**Spring Boot Security**

**JSON Web Token (JWT)**

It is just a way that enables two applications to communicate with each other and transmit/exchange the information with another.

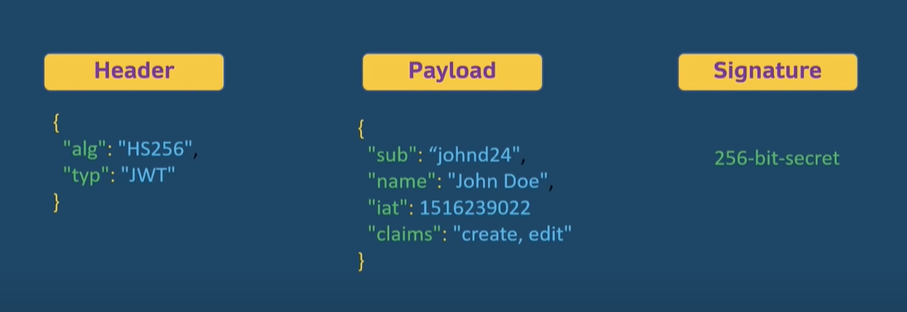
Our application is going to give a token to a user. They’re going to create the token and the user information and then sign the token. That signature is digital.

Now whenever the user wants to get access to the information, the user will send the token as the ***authorization***.

***JWT is mostly used for authorization.***

Let's say we have an API and I need to access it. I would like some sort of authentication. I would tell the application this is me with my username and password. The application then will verify who I am and this is called ***authentication***.

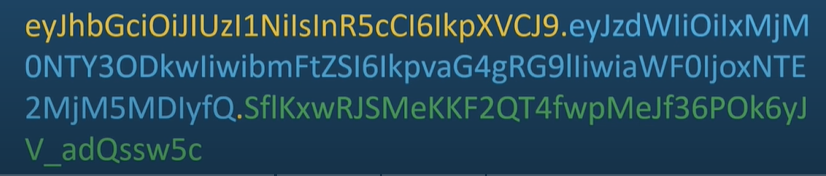
The application then gives the ***JSON web token*** to me. This token contains my information, my permissions. So, whenever I need to access a resource on the server, I will send that token as a request. Then this is going to be the ***authorization***. The server in return, checks the token to see if it’s a valid token or not. After that it checks what all permissions that token has. And if the token has the permission to access that resource, it returns that resource as the response.



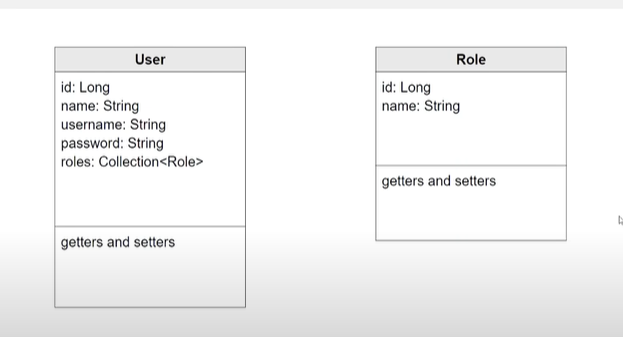
The ***JSON web token*** has three parts which we can see above.

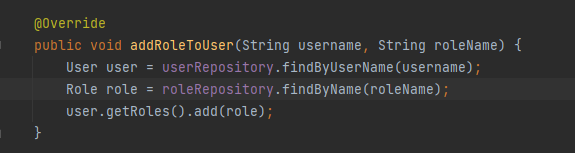
The ***payload*** contains all the information about the owner of the token, what all permissions that owner has etc.

For the ***signature***, we have to take the encoded header, encoded payload and use a secret with the algorithm. After this we sign the token.



**Design**

****

****

As we have used the ***@Transactional*** annotation, we don’t have to again call the userRepository to save the changes. It will do by itself.

If we use ***@RequiredArgsConstructor***, then it will auto-inject all the objects that we specified. We don’t have to write ***@Autowired*** for every object to get injected.

