


JWT Authentication

Theory

Practice

 0% completed, 0 problems solved

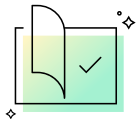
▼

Theory

🕒 25 minutes reading

Verify to skip

Start practicing



You're starting a new topic!

Each topic introduces one new concept with a short article. Read it, then apply what you've learned by solving a few problems. Practice makes perfect!

You are already familiar with basic authentication and form authentication as well as their use in Ktor. Today we are going to study another important type of authentication, **JSON Web Tokens**, JWT for short.

§1. What are JSON Web Tokens

JWT is a mechanism for verifying the owner of some JSON data. It is an encoded, URL-safe string. Unlike cookies, it can contain an unlimited amount of data. It has a cryptographic signature. When a server receives a JWT, it can ensure that the data it contains can be trusted because it is signed by the source. No middleman can change a JWT once it has been sent.

The signature allows you to determine if the data has been modified by third parties, but it does not encrypt the data. Anyone can read the data stored in a JWT token, but only the owner of the secret key can change it. Otherwise, the signature will no longer be correct and the token will be considered invalid.

JWT is more often used as an authentication method where access to protected resources is granted using a token, a special string that the user receives when they enter the correct credentials. Unlike the token in the session mechanism, a JWT token is not a randomly generated string. It encodes certain information about the user, which the server can easily retrieve directly from the token without having to query the database. Since the token is signed, the information about the user stored in the token cannot be faked.

Let's look at the structure of a typical JWT token.

§2. JWT token structure

For example, take a token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bW11IjoIbWlrZSJ9.PHhH0mDSTTHbwsOGearWBWFKqPuoAumgn2h8ANl2B1I
```

Each JWT token consists of three parts, separated by dots: `xxxxx.yyyyy.zzzzz`

In our case, the token decomposes into three parts.

The first part is the **header**: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9`

The second part is the **payload**, which contains information about the user:

```
eyJ1bW11IjoIbWlrZSJ9
```

The third part is the **signature**: `PHhH0mDSTTHbwsOGearWBWFKqPuoAumgn2h8ANl2B1I`

Let's look at the purpose of each of them.

§3. JWT header

The first part is the header. The header usually contains the type of token and the algorithm used to create the signature. In our case, this will be information about the JWT token. For compactness, the header is encoded with Base64URL before being sent over the network. We have to decode it to see its structure. The easiest way to do this is to use an online Base64URL [encoder/decoder](#). So our header,

`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9` looks like this when decoded:

```
1  {"alg": "HS256", "typ": "JWT"}
```

It's just JSON containing the `alg` and `typ` fields. The `typ` field tells us that the token is a JWT token. The `alg` field says that the `HS256` algorithm was used to create the token signature. This is the algorithm most often used.

§4. JWT payload

The second part of the token is the payload: `eyJ0eXB1IjoiaWlrZSJ9`. It is also encoded in Base64URL. Let's decode it:

```
1  {"name": "Mike"}
```

Payload is a plain JSON object containing information about the authenticated user. Web services usually decide for themselves what they want to store in the payload. In our case, it only contains the name of the authorized user. This is very convenient because we can get the name of the authorized user right from the token itself. We won't have to query the database.

The payload can also contain additional fields, such as `iss`, `sub`, `aud`, `exp`.

The `iss`, issuer field is the identifier of the server that issued the token. For example, the URL of the domain where the server is hosted.

The `sub`, subject field contains the identifier of the client to whom the token was issued. For example, the client's email address.

The `aud`, audience field contains the identifier of the recipient of the token. For example, the URL of the protected pages which require the token to access.

Finally, `exp`, expiration time, contains a timestamp of when the token expires and is considered invalid.

So our payload could look like this:

```
1  {  
2    "name": "Mike",  
3    "iss": "http://example.com/",  
4    "sub": "mike@email.com",  
5    "aud": "http://example.com/protected",  
6    "exp": "1636027948"  
7  }
```

§5. JWT signature

The third and the most important part of the token is the signature. It allows the server to verify that the payload has not been spoofed. It is created as follows:

```
1  HMACSHA256(headerInBase64URL + "." + payloadInBase64URL, secret)
```

We take our header and payload encoded in Base64URL and concatenate them into one string with a dot between them. This is the JWT token with a discarded signature. Then we hash the resulting string using the HMACSHA256 algorithm, passing our secret key to it. The resulting signature is concatenated with the header and payload by adding a dot between them and we finally get a JWT token:

```
1 headerInBase64URL + "." + payloadInBase64URL + "." +
  HMACSHA256(headerInBase64URL + "." + payloadInBase64URL, secret)
```

In our case:

```
1 HMACSHA256("eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9" + "." +
  "eyJyZW11IjoiTWlrZSJ9", "secret")
```

We used the string `"secret"` as the secret key. In real projects, you have to think of a more sophisticated key, because if the hackers find it out, they can use it to give themselves tokens.

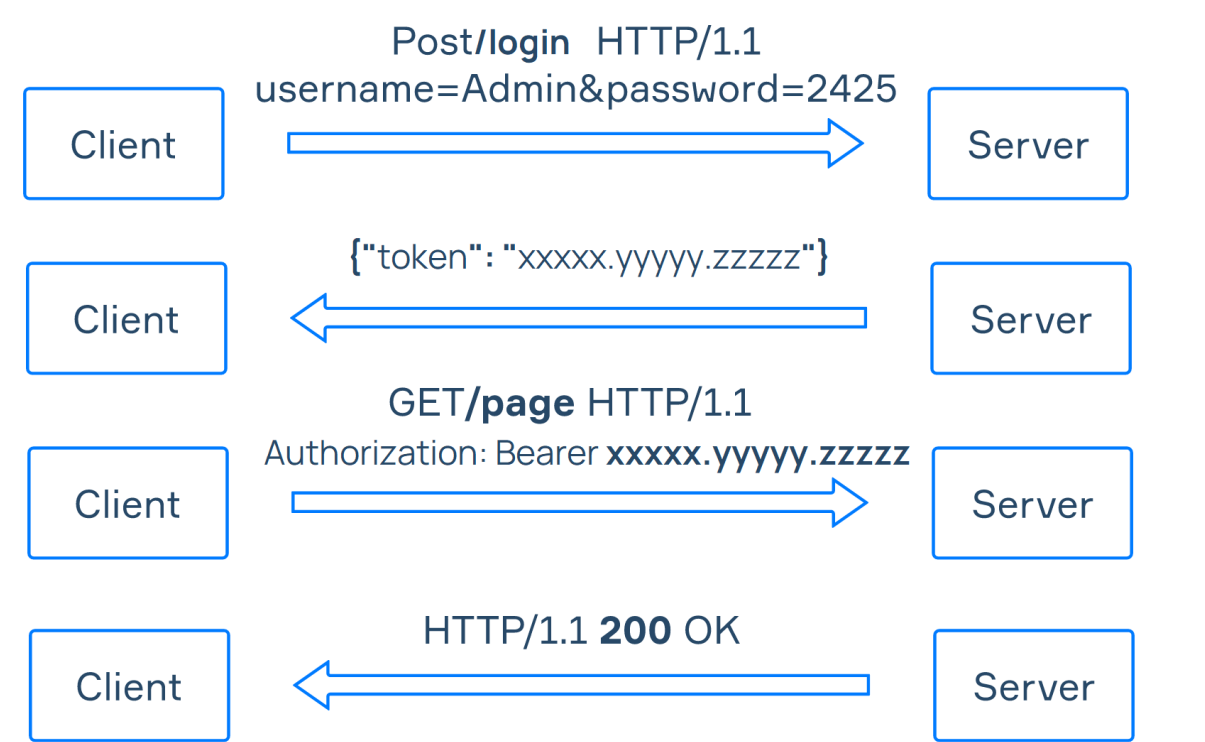
You can also use an [online service](#) to hash HMACSHA256. Give it the hash string `"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyZW11IjoiTWlrZSJ9"` and the key `"secret"` and set output text format, Base64URL. We will get a string which is our signature: `PHzH0mDSTTHbwsOGearWBWFKqPuoAumgn2h8ANl2B1I=` The "=" at the end of the signature is usually cut off.

Every time the server gets a JWT token, it checks the signature, doing all the same steps. It takes the header and the payload, the first two parts of the token, and the dot between them. Hashes them with a secret key and checks the received signature with the one that is in the token, which is the third part of the token. Thus, if hackers change the payload to `{"name": "Admin"}`, for example, the validation of the token will fail because the signatures for the different payloads will be different. To pass the validation, the hackers have to generate a new signature for their payload `{"name": "Admin"}`. But they can't do that because they don't know the secret key used to sign the tokens.

§6. JWT authorization flow in Ktor

Now let's look at the client-server communication scheme for JWT authentication in Ktor.

Suppose we have a page `localhost:8080/page`, protected by JWT authentication. Page `localhost:8080/login` accepts user authorization requests by username and password. The mechanism of interaction between the client and the server, in this case, will look like this:



- 1) First, the client makes a POST authorization request to `localhost:8080/login` and specifies its credentials in the request body.
- 2) The server checks the user's login and password and, if successful, generates a new JWT token with the user's information in the payload.
- 3) After receiving the token, the client can make a GET request to the protected `localhost:8080/page`, specifying the header `Authorization: Bearer <auth_token>`. It contains the token received in the previous step. The server sees that the requested page is protected by JWT authentication and retrieves the token from the `Authorization` header.
- 4) Next, the server checks the signature of the received token and, if successful, returns the requested page to the user.

We can implement this whole authentication mechanism ourselves, but fortunately, Ktor has a handy plugin that makes it easy to connect JWT authorization to your Routing handlers easily.

§7. JWT authentication plugin

Let's install the JWT authentication plugin.

First, you need to add dependencies to `build.gradle.kts`:

```
1 implementation("io.ktor:ktor-server-auth:$ktor_version")
2 implementation("io.ktor:ktor-server-auth-jwt:$ktor_version")
3 implementation("io.ktor:ktor-serialization-kotlinx-
json:$ktor_version")
```

The first two dependencies are responsible for the authentication plugin itself. The third dependency connects the `kotlinx.serialization` library. We need it to return JSON to the user.

Before setting up the plugin, let's create global variables that contain the main settings. For simplicity, we will write all the code in the `Application.kt` file.

```
1 const val secret = "secret"
2 const val issuer = "http://0.0.0.0:8080/"
3 const val audience = "http://0.0.0.0:8080/page"
4 const val myRealm = "Access to 'page'"
5
6 fun main() {
7     embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
8         ...
```

The `secret` is our key for creating a token signature.

The `issuer` is the URL of the domain that produced the token.

The `audience` is the unique identifier of the audience for an issued token. In our case, this is the URL of the protected page that will accept JWT tokens.

`myRealm` is the text message from the server that your application can show to the user.

In the process of configuring the plug-in, the IDE will ask you to add various imports. For your convenience, we'll list all of the required imports here:

```
1 import io.ktor.server.engine.*
2 import io.ktor.server.netty.*
3
4 import io.ktor.server.application.*
5 import io.ktor.server.auth.*
6 import io.ktor.server.auth.jwt.*
7 import com.auth0.jwt.JWT
```

```

8  import com.auth0.jwt.algorithms.Algorithm
9  import io.ktor.http.*
1
0  import io.ktor.server.request.*
1
1  import io.ktor.server.response.*
1
2  import io.ktor.server.routing.*
1
3  import kotlinx.serialization.*
1
4  import kotlinx.serialization.json.Json
1
5  import java.util.*

```

To set the JWT authentication provider, we call the `jwt` function inside the `install` block. The `jwt` function takes the authorization name `"myAuth"` as a parameter. We will need this name to refer to the authorization in the Routing handler.

```

1  embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
2      install(Authentication) {
3          jwt("myAuth") {
4              // Configure jwt authentication
5          }
6      }
7      ...

```

Inside the configuration we must define the `realm` field and the three functions: `verifier`, `validate` and `challenge`:

```

1  install(Authentication) {
2      jwt("myAuth") {
3          realm = myRealm
4          verifier(
5              JWT
6              .require(Algorithm.HMAC256(secret))
7              .withAudience(audience)
8              .withIssuer(issuer)
9              .build())
1
0          validate { credential ->
1
1              if (credential.payload.getClaim("username").asString()
!= "") {
1
2                  JWTPrincipal(credential.payload)
1
3              } else {
1
4                  null
1
5              }
1
6          }
1
7          challenge { defaultScheme, realm ->
1
8              call.respond(HttpStatusCode.Unauthorized, "Token is not
valid or has expired")
1
9          }
2
0      }
2
1  }

```

The `verifier` function checks the format of the token and its signature. We use the `secret` key, `audience`, and `issuer` to set it up.

The `validate` function allows us to perform additional checks on the token's payload and, if everything is good, returns a `JWTPrincipal` object containing a username. With this object, we can get in the Routing handler the name of the user who has logged in to our page. If the payload check fails, `validate` should return null. In this case, the authentication is considered unsuccessful.

The `challenge` function allows us to configure a response to be sent if authentication fails.

We have configured the authentication plugin. Let's make an API for authorization (`localhost:8080/login`). We have to make this page ourselves.

§8. Generating a token

Now we need to make a login page that will check the user's credentials and, if successful, generate a token for the user.

Create a `/login` handler and add the following code to it:

```
1 post("/login") {
2     val parameters = call.receiveParameters()
3     val username = parameters["username"].toString()
4     val password = parameters["password"].toString()
5
6     if (username == "Admin" && password == "2425") {
7         val token = JWT.create()
8             .withAudience(audience)
9             .withIssuer(issuer)
10
11             .withClaim("username", username)
12
13             .withExpiresAt(Date(System.currentTimeMillis() +
14 24*60*60000))
15
16             .sign(Algorithm.HMAC256(secret))
17
18         call.respondText(
19
20             Json.encodeToString(hashMapOf("token" to token)),
21
22             ContentType.Application.Json
23
24         )
25
26     } else {
27
28         call.response.status(HttpStatusCode.Unauthorized)
29
30         call.respondText(
31
32             Json.encodeToString(hashMapOf("error" to "Wrong login or
33 password")),
34
35             ContentType.Application.Json
36
37         )
38
39     }
40 }
```

First, we get the post request parameters, `username`, and `password` and check that they are correct. There can be any validation of the entered data, including database calls. To be simple, we compare them to `"Admin"` and `"2425"`.

After a successful login and password check, we generate a token and return it to the user. Token generation is done with `JWT.create()` and other additional methods: `withAudience`, `withIssuer`, `withClaim`, `withExpiresAt`, and `sign`.

`withAudience` and `withIssuer` add our `audience` and `issuer` to payload.

The `withClaim` method allows us to add arbitrary data to the payload object. We add the username field.

`withExpiresAt` allows you to specify the timestamp in milliseconds, after which the token becomes invalid, and the user has to authorize again. We take the current time and add the number of milliseconds contained in one day. So our token will live for one day.

The `sign` method sets the algorithm and the secret key to create the signature.

After the generation, we send the token to the user as JSON like

`{"token": "xxxxx.yyyyy.zzzzz"}`. We could use simple string concatenation to do this. It is proper to do it by converting the object to a string with `Json.encodeToString`. Because, we might want to send a more complex JSON object in the future. We also set `Content-type: application/json` so that the client understands that it's JSON and not just plain text.

If the user sends the wrong credentials, we'll return the error as JSON: `{"error": "Wrong login or password"}`. We also set the corresponding `401 Unauthorized` response code, so that the client understands that the authorization failed.

Great! The login API is ready. Now the easy part is to connect JWT authentication to our page.

§9. Connecting a JWT to a page

To protect the page, we need to wrap the appropriate handler in the `authenticate` function and pass the authentication name to it:

```
1  authenticate("myAuth") {
2      get("/page") {
3          call.respondText("Hi, authorized user!")
4      }
5  }
```

Also, inside the handler, we can get the name under which the user is logged in, as well as the expiration time of the token. To do this, we need to use the

`call.principal` function:

```
1  authenticate("myAuth") {
2      get("/page") {
3          val principal = call.principal<JWTPrincipal>()
4          val username =
principal!!.payload.getClaim("username").asString()
5          val expiresAt =
principal.expiresAt?.time?.minus(System.currentTimeMillis())
6          call.respondText("Hi, $username! Token is expired at
$expiresAt ms.")
7      }
8  }
```

Our JWT authentication is done. You can see the full code of the `Application.kt` by expanding the hint:

Tip:

```
1  package com.example
2
3  import io.ktor.server.engine.*
```

```
4 import io.ktor.server.netty.*
5 import io.ktor.server.application.*
6 import io.ktor.server.auth.*
7 import io.ktor.server.auth.jwt.*
8 import com.auth0.jwt.JWT
9 import com.auth0.jwt.algorithms.Algorithm
1
0 import io.ktor.http.*
1
1 import io.ktor.server.request.*
1
2 import io.ktor.server.response.*
1
3 import io.ktor.server.routing.*
1
4 import kotlinx.serialization.*
1
5 import kotlinx.serialization.json.Json
1
6 import java.util.*
1
7
1
8 const val secret = "secret"
1
9 const val issuer = "http://0.0.0.0:8080/"
2
0 const val audience = "http://0.0.0.0:8080/page"
2
1 const val myRealm = "Access to 'page'"
2
2
2
3 fun main() {
2
4     embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
2
5         install(Authentication) {
2
6             jwt("myAuth") {
2
7                 realm = myRealm
2
8                 verifier(
2
9                     JWT
3
0                     .require(Algorithm.HMAC256(secret))
3
1                     .withAudience(audience)
3
2                     .withIssuer(issuer)
3
3                     .build()
4
5                 )
3
6                 validate { credential ->
3
7                     if
8 (credential.payload.getClaim("username").asString() != "") {
3
9                         JWTPrincipal(credential.payload)
3
0
1                     } else {
3
2                         null
4
3                     }
4
5                 }
1
6             }
1
```



```

4
2         challenge { defaultScheme, realm ->
4
3             call.respond(HttpStatusCode.Unauthorized, "Token
is not valid or has expired")
4
4             }
4
5         }
4
6     }
4
7
4
8     routing {
4
9         post("/login") {
5
0             val parameters = call.receiveParameters()
5
1             val username = parameters["username"].toString()
5
2             val password = parameters["password"].toString()
5
3
5
4             if (username == "Admin" && password == "2425") {
5
5                 val token = JWT.create()
5
6                 .withAudience(audience)
5
7                 .withIssuer(issuer)
5
8                 .withClaim("username", username)
5
9
0                 .withExpiresAt(Date(System.currentTimeMillis() + 24 * 60 * 60000))
6
0                 .sign(Algorithm.HMAC256(secret))
6
1                 call.respondText(
6
2                     Json.encodeToString(hashMapOf("token" to
token)),
6
3                     ContentType.Application.Json
6
4                     )
6
5                 } else {
6
6
call.response.status(HttpStatusCode.Unauthorized)
6
7                 call.respondText(
6
8                     Json.encodeToString(hashMapOf("error" to
"Wrong login or password")),
6
9                     ContentType.Application.Json
7
0                     )
7
1                 }
7
2             }
7
3
7
4             authenticate("myAuth") {

```

Table of contents:

[↑ JWT Authentication](#)

[§1. What are JSON Web Tokens](#)

```
7
5      get("/page") {
7
6          val principal = call.principal<JWTPrincipal>()
7
7          val username =
principal!!.payload.getClaim("username").asString()
7
8          val expiresAt =
principal.expiresAt?.time?.minus(System.currentTimeMillis())
7
9          call.respondText("Hi, $username! Token is
expired at $expiresAt ms.")
8
0      }
8
1    }
8
2  }
8
3    }.start(wait = true)
8
4 }
```

[§2. JWT token structure](#)

[§3. JWT header](#)

[§4. JWT payload](#)

[§5. JWT signature](#)

[§6. JWT authorization flow in Ktor](#)

[§7. JWT authentication plugin](#)

[§8. Generating a token](#)

[§9. Connecting a JWT to a page](#)

[§10. Checking authentication with Postman](#)

[§11. Conclusion](#)

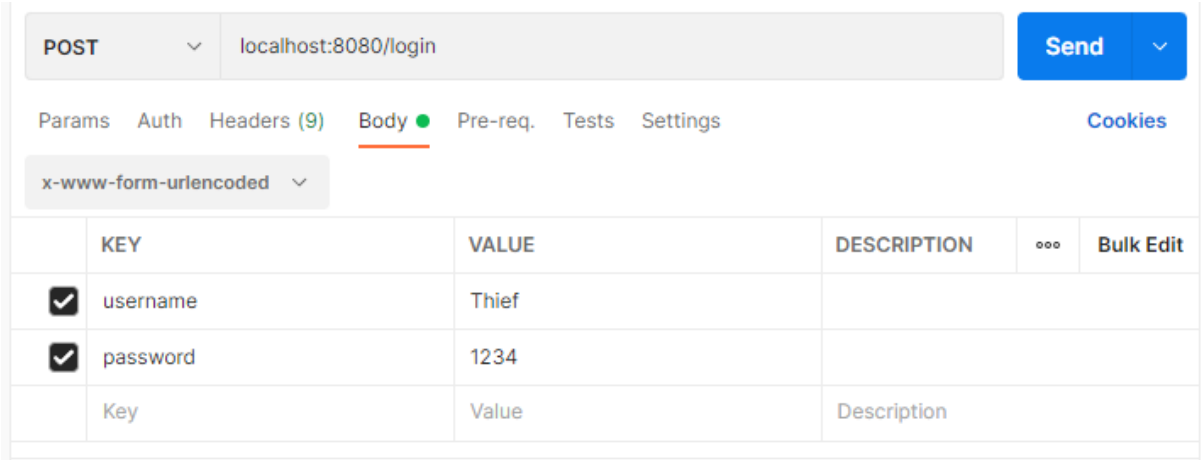
[Discussion](#)

§10. Checking authentication with Postman

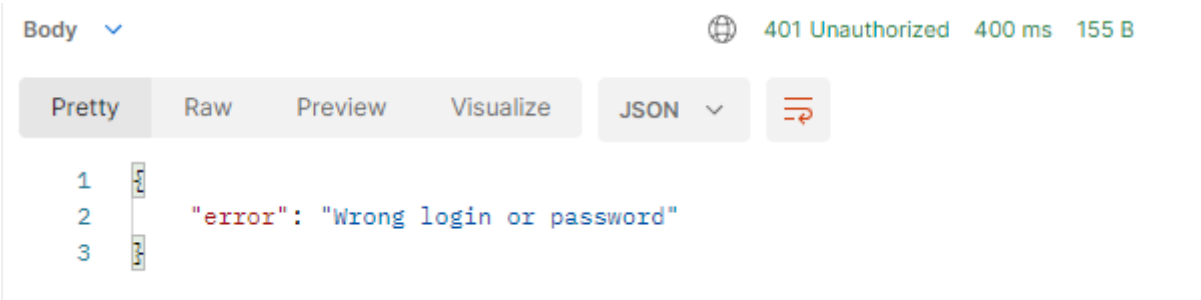
Unlike basic authentication, JWT authentication is not part of the HTTP protocol, so it is not supported by default browsers. To receive tokens, store them, and send them along with requests to secure pages, we need JavaScript. To use JWT authentication, we need to create a client to communicate with the server. In this topic we won't create a client, as the topic is not about JavaScript. To check if the authentication we made works, we will use Postman. It will send requests to the server and act as a client.

Let's run our server and open the Postman.

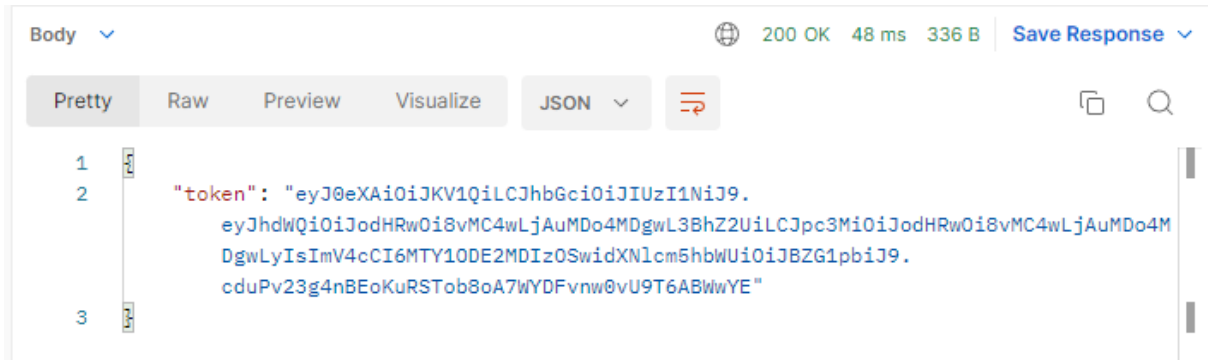
First, we have to make a POST request to `localhost:8080/login` to get the JWT token.



If you enter the wrong data, the server will return an error:

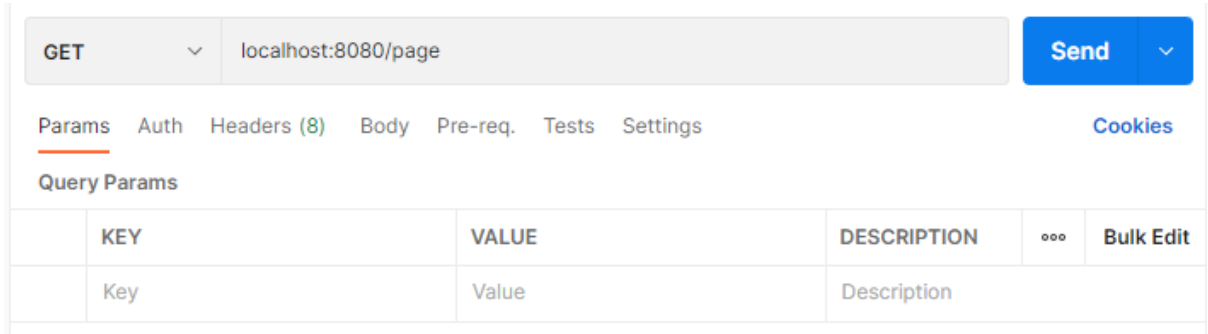


Now let's try to enter the correct login and password, `"Admin"` and `"2425"`. The server will see that the username and password are correct and give us a JWT token!

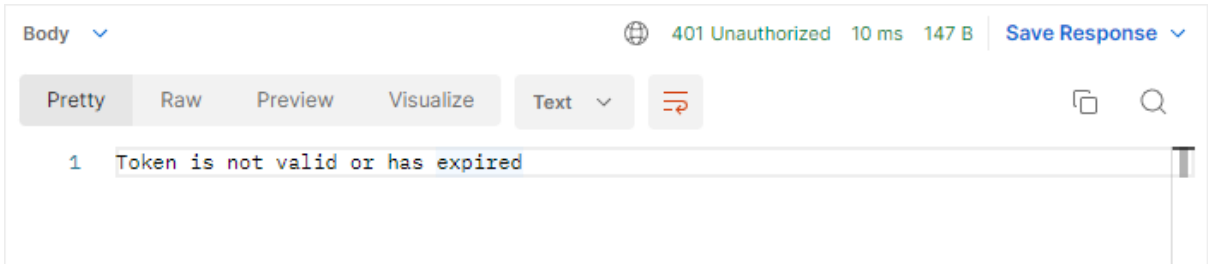


Let's copy it to the clipboard and try to access the protected page, `localhost:8080/page` with this token.

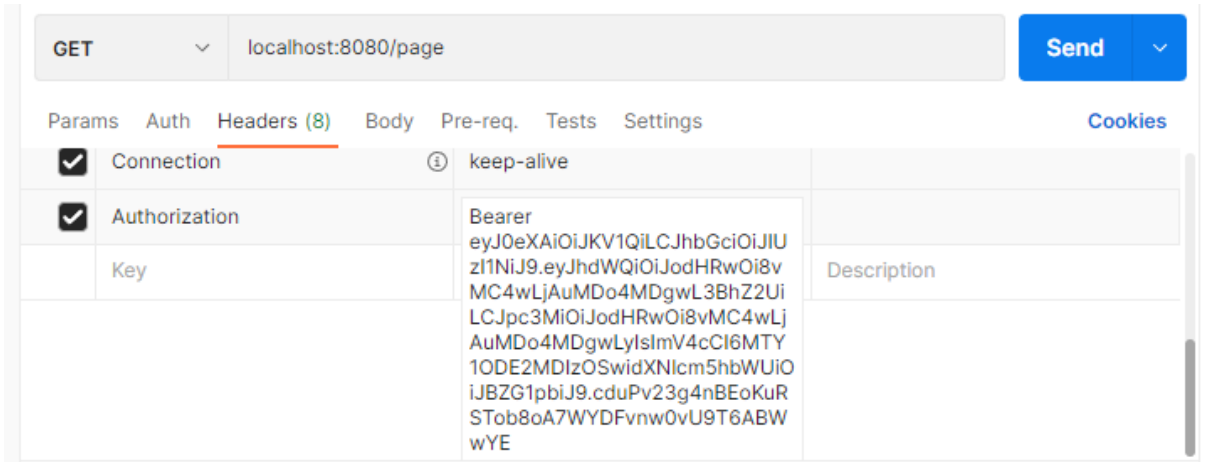
First, let's try to make a normal GET request without a token.



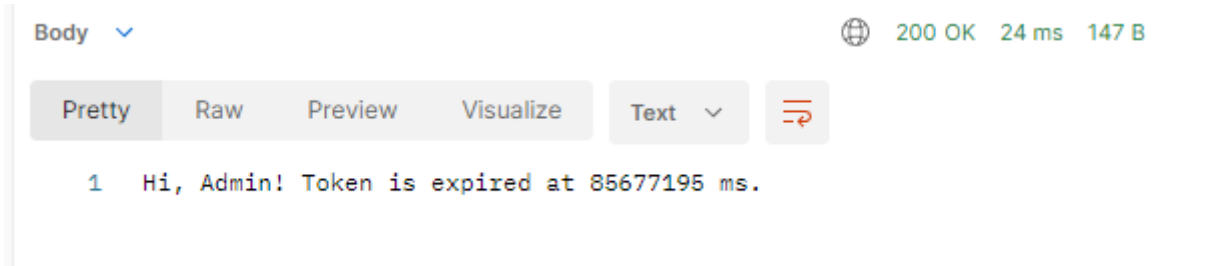
Since we can only access the protected page with a token, the server returned an error.



Now add an `Authorization` header, in which we put the token obtained in the previous step. The header should look like this: `Authorization: Bearer <your_auth_token>`



Now we can access the `localhost:8080/page` !



Everything works perfectly.

§11. Conclusion

In this topic, we studied JWT authentication.

We learned about the structure of the JWT token and how it is signed. A JWT token looks like this:

```
1 headerInBase64URL + "." + payloadInBase64URL + "." +  
  HMACSHA256(headerInBase64URL + "." + payloadInBase64URL, secret)
```

We discovered how to connect the JWT authentication plugin in Ktor and protect your pages with it. Remember the methods that must be specified during the configuration of this plugin, `verifier`, `validate`, and `challenge`.

We also looked at how to generate tokens using the `JWT.create()` method and send them as JSON to the user using `Json.encodeToString()`.

Finally, we tested the functionality of our authentication with Postman.

Now let's apply the information from this topic to tasks.

10 users liked this piece of theory. 0 didn't like it. **What about you?**



Start practicing

Verify to skip

Provided by: [JetBrains Academy](#)

Comments (0)

Useful links (0)

Show discussion