Securing applications with JWT Spring Boot

Ignacio Oliveto Follow
Jul 20, 2019 · 9 min read ★

An application's security is one of the biggest issues to consider.

Every day millions of users put sensitive information on the internet, and keeping that information safe is one of the biggest challenges for developers. Fortunately, this is not a new problem and we have several tools at our disposal to keep our app users' data safe. One of those tools that is particularly practical and easy to implement, is JWT (JSON Web Tokens).

Description

Throughout this post we'll learn how to implement a basic user authentication with JWT and Spring Boot, but first we need to know what a JWT is. If we do a quick google search we'll find that JWT is an RFC 7519 open standard. If you go to that link you'll find this definition:

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.

These kinds of definitions can sometimes seem more complex than they are, so we'll try to explain it in an easier way. In any application that has private information or restricted access, we need a way to validate requests without processing the client's login credentials for every request. For this we have JWT, which authenticates the user without cookies or login info.

Explanation

When a user logs in, the backend generates a JWT composed as follows:

```
header.payload.signature
```

HEADER

We can see that the JWT starts with the header which contains all pertinent info about how a token must be interpreted.

PAYLOAD

The payload contains the claims. In short, the claims are the user's data (or any entity) plus additional important information (not mandatory) that adds functionality to the token.

We can find three types of claims: registered, public, private.

Registered claims are used to provide additional information about the token, such as the time it was created or when the token expires. These claims are not mandatory.

Public claims are defined by those who use JWT. One should be careful about which names they use since they can cause a collision.

Private claims can be used to store information without using *registered claims* or *public claims*. Keep in mind that they are susceptible to collision.

SIGNATURE

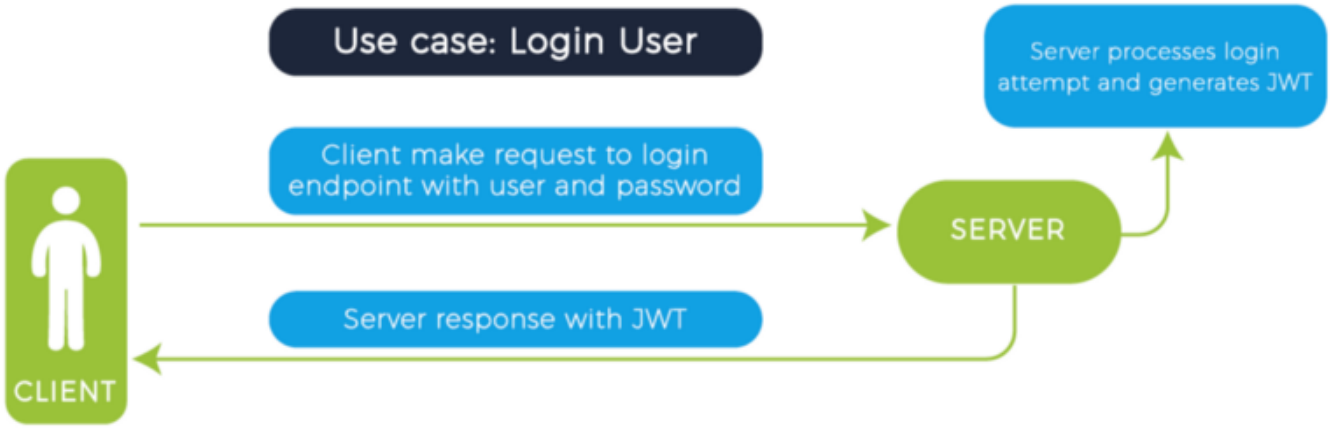
The signature is composed of the encoded header, encoded payload, a secret and the coding algorithm (also present in the header). And all that is signed.

This is a rough look at the composition of a JWT. For more information visit this link to find an example of a JWT and its parts, and in case you want to analyze it in more detail check out this link.

