envato tuts+

Code > Python

JWT Authentication in Django

Esther Vaati Last updated Dec 15, 2022

10 min | English

This tutorial will give an introduction to JSON Web Tokens (JWT) and how to implement JWT authentication in Django.

What Is JWT?

JWT is an encoded JSON string that is passed in headers to authenticate requests. It is usually obtained by hashing JSON data with a secret key. This means that the server doesn't need to query the database every time to retrieve the user associated with a given token.

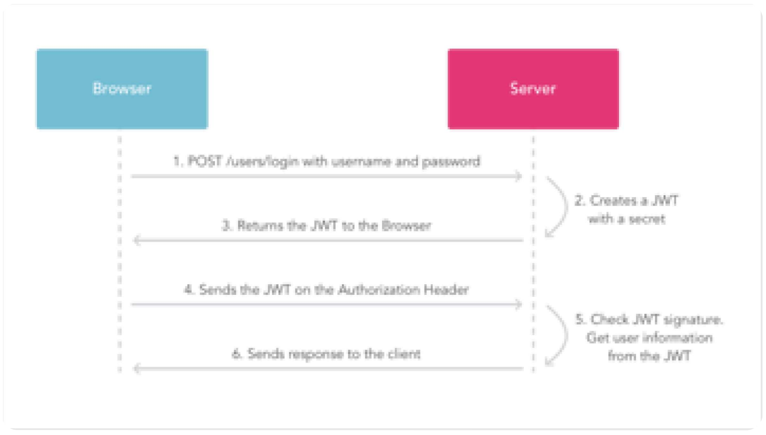
How JSON Web Tokens Work

When a user successfully logs in using their credentials, a JSON Web Token is obtained and saved in local storage. Whenever the user wants to access a protected URL, the token is sent in the header of the request. The server then checks for a valid JWT in the Authorization header, and if found, the user will be allowed access.

A typical content header will look like this:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsI
```

Below is a diagram showing this process:


Browser
Server
1. POST /users/login with username and password
2. Creates a JWT with a secret
3. Returns the JWT to the Browser
4. Sends the JWT on the Authorization Header
5. Check JWT signature. Get user information from the JWT
6. Sends response to the client

Advertisement

The Concept of Authentication and Authorization

Authentication is the process of identifying a logged-in user, while authorization is the process of identifying if a certain user has the right to access a web resource.

API Example

In this tutorial, we are going to build a simple user authentication system in Django JWT as the authentication mechanism.

Requirements

- Django
- Python

Let's get started.

Create a directory where you will keep your project and also a virtual environment to install the project dependencies.

```
mkdir myprojects

cd myprojects

virtual venv

```

Activate the virtual environment:

```
source venv/bin/activate
```

Create a Django project.

```
django-admin startproject django_auth

```

Install DRF and django-rest-framework-jwt using pip.

```
pip install djangorestframework
pip install djangorestframework-jwt
pip install django
```

Let's go ahead and add DRF to the list of installed apps in the `settings.py` file.

Configure the JWT Settings

In order to use a simple JWT, we need to configure django-rest-framework permissions to accept JSON Web Tokens.

In the `settings.py` file, add the following configurations:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
      'rest_framework_jwt.authentication.JSONWebTokenAuthentication',
    ),
}
```

Create a new app called users which will handle user authentication and management.

```
cd django-auth
django-admin.py startapp users
```

Add the users application to the list of installed apps in the `settings.py` file.

Setting Up the Database

We are going to use the PostgreSQL database because it's more stable and robust.

Create the `auth` database and assign a user.

Switch over to the Postgres account on your machine by typing:

```
sudo su postgres
```

Access the Postgres prompt and create the database:

```
psql
postgres=# CREATE DATABASE auth;
```

Create a role:

```
postgres=# CREATE ROLE django_auth WITH LOGIN PASSWORD 'asdfgh';
```

Grant database access to the user:

```
postgres=# GRANT ALL PRIVILEGES ON DATABASE auth TO django_auth;
```

Install the psycopg2 package, which will allow us to use the database we configured:

```
pip install psycopg2
```

Edit the currently configured SQLite database and use the Postgres database.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'auth',
        'USER': 'django_auth',
        'PASSWORD': 'asdfgh',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

Creating Models

Django comes with a built-in authentication system which is very elaborate, but sometimes we need to make adjustments, and thus we need to create a custom user authentication system. Our user model will be inheriting from the `AbstractBaseUser` class provided by `django.contrib.auth.models`.