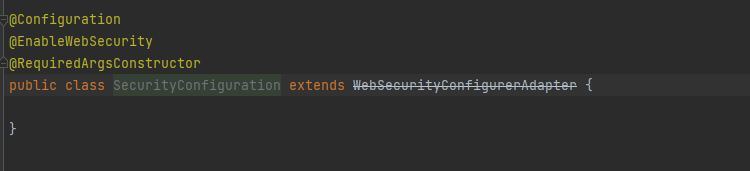
With the below url:

<http://localhost:8080/api/users>

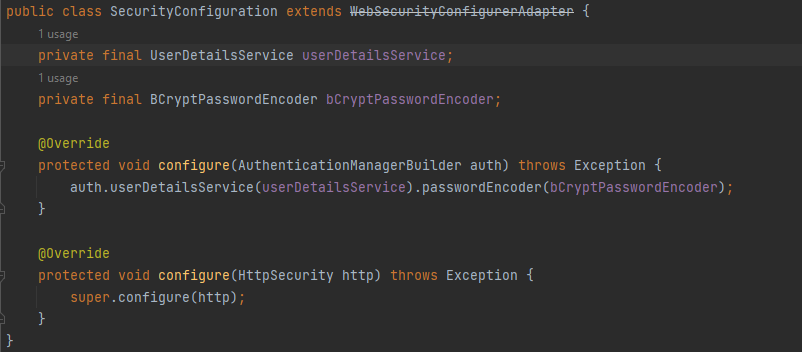
Anyone can access the link with the password generated in the command line. We have to do something such that we can restrict the access to only those users which we have created and also based on the role, we will be restricting the access to particular sets of users.

Spring has its own default configuration which generates a random password which we can see in the logs and username mainly ***user***, ***admin*** to login and access the APIs. But we need to tell Spring Security that we have our own sets of users that we want to use to login the system. And also we need to specify the roles of each user and make Spring security to understand those so that we can restrict the access of some paths based on the roles of the users.

This is where **Spring Security** comes into action.



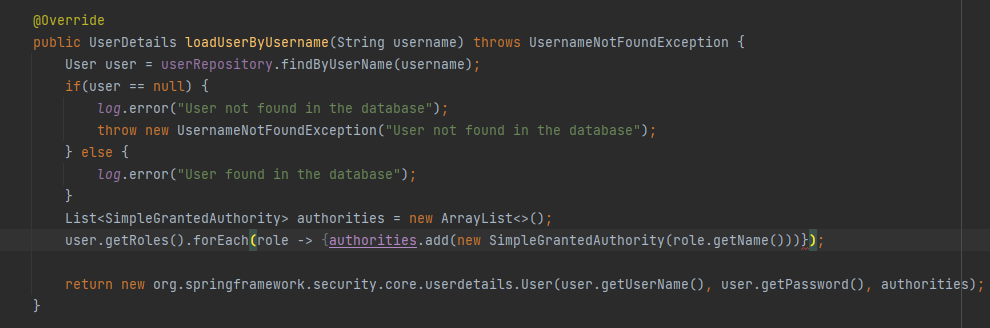
***WebSecurityConfigurerAdapter*** is the main security class.



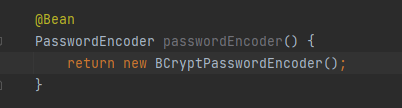
There are many ways we can tell Spring how to look for the users. One is ***inMemoryAuthentication*** and here I can pass the username and password that Spring will use to check for users whenever users are trying to log into the application.

But we will be using ***UserDetailService*** to tell Spring how to look for the users.

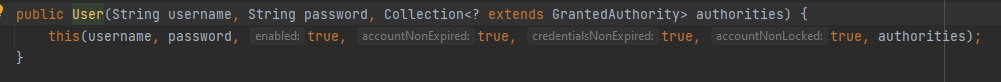
We don’t have any configurations or overrides for the two beans that we have used. So, we have to create two beans in our applications.