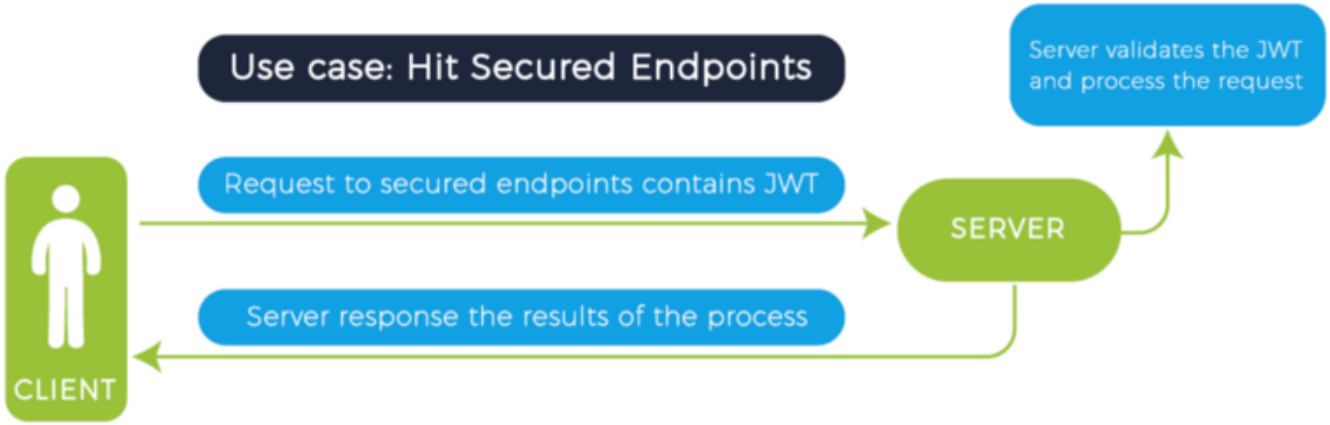
IMPLEMENTATION

For the implementation we'll use *Java Spring Boot* to make a small application with a login, and we'll use *postman* to consume that app. Regarding persistence, we'll use *PostgreSQL*, but keep in mind that it is merely to persist users without any kind of relationship (Since the goal is to teach a basic session login, using other entities such as roles would deviate us from the main concept).

To get started let's divide the use cases in two: when the user logs in and when the user tries to consume a protected endpoint.

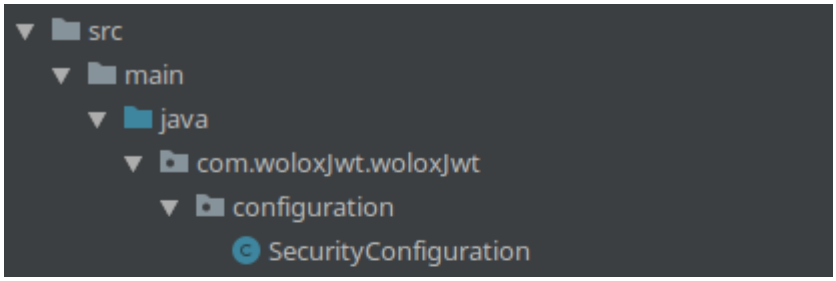


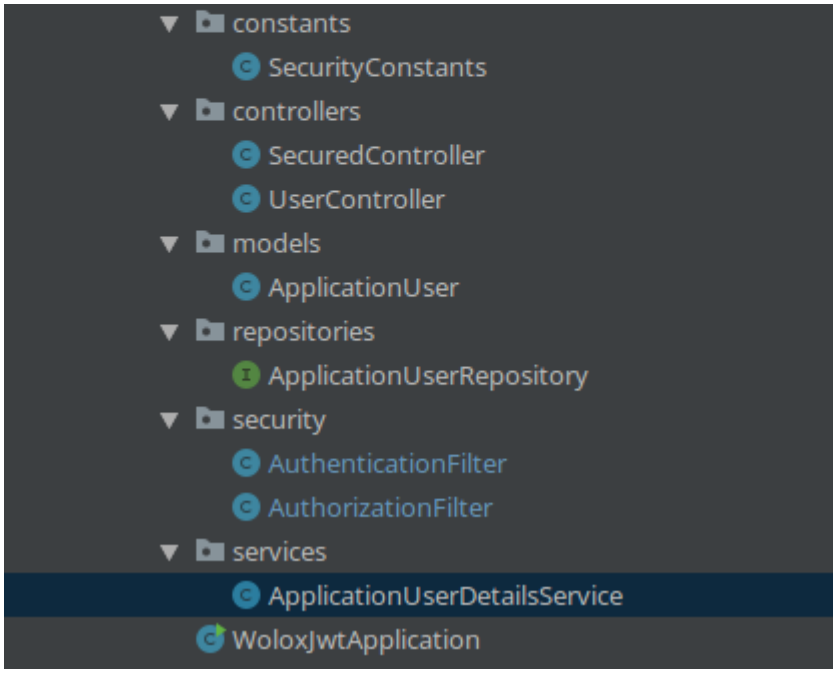
In this diagram we can see that the user tries to log in with their username and password, the request is successful and in return the user receives a JWT that will be used to consume protected endpoints.



