# OAuth2 with Password (and hashing), Bearer with JWT tokens

Now that we have all the security flow, let's make the application actually secure, using JWT tokens and secure password hashing.

This code is something you can actually use in your application, save the password hashes in your database, etc.

We are going to start from where we left in the previous chapter and increment it.

## About JWT

JWT means "JSON Web Tokens".

It's a standard to codify a JSON object in a long dense string without spaces. It looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaW
```

It is not encrypted, so, anyone could recover the information from the contents.

But it's signed. So, when you receive a token that you emitted, you can verify that you actually emitted it.

That way, you can create a token with an expiration of, let's say, 1 week. And then when the user comes back the next day with the token, you know that user is still logged in to your system.

After a week, the token will be expired and the user will not be authorized and will have to sign in again to get a new token. And if the user (or a third party) tried to modify the token to change the expiration, you would be able to discover it, because the signatures would not match.

If you want to play with JWT tokens and see how they work, check https://jwt.io.

## Install `python-jose`

We need to install `python-jose` to generate and verify the JWT tokens in Python:

```
$ pip install "python-jose[cryptography]"

---> 100%
```

Python-jose requires a cryptographic backend as an extra.

Here we are using the recommended one: pyca/cryptography.

> **Tip**
>
> This tutorial previously used PyJWT.
>
> But it was updated to use Python-jose instead as it provides all the features from PyJWT plus some extras that you might need later when building integrations with other tools.

## Password hashing

"Hashing" means converting some content (a password in this case) into a sequence of bytes (just a string) that looks like gibberish.

Whenever you pass exactly the same content (exactly the same password) you get exactly the same gibberish.

But you cannot convert from the gibberish back to the password.

### Why use password hashing

If your database is stolen, the thief won't have your users' plaintext passwords, only the hashes.

So, the thief won't be able to try to use that password in another system (as many users use the same password everywhere, this would be dangerous).

## Install `passlib`

PassLib is a great Python package to handle password hashes.

It supports many secure hashing algorithms and utilities to work with them.

The recommended algorithm is "Bcrypt".

So, install PassLib with Bcrypt:

```
$ pip install "passlib[bcrypt]"

---> 100%
```

> **Tip**
>
> With `passlib`, you could even configure it to be able to read passwords created by **Django**, a **Flask** security plug-in or many others.
>
> So, you would be able to, for example, share the same data from a Django application in a database with a FastAPI application. Or gradually migrate a Django application using the same database.
>
> And your users would be able to login from your Django app or from your **FastAPI** app, at the same time.

## Hash and verify the passwords

Import the tools we need from `passlib`.

Create a PassLib "context". This is what will be used to hash and verify passwords.

> **Tip**
>
> The PassLib context also has functionality to use different hashing algorithms, including deprecated old ones only to allow verifying them, etc.
>
> For example, you could use it to read and verify passwords generated by another system (like Django) but hash any new passwords with a different algorithm like Bcrypt.
>
> And be compatible with all of them at the same time.

Create a utility function to hash a password coming from the user.

And another utility to verify if a received password matches the hash stored.

And another one to authenticate and return a user.

**Python 3.6 and above**

```
{!> ../../../docs_src/security/tutorial004.py!}
```

**Python 3.10 and above**

```
{!> ../../../docs_src/security/tutorial004_py310.py!}
```

> **Note**
>
> If you check the new (fake) database `fake_users_db`, you will see how the hashed password looks like now: `"$2b$12$EixZaYVK1fsbw1ZfbX3OXePaWxn96p36WQoeG6Lruj3vjPGga31lW"`.

## Handle JWT tokens

Import the modules installed.

Create a random secret key that will be used to sign the JWT tokens.

To generate a secure random secret key use the command:

```
$ openssl rand -hex 32

09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7
```

And copy the output to the variable `SECRET_KEY` (don't use the one in the example).

Create a variable `ALGORITHM` with the algorithm used to sign the JWT token and set it to `"HS256"`.

Create a variable for the expiration of the token.

Define a Pydantic Model that will be used in the token endpoint for the response.

Create a utility function to generate a new access token.

**Python 3.6 and above**

