we have the TokenAuthentication in place, let’s try to make another request to our /hello/ endpoint:

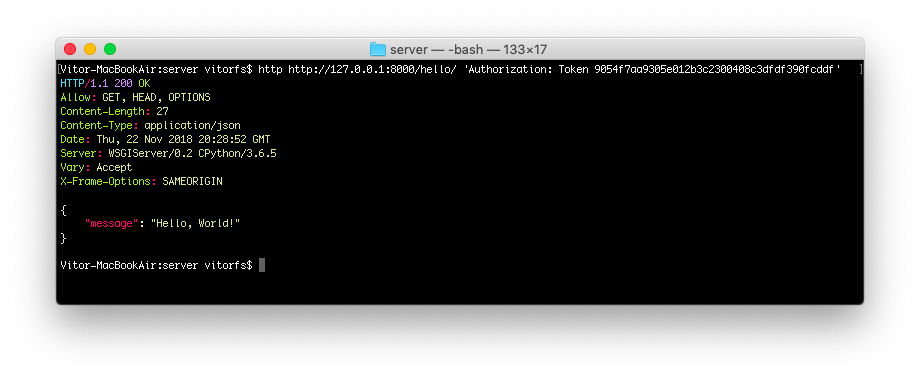
http http://127.0.0.1:8000/hello/



Notice how our API is now providing some extra information to the client on the required authentication method.

So finally, let’s use our token!

http http://127.0.0.1:8000/hello/ 'Authorization: Token 9054f7aa9305e012b3c2300408c3dfdf390fcddf'



And that’s pretty much it. For now on, on all subsequent request you should include the header Authorization: Token 9054f7aa9305e012b3c2300408c3dfdf390fcddf.

The formatting looks weird and usually it is a point of confusion on how to set this header. It will depend on the client and how to set the HTTP request header.

For example, if we were using cURL, the command would be something like this:

curl http://127.0.0.1:8000/hello/ -H 'Authorization: Token 9054f7aa9305e012b3c2300408c3dfdf390fcddf'

Or if it was a Python [requests](http://docs.python-requests.org/) call:

import requests

url **=** 'http://127.0.0.1:8000/hello/'

headers **=** {'Authorization': 'Token 9054f7aa9305e012b3c2300408c3dfdf390fcddf'}

r **=** requests**.**get(url, headers**=**headers)

Or if we were using Angular, you could implement an HttpInterceptor and set a header:

**import** { Injectable } from '@angular/core';

**import** { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '@angular/common/http';

**import** { Observable } from 'rxjs';

@Injectable()

**export** **class** AuthInterceptor **implements** HttpInterceptor {

intercept(request: HttpRequest**<**any**>**, next: HttpHandler): Observable**<**HttpEvent**<**any**>>** {

**const** user **=** JSON.parse(localStorage.getItem('user'));

**if** (user **&&** user.token) {

request **=** request.clone({

setHeaders: {

Authorization: `Token ${user.accessToken}`

}

});

}

**return** next.handle(request);

}

}

#### **User Requesting a Token**

The DRF provide an endpoint for the users to request an authentication token using their username and password.

Include the following route to the **urls.py** module:

**myapi/urls.py**

from django.urls import path

from rest\_framework.authtoken.views import obtain\_auth\_token *# <-- Here*

from myapi.core import views

urlpatterns **=** [

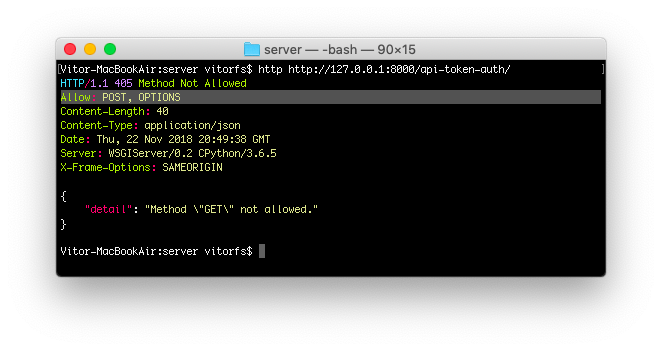
path('hello/', views**.**HelloView**.**as\_view(), name**=**'hello'),

path('api-token-auth/', obtain\_auth\_token, name**=**'api\_token\_auth'), *# <-- And here*

]

So now we have a brand new API endpoint, which is /api-token-auth/. Let’s first inspect it:

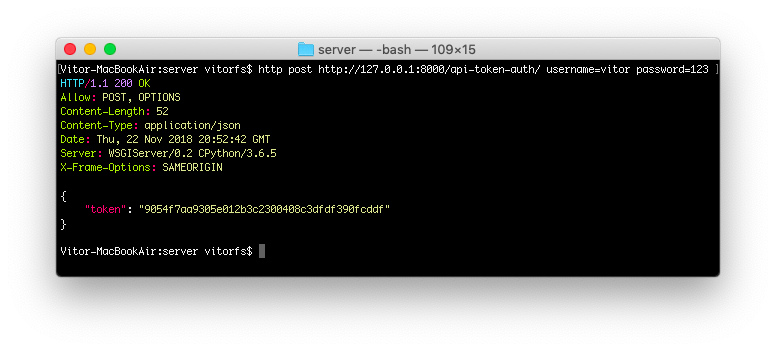
http http://127.0.0.1:8000/api-token-auth/



It doesn’t handle GET requests. Basically it’s just a view to receive a POST request with username and password.

Let’s try again:

http post http://127.0.0.1:8000/api-token-auth/ username**=**vitor password**=**123



The response body is the token associated with this particular user. After this point you store this token and apply it to the future requests.

Then, again, the way you are going to make the POST request to the API depends on the language/framework you are using.

If this was an Angular client, you could store the token in the localStorage, if this was a Desktop CLI application you could store in a text file in the user’s home directory in a dot file.

#### **Conclusions**

Hopefully this tutorial provided some insights on how the token authentication works. I will try to follow up this tutorial providing some concrete examples of Angular applications, command line applications and Web clients as well.

It is important to note that the default Token implementation has some limitations such as only one token per user, no built-in way to set an expiry date to the token.

You can grab the code used in this tutorial at [github.com/sibtc/drf-token-auth-example](https://github.com/sibtc/drf-token-auth-example).

### How to Use JWT Authentication with Django REST Framework

JWT stand for **JSON Web Token** and it is an authentication strategy used by client/server applications where the client is a Web application using JavaScript and some frontend framework like Angular, React or VueJS.

In this tutorial we are going to explore the specifics of JWT authentication. If you want to learn more about Token-based authentication using Django REST Framework (DRF), or if you want to know how to start a new DRF project you can read this tutorial: [How to Implement Token Authentication using Django REST Framework](https://simpleisbetterthancomplex.com/tutorial/2018/11/22/how-to-implement-token-authentication-using-django-rest-framework.html). The concepts are the same, we are just going to switch the authentication backend.

* [How JWT Works?](https://simpleisbetterthancomplex.com/tutorial/2018/12/19/how-to-use-jwt-authentication-with-django-rest-framework.html#how-jwt-works)
* [Installation & Setup](https://simpleisbetterthancomplex.com/tutorial/2018/12/19/how-to-use-jwt-authentication-with-django-rest-framework.html#installation--setup)
* [Example Code](https://simpleisbetterthancomplex.com/tutorial/2018/12/19/how-to-use-jwt-authentication-with-django-rest-framework.html#example-code)
* [Usage](https://simpleisbetterthancomplex.com/tutorial/2018/12/19/how-to-use-jwt-authentication-with-django-rest-framework.html#usage)
  + [Obtain Token](https://simpleisbetterthancomplex.com/tutorial/2018/12/19/how-to-use-jwt-authentication-with-django-rest-framework.html#obtain-token)
  + [Refresh Token](https://simpleisbetterthancomplex.com/tutorial/2018/12/19/how-to-use-jwt-authentication-with-django-rest-framework.html#refresh-token)
* [What’s The Point of The Refresh Token?](https://simpleisbetterthancomplex.com/tutorial/2018/12/19/how-to-use-jwt-authentication-with-django-rest-framework.html#whats-the-point-of-the-refresh-token)
* [Further Reading](https://simpleisbetterthancomplex.com/tutorial/2018/12/19/how-to-use-jwt-authentication-with-django-rest-framework.html#further-reading)

#### **How JWT Works?**

The JWT is just an authorization token that should be included in all requests:

curl http://127.0.0.1:8000/hello/ -H 'Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNTQzODI4NDMxLCJqdGkiOiI3ZjU5OTdiNzE1MGQ0NjU3OWRjMmI0OTE2NzA5N2U3YiIsInVzZXJfaWQiOjF9.Ju70kdcaHKn1Qaz8H42zrOYk0Jx9kIckTn9Xx7vhikY'

The JWT is acquired by exchanging an username + password for an **access token** and an **refresh token**.

The **access token** is usually short-lived (expires in 5 min or so, can be customized though).

The **refresh token** lives a little bit longer (expires in 24 hours, also customizable). It is comparable to an authentication session. After it expires, you need a full login with username + password again.

Why is that?

