Learning Objectives

- **Explain what the JWT library does and why it is beneficial**
- **Differentiate the access and response tokens from a JWT response**
- **Identify some of the common settings for JWT**
- **Add JWT authentication to Blango**
- **Verify JWT works as expected with Postman**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

JWT Introduction

JWT Introduction

JWT stands for JSON Web Token. It is a way of encoding authorization information into JSON structure.

JWTs consist of three parts: header, payload and signature. The header consists of a `type` (usually `JWT`) and the algorithm (`alg`) that was used to generate the signature. For example:

```
{
  "alg": "HS256",
  "type": "JWT"
}
```

The payload can consist of any data you want. At a minimum you probably want some way of finding out which user the token is for, for example, their user ID. Sometimes the expiry time of the token (`exp`, in seconds since the Unix epoch) is also included, along with a `token_type`. A payload might look like this:

```
{
  "user_pk": 1,
  "token_type": "access",
  "exp": 1630462029
}
```

Then finally, there's the signature. It's generated using a hash function as specified in the header, in our case a function called *HMACSHA256*.

We saw the SHA256 hash function in Course 2, Module 2. The HMAC256 function is a little bit different. Instead of just hashing a value, it also takes into account a secret. This means that even if you know the value being hashed, and the hash function, you won't be able to generate the same output hash. We'll return to this concept in a moment. For now, back to the JWT signature.

For a JWT, the input value to the HMAC256 is the base-64 encoded header, and the base-64 encoded payload, joined with a `.`. The secret that's used is kept private.

Now that we have the three parts of the JWT, they're joined together with a `.` between each component, to build the full token.

In our case, the base-64 encoded header is `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9`. The base-64 encoded payload is `eyJ1c2VyX3BrIjoxLCJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjMwNDYyMDI5fQ` and the signature is (generated using the secret `secret`) is `r8-OzTcodLgFSk1iEjFQc_i9aqyd6l84qNfhcw78bd8`. Therefore, our whole JWT is:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX3BrIjoxLCJ0b2tlbl9
0eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjMwNDYyMDI5fQ.r8-
OzTcodLgFSk1iEjFQc_i9aqyd6l84qNfhcw78bd8
```

To authenticate using JWT, the client sends the token in the `Authorization` HTTP header, usually with a `Bearer` prefix:

```
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX3BrIjoxLCJ0b2tlbl9
0eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjMwNDYyMDI5fQ.r8-
OzTcodLgFSk1iEjFQc_i9aqyd6l84qNfhcw78bd8
```

The server can then decode the token, verify the signature and log in the user based on the information in the token (`user_pk`, in our case).

Notice that the JWT doesn't contain a password, so how can we safely authenticate the user just based on the payload? What's to stop the client editing the payload, and changing the `user_pk` to another value?

If they did that, they wouldn't be able to generate the signature for it, since they don't know the secret parameter for the HMAC256 function. If they were to try to authenticate with the new value, the server would reject the token due to it not having the right signature.

This gives JWTs some advantages over other token types. JWTs can be moved around between systems and validated even without talking to the system that generated the JWT originally.

For example, let's say we had a sister site to Blango (a fruit-specific blog called Mango). We want users to be able to log in to Blango, receive a token, then pass that token to Mango and be authenticated without having to log in there again.

Using our original token system that we previously set up in DRF, the user would receive a token, but when that was passed to Mango, it would have to query the Blango database to check the validity of the token. This means more management overhead to set up this link.

Compare that to a JWT. Blango could generate a JWT, which would then be passed to Mango by the client. Provided we'd previously shared the secret between the sites, Mango would be able to verify the JWT just using the signature, without having to have any link to Blango.

Another cool feature of JWTs is the ability to include arbitrary data in the payload. Perhaps we want only some of the users to be able to post blog entries on Mango, and the rest are just read-only. Blango can add a field like `"can_post": true` to the JWT payload, and Mango can read this to check if the user has author permissions. Remember that the client can't just change the value from `false` to `true` as it would cause the signature to be invalid.

It's important to point out that even though JWTs are tamper-resistant, they should still be kept private by the client. If our JWTs are leaked then another client could authenticate as us. Using the expiry (`exp` in the payload) means that even if leaked, the time they can be used for is limited.

Now that you know the fundamentals of JWT, let's see how it can be integrated into DRF.

Simple JWT Setup

SimpleJWT Setup

The SimpleJWT library provides a simple way of integrating JWT into your Django Rest Framework application.

Once installed with `pip`, all you need to do is add `rest_framework_simplejwt.authentication.JWTAuthentication` to the list of `DEFAULT_AUTHENTICATION_CLASSES` in the `REST_FRAMEWORK` settings. Then, set up URL patterns to point to two provided views:

- `rest_framework_simplejwt.views.TokenObtainPairView`: This view accepts user credentials (e.g. username or email address, and password), then returns an access token and refresh token (more about these token types in a minute).
- `rest_framework_simplejwt.views.TokenRefreshView`: This view accepts a refresh token and returns a new access token (and sometimes another refresh token, depending on settings).

Once you have an access token, you can use it in the `Authorization` HTTP header similarly to how we've used standard DRF tokens before, except it should have the `Bearer` prefix.

Access and Refresh Tokens

As we mentioned, the `TokenObtainPairView` returns two tokens, an access token and a refresh token. For example, here's the JSON that's sent to the view in a POST request:

