Spring Security

7 more properties

GitHub Repository

You can use this repository as an reference for understanding spring security-

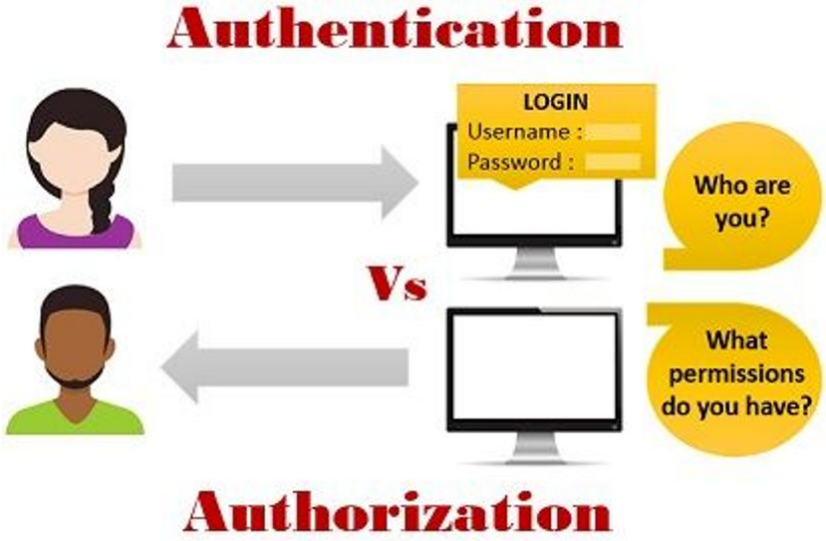
https://github.com/Arora-Shivam/MasaiSpringSecurity

Introduction

Spring Security provides comprehensive security services for J2EE-based enterprise software
applications.

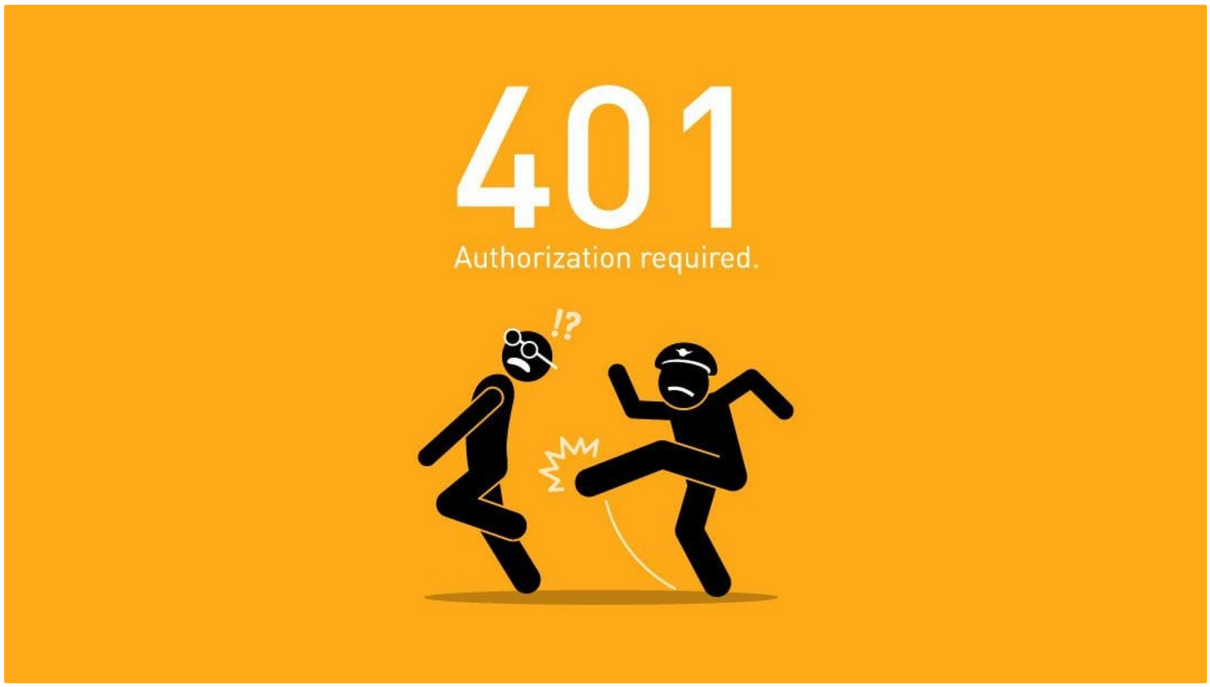
Spring Security is a framework that provides various security features like authentication, and authorization to create secure Java Enterprise Applications.

This framework targets two major areas of application - authentication and authorization.



Authentication

Authentication is how we verify the identity of who is trying to access a particular resource. A common way to authenticate users is by requiring the user to enter a username and password. Once authentication is performed we know the identity and can perform authorization. Http response code in case of failed authentication is 401.