It’s a security feature and also it’s because the JWT holds a little bit more information. If you look closely the example I gave above, you will see the token is composed by three parts:

xxxxx.yyyyy.zzzzz

Those are three distinctive parts that compose a JWT:

header.payload.signature

So we have here:

header **=** eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

payload **=** eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNTQzODI4NDMxLCJqdGkiOiI3ZjU5OTdiNzE1MGQ0NjU3OWRjMmI0OTE2NzA5N2U3YiIsInVzZXJfaWQiOjF9

signature **=** Ju70kdcaHKn1Qaz8H42zrOYk0Jx9kIckTn9Xx7vhikY

This information is encoded using Base64. If we decode, we will see something like this:

**header**

{

"typ": "JWT",

"alg": "HS256"

}

**payload**

{

"token\_type": "access",

"exp": 1543828431,

"jti": "7f5997b7150d46579dc2b49167097e7b",

"user\_id": 1

}

**signature**

The signature is issued by the JWT backend, using the header base64 + payload base64 + SECRET\_KEY. Upon each request this signature is verified. If any information in the header or in the payload was changed by the client it will invalidate the signature. The only way of checking and validating the signature is by using your application’s SECRET\_KEY. Among other things, that’s why you should always keep your SECRET\_KEY **secret**!

#### **Installation & Setup**

For this tutorial we are going to use the [djangorestframework\_simplejwt](https://github.com/davesque/django-rest-framework-simplejwt) library, recommended by the DRF developers.

pip install djangorestframework\_simplejwt

**settings.py**

REST\_FRAMEWORK **=** {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

'rest\_framework\_simplejwt.authentication.JWTAuthentication',

],

}

**urls.py**

from django.urls import path

from rest\_framework\_simplejwt import views **as** jwt\_views

urlpatterns **=** [

*# Your URLs...*

path('api/token/', jwt\_views**.**TokenObtainPairView**.**as\_view(), name**=**'token\_obtain\_pair'),

path('api/token/refresh/', jwt\_views**.**TokenRefreshView**.**as\_view(), name**=**'token\_refresh'),

]

#### **Example Code**

For this tutorial I will use the following route and API view:

**views.py**

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework.permissions import IsAuthenticated

**class** **HelloView**(APIView):

permission\_classes **=** (IsAuthenticated,)

**def** **get**(self, request):

content **=** {'message': 'Hello, World!'}

**return** Response(content)

**urls.py**

from django.urls import path

from myapi.core import views

urlpatterns **=** [

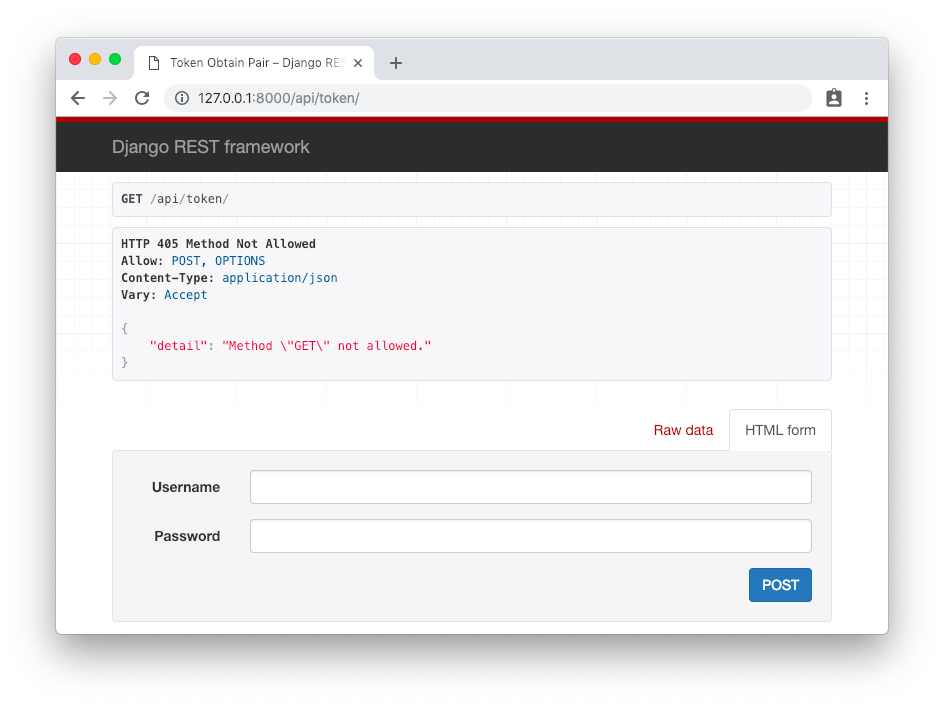
path('hello/', views**.**HelloView**.**as\_view(), name**=**'hello'),

]

#### **Usage**

I will be using [HTTPie](https://httpie.org/) to consume the API endpoints via the terminal. But you can also use [cURL](https://curl.haxx.se/) (readily available in many OS) to try things out locally.