In users/models.py, we start by creating the User model to store the user details.

```
# users/models.py
from __future__ import unicode_literals
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import (
    AbstractBaseUser, PermissionsMixin
)

```

```
class User(AbstractBaseUser, PermissionsMixin):
    """
An abstract base class implementing a fully featured User model with
admin-compliant permissions.

"""
    email = models.EmailField(max_length=40, unique=True)
    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=30, blank=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)
    date_joined = models.DateTimeField(default=timezone.now)

    objects = UserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name']

    def save(self, *args, **kwargs):
        super(User, self).save(*args, **kwargs)
        return self
```

`REQUIRED_FIELDS` contains all required fields on your user model, except the username field and password, as these fields will always be prompted for.

`UserManager` is the class that defines the `create_user` and `createsuperuser` methods. This class should come before the `AbstractBaseUser` class we defined above. Let's go ahead and define it.

```
from django.contrib.auth.models import (
    AbstractBaseUser, PermissionsMixin, BaseUserManager
)

class UserManager(BaseUserManager):

    def _create_user(self, email, password, **extra_fields):
        """
Creates and saves a User with the given email,and password.
"""
        if not email:
            raise ValueError('The given email must be set')
        try:
            with transaction.atomic():
                user = self.model(email=email, **extra_fields)
                user.set_password(password)
                user.save(using=self._db)
                return user
        except:
            raise

    def create_user(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', False)
        extra_fields.setdefault('is_superuser', False)
        return self._create_user(email, password, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)

        return self._create_user(email, password=password, **extra_fields)
```

Migrations

Migrations provide a way of updating your database schema every time your models change, without losing data.

Create an initial migration for our users model, and sync the database for the first time.

```
python manage.py make migrations users

python manage.py migrate
```

Creating a Superuser

Create a superuser by running the following command:

```
python manage.py createsuperuser
```

Creating New Users

Let's create an endpoint to enable registration of new users. We will start by serializing the User model fields. Serializers provide a way of changing data to a form that is easier to understand, like JSON or XML. Deserialization does the opposite, which is converting data to a form that can be saved to the database.

Create users/serializers.py and add the following code.

```
# users/serializers.py
from rest_framework import serializers
from.models import User


class UserSerializer(serializers.ModelSerializer):

    date_joined = serializers.ReadOnlyField()

    class Meta(object):
        model = User
        fields = ('id', 'email', 'first_name', 'last_name',
                  'date_joined', 'password')
        extra_kwargs = {'password': {'write_only': True}}
```

CreateUserAPIView

Next, we want to create a view so the client will have a URL for creating new users.

In users.views.py, add the following:

```
# users/views.py
class CreateUserAPIView(APIView):
    # Allow any user (authenticated or not) to access this url
    permission_classes = (AllowAny,)

    def post(self, request):
        user = request.data
        serializer = UserSerializer(data=user)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
```

We set `permission_classes` to `(AllowAny,)` to allow any user (authenticated or not) to access this URL.

Configuring URLs

Create a file `users/urls.py` and add the URL to match the view we created. Also add the following code.

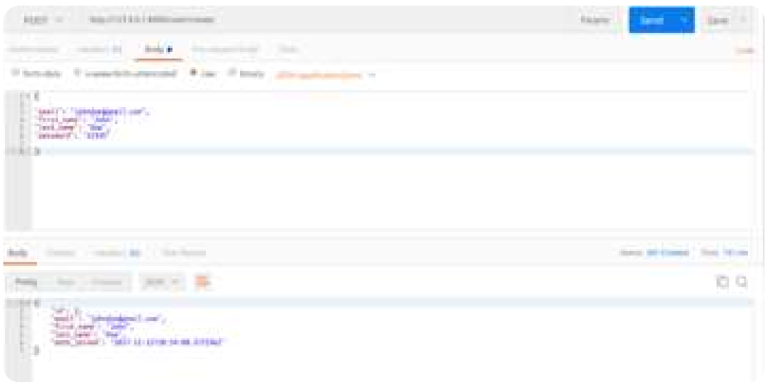
```
# users/urls.py

from django.conf.urls import url, patterns
from .views import CreateUserAPIView

urlpatterns = [
    url(r'^create/$', CreateUserAPIView.as_view()),
]
```

We also need to import URLs from the users application to the main `django_auth/urls.py` file. So go ahead and do that. We are using the `include` function here, so don't forget to import it.

```
# django_auth/urls.py
from django.conf.urls import url, include
```

```python
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^user/', include('users.urls', namespace='users')),

]
```

Now that we are done creating the endpoint, let's do a test and see if we are on track. We will use Postman to do the tests. If you are not familiar with Postman, it's a tool which presents a friendly GUI for constructing requests and reading responses.