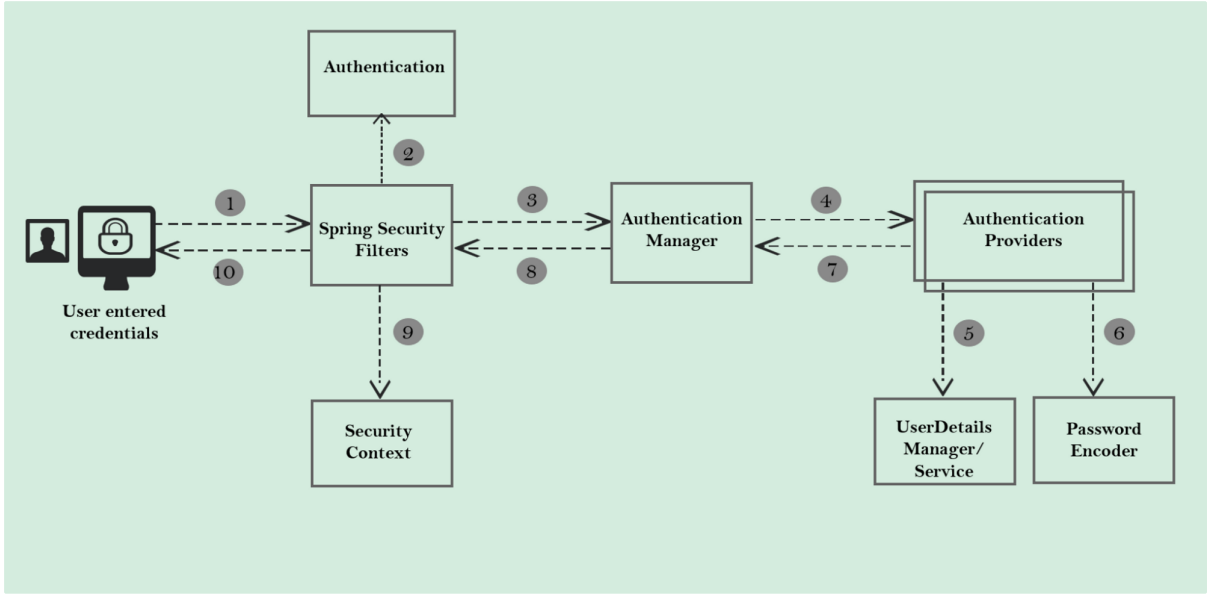
Authorization

Authorization is the process to allow the authority to perform actions in the application. We can apply authorization to authorize web requests, methods, and access to the individual domain. Http response code in case of failed authorization is 403.

Spring Security Dependency for Maven

```
<dependency> <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId> </dependency>
```

Spring Security Components



Spring Security Filters

A series of Spring Security filters intercept each request & work together to identify if Authentication is required or not. If Authentication is required, accordingly navigate the user to the login page or use the existing details stored during initial authentication.

Authentication

Filters like UsernamePasswordAuthenticationFilter will extract username/password from HTTP request & prepare Authentication type object. Because Authentication is the core standard of storing authenticated user details inside the Spring Security framework.

AuthenticationManager

Once received a request from the filter, it delegates the validation of the user details to the authentication providers available. Since there can be multiple providers inside an app, it is the responsibility of the Authentication Manager to manage all the authentication providers available.

AuthenticationProvider

Authentication Providers have all the core logic of validating user details for authentication.

UserDetailsManager/UserDetailsService

UserDetailsManager/UserDetailsService helps in retrieving, creating, updating, and deleting the User Details from the DB/storage systems. It is one of the core interfaces of Spring Security. The authentication of any request mostly depends on the implementation of the UserDetailsService interface. It is most commonly used in database-backed authentication to retrieve user data. The data is retrieved with the implementation of the lone loadUserByUsername() method where we can provide our logic to fetch the user details for a user. The method will throw a UsernameNotFoundException if the user is not found.

PasswordEncoder

Service interface that helps in encoding & hashing passwords. Otherwise, we may have to live with plain text passwords.

SecurityContext

Once the request has been authenticated, the Authentication will usually be stored in a thread-local SecurityContext managed by the SecurityContextHolder. This helps during the upcoming requests from the same user.

Spring Security Flow

1) User trying to access a secure page for the first time

2)Behind the scenes few filters like AbstractSecurityInterceptor, DefaultLoginPageGeneratingFilter

identify that the user is not logged in & redirect the user to the login page

3) User entered his credentials and the request is intercepted by filters

4)Filters like UsernamePasswordAuthenticationFilter, extract the username, and password from the request and form an object of UsernamePasswordAuthenticationToken which is an implementation of the Authentication interface. With the object created it invokes authenticate() method of ProviderManager.

5) ProviderManager which is an implementation of AnthenticationManager, identifies the list of Authentication providers available that are supporting a given authentication object style. In the default behavior, authenticate( ) method of the DaoAuthenticationProvider will be invoked by ProviderManager.

6) DaoAuthenticationProvider invokes the method loadUserByUsername() of UserDetailsService to load the user details. Once the user details are loaded, it takes help from the default password encoder implementation to compare the password and validate if the user is authentic or not.

7) At last it returns the Authentication object with the details of the authentication