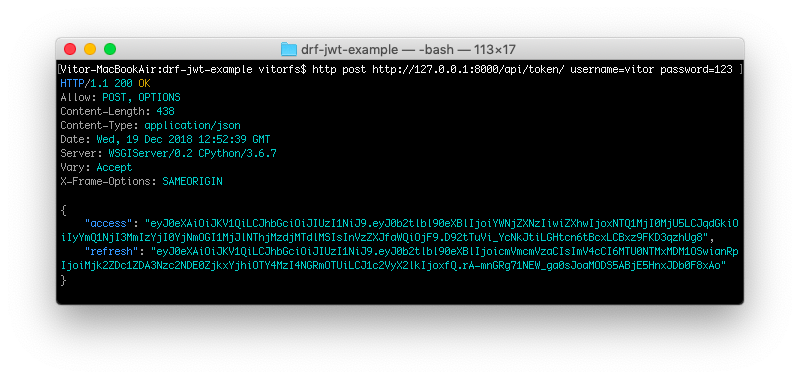
Or alternatively, use the DRF web interface by accessing the endpoint URLs like this:



##### **Obtain Token**

First step is to **authenticate and obtain the token**. The endpoint is /api/token/ and it only accepts **POST** requests.

http post http://127.0.0.1:8000/api/token/ username**=**vitor password**=**123



So basically your response body is the two tokens:

{

"access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNTQ1MjI0MjU5LCJqdGkiOiIyYmQ1NjI3MmIzYjI0YjNmOGI1MjJlNThjMzdjMTdlMSIsInVzZXJfaWQiOjF9.D92tTuVi\_YcNkJtiLGHtcn6tBcxLCBxz9FKD3qzhUg8",

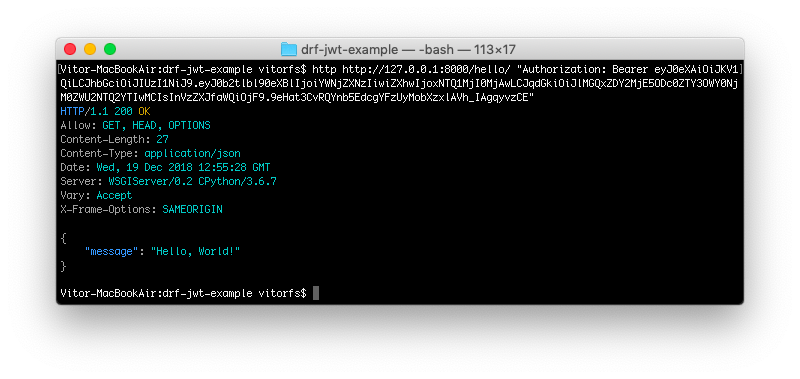
"refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTU0NTMxMDM1OSwianRpIjoiMjk2ZDc1ZDA3Nzc2NDE0ZjkxYjhiOTY4MzI4NGRmOTUiLCJ1c2VyX2lkIjoxfQ.rA-mnGRg71NEW\_ga0sJoaMODS5ABjE5HnxJDb0F8xAo"

}

After that you are going to store both the **access token** and the **refresh token** on the client side, usually in the [localStorage](https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage).

In order to access the protected views on the backend (i.e., the API endpoints that require authentication), you should include the **access token** in the header of all requests, like this:

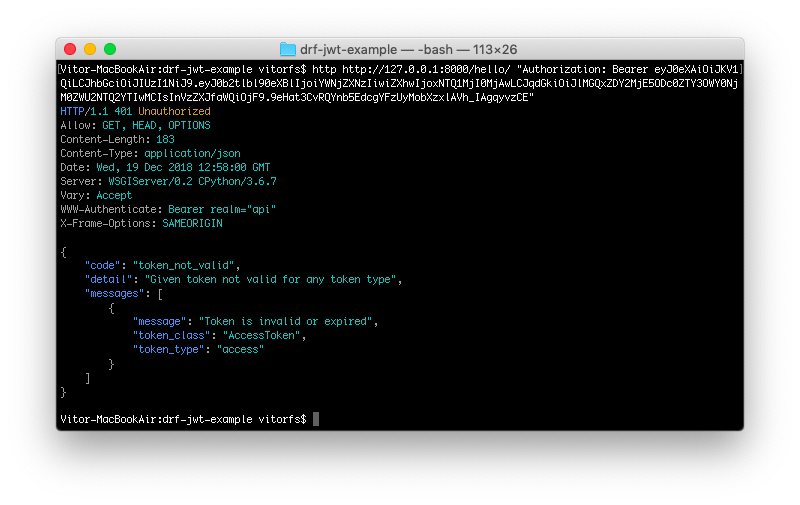
http http://127.0.0.1:8000/hello/ "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNTQ1MjI0MjAwLCJqdGkiOiJlMGQxZDY2MjE5ODc0ZTY3OWY0NjM0ZWU2NTQ2YTIwMCIsInVzZXJfaWQiOjF9.9eHat3CvRQYnb5EdcgYFzUyMobXzxlAVh\_IAgqyvzCE"