As you can see above, the endpoint is working as expected.

Authenticating Users

We will make use of the Django-REST Framework JWT Python module we installed at the beginning of this tutorial. It adds a simple JWT authentication support for Django Rest Framework JWT apps.

But first, let's define some configuration parameters for our tokens and how they are generated in the settings.py file.

```python
# settings.py
import datetime
JWT_AUTH = {

    'JWT_VERIFY': True,
    'JWT_VERIFY_EXPIRATION': True,
    'JWT_EXPIRATION_DELTA': datetime.timedelta(seconds=3000),
    'JWT_AUTH_HEADER_PREFIX': 'Bearer',

}
```

- `JWT_VERIFY`: It will raise a jwt.DecodeError if the secret is wrong.

- `JWT_VERIFY_EXPIRATION`: Sets the expiration to True, meaning Tokens will expire after a period of time. The default time is five minutes.
- `JWT_AUTH_HEADER_PREFIX`: The Authorization header value prefix that is required to be sent together with the token. We have set it as `Bearer`, and the default is `JWT`.

In `users/views.py`, add the following code.

```
@api_view(['POST'])
@permission_classes([AllowAny, ])
def authenticate_user(request):

    try:
        email = request.data['email']
        password = request.data['password']

        user = User.objects.get(email=email, password=password)
        if user:
            try:
                payload = jwt_payload_handler(user)
                token = jwt.encode(payload, settings.SECRET_KEY)
                user_details = {}
                user_details['name'] = "%s %s" % (
                    user.first_name, user.last_name)
                user_details['token'] = token
                user_logged_in.send(sender=user.__class__,
                                    request=request, user=user)
                return Response(user_details, status=status.HTTP_200_OK)

            except Exception as e:
                raise e
        else:
            res = {
                'error': 'can not authenticate with the given credentials or the acc
            return Response(res, status=status.HTTP_403_FORBIDDEN)
    except KeyError:
        res = {'error': 'please provide a email and a password'}
        return Response(res)
```

In the code above, the login view takes username and password as input, and it then creates a token with the user information corresponding to the passed credentials as payload and returns it to the browser. Other user details such as name are also returned to the browser together with the token. This token will be used to authenticate in future requests.

The permission classes are set to `allowAny` since anyone can access this endpoint.

We also store the last login time of the user with this code.

```python
user_logged_in.send(sender=user.__class__,
                                    request=request, user=user)
```

Every time the user wants to make an API request, they have to send the token in Auth Headers in order to authenticate the request.

Let's test this endpoint with Postman. Open Postman and use the request to authenticate with one of the users you created previously. If the login attempt is successful, the response will look like this:

Retrieving and Updating Users

So far, users can register and authenticate themselves. However, they also need a way to retrieve and update their information. Let's implement this.

In `users.views.py`, add the following code.

```python
class UserRetrieveUpdateAPIView(RetrieveUpdateAPIView):

    # Allow only authenticated users to access this url
    permission_classes = (IsAuthenticated,)
    serializer_class = UserSerializer

    def get(self, request, *args, **kwargs):
        # serializer to handle turning our `User` object into something that
        # can be JSONified and sent to the client.
        serializer = self.serializer_class(request.user)

        return Response(serializer.data, status=status.HTTP_200_OK)

    def put(self, request, *args, **kwargs):
        serializer_data = request.data.get('user', {})

        serializer = UserSerializer(
            request.user, data=serializer_data, partial=True
        )
        serializer.is_valid(raise_exception=True)
        serializer.save()
```

```
            return Response(serializer.data, status=status.HTTP_200_OK)
```

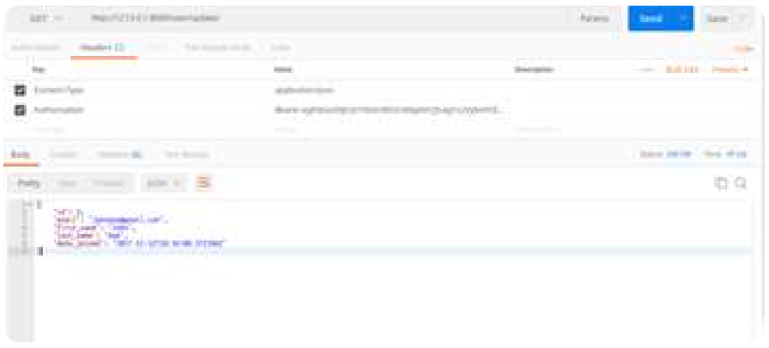
We first define the permission classes and set to `IsAuthenticated` since this is a protected URL and only authenticated users can access it.