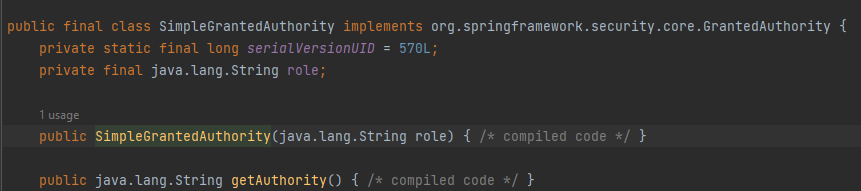
***UserDetailService*** has the function ***loadUserbyUsername*** that we need to override and create our own logic to tell Spring Security of all the users that we have in the database and roles that each user has.



Here we have overridden this ***passwordEncoder*** bean where we will be encoding the password of the user who is logged in using ***BCryptPasswordEncoder***.



Here the authorities can be any collection which extends ***GrantedAuthority***.



***SimpleGrantedAuthority*** implements ***GrantedAuthority*** and so we can use it.

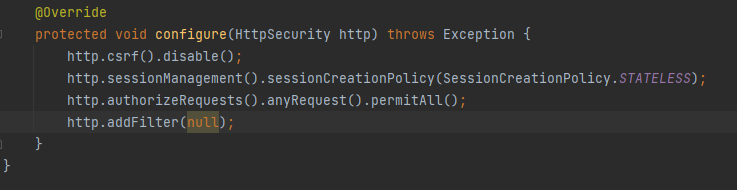
**Authentication & Authorization**



Now if we tie this with a ***JSON web token*** then we can say that when the user logs in for the first time, the application creates a token and gives it to the user if that user information is present in the database (***Authentication***).

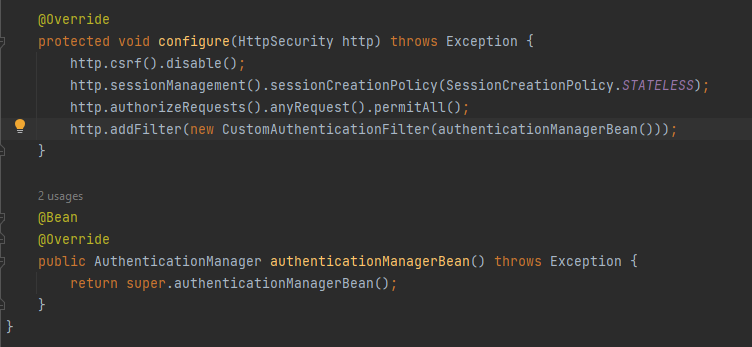
After the user is logged in, it will make subsequent requests to the server when it is accessing the resources of the application. Then with that request, that token will also be sent which the servers will see and check all the permissions associated with that token. On the basis of that, it decides whether to grant the access of that resource to that user or not. (***Authorization***).

Now, when Spring uses the default configuration as we know when we were trying to access the application. It uses a session policy that was stateful. So, the Spring uses a session and then saves something in memory tracking the user by giving them a cookie. But we don’t want this system. We want a ***JSON web token based system***.