You can use this **access token** for the next five minutes.

After five min, the token will expire, and if you try to access the view again, you are going to get the following error:

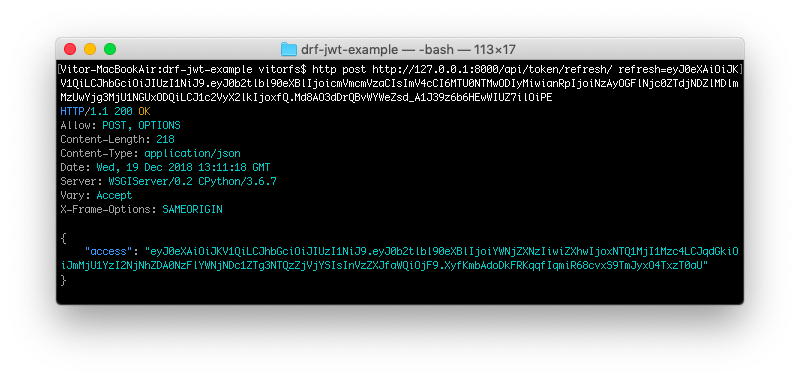
http http://127.0.0.1:8000/hello/ "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNTQ1MjI0MjAwLCJqdGkiOiJlMGQxZDY2MjE5ODc0ZTY3OWY0NjM0ZWU2NTQ2YTIwMCIsInVzZXJfaWQiOjF9.9eHat3CvRQYnb5EdcgYFzUyMobXzxlAVh\_IAgqyvzCE"



##### **Refresh Token**

To get a new **access token**, you should use the refresh token endpoint /api/token/refresh/ posting the **refresh token**:

http post http:**//**127.0**.**0.1:8000**/**api**/**token**/**refresh**/** refresh**=**eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9**.**eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTU0NTMwODIyMiwianRpIjoiNzAyOGFlNjc0ZTdjNDZlMDlmMzUwYjg3MjU1NGUxODQiLCJ1c2VyX2lkIjoxfQ**.**Md8AO3dDrQBvWYWeZsd\_A1J39z6b6HEwWIUZ7ilOiPE



The return is a new **access token** that you should use in the subsequent requests.

The **refresh token** is valid for the next 24 hours. When it finally expires too, the user will need to perform a full authentication again using their username and password to get a new set of **access token** + **refresh token**.

#### **What’s The Point of The Refresh Token?**

At first glance the **refresh token** may look pointless, but in fact it is necessary to make sure the user still have the correct permissions. If your **access token** have a long expire time, it may take longer to update the information associated with the token. That’s because the authentication check is done by cryptographic means, instead of querying the database and verifying the data. So some information is sort of cached.

There is also a security aspect, in a sense that the **refresh token** only travel in the POST data. And the **access token** is sent via HTTP header, which may be logged along the way. So this also give a short window, should your **access token** be compromised.

#### **Further Reading**

This should cover the basics on the backend implementation. It’s worth checking the [djangorestframework\_simplejwt settings](https://github.com/davesque/django-rest-framework-simplejwt) for further customization and to get a better idea of what the library offers.

The implementation on the frontend depends on what framework/library you are using. Some libraries and articles covering popular frontend frameworks like angular/react/vue.js:

* [Angular JWT library](https://github.com/auth0/angular-jwt)
* [Angular 2 JWT library](https://github.com/auth0/angular2-jwt)
* [Secure Your React and Redux App with JWT Authentication](https://auth0.com/blog/secure-your-react-and-redux-app-with-jwt-authentication/)
* [Authenticating via JWT using Django, Axios, and Vue](https://www.pydanny.com/drf-jwt-axios-vue.html)

The code used in this tutorial is available at [github.com/sibtc/drf-jwt-example](https://github.com/sibtc/drf-jwt-example).