```
{
    "email": "ben@example.com",
    "password": "password"
}
```

And here's the response:

```
{
    "refresh":
        "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTYzMDU1NDA5Myw
        ELopei8",

    "access":
        "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjMwNDY3OTkzLCJ
        wqzqAB7wrjWE3ivAPHahJgppYhw"

}
```

For brevity, we'll truncate these tokens to `eyJ0e...refresh...opei8` and `eyJ0e...access...ppYhw`.

When we want to authenticate to the API, we use the `access` token, so our requests would have the `Authorization` HTTP header like this:

```
Authorization: Bearer eyJ0e...access...ppYhw
```

By default, SimpleJWT generates access tokens which are only valid for five minutes. We can use the refresh token, which is valid for one day, to get a new access token. This is done by sending the refresh token with a POST request to the `TokenRefreshView`, with a body like this:

```
{
    "refresh": "eyJ0E...refresh...opei8"
}
```

Which returns a response containing a new access token:

```
{
    "access":
        "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjMwNDg0MDk0LCJ
        Uync503zJIN8sDo0n-dT2jFr_0"


}
```

For a client to get a new refresh token, they must authenticate to the `JWTAuthentication` with their credentials again.

Now let's look at some options for customizing SimpleJWT.

SimpleJWT Settings

The full list of SimpleJWT settings is available at the official documentation, but here are some of the ones you'll be most likely to customize.

They are all contained in a dictionary called `SIMPLE_JWT` in `settings.py` – similar to the `REST_FRAMEWORK` settings dictionary.

- `ACCESS_TOKEN_LIFETIME`: This is how long the access token is valid for. Must be set as a `datetime.timedelta` instance. Defaults to `timedelta(minutes=5)`.
- `REFRESH_TOKEN_LIFETIME`: How long the refresh token is valid for, must also be set as a `timedelta` instance. Defaults to `timedelta(days=1)`.
- `ROTATE_REFRESH_TOKENS`: Normally when a token is refreshed the refresh token is not refreshed (i.e. the same refresh token should be used again, and the validity of it is unchanged). If this setting is set to `True`, then a new refresh token will also be issued with new expiration date, when the token is refreshed.
- `BLACKLIST_AFTER_ROTATION`: If this is set to `True`, then a refresh token is blacklisted after it is used (i.e. it is only usable once). This requires that `ROTATE_REFRESH_TOKENS` is also `True`, and that the `rest_framework_simplejwt.token_blacklist` is added to `INSTALLED_APPS`, since the token blacklist is maintained in the database.
- `SIGNING_KEY`: The secret key used to sign in the hash function (HMACSHA256 by default). This is set to the Django `SECRET_KEY` setting by default but it's recommended to change it to something else so that these are independent.
- `LEEWAY`: The amount of leeway to allow in expiry checking, this can be an integer number of seconds or a `timedelta` instance. Defaults to 0.
- `AUTH_HEADER_TYPES`: The token prefix(es) that can be used in the `Authorization` header. Defaults to the tuple `("Bearer",)`. It's common to be more specific about your type of token by using something like `("JWT",)` which would allow the prefix `JWT` before the token.

You won't always need to change all of these for each project. SimpleJWT is conservative with its security so usually you'll be fine bumping up the `ACCESS_TOKEN_LIFETIME`, but changing the `SIGNING_KEY` is a good idea too.

It's also worthwhile to consider some of the other signing methods if you need more security. Since this is quite an in-depth discussion on its own we refer back to the official documentation as a starting point.

Now, let's set up JWT authentication in Blango.

Try It Out

Try It Out

As is usual with any third party app, we'll need to install it with `pip`. The package name is `djangorestframework-simplejwt`:

```
pip3 install djangorestframework-simplejwt PyJWT==1.7.1
```

Once installed, we can start by adding `rest_framework_simplejwt.authentication.JWTAuthentication` to the `REST_FRAMEWORK`'s `DEFAULT_AUTHENTICATION_CLASSES` list in `settings.py`. It will end up looking something like this:

Open settings.py

```
    REST_FRAMEWORK = {
        "DEFAULT_AUTHENTICATION_CLASSES": [
            "rest_framework.authentication.BasicAuthentication",

        "rest_framework.authentication.SessionAuthentication",
            "rest_framework.authentication.TokenAuthentication",

        "rest_framework_simplejwt.authentication.JWTAuthenticati
        on"
        ],
        # existing REST_FRAMEWORK settings omitted
    }
```

Next we'll need to add URL patterns to the `TokenObtainPairView` and `TokenRefreshView` views. We'll make these URLs `/api/v1/jwt/` and `/api/v1/jwt/refresh/`, respectively.

Open `blog/api/urls.py`, and add an import for the views:

Open api/urls.py

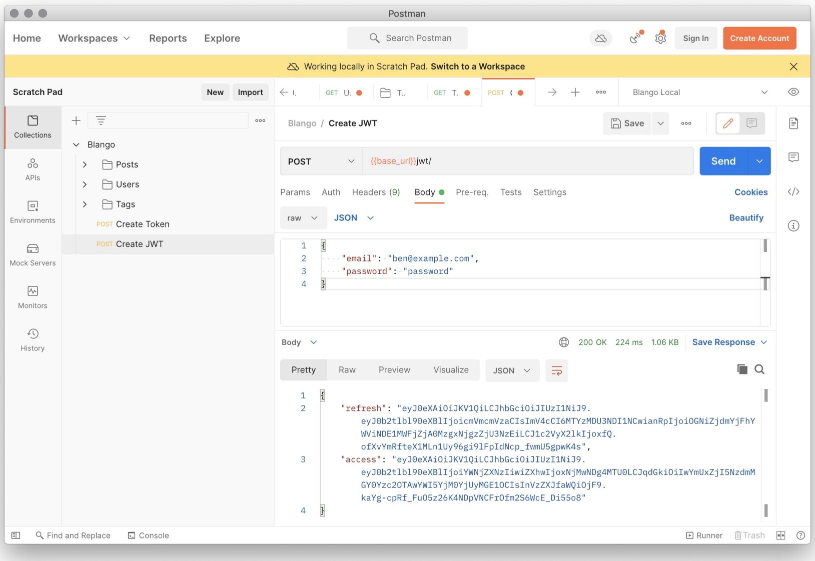
```
from rest_framework_simplejwt.views import TokenObtainPairView,
TokenRefreshView
```

Then add these URL patterns, after the existing `token-auth/` route:

```
    path("jwt/", TokenObtainPairView.as_view(),
        name="jwt_obtain_pair"),
    path("jwt/refresh/", TokenRefreshView.as_view(),
        name="jwt_refresh"),
```

That's all that's required. To try this out, it's best to use Postman. Start by creating a new request under the *Blango* collection in the *Tags* folder called `Create JWT`. It should be a `POST` request with URL `{{base_url}}jwt/`. The request body should be a JSON dictionary with "email" and "password" entries for the user.

After making the request, you should receive back your `access` and `refresh` tokens.



POST `{{base_url}}jwt/`