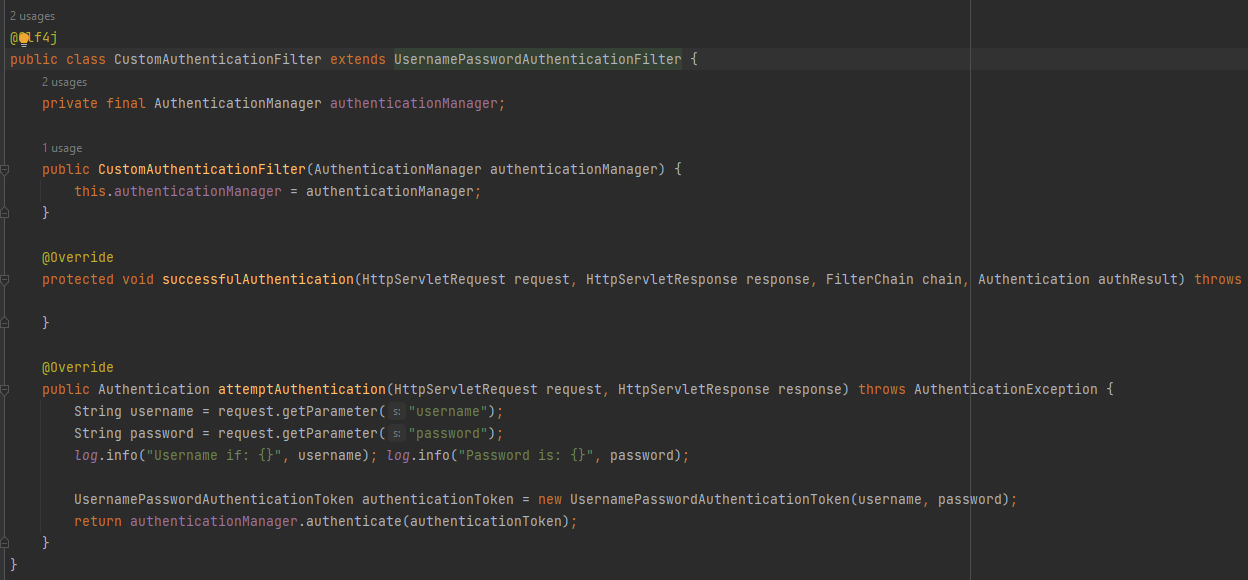
For now, we are allowing everyone to access our application at this point by using ***permitAll()*** method.

We also need an ***authentication filter*** so that we can check the user whenever they are trying to log into our application.

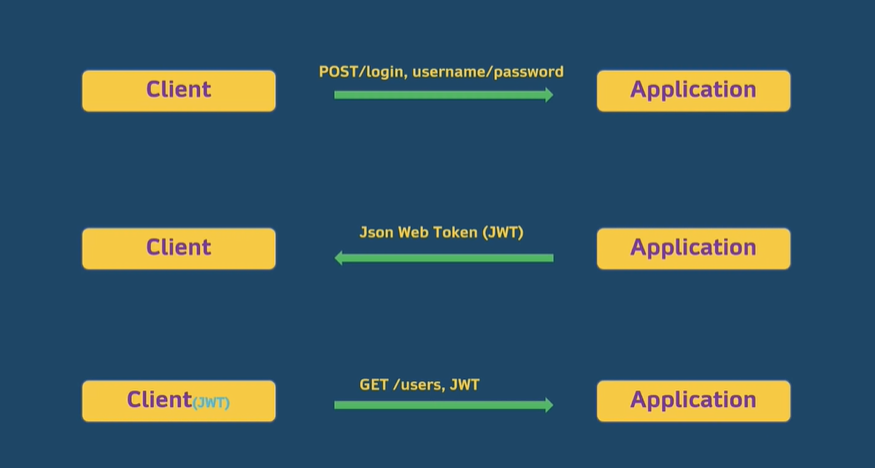


Here we added our ***custom filter*** meaning what we have to do when the user is successfully logged in or what to do when the user is not able to log in.

To this custom filter, we need to pass the ***authenticationManager*** which the interface ***WebSecurityConfigurerAdapter*** has. ***AuthenticationManager*** is the one who is going to authenticate the user and decide whether we need to add the token to that user or not.

