



Practical: Authentication and sessions in FastAPI with JSON Web Tokens (Python)

Table of Contents

- [Prerequisites and references](#)
- [User sessions and authentication](#)
- [Sample FastAPI app](#)
- [Testing the app](#)
 - [Logging on](#)
 - [Accessing privileged content](#)
 - [Users with additional permissions](#)
- [Read the code](#)

In this practical we'll see how to use JSON web tokens to manage sessions in Python with FastAPI.

Prerequisites and references

- [FastAPI](https://FastAPI.tiangolo.com/)  (<https://FastAPI.tiangolo.com/>)
- [PyJWT module](https://github.com/jpadilla/pyjwt)  (<https://github.com/jpadilla/pyjwt>)
- [JWT FastAPI demo app](https://canvas.qut.edu.au/courses/20367/files/6717519/download) (<https://canvas.qut.edu.au/courses/20367/files/6717519/download>)
- [Hoppscotch request collection](https://canvas.qut.edu.au/courses/20367/files/6717520/download) (<https://canvas.qut.edu.au/courses/20367/files/6717520/download>)

User sessions and authentication

JSON Web Tokens (JWTs) are a standardised way of authenticating users using HTTP requests. We will explore them in more detail later in the unit. For now, it's enough to know that a JWT contains information about a user, cryptographically signed so that it can later be validated.

Our process for creating authenticated user sessions using JWTs will be as follows:

- Logging in:
 1. The client makes a request using a username and password
 2. The server checks that the user is authorised and the password is correct
 3. The server responds with a JWT containing the username
- Accessing content:
 1. The client includes the JWT in the header of their request
 2. The server verifies that the JWT is valid and extracts the username
 3. The server checks that the JWT is authorised to make the request
 4. The server proceeds with the request

Naturally, if anything goes wrong (eg. password not correct) the server will not proceed and will send an appropriate response (eg. 401).

Sample FastAPI app

For this practical we've created a simple FastAPI app that implements the basic functionality outlined in the previous section.

[jwtfastapi.zip](https://canvas.qut.edu.au/courses/20367/files/6717519/download) (<https://canvas.qut.edu.au/courses/20367/files/6717519/download>)

- Download this file and unzip it.

To run the application you have a few choices:

- Run locally on your own computer (requires Python already installed)
- Run on an EC2 instance with Python installed
- Run using Docker (locally or on an EC2 instance)

If you are using an EC2 instance, use `scp` to copy the `jwtfastapi` directory to your EC2 instance as in previous tutorials. On your computer, open the terminal, change to the directory containing the `jwtfastapi` directory and run

```
scp -i <pathToYourKeyFile> -r jwtfastapi ubuntu@<yourEC2PublicDNSName>:/home/ubuntu
```

with `<pathToYourKeyFile>` and `<yourEC2PublicDNSName>` set appropriately.

We now assume that you have a terminal open (either on your local machine if running locally or SSH'ed in to your EC2 instance if running there) and with the `jwtfastapi` directory as the working directory.

To run directly using Python/FastAPI:

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
uvicorn app:app --host 0.0.0.0 --port 3000
```

To run with Docker:

```
docker build -t jwtfastapi .
docker run --rm -dp 3000:3000 jwtfastapi
```

Either way, the server should now be accessible on port 3000 of either your local machine (`localhost`) or on your EC2 instance (look up the public DNS name in the AWS console).

Testing the app

We will use [Hoppscotch](https://hoppscotch.com/download) (<https://hoppscotch.com/download>) for testing along with a [Hoppscotch request collection](https://canvas.qut.edu.au/courses/20367/files/6717520/download) (<https://canvas.qut.edu.au/courses/20367/files/6717520/download>). We'll assume that you are familiar with its usage from previous practicals.

Logging on

- Start up Hoppscotch and import (*Import from Hoppscotch*) the [Hoppscotch request collection](https://canvas.qut.edu.au/courses/20367/files/6717520/download) (<https://canvas.qut.edu.au/courses/20367/files/6717520/download>)
 - Mac users may find that the file doesn't import correctly. You can use the web browser version at [Hoppscotch.io](https://hoppscotch.io) (<https://hoppscotch.io>) instead, but note that this will not work if you are running the app locally; you must run it on an EC2 instance as the web version of Hoppscotch cannot access local addresses (eg. `127.0.0.1` or `localhost`)
- Create new environment with `baseUrl` set to `http://localhost:3000` or `http://<your EC2 instance public DNS name>:3000` as appropriate
- In the *JWT-FastAPI demo* collection, select and run the *Login* request
- In the request body, note that we have specified the username and password in JSON format.
- In the response you should see `authToken` followed by a long random-looking string. This string is the JWT.

Accessing privileged content

The server requires a valid JWT to access the `/` route.

- Find the *Authorised user content* request and run it. The response should be *Unauthorized*.
- Go back to the *Login* request and copy the JWT from the response
- In the *Authorised user content* request, find the *Authorization* tab. The *Authorization Type* should already be set to *Bearer*. Paste the JWT in to the *Token* field.
- Run the request again. You should see `This content for authorised users only.` in the response

Users with additional permissions

- Find the *Admin User Content* request and run it. The response should be *Unauthorized*.
- As in the previous section, copy the JWT into the *Token* field in the *Authorization* tab. You should now see *Forbidden*.
- Go back to the *Login* request. In the body, change the username and password both to be `admin`. Run the request.
- Copy the new JWT from the admin login request in *Token* field in the *Authorization* tab on the *Admin User Content* request.
- Run the request again. You should now see `Admin only content.` in the response.

Read the code

We won't provide a complete analysis of the code; we leave that to you. Here we'll highlight some important things to look for.

app.py

- We're using a hard-coded user database. This is fine for the first assessment. In the second assessment you'll use a cloud service for this task.
- The `login` route uses `POST` so that the username and password are not visible in the URL (as with a `GET`). We need to check the username and password to make sure that they match what is in the database.
- For the `/` and `/admin` routes we use a middleware function `authenticate_token` to handle the JWT.
- The middleware function grabs the JWT from the header and verifies it. If verification fails then we immediately send a 401. Later handlers can assume that verification succeeded since they are not called if it fails. Also, we attach a `user` object to the request so that later handlers can make use of that information.
- For the `/admin` route we check that the user has admin privileges, but you could do other things such as provide a response that is customised to the particular user (eg. show the list of files that this user owns)
- We're using a hard-coded secret for signing the JWTs. This is OK for testing, but not OK for production. We'll explore secrets management later in the unit.

TEQSA PRV12079 | CRICOS 00213J | ABN 83 791 724 622