```
{
    "email": "ben@example.com",
    "password": "password"
}
```

200 OK 224 ms 1.06 KB

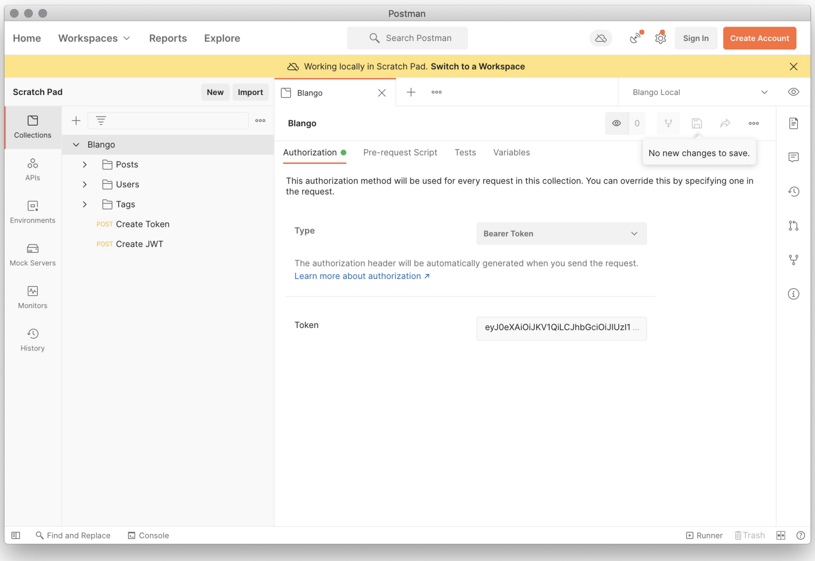
```
{
    "refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
        eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTYzMDU3NDI1NCwianRpIjoiOGNiZjdmYjFhY
        WViNDE1MWFjZjA0MzgxNjgzZjU3NzEiLCJ1c2VyX2lkIjoxfQ.
        ofXvYmRfteX1MLn1Uy96gi9lFpIdNcp_fwmU5gpwK4s",
    "access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
        eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjMwNDg4MTU0LCJqdGkiOiIwYmUxZjI5NzdmM
        GY0Yzc2OTAwYWI5YjM0YjUyMGE1OCIsInVzZXJfaWQiOjF9.
        kaYg-cpRf_Fu05z26K4NDpVNCFrOfm2S6WcE_Di55o8"
}
```

Click here to see a larger version of the image

▼ **jwt.io**

jwt.io is an online tool that lets you decode, verify and generate JWTs. If you're curious about the content of the access and refresh tokens you can paste them into this site and have them decoded. Note that even though the site doesn't know the secret it can still decode them, but without a matching secret, can't make changes to them.

You can now set Postman to use your access token, by going into the **Authorization** tab for the *Blango* collection. Select type *Bearer*, then enter the access token into the **Token** field.



This authorization method will be used for every request in this collection. You can override this by specifying one in the request.

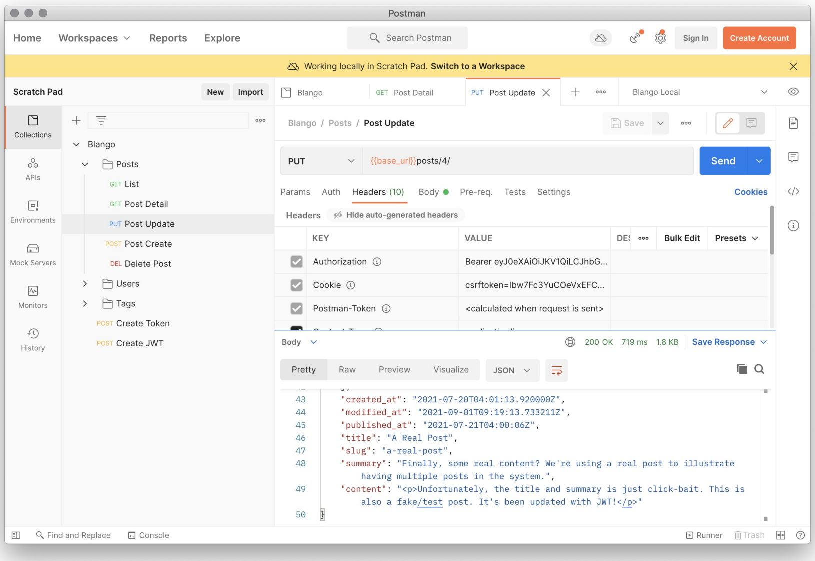
Type: Bearer Token

The authorization header will be automatically generated when you send the request.
Learn more about authorization ↗

Token: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1...

[Click here to see a larger version of the image]

Finally, make a request that requires authentication, like updating or creating a `Post`. Check the `Headers` tab for the request, you should see `Bearer <token>` listed there, and you shouldn't have any trouble making the request.



PUT {{base_url}}posts/4/

KEY	VALUE
Authorization	Bearer eyJ0eXAiOiJKV1QiLCJhbG...
Cookie	csrftoken=lbw7Fc3YuCOeVxEFC...
Postman-Token	<calculated when request is sent>

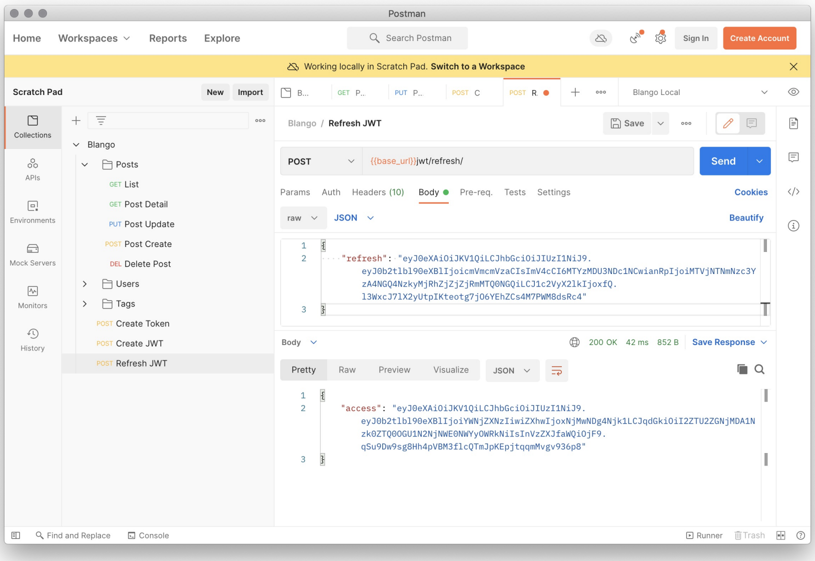
200 OK 719 ms 1.8 KB