In this diagram we can see that after logging in and obtaining a JWT token, the user makes a request to a protected endpoint, the server validates that JWT, processes the request and sends the response to the client.

Now that we have an idea of how the client-server interaction works, we can start looking at our project's structure:





Let's start then with the User class

```
1  package com.woloxJwt.woloxJwt.models;
2
3
4  import javax.persistence.Entity;
5  import javax.persistence.GeneratedValue;
6  import javax.persistence.GenerationType;
7  import javax.persistence.Id;
8
9
10 @Entity
11 public class ApplicationUser {
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private long id;
15     private String username;
16     private String password;
17
18     public long getId() {
19         return id;
20     }
21
22     public void setId(long id) {
23         this.id = id;
24     }
25
```

```
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

ApplicationUser.java hosted with ♡ by **GitHub** view raw

Here we have a typical User class from our application. As we mentioned before, we omitted the implementation of roles in order to not deviate from the main concept.

In this case we have an id that represents the user in the database, and we have a username and a password that will allow us to persist the credentials of the user and use them in the login process.

```
package com.woloxJwt.woloxJwt.repositories;

import com.woloxJwt.woloxJwt.models.ApplicationUser;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ApplicationUserRepository extends JpaRepository<ApplicationUser, Long> {
    ApplicationUser findByUsername(String username);
}
```

ApplicationUserRepository.java hosted with ♡ by **GitHub** view raw

To persist the user we'll use a repository. In this repository we'll have a method that allows us to obtain a user by the username which will be useful for the authentication process.

```
package com.woloxJwt.woloxJwt.controllers;
```

```java
import com.woloxJwt.woloxJwt.models.ApplicationUser;
import com.woloxJwt.woloxJwt.repositories.ApplicationUserRepository;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/users")
public class UserController {

    private ApplicationUserRepository applicationUserRepository;
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public UserController(ApplicationUserRepository applicationUserRepository,
                          BCryptPasswordEncoder bCryptPasswordEncoder) {
        this.applicationUserRepository = applicationUserRepository;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }

    @PostMapping("/record")
    public void signUp(@RequestBody ApplicationUser user) {
        user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
        applicationUserRepository.save(user);
    }
}
```

UserController.java hosted with ♡ by **GitHub** view raw

This controller will be responsible for registering a user in the database using the username and password. It's important to note that we are encrypting the password so that our database does not contain the actual password information.

```java
package com.woloxJwt.woloxJwt.controllers;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/secure")
public class SecuredController {

```

```
13      @GetMapping
14      public ResponseEntity reachSecureEndpoint() {
15
16          return new ResponseEntity("If your are reading this you reached a secure endpoint", Http
17      }
18  }
```

SecuredController.java hosted with ♡ by **GitHub** view raw

This will be our secure controller to test that the authentication was successful and that we get a response that we normally would not have access to without the correct JWT.

Spring Boot security configuration and filters

First we'll start by defining some constants that we'll use throughout the implementation.