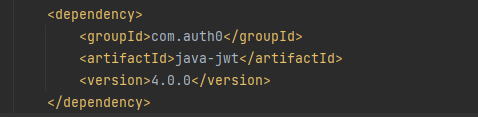


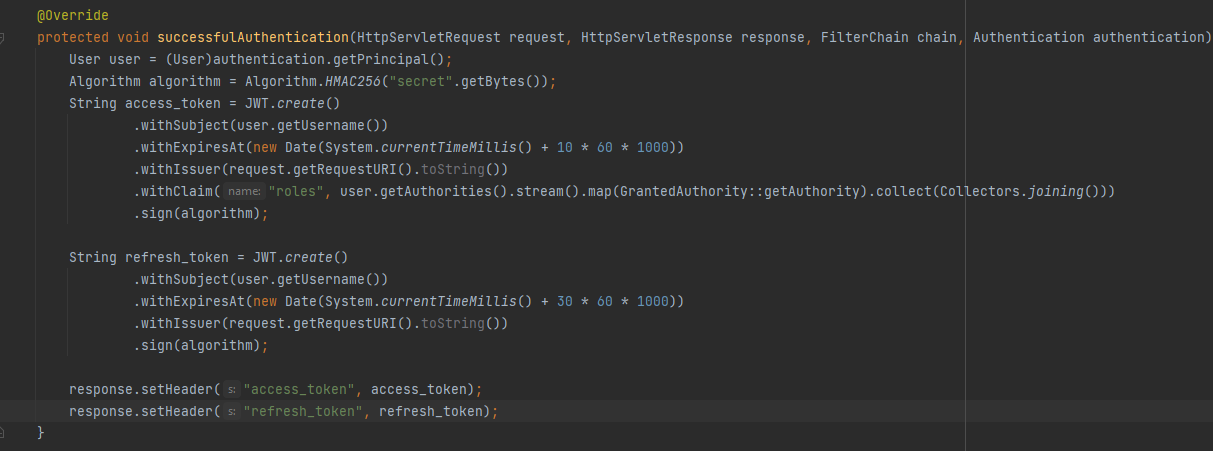
**Simple Flow of Security with JSON Web Token**



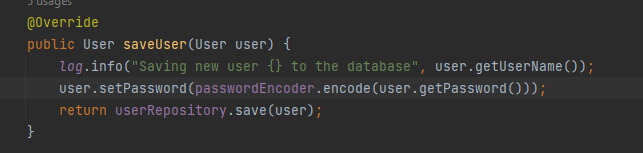
There is something called a ***refresh token*** that the client also has. So when the actual access token expires, with a refresh token, the server can create a new token for the user.

After the successful authentication, we need to create the JSON token and then sign the token and give it to the user. For this we can use the external library to do that work for us.