```
    "created_at": "2021-07-20T04:01:13.920000Z",
    "modified_at": "2021-09-01T09:19:13.733211Z",
    "published_at": "2021-07-21T04:00:06Z",
    "title": "A Real Post",
    "slug": "a-real-post",
    "summary": "Finally, some real content? We're using a real post to illustrate
        having multiple posts in the system.",
    "content": "<p>Unfortunately, the title and summary is just click-bait. This is
        also a fake/test post. It's been updated with JWT!</p>"
}
```

[Click here to see a larger version of the image]

Finally, try refreshing your token. Create a new `POST` request with the URL `{{base_url}}jwt/refresh/`. The body should be a JSON dictionary with a key `refresh` containing the refresh token you received.

After sending the request, you'll receive back an updated access token in a JSON dictionary.



POST {{base_url}}jwt/refresh/

```
{
    "refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
        eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTYzMDU3NDc1NCwianRpIjoiMTVjNTNmNzc3Y
        zA4NGQ4NzkyMjRhZjZjZjRmMTQ0NGQiLCJ1c2VyX2lkIjoxfQ.
        l3WxcJ7lX2yUtpIKteotg7jO6YEhZCs4M7PWM8dsRc4"
}
```

200 OK 42 ms 852 B

```
{
    "access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
        eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjMwNDg4Njk1LCJqdGkiOiI2ZTU2ZGNjMDA1N
        zk0ZTQ0OGU1N2NjNWE0NWYyOWRkNiIsInVzZXJfaWQiOjF9.
        qSu9Dw9sg8Hh4pVBM3flcQTmJpKEpjtqqmMvgv936p8"
}
```

Click here to see a larger version of the image

Before moving on, let's make a few settings changes to make the JWT a bit nicer to work with. Since we're not very concerned about the security of our application while we develop it, we'll make the JWTs last a bit longer – one day. We'll also increase the time the refresh token is valid for to 7 days.

Open `settings.py` again. We need to start by importing the `timedelta` class at the start of the file.

Open settings.py

```
from datetime import timedelta
```

Then add this `SIMPLE_JWT` setting to the `Dev` class:

```
    SIMPLE_JWT = {
        "ACCESS_TOKEN_LIFETIME": timedelta(days=1),
        "REFRESH_TOKEN_LIFETIME": timedelta(days=7),
    }
```

Try fetching a new token with Postman, then use jwt.io to decode it. The expiry time (`exp`) is the number of seconds since Jan 1, 1970 (the Unix epoch). You can convert this to a human readable format at the Epoch Converter site. Some browsers will convert this for you if you put your mouse over the number of seconds. You should see that the access token is now valid for one day and the refresh token for seven days.

You now should have a good understanding of JSON Web Tokens, and how to use them with Django Rest Framework. In the next section we're going to finish up our look at third-party libraries with *Django Versatile Image Field*.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish simplejwt"
```

- Push to GitHub:

```
git push
```