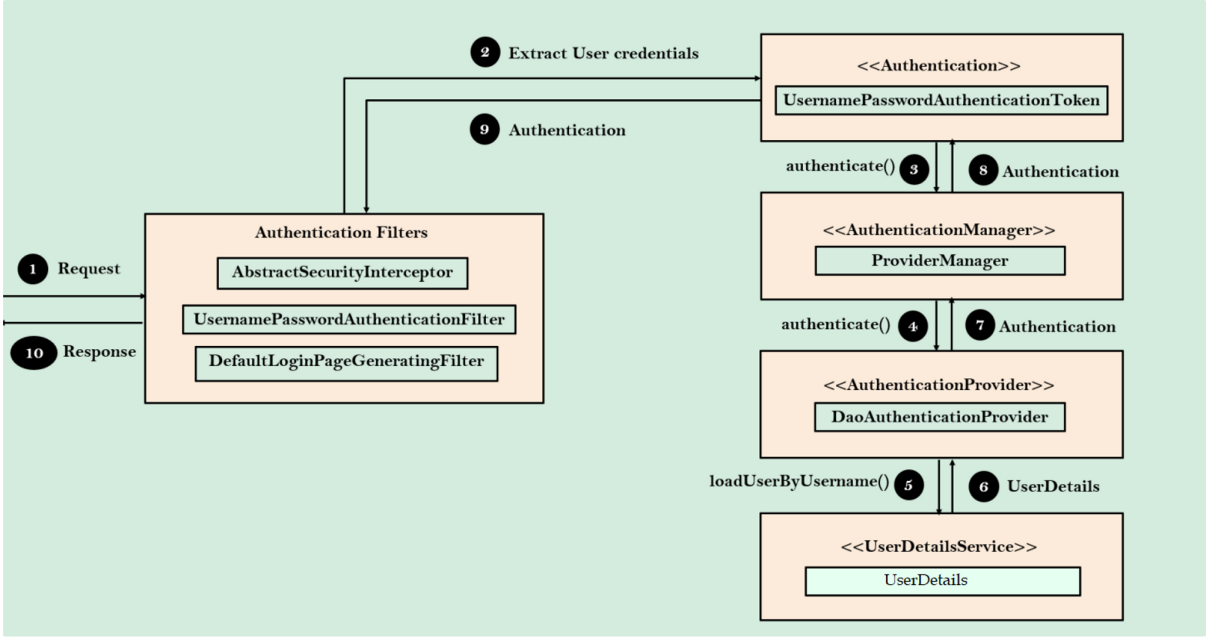
success or not to ProviderManager.

8) ProviderManager checks if authentication is successful or not. If not, it will try with

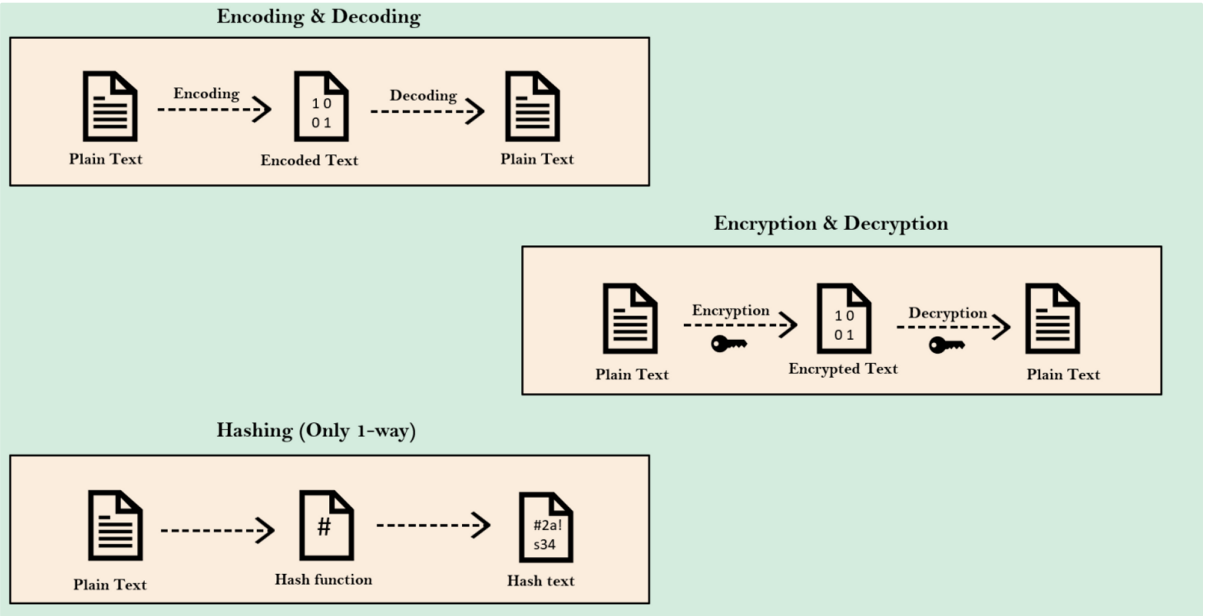
other available AuthenticationProviders Otherwise, it simply returns the authentication details to the filters.

9) The Authentication object is stored in the SecurityContext object by the filter

for future use and the response will be returned to the end user.



Different Ways of Password Management



Encoding

Encoding is defined as the process of converting data from one form to another and has nothing to do with cryptography. It involves no secret and is completely reversible. Encoding can't be used for securing data. Below are the various publicly available algorithms used for encoding

Ex: ASCII, BASE64, UNICODE

Encryption

Encryption is defined as the process of transforming data in such a way that guarantees confidentiality. To achieve confidentiality encryption requires the use of a secret which, in

cryptographic terms, we call a "key".

Encryption can be reversible by using decryption with the help of the "key". As long as the "key" is

confidential, encryption can be considered secure.

Hashing

In hashing, data is converted to the hash value using some hashing function. Data once hashed is non-reversible. One cannot determine the original data from a hash value generated. Given some arbitrary data along with the output of a hashing algorithm, one can verify whether this data matches the original input data without needing to see the original data.

Several algorithms have been developed especially for password hashing:

- bcrypt
- scrypt
- PBKDF2
- NoOpPasswordEncoder
- argon2

Spring Security Project with default Configuration

Step1- Create a Spring Boot Project

Step 2 - Add the following Dependencies

- **Spring Web** – It bundles all dependencies related to web development including Spring MVC, REST, and an embedded Tomcat Server.
- **Spring Security** – For the implementation of security features provided by Spring Security.