```
{!> ../../../docs_src/security/tutorial004.py!}
```

**Python 3.10 and above**

```
{!> ../../../docs_src/security/tutorial004_py310.py!}
```

## Update the dependencies

Update `get_current_user` to receive the same token as before, but this time, using JWT tokens.

Decode the received token, verify it, and return the current user.

If the token is invalid, return an HTTP error right away.

**Python 3.6 and above**

```
{!> ../../../docs_src/security/tutorial004.py!}
```

**Python 3.10 and above**

```
{!> ../../../docs_src/security/tutorial004_py310.py!}
```

## Update the `/token` *path operation*

Create a `timedelta` with the expiration time of the token.

Create a real JWT access token and return it.

**Python 3.6 and above**

```
{!> ../../../docs_src/security/tutorial004.py!}
```

**Python 3.10 and above**

```
{!> ../../../docs_src/security/tutorial004_py310.py!}
```

### Technical details about the JWT "subject" `sub`

The JWT specification says that there's a key `sub`, with the subject of the token.

It's optional to use it, but that's where you would put the user's identification, so we are using it here.

JWT might be used for other things apart from identifying a user and allowing them to perform operations directly on your API.

For example, you could identify a "car" or a "blog post".

Then you could add permissions about that entity, like "drive" (for the car) or "edit" (for the blog).

And then, you could give that JWT token to a user (or bot), and they could use it to perform those actions (drive the car, or edit the blog post) without even needing to have an account, just with the JWT token your API generated for that.

Using these ideas, JWT can be used for way more sophisticated scenarios.

In those cases, several of those entities could have the same ID, let's say `foo` (a user `foo`, a car `foo`, and a blog post `foo`).

So, to avoid ID collisions, when creating the JWT token for the user, you could prefix the value of the `sub` key, e.g. with `username:`. So, in this example, the value of `sub` could have been: `username:johndoe`.

The important thing to have in mind is that the `sub` key should have a unique identifier across the entire application, and it should be a string.

## Check it

Run the server and go to the docs: http://127.0.0.1:8000/docs.

You'll see the user interface like:

Authorize the application the same way as before.

Using the credentials:

Username: `johndoe` Password: `secret`

> **Check**
>
> Notice that nowhere in the code is the plaintext password "`secret`", we only have the hashed version.

Call the endpoint `/users/me/`, you will get the response as:

```
{
  "username": "johndoe",
  "email": "johndoe@example.com",
  "full_name": "John Doe",
  "disabled": false
}
```

If you open the developer tools, you could see how the data sent and only includes the token, the password is only sent in the first request to authenticate the user and get that access token, but not afterwards:

> **Note**
>
> Notice the header `Authorization`, with a value that starts with `Bearer`.

## Advanced usage with `scopes`

OAuth2 has the notion of "scopes".

You can use them to add a specific set of permissions to a JWT token.

Then you can give this token to a user directly or a third party, to interact with your API with a set of restrictions.

You can learn how to use them and how they are integrated into **FastAPI** later in the **Advanced User Guide**.

## Recap

With what you have seen up to now, you can set up a secure **FastAPI** application using standards like OAuth2 and JWT.

In almost any framework handling the security becomes a rather complex subject quite quickly.

Many packages that simplify it a lot have to make many compromises with the data model, database, and available features. And some of these packages that simplify things too much actually have security flaws underneath.

---

**FastAPI** doesn't make any compromise with any database, data model or tool.

It gives you all the flexibility to choose the ones that fit your project the best.

And you can use directly many well maintained and widely used packages like `passlib` and `python-jose`, because **FastAPI** doesn't require any complex mechanisms to integrate external packages.

But it provides you the tools to simplify the process as much as possible without compromising flexibility, robustness, or security.

And you can use and implement secure, standard protocols, like OAuth2 in a relatively simple way.

You can learn more in the **Advanced User Guide** about how to use OAuth2 "scopes", for a more fine-grained permission system, following these same standards. OAuth2 with scopes is the mechanism used by many big authentication providers, like Facebook, Google, GitHub, Microsoft, Twitter, etc. to authorize third party applications to interact with their APIs on behalf of their users.

---

Last update: September 11, 2022

Created: September 11, 2022