We then define a `get` method to retrieve user details. After retrieving user details, an authenticated user will then update their details as desired.

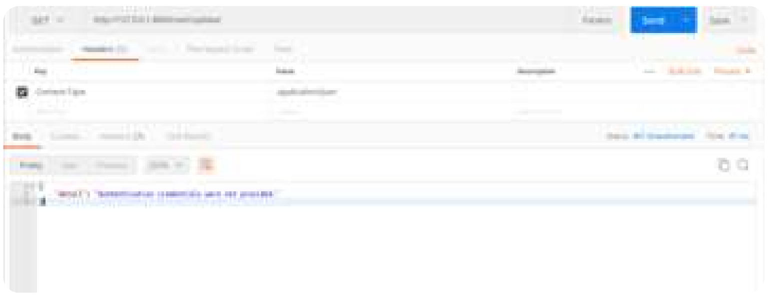
Update your URLs to define the endpoint as follows.

```
users/urls.py
from .views import CreateUserAPIView, UserRetrieveUpdateAPIView

urlpatterns = [

    url(r'^update/$', UserRetrieveUpdateAPIView.as_view()),
]
```

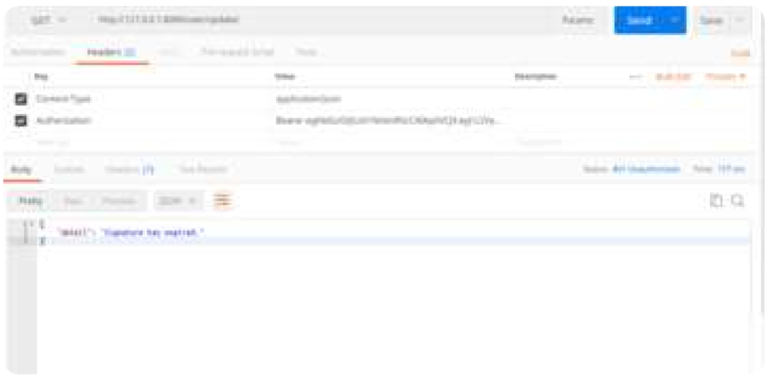
In order for the request to be successful, the headers should contain the JWT token as shown below.



If you attempt to request a resource without the authentication header, you will get the following error.



If a user stays beyond the time specified in `JWT_EXPIRATION_DELTA` without making a request, the token will expire and they will have to request another token. This is a demonstrated below.



```
{
    "detail": "Signature has expired."
}
```

Advertisement

Conclusion

This tutorial has covered what is necessary to successfully build a solid back-end authentication system with JSON Web Tokens. We also covered Django JWT authentication or Django authorization.

Check out more Python and Django JWT articles and tutorials:

How to Use Python to Find the Zipf Distribution of a Text File
Abder-Rahman Ali
23 Aug 2022

How to Run Unix Commands in Your Python Program

Abder-Rahman Ali

13 Jul 2022

How to Make Changes to Multiple Files Using Python

Abder-Rahman Ali

27 Jun 2022

How to Work With JSON Data Using Python

Abder-Rahman Ali

30 Jun 2022

Getting Started With Django: Newly Updated Course

Andrew Blackman

02 Sep 2019

Advertisement

Python Django Web Development

Did you find this post useful?

Yes No

Want a weekly email summary?

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Sign up

Esther Vaati

Software developer

Software developer and content creator. Student of Life | #Pythonist | Loves to code and write Tutorials

vaatiesther_

Advertisement

QUICK LINKS - Explore popular categories

ENVATO TUTS+ +

HELP +

tuts+

30,874 Tutorials

1,281 Courses

47,365 Translations

Certified B Corporation

Envato Envato Elements Envato Market Placeit by Envato All products Careers Sitemap