Step 3 - We will create a Controller

```
@RestController @RequestMapping("/masai") public class WelcomeController { @GetMapping("/welcome") public ResponseEntity<String> welcome(){ return new ResponseEntity<String>("Welcome to Masai App without security",HttpStatus. ACCEPTED); } @GetMapping("/welcomeP") public ResponseEntity<String> welcomeP(){ return new ResponseEntity<String>("Welcome to Masai App with Security",HttpStatus .ACCEPTED); } @GetMapping("/admin") public ResponseEntity<String> admin(){ return new ResponseEntity<String>("Welcome to Masai App for Admin",HttpStatus. ACCEPTED); } }
```

Requirements-

- Services with Security-

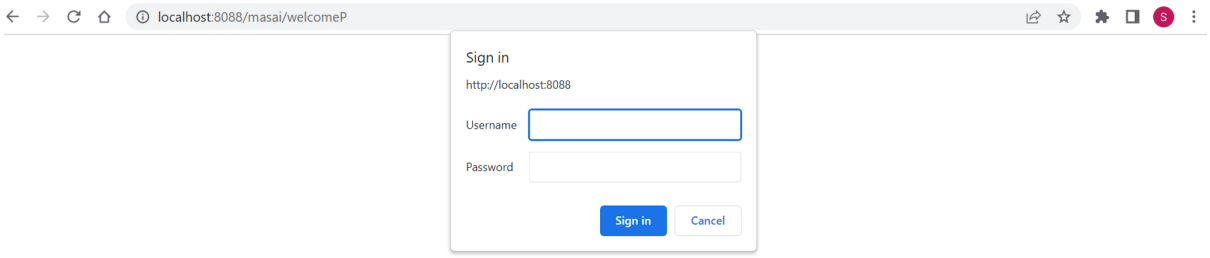
```
/masai/welcomeP
```

- Services without Security-

```
/masai/welcome
```

When we add Spring Security to an existing Spring application it adds a login form and sets up a dummy user. This is Spring Security in auto-configuration mode. In this mode, it also sets up the default filters, authentication managers, authentication providers, and so on. This setup is an in-memory authentication setup.

By default, Spring Security Framework protects all the paths present inside the web application and the login form will ask for a username and password. The username is “user” and Password will be auto-generated.



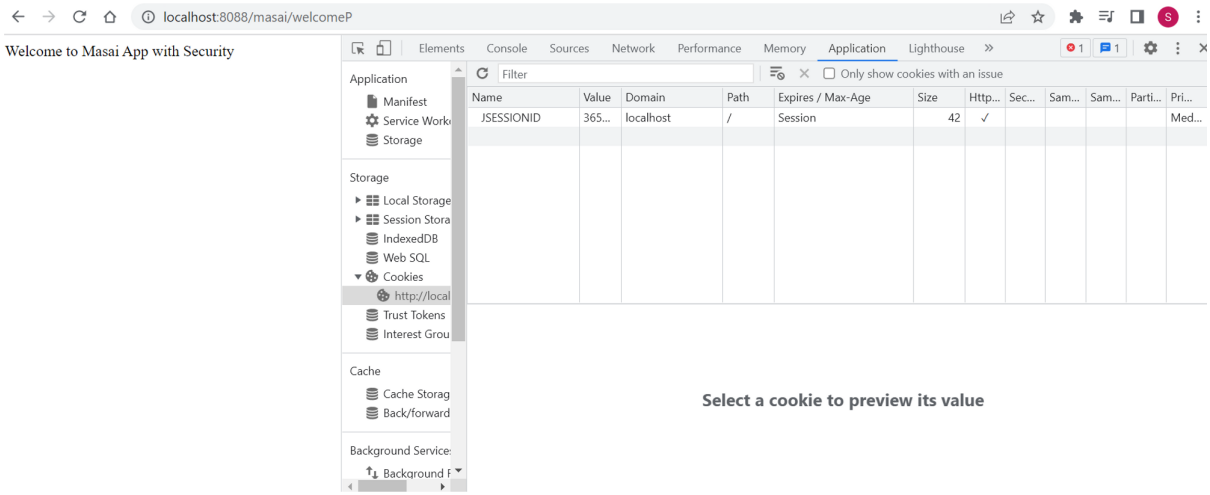
Step 4- Custom Security Configuration

We can secure the web application APIs path as per our custom requirements-

```
@Configuration public class ProjectSecurityConfig { @Bean public Security
FilterChain masaiSecurityConfig(HttpSecurity http) throws Exception { htt
p.authorizeHttpRequests( (auth)->auth .requestMatchers("/masai/welcome
P").authenticated() .requestMatchers("/masai/welcome").permitAll() ).http
Basic(); return http.build(); } @Bean public PasswordEncoder passwordEnco
der() { return NoOpPasswordEncoder.getInstance(); //Not recommended } }
```

Spring Security Remember Me Authentication

Once the user is authenticated and if the user try to access the protected URL again then Spring Security will not ask user to login again because once a user is authenticated a JSESSIONID cookie is generated.The next visit will read this browser cookie and if the cookie is valid, it will perform the auto login.