```
1  package com.woloxJwt.woloxJwt.constants;
2
3  public class SecurityConstants {
4      public static final String SIGN_UP_URL = "/users/record";
5      public static final String KEY = "q3t6w9z$C&F)J@NcQfTjWnZr4u7x!A%D*G-KaPdSgUkXp2s5v8y/B?E(H+M
6      public static final String HEADER_NAME = "Authorization";
7      public static final Long EXPIRATION_TIME = 1000L*60*30;
8  }
```

SecurityConstants.java hosted with ♡ by **GitHub** view raw

Here we can see a series of important data:

- SIGN_UP_URL: Determines a public endpoint to register the user.
- KEY: Contains the key to sign the token and it has a length of 512 bytes because it'll be used by an algorithm that requires a string of at least that length. (Normally the key would be obtained from a secret and would never be hardcoded).

- HEADER_NAME: Contains the name of the header you are going to add the JWT to when doing a request.
- EXPIRATION_DATE: Contains the time (in milliseconds) during which the token is valid before expiring.

```
package com.woloxJwt.woloxJwt.configuration;

import com.woloxJwt.woloxJwt.security.AuthenticationFilter;
import com.woloxJwt.woloxJwt.security.AuthorizationFilter;
import com.woloxJwt.woloxJwt.services.ApplicationUserDetailsService;
import org.springframework.context.annotation.Bean;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationMana
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAda
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

import static com.woloxJwt.woloxJwt.constants.SecurityConstants.SIGN_UP_URL;

@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    private ApplicationUserDetailsService userDetailsService;
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public SecurityConfiguration(ApplicationUserDetailsService userDetailsService, BCryptPasswor
        this.userDetailsService = userDetailsService;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable().authorizeRequests()
                .antMatchers(HttpMethod.POST, SIGN_UP_URL).permitAll()
                .anyRequest().authenticated()
                .and()
                .addFilter(new AuthenticationFilter(authenticationManager()))
```

```
                .addFilter(new AuthorizationFilter(authenticationManager()))
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }
    @Bean
    CorsConfigurationSource corsConfigurationSource() {
        final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", new CorsConfiguration().applyPermitDefaultValues
        return source;
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder);
    }


}
```

SecurityConfiguration.java hosted with ♡ by **GitHub** view raw

This class contains the configuration of Spring Boot security. With this configuration we can specify, among several things, which url we can register the user in, which URLs are protected and the type of session we want to use for authentication.

Let's see the most important configurations that this class does.

The first thing that we can observe is the annotation **@EnableWebSecurity**. This annotation activates the web security integrated in Spring Boot whose configuration we are going to change.

Do you remember the constant ***SIGN_UP_URL*** that we defined in the ***SecurityConstants*** class? That constant is used by the ***configure (HttpSecurity http)*** method in this class to define the endpoint we can register a user in. In addition, this method is responsible for very important configurations such as: the configuration of ***CORS***, the definition of ***authentication*** and ***authorization*** filters (whose implementation we'll later see), and also establishes that the session will be stateless (which avoids sending cookies along with the response to handle the session).
In the CorsConfigurationSource **corsConfigurationSource()** method we'll allow all urls to support endpoint CORS and this will allow us to limit it to only some or none.

Finally the ***configure (AuthenticationManagerBuilder auth)*** method allows us to use a custom implementation of a service to obtain user data when authentication is correct (we'll see the implementation of this service next).

```java
package com.woloxJwt.woloxJwt.services;

import com.woloxJwt.woloxJwt.models.ApplicationUser;
import com.woloxJwt.woloxJwt.repositories.ApplicationUserRepository;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import static java.util.Collections.emptyList;

@Service
public class ApplicationUserDetailsService implements UserDetailsService {
    private ApplicationUserRepository applicationUserRepository;

    public ApplicationUserDetailsService(ApplicationUserRepository applicationUserRepository) {
        this.applicationUserRepository = applicationUserRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        ApplicationUser applicationUser = applicationUserRepository.findByUsername(username);
        if (applicationUser == null) {
            throw new UsernameNotFoundException(username);
        }
        return new User(applicationUser.getUsername(), applicationUser.getPassword(), emptyList(
    }
}
```

ApplicationUserDetailsService.java hosted with ♡ by **GitHub** view raw

This service is responsible for comparing the user's data in the database to the submitted credentials and if they match, authenticating the user. It is important to highlight that the method instantiates a User object, which is used by the core of Spring Boot security to generate the user's session. For more information about this you can visit this link.

Now that we have the necessary services to obtain the user data from the database and the necessary configuration for Spring Boot to generate a JWT token, we are going to see two very important classes: the first one that is in charge of authenticating the user (generating a token that is sent in the headers when the login is correct), and the second one that validates the token sent by the client to allow it to consume the protected endpoints.

```
1  package com.woloxJwt.woloxJwt.security;
2
3  import com.fasterxml.jackson.databind.ObjectMapper;
4  import com.woloxJwt.woloxJwt.models.ApplicationUser;
5  import io.jsonwebtoken.Claims;
6  import io.jsonwebtoken.Jwts;
7  import io.jsonwebtoken.SignatureAlgorithm;
8  import io.jsonwebtoken.security.Keys;
9  import org.springframework.security.authentication.AuthenticationManager;
10 import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
11 import org.springframework.security.core.Authentication;
12 import org.springframework.security.core.AuthenticationException;
13 import org.springframework.security.core.userdetails.User;
14 import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
15
16 import javax.servlet.FilterChain;
17 import javax.servlet.ServletException;
18 import javax.servlet.http.HttpServletRequest;
19 import javax.servlet.http.HttpServletResponse;
20 import java.io.IOException;
21 import java.security.Key;
22 import java.util.ArrayList;
23 import java.util.Date;
24
25 import static com.woloxJwt.woloxJwt.constants.SecurityConstants.*;
26
27 public class AuthenticationFilter extends UsernamePasswordAuthenticationFilter {
28     private AuthenticationManager authenticationManager;
29
30     public AuthenticationFilter(AuthenticationManager authenticationManager) {
31         this.authenticationManager = authenticationManager;
32     }
33
34     @Override
35     public Authentication attemptAuthentication(HttpServletRequest req,
```

```
                                                HttpServletResponse res) throws AuthenticationEx
        try {
            ApplicationUser applicationUser = new ObjectMapper().readValue(req.getInputStream(),

            return authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(applicationUser.getUsername(),
                            applicationUser.getPassword(), new ArrayList<>())
            );
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest req, HttpServletResponse res, Fil
                                            Authentication auth) throws IOException, ServletExce

        Date exp = new Date(System.currentTimeMillis() + EXPIRATION_TIME);
        Key key = Keys.hmacShaKeyFor(KEY.getBytes());
        Claims claims = Jwts.claims().setSubject(((User) auth.getPrincipal()).getUsername());
        String token = Jwts.builder().setClaims(claims).signWith(key, SignatureAlgorithm.HS512).
        res.addHeader("token", token);


    }
}
```

AuthenticationFilter.java hosted with ♡ by **GitHub** view raw

The first thing that we can notice in this class is that it extends from “UsernamePasswordAuthenticationFilter”, a class responsible for processing the user session. Going through the class we see that we instantiate an object of the class “AuthenticationManager” that we are going to use in the “attemptAuthentication” method to process the session start attempt. In the case that the attempt is satisfactory, the token is generated in the “successfulAuthentication” method. First we’ll define the expiration time (we’ll use one of the previously defined constants), and then we’ll generate the key with which the token will be signed. This key is also defined in our constants, we use the “hmacShaKeyFor” method to generate this key in a secure way (previously the literal string was used, but it is safer to use a Key object for this). It’s important to note that the key must be a certain size. As an example, we are going to use

the encryption algorithm “HS512”. For this algorithm we need a string that is no less than 512 bits, otherwise we’ll get an exception informing us that this isn’t safe enough.

To generate a string of exactly 512 bits, I recommend using this site.

Now that we’ve seen the class in charge of authenticating users, we are going to take a look at the class that is in charge of authorizing users to consume protected endpoints. For this, this class will validate the token that the client sends to us and check if it matches our signature.

