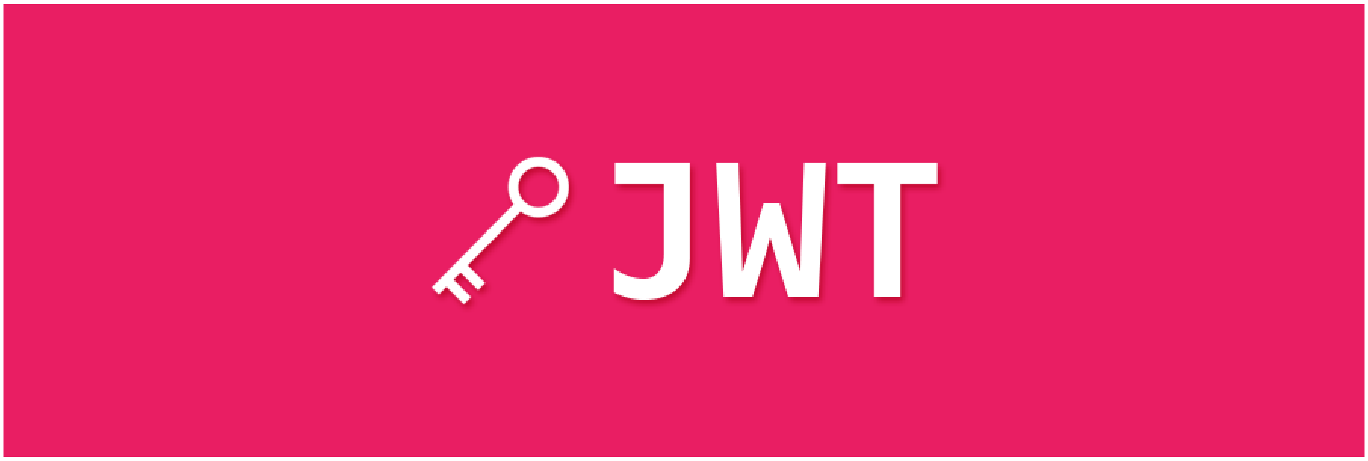
Implementing JWT with Spring Boot and Spring Security

Spring Boot JWT

XOOR Nov 11, 2017 · 8 min read



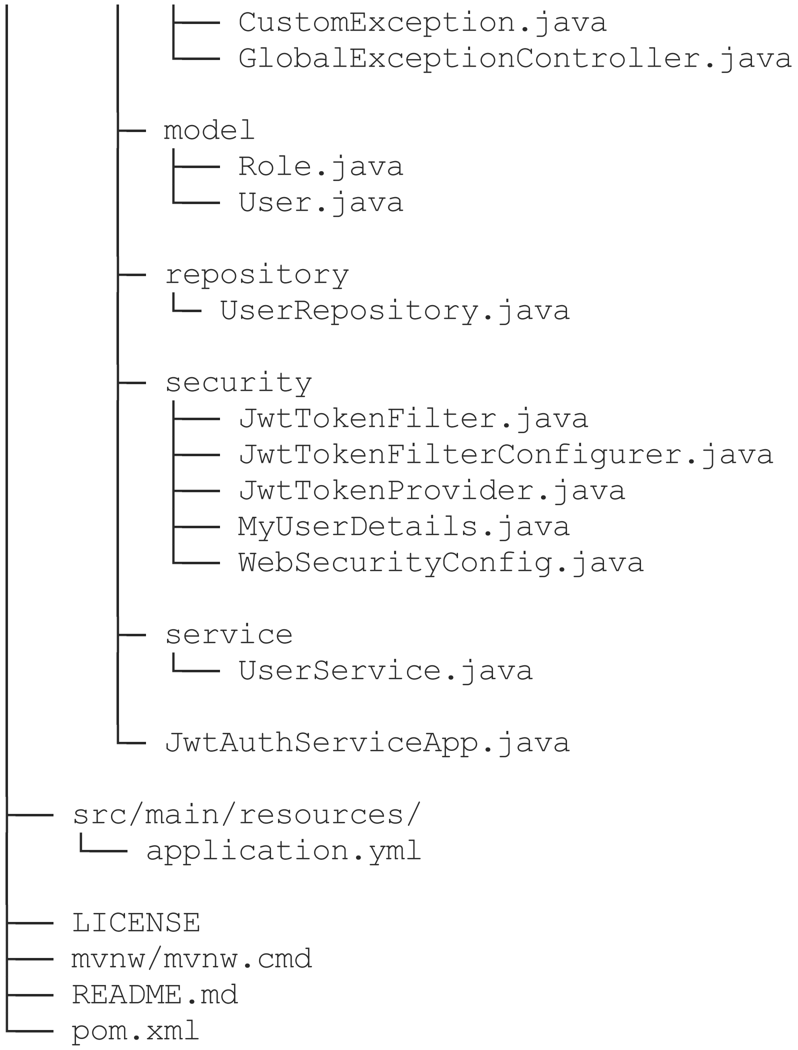
Written by Mauricio Urraco, Full Stack Developer @ XOOR

You can find the code for this tutorial here.

Let’s start by catching a glimpse of the file structure of the project:

File Structure

```
spring-boot-jwt/
 │
 ├─ src/main/java/
 │  └─ murraco
 │     ├─ configuration
 │     │  └─ SwaggerConfig.java
 │     │
 │     ├─ controller
 │     │  └── UserController.java
 │     │
 │     ├─ dto
 │     │  ├── UserDataDTO.java
 │     │  └── UserResponseDTO.java
 │     │
 │     ├─ exception
```



```
│         │    ├── CustomException.java
│         │    └── GlobalExceptionController.java
│         │
│         ├─ model
│         │    ├── Role.java
│         │    └── User.java
│         │
│         ├─ repository
│         │    └─ UserRepository.java
│         │
│         ├─ security
│         │    ├── JwtTokenFilter.java
│         │    ├── JwtTokenFilterConfigurer.java
│         │    ├── JwtTokenProvider.java
│         │    ├── MyUserDetails.java
│         │    └── WebSecurityConfig.java
│         │
│         ├─ service
│         │    └── UserService.java
│         │
│         └─ JwtAuthServiceApp.java
│
├── src/main/resources/
│    └── application.yml
│
├── LICENSE
├── mvnw/mvnw.cmd
├── README.md
└── pom.xml
```

Introduction (https://jwt.io)

Just to throw some background in, we have a wonderful introduction, courtesy of **jwt.io**! Let’s take a look:

What is JSON Web Token?

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA.

Let’s explain some concepts of this definition further.

Compact: Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast.

Self-contained: The payload contains all the required information about the user, avoiding the need to query the database more than once.

When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

Authentication: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

Information Exchange: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed — for example, using public/private key pairs — you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

What is the JSON Web Token structure?

JSON Web Tokens consist of three parts separated by dots **(.)**, which are:

1. Header
2. Payload
3. Signature

Therefore, a JWT typically looks like the following.

```
xxxxx.yyyyy.zzzzz
```

Let's break down the different parts.

Header

The header typically consists of two parts: the type of the token, which is JWT, and the hashing algorithm being used, such as HMAC SHA256 or RSA.

For example:

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

Then, this JSON is Base64Url encoded to form the first part of the JWT.

Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims: reserved, public, and private claims.

- **Reserved claims**: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: iss (issuer), exp (expiration time), sub (subject), aud (audience), and others.

> Notice that the claim names are only three characters long as JWT is meant to be compact.

- **Public claims**: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

- **Private claims**: These are the custom claims created to share information between parties that agree on using them.

An example of payload could be:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

The payload is then Base64Url encoded to form the second part of the JSON Web Token.

Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

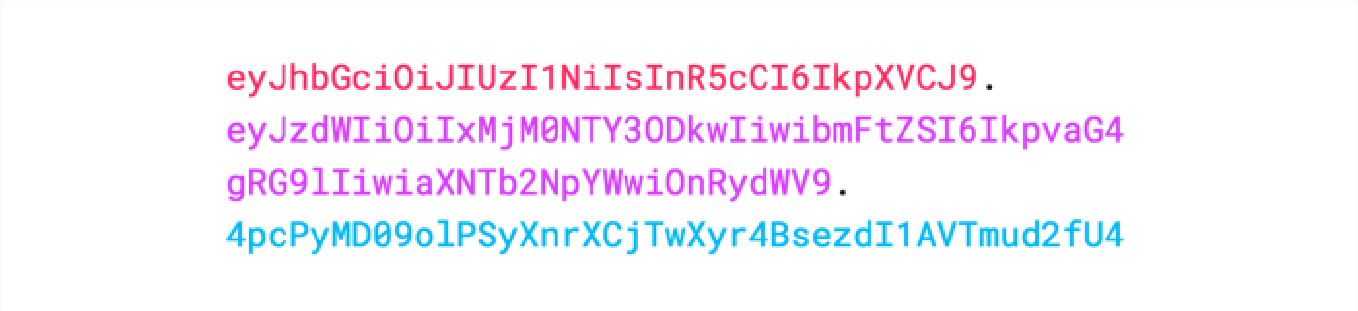
For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way. Putting all together

The output is three Base64 strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret. Encoded JWT



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

How do JSON Web Tokens work?

In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used), instead of the traditional approach of creating a session in the server and returning a cookie.

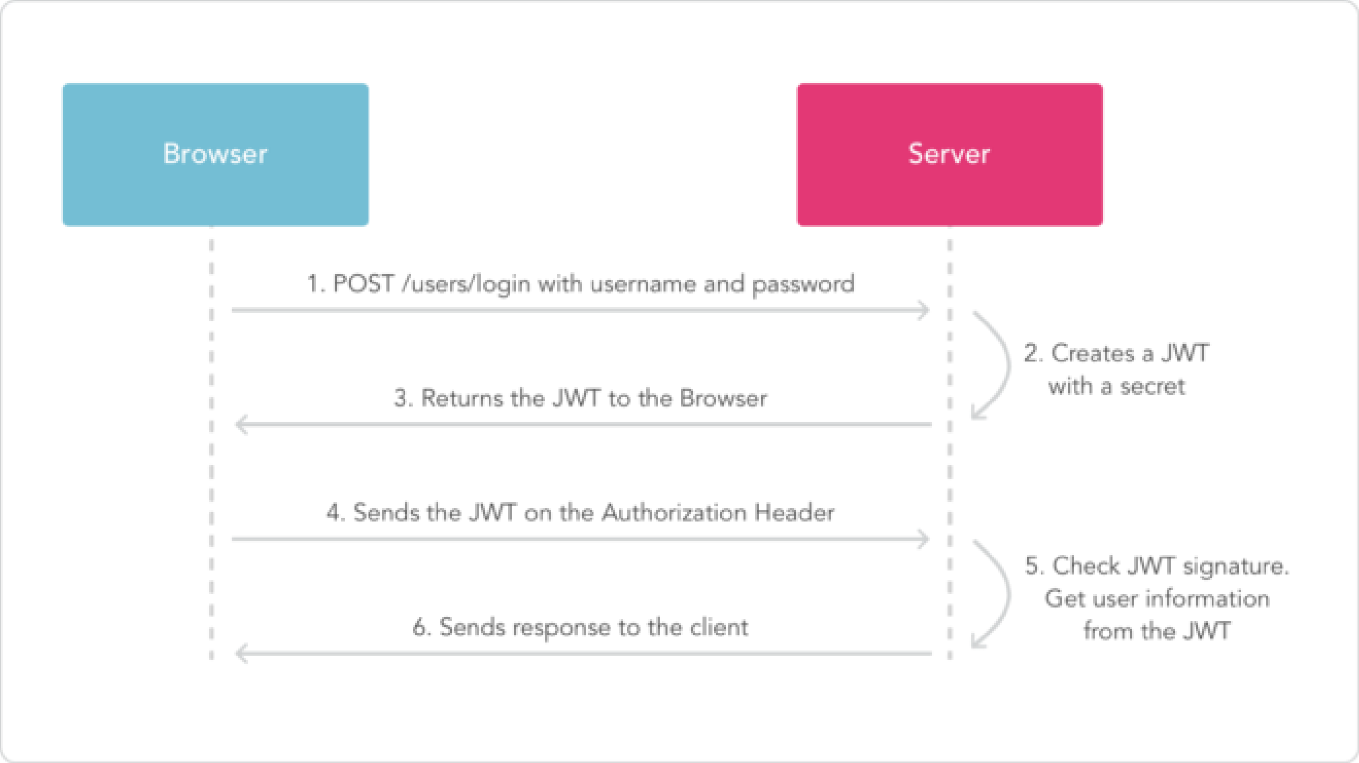
Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

This is a stateless authentication mechanism as the user state is never saved in server memory. The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. As JWTs are self-contained, all the necessary information is there, reducing the need to query the database multiple times.

This allows you to fully rely on data APIs that are stateless and even make requests to downstream services. It doesn't matter which domains are serving your APIs, so Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies.

The following diagram shows 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

JWT Authentication Summary

Token based authentication schema's became immensely popular in recent times, as they provide important benefits when compared to sessions/cookies:

- CORS
- No need for CSRF protection
- Better integration with mobile
- Reduced load on authorization server
- No need for distributed session store

Some trade-offs have to be made with this approach:

- More vulnerable to XSS attacks
- Access token can contain outdated authorization claims (e.g when some of the user privileges are revoked)
- Access tokens can grow in size in case of increased number of claims
- File download API can be tricky to implement
- True statelessness and revocation are mutually exclusive

JWT Authentication flow is very simple

1. User obtains Refresh and Access tokens by providing credentials to the Authorization server
2. User sends Access token with each request to access protected API resource
3. Access token is signed and contains user identity (e.g. user id) and authorization claims.

It’s important to note that authorization claims will be included with the Access token. Why is this important? Well, let’s say that authorization claims (e.g user privileges in the database) are changed during the life time of Access token. Those changes will not become effective until new Access token is issued. In most cases this is not big issue, because Access tokens are short-lived. Otherwise go with the opaque token pattern.

Implementation Details

Let’s see how can we implement the JWT token based authentication using Java and Spring, while trying to reuse the Spring security default behavior where we can. The Spring Security framework comes with plug-in classes that already deal with authorization mechanisms such as: session cookies, HTTP Basic, and HTTP Digest. Nevertheless, it lacks from native support for JWT, and we need to get our hands dirty to make it work.

MySQL DB

This demo is currently using a MySQL database called **user_db** that’s automatically configured by Spring Boot. If you want to connect to another database you have to specify the connection in the `application.yml` file inside the resource directory. Note that `hibernate.hbm2ddl.auto=create-drop` will drop and create a clean database each time we deploy (you may want to change it if you are using this in a real project). Here's the example from the project:

Code Code

1. `JwtTokenFilter`

2. `JwtTokenFilterConfigurer`
3. `JwtTokenProvider`
4. `MyUserDetails`
5. `WebSecurityConfig`

JwtTokenFilter

The `JwtTokenFilter` filter is applied to each API (`/**`) with exception of the signin token endpoint (`/users/signin`) and singup endpoint (`/users/signup`).

This filter has the following responsibilities:

1. Check for access token in Authorization header. If Access token is found in the header, delegate authentication to `JwtTokenProvider` otherwise throw authentication exception
2. Invokes success or failure strategies based on the outcome of authentication process performed by JwtTokenProvider

Please ensure that `chain.doFilter(request, response)` is invoked upon successful authentication. You want processing of the request to advance to the next filter, because very last one filter *FilterSecurityInterceptor#doFilter* is responsible to actually invoke method in your controller that is handling requested API resource.

JwtTokenFilterConfigurer

Adds the `JwtTokenFilter` to the `DefaultSecurityFilterChain` of spring boot security.

JwtTokenProvider

The `JwtTokenProvider` has the following responsibilities:

1. Verify the access token's signature
2. Extract identity and authorization claims from Access token and use them to create UserContext
3. If Access token is malformed, expired or simply if token is not signed with the appropriate signing key Authentication exception will be thrown

MyUserDetails

Implements `UserDetailsService` in order to define our own custom loadUserbyUsername function. The UserDetailsService interface is used to retrieve user-related data. It has one method named loadUserByUsername() which finds a user entity based on the username and can be overridden to customize the process of finding the user.

It is used by the `DaoAuthenticationProvider` to load details about the user during authentication.

WebSecurityConfig

The `WebSecurityConfig` class extends `WebSecurityConfigurerAdapter` to provide custom security configuration.

Following beans are configured and instantiated in this class:

1. `JwtTokenFilter`
2. `PasswordEncoder`

Also, inside `WebSecurityConfig#configure(HttpSecurity http)` method we'll configure patterns to define protected/unprotected API endpoints. Please note that we have disabled CSRF protection because we are not using Cookies.

Conclusion

As you can see from this post, thanks to Spring Boot and Spring Security, we can have a JWT authentication service up and running in record time. This is also incredibly easy to integrate into a Microservices ecosystem if we make use of Spring Cloud and the Netflix Stack: Eureka, Zuul, Ribbon and Hystrix (we'll see more about this in future posts). In the meantime remember to check the code on GitHub to try it out or to add JWT authentication to your own project!

What am I missing here? Let me know in the comments and I'll add it in! Thanks for reading!