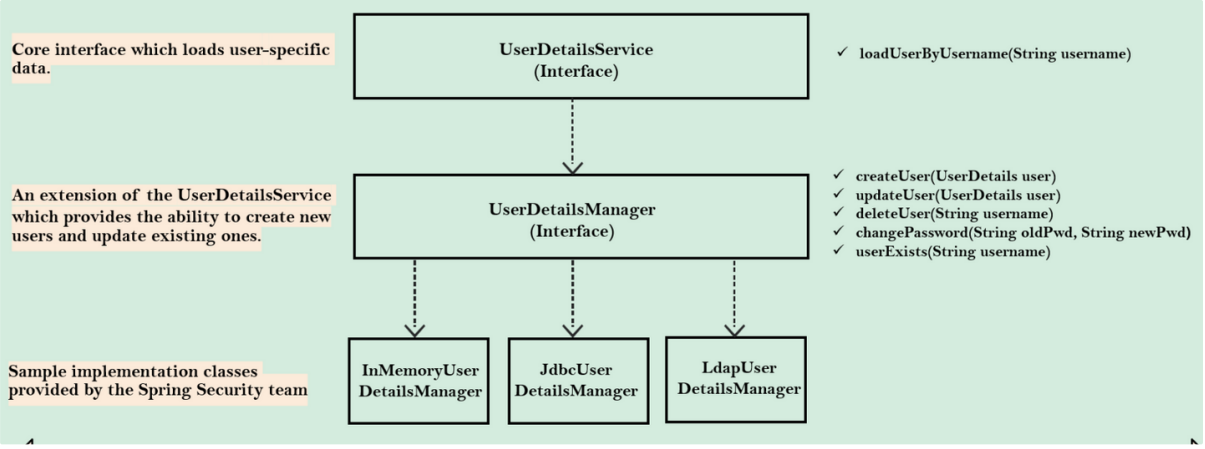


Spring Security Project with Custom User using InMemoryUserDetailsManager

Step1, Step2 & Step 3 are the same.

Step 4- Custom Security Configuration with Custom User

It is very difficult to remember the auto-generated password because this is a random password and Spring Security generates a random password every time when we execute the Spring Application. If we want to make our Spring application for multiple users it is difficult to configure their credentials. So to overcome this situation when we handle multiple authentications along with their respective roles. We will use in-memory authentication in the Spring Application.




```
@Configuration public class ProjectSecurityConfig { @Bean public Security
FilterChain masaiSecurityConfig(HttpSecurity http) throws Exception { htt
p.authorizeHttpRequests( (auth)->auth .requestMatchers("/masai/welcome
P").authenticated() .requestMatchers("/masai/welcome").permitAll() ).http
Basic(); return http.build(); } @Bean public InMemoryUserDetailsManager u
serDetails() { InMemoryUserDetailsManager userDetailsService = new InMemo
ryUserDetailsManager(); UserDetails admin = User.withUsername("admin").pa
ssword("12345").authorities("admin").build(); UserDetails user = User.wit
hUsername("user").password("12345").authorities("read").build(); userDeta
ilsService.createUser(admin); userDetailsService.createUser(user); return
userDetailsService; } @Bean public PasswordEncoder passwordEncoder() { re
turn NoOpPasswordEncoder.getInstance(); } }
```

UserDetails

UserDetails is an interface in Spring Security. This holds the detail of the user.

```
UserDetails admin =
User.withUsername("admin").password("12345").authorities("admin").build();
```

Here, we have created a user admin.

User is a class that implements UserDetails and it is provided by spring security and here we created the object of User with username “admin” and password “12345” and authorities “admin”.

UserDetailsService

UserDetailsService is used by DaoAuthenticationProvider for retrieving a username, a password, and other attributes for authenticating with a username and password. Spring Security provides in-memory and JDBC implementations of UserDetailsService.

.

UserDetailsService is used to create a UserDetails when the username is passed. It has only one method-

```
UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException;
```

This returns the UserDetails which is then passed to Authentication Provider which further checks whether the user credentials.

InMemoryUserDetailsManager

InMemoryUserDetailsManager internally stores and retrieve the user-related information which is required for Authentication. InMemoryUserDetailsManager indirectly implements the UserDetailsService interface.

Now the InMemoryUserDetailsManager reads the in-memory hashmap and loads the UserDetails
by calling the loadUserByUsername() method.

PasswordEncoder

```
@Bean public PasswordEncoder passwordEncoder() { return
NoOpPasswordEncoder.getInstance(); }
```

Here, we are creating a Bean of type NoOpPasswordEncoder (deprecated), this is not recommended for production-level applications although for simplicity we have used this. In the next section, we will use BCryptPasswordEncoder and hash the Password before storing it in the Database.

Methods inside PasswordEncoder Interface.

```
public interface PasswordEncoder {

        String encode(CharSequence rawPassword);

        boolean matches(CharSequence rawPassword, String encodedPassword);

        default boolean upgradeEncoding(String encodedPassword) {
                return false;
        }
}
```

Spring Security Project with SpringDataJPA

Step1- Create a Spring Boot Project

Step 2 - Add the following Dependencies

- **Spring Web** – It bundles all dependencies related to web development including Spring MVC, REST, and an embedded Tomcat Server.
- **Spring Security** – For the implementation of security features provided by Spring Security.
- **Spring Data JPA** – In addition to using all features defined by JPA specification, Spring Data JPA adds its own features such as the no-code implementation of the repository pattern and the creation of database queries from the method name.
- **Mysql Driver** – For the MySQL database driver.