```
package com.woloxJwt.woloxJwt.security;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.security.Keys;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.authentication.www.BasicAuthenticationFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.security.Key;
import java.util.ArrayList;

import static com.woloxJwt.woloxJwt.constants.SecurityConstants.*;

public class AuthorizationFilter extends BasicAuthenticationFilter {

    public AuthorizationFilter(AuthenticationManager authManager) {
        super(authManager);
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain) throws IOException, ServletException {
        String header = request.getHeader(HEADER_NAME);
```

```
33
34          if (header == null) {
35              chain.doFilter(request, response);
36              return;
37          }
38
39          UsernamePasswordAuthenticationToken authentication = authenticate(request);
40
41          SecurityContextHolder.getContext().setAuthentication(authentication);
42          chain.doFilter(request, response);
43      }
44
45      private UsernamePasswordAuthenticationToken authenticate(HttpServletRequest request) {
46          String token = request.getHeader(HEADER_NAME);
47          if (token != null) {
48              Claims user = Jwts.parser()
49                      .setSigningKey(Keys.hmacShaKeyFor(KEY.getBytes()))
50                      .parseClaimsJws(token)
51                      .getBody();
52
53              if (user != null) {
54                  return new UsernamePasswordAuthenticationToken(user, null, new ArrayList<>());
55              }else{
56                  return  null;
57              }
58
59          }
60          return null;
61      }
62  }
```

AuthorizationFilter.java hosted with ♡ by **GitHub** view raw

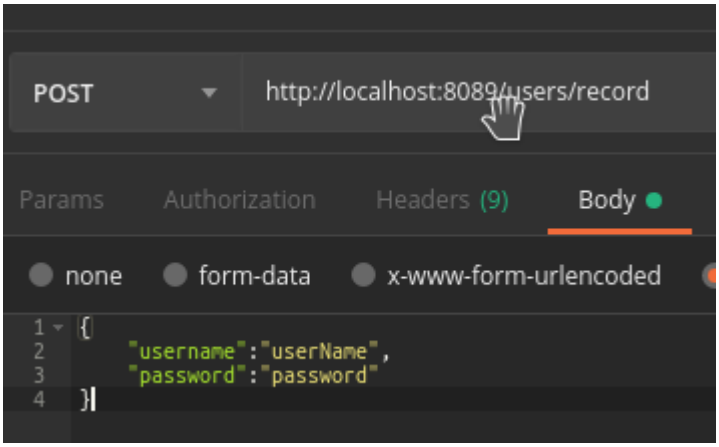
In this class we can see that the “authentication” method will check if the token exists and if so, if it’s valid or not. In the case that it is, it will get an encrypted user from the token and will incorporate it into the “SecurityContextHolder”, then it will run a series of filters (some of which we define in the class “SecurityConfiguration”) and if these filters pass through correctly, the method will allow the user to reach the endpoint.

Note that until now we do not specifically define the endpoint in which a user is going to log in.This is why Spring Boot already pre-defines it when we extend our “UsernamePasswordAuthenticationFilter” authorization filter. At this point we have

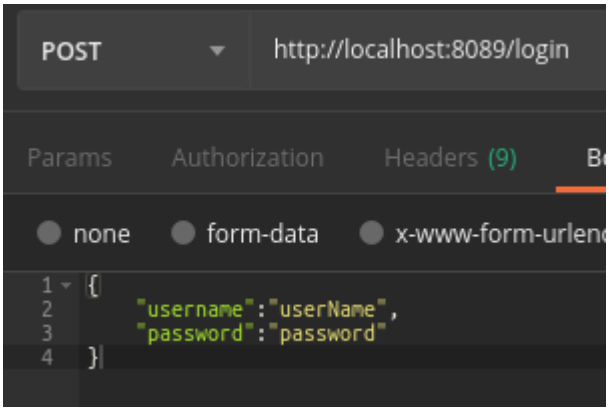
already implemented all the logic necessary to log in a user and authorize the use of protected endpoints.

Testing the implementation

To test the implementation we are going to use Postman. First, we have to register a user. To do this we'll send the credentials that the user will have in a POST and they'll be persisted in the database.



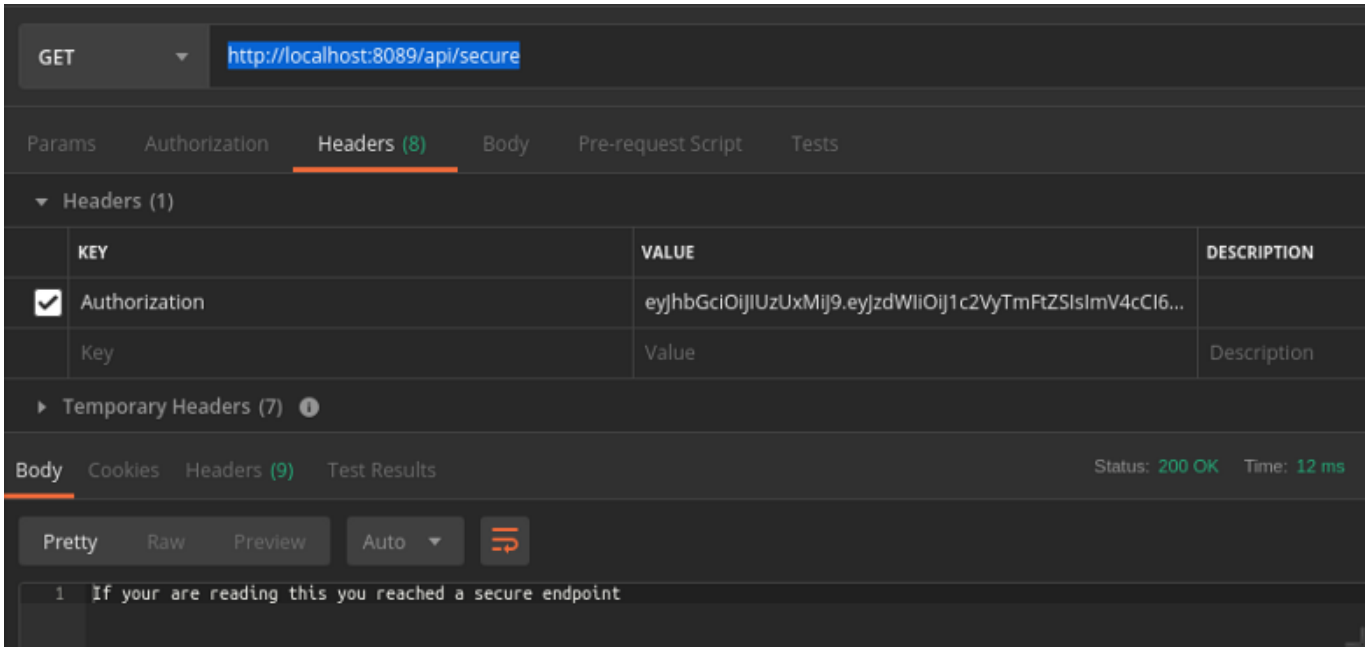
If everything went well we'll receive an HTTP status 200 confirming that the credentials have been persisted. Now we can try to log in.



We are going to make a request to: "http://localhost:8089/login", if the credentials are correct, we'll get an HTTP status 200 response and the token generated by the server included in the headers.

Now that we have the token that confirms our identity we can consume secured endpoints. Let's try to access the endpoint that we previously defined. This time we'll send a request to "http://localhost:8089/api/secure" with a GET method, but we have to add the token to the headers. For this we're going to put the word "Authorization" as an identifier and the token next to it.

If the token is correct, it'll reply a 200 response status in addition to the confirmation that we are reading protected information.



By achieving the consumption of this endpoint we can confirm that the authentication and authorization of users in our application works

Conclusion

In this post we implemented a basic user class, with its credentials, controllers that define endpoints where a user can log in and an endpoint that can only be consumed if the user is authorized. For this we configured the security of Spring Boot, implemented a series of filters and developed the logic that generates the token that we send in each request.

After doing all of this, we can then rest assured that our information will remain safe using JWT.

API Software Development Java Spring Boot Backend