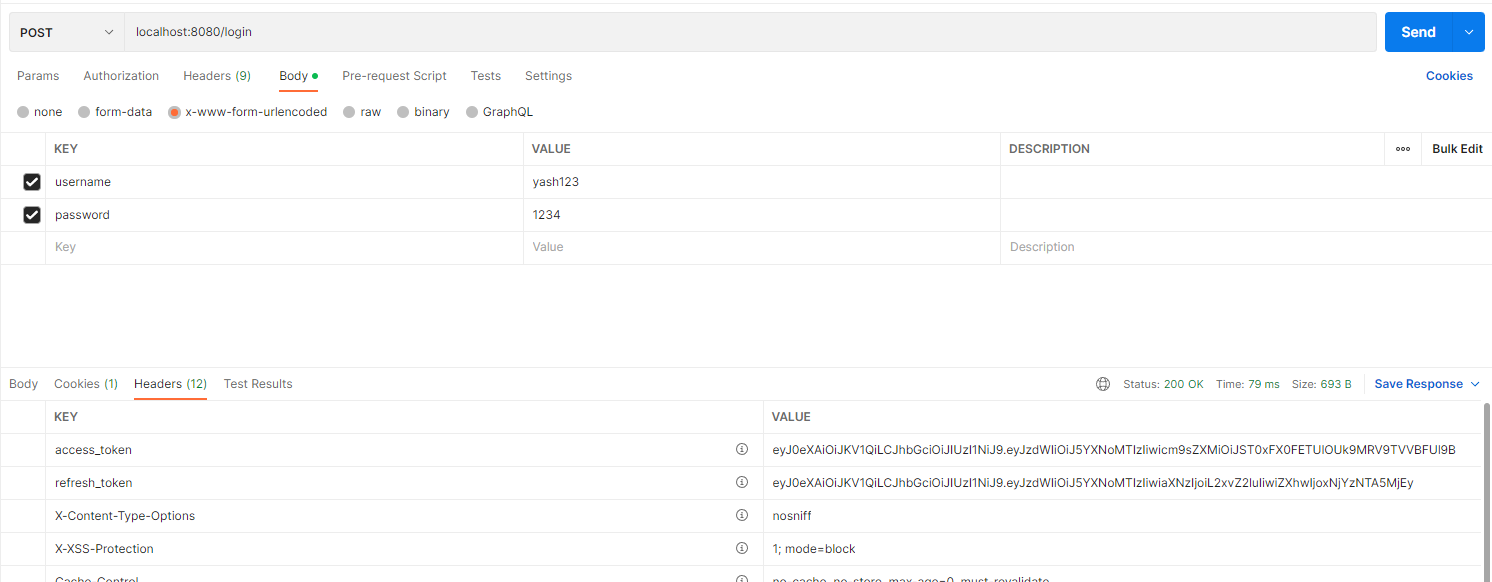




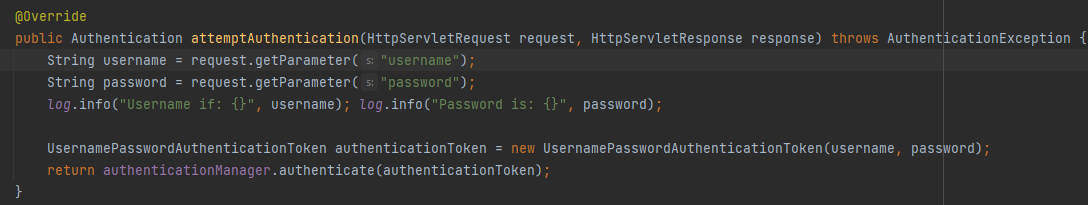
***.getPrincipal()*** would give us the successfully authenticated user.



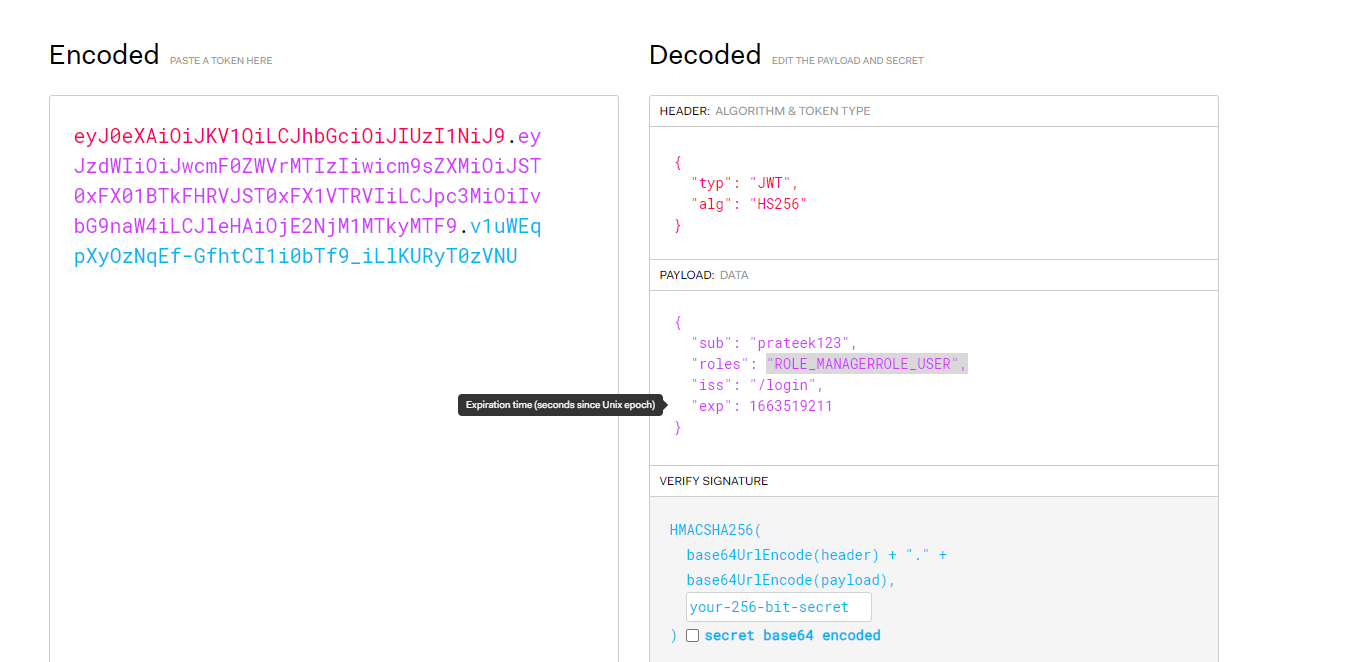
Before saving the user, we are encoding the password of the user.



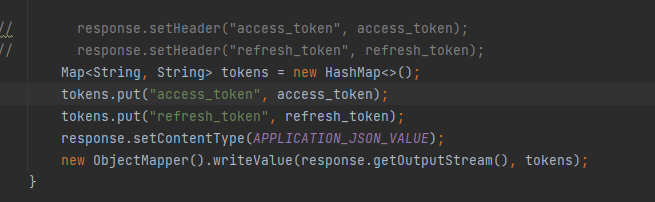
Here we have to pass the username and the password as the parameter because of the following:

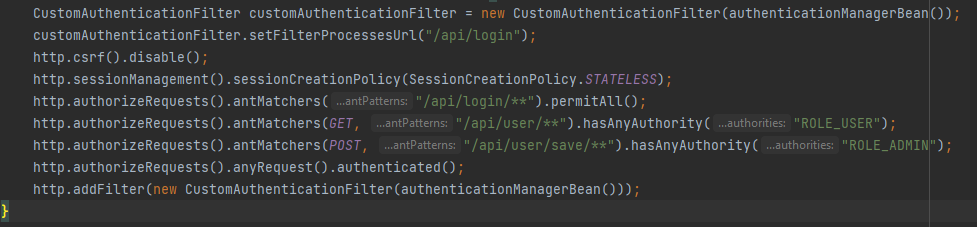


But if we give some other user that is not stored in the database, then it will throw an error of ***not being authorized.***



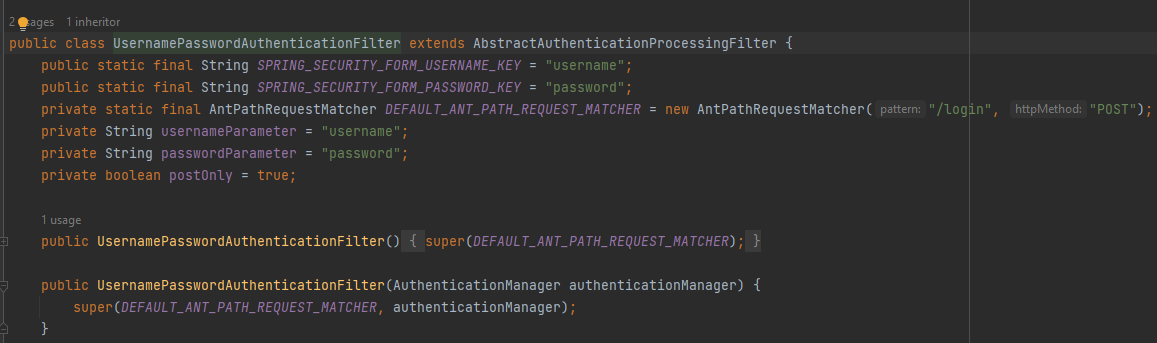
We can also go to the <https://jwt.io/> to see what ***access-token*** actually contains.