Step 3 - We will create an Employee Entity

```
@Data @Entity @AllArgsConstructor @NoArgsConstructor public class Employe
e{ @Id @GeneratedValue(strategy = GenerationType.AUTO) private int id; pr
ivate String userName; private String address; private String password; p
rivate String role; }
```

Step 4- Create your own implemented class of UserDetails

```
public class MasaiSecurityUser implements UserDetails{ private Employee e
mployee; public MasaiSecurityUser(Employee emp) { this.employee=emp; } @O
verride public Collection<? extends GrantedAuthority> getAuthorities() {
List<GrantedAuthority> grantedAuthorityList = new ArrayList<>(); grantedA
uthorityList.add(new SimpleGrantedAuthority(employee.getRole())); return
grantedAuthorityList; } @Override public String getPassword() { // TODO A
uto-generated method stub return employee.getPassword(); } @Override publ
ic String getUsername() { // TODO Auto-generated method stub return emplo
yee.getUserName(); } @Override public boolean isAccountNonExpired() { //
TODO Auto-generated method stub return true; } @Override public boolean i
sAccountNonLocked() { // TODO Auto-generated method stub return true; } @
Override public boolean isCredentialsNonExpired() { // TODO Auto-generate
d method stub return true; } @Override public boolean isEnabled() { // TO
DO Auto-generated method stub return true; } }
```

Step 5 - Create your own implemented class of UserDetailsService

Here, we will override loadUserByUsername and Obtain the user details with the help of DataJPA and pass it to the DaoAuthenticationProvider.

```
@Service public class MasaiUserDetailService implements UserDetailsServic
e{ @Autowired private EmployeeDao employeeDao; @Override public UserDetai
ls loadUserByUsername(String username) throws UsernameNotFoundException {
// TODO Auto-generated method stub List<Employee> employee=employeeDao.fi
ndByUserName(username); if(employee.size()>0) { return new MasaiSecurityU
ser(employee.get(0)); } else { throw new UsernameNotFoundException("No Us
er Exist with this username"); } } }
```

Step 6- Custom Security Configuration

```
@Configuration public class ProjectSecurityConfig { @Bean public Security
FilterChain masaiSecurityConfig(HttpSecurity http) throws Exception { htt
p.authorizeHttpRequests( (auth)->auth .antMatchers("/masai/employee/profi
le","/masai/welcomeP").authenticated() .antMatchers("/masai/admin").hasRo
le("ADMIN") .antMatchers("/masai/employee/register","/masai/welcome").per
mitAll() ).csrf().disable() .httpBasic(); return http.build(); } @Bean pu
blic PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder
(); } }
```

Authorization


```
http.authorizeHttpRequests( (auth)->auth
.antMatchers("/masai/employee/profile","/masai/welcomeP").authenticated()
.antMatchers("/masai/admin").hasRole("ADMIN")
.antMatchers("/masai/employee/register","/masai/welcome").permitAll()
).csrf().disable() .httpBasic();
```

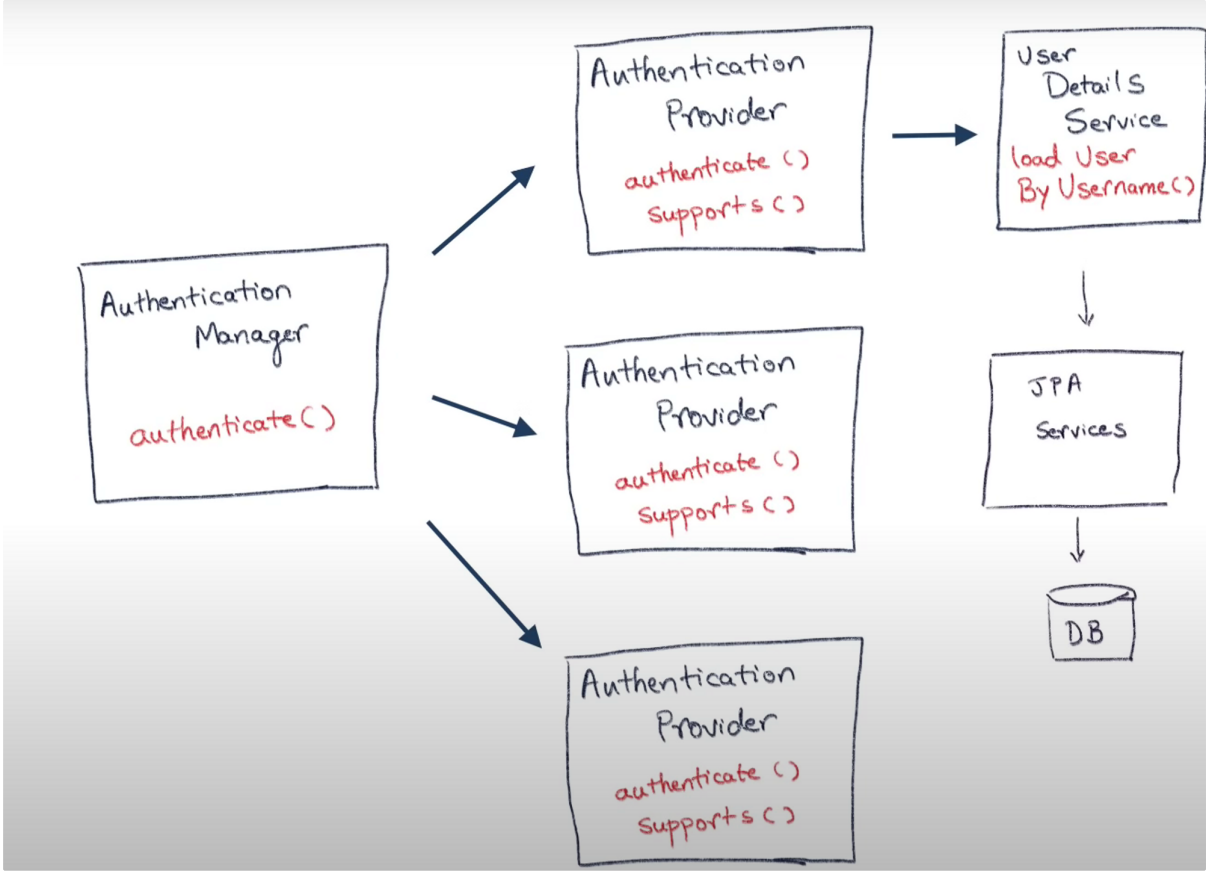
Here, we are specifying that the Api "/masai/admin" will allow only those user which have the role of "ADMIN".

SecurityContextHolder

The most fundamental object is SecurityContextHolder. This is where we store details of the
present security context of the application, which includes details of the principal currently using the application

Inside the SecurityContextHolder we store details of the principal currently interacting with
the application. Spring Security uses an Authentication object to represent this information.

Internal Flow



DaoAuthenticationProvider is an AuthenticationProvider implementation that uses a UserDetailsService and PasswordEncoder to authenticate a username and password.

The authentication Filter from the Reading the Username & Password section passes a UsernamePasswordAuthenticationToken to the AuthenticationManager, which is implemented by ProviderManager.

The ProviderManager is configured to use an AuthenticationProvider of type DaoAuthenticationProvider. DaoAuthenticationProvider looks up the UserDetails from the UserDetailsService.

DaoAuthenticationProvider uses the PasswordEncoder to validate the password on the UserDetails returned in the previous step.

When authentication is successful, the Authentication that is returned is of type UsernamePasswordAuthenticationToken and has a principal that is the UserDetails returned by the configured UserDetailsService. Ultimately, the returned UsernamePasswordAuthenticationToken is set on the SecurityContextHolder by the authentication Filter.

Cross-Origin Resource Sharing (CORS)

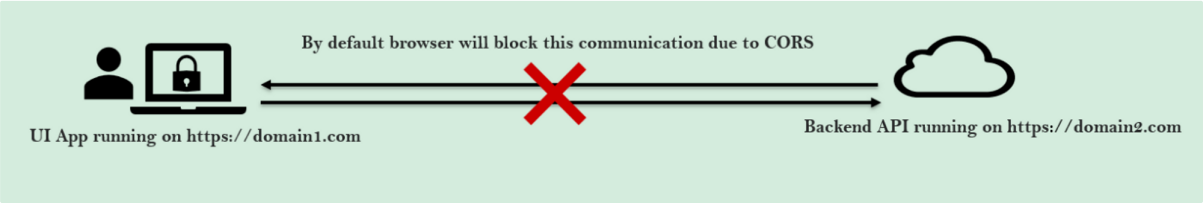
CORS (Cross-Origin Resource Sharing) is a mechanism by which data or any other resource of a site could be shared intentionally with a third-party website when there is a need. Generally, access to resources that are residing in a third-party site is restricted by the browser clients for security purposes.

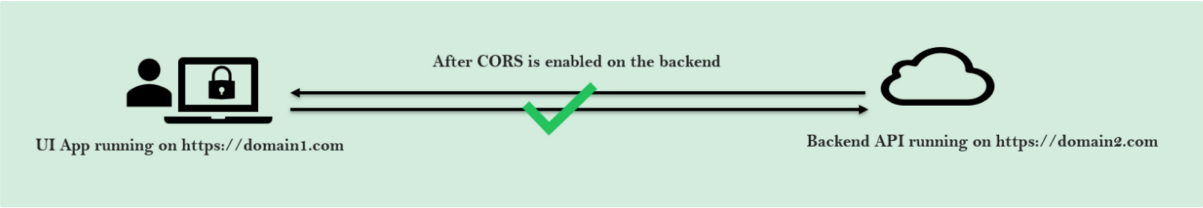
By default, the browser does not allow to share of data between different origins.

In this context, "other origins" means-

- a different scheme (HTTP or HTTPS)
- a different domain
- a different port

However, there are legitimate scenarios where cross-origin access is desirable or even necessary. For example, if a UI App wishes to make a call to API call on a different domain, it would be blocked by doing so by default due to CORS.





Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is one of the most severe vulnerabilities which can be exploited in various ways- from changing a user's info without his knowledge to gaining full access to the user's account.

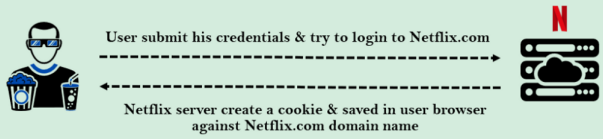
By Default Spring Security gives protection against CSRF attacks by not allowing POST and PUT requests to execute and you will get an HTTP 403 error code.

So here, we are disabling CSRF so that we can POST a request. (Not a recommended approach)

```
http.authorizeHttpRequests( (auth)->auth
.antMatchers("/masai/employee/profile","/masai/welcomeP").authenticated()
.antMatchers("/masai/admin").hasRole("ADMIN")
.antMatchers("/masai/employee/register","/masai/welcome").permitAll()
).csrf().disable() .httpBasic();
```

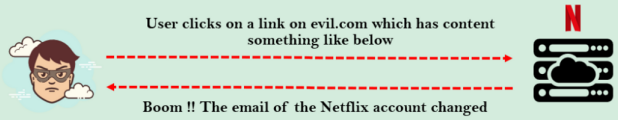
CSRF Attack

Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com



Step 2 : The same Netflix user opens an evil.com website in another tab of the browser.