Now there are couples of things to understand:

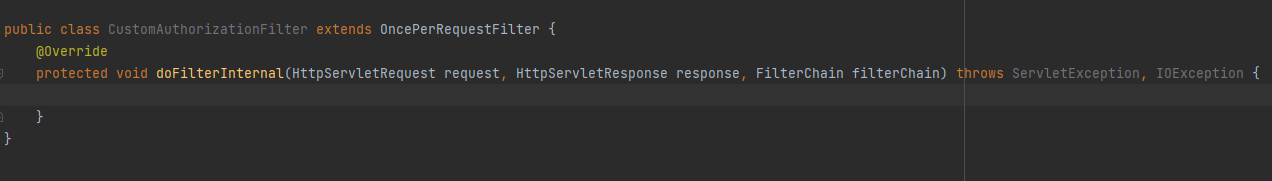
1. We can change the url ***/login*** to something else which gives us the login screen. We can do that with the help of ***CustomAuthenticatioFilter*** object since it implements ***UserPasswordAuthenticationFilter***.



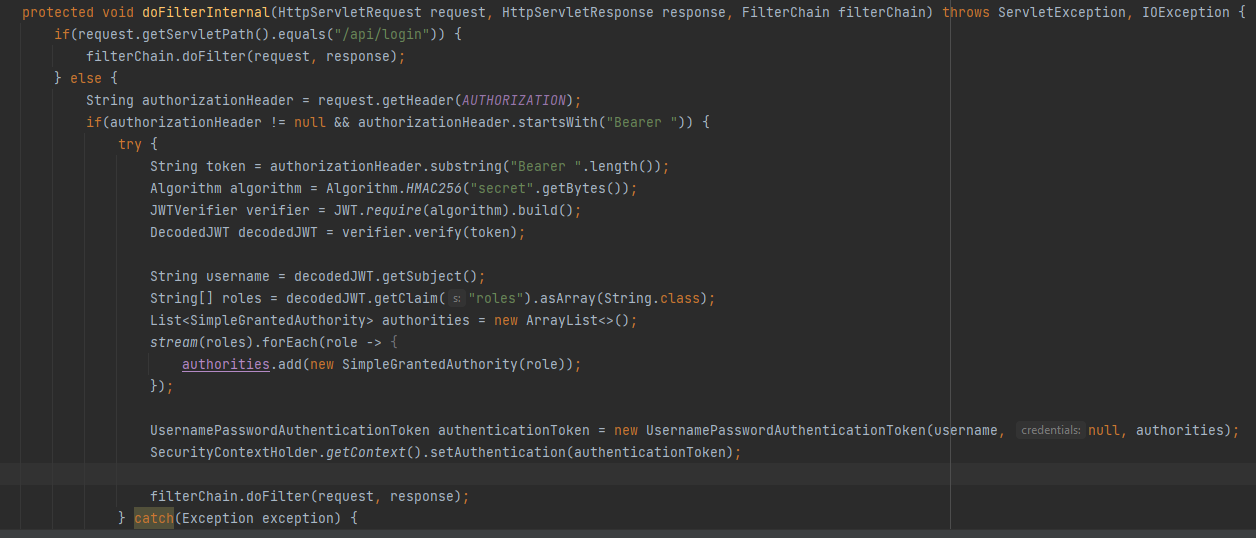
1. Here earlier we were permitting all the requests which will come to the server. So, there was no security as we were allowing every user to access any resource on the application. But now with the help of ***antMatchers*** we can specify the urls which can be accessed and by whom it can be accessed.
2. Also the order of writing these ***antMatchers*** is very important.

Now, after the user is authenticated, we are providing the user with an ***access\_token*** and a ***refresh\_token***. Now what we want is that to access the resources/paths in our application, the user should pass this token to his request to the server. Then the server would check if this is a valid token and has the permissions to access that resource. If it has then it will give the response.

For this to work, we need an ***authorization filter.***



Every request that will be coming from the user will be intercepted by ***OncePerRequestFilter*** and all the logic to handle the request i.e. to provide the access to the resource or not, will be done inside the ***addFilterInternal*** method.



Every access-token that we are going to send with the request, should start with ***Bearer <space>access\_toke***n. This means that the one who is sending the request is the owner of the access\_token and once they check if the token is valid, no further validations will be done.