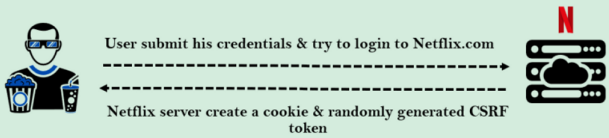
Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com, the backend server of Netflix.com can't differentiate from where the request came. So here the evil.com forged the request as if it is coming from a Netflix.com UI page.



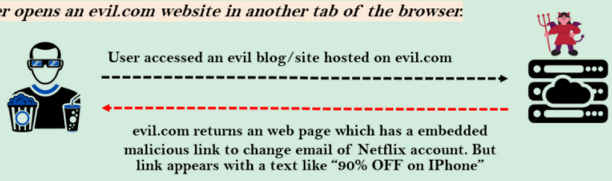
The solution to CSRF Attack

- To defeat a CSRF attack, applications need a way to determine if the HTTP request is legitimately generated via the application's user interface. The best way to achieve this is through a CSRF token. A CSRF token is a secure random token that is used to prevent CSRF attacks. The token needs to be unique per user session and should be of large random value to make it difficult to guess.
- Let's see how this solve CSRF attack by taking the previous Netflix example again,

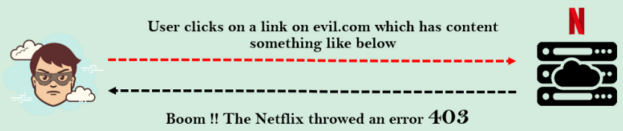
Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com along with a randomly generated unique CSRF token for this particular user session. CSRF token is inserted within hidden parameters of HTML forms to avoid exposure to session cookies.



Step 2 : The same Netflix user opens an evil.com website in another tab of the browser.

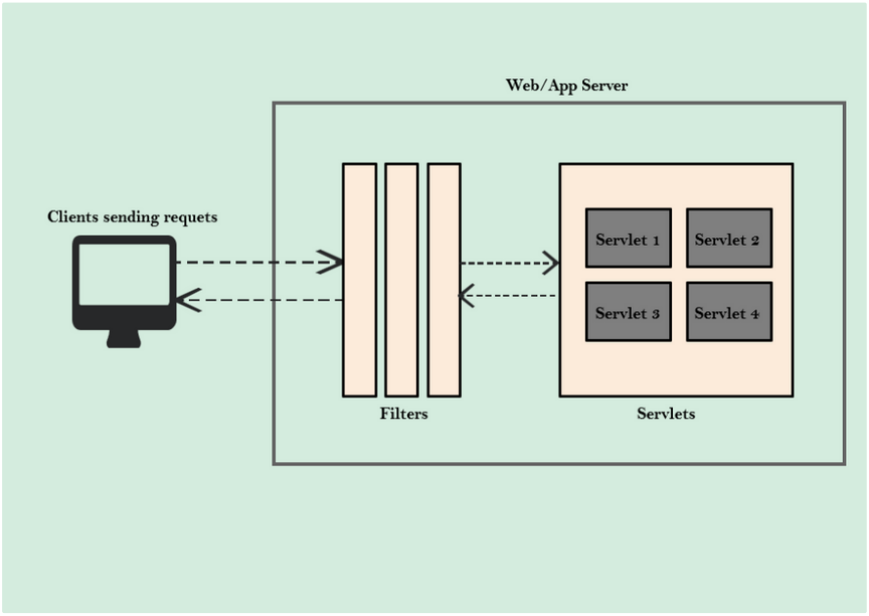


Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com. This time the Netflix.com backend server expects CSRF token along with the cookie. The CSRF token must be same as initial value generated during login operation



The CSRF token will be used by the application server to verify the legitimacy of the end-user request if it is coming from the same App UI or not. The application server rejects the request if the CSRF token fails to match the test.

Filters In Spring Security



Filters inside java web applications can be used to intercept each request/response and do some pre-work before our business logic.

✓ We can always check the registered filters inside Spring Security with the below configurations,

1. @EnableWebSecurity(debug = true) – We need to enable the debugging of the security details
2. Enable logging of the details by adding the below property in application.properties

```
logging.level.org.springframework.security.web.FilterChainProxy=DEBUG
```

Some of the internal filters that are executed by spring security in the authentication flow

Security filter chain: [
DisableEncodeUrlFilter
WebAsyncManagerIntegrationFilter
SecurityContextHolderFilter
HeaderWriterFilter
LogoutFilter
BasicAuthenticationFilter
RequestCacheAwareFilter
SecurityContextHolderAwareRequestFilter
AnonymousAuthenticationFilter
ExceptionTranslationFilter
AuthorizationFilter
]

Implementing Custom Filter

We can implement our own custom filter by implementing the Filter interface from jakarta.servlet package.

```
public class MyCustomFilterBeforeBasicAuthenticationFilter extends
OncePerRequestFilter{ @Override protected void
doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain) throws ServletException, IOException {
System.out.println("We are inside
MyCustomFilterBeforeBasicAuthenticationFilter");
filterChain.doFilter(request, response); } }
```

Here, we have created our own filter by extending the implemented class of filter interface i.e OncePerRequestFilter class then we have overridden this doFilterInternal method to write our logic.

This method accepts 3 parameters HttpServletRequest, HttpServletResponse & FilterChain.

ServletRequest - It represents the HTTP request. We use the ServletRequest object to retrieve details about the request from the client.

ServletResponse - It represents the HTTP response. We use the ServletResponse object to modify the response before sending it back to the client or further along the filter chain.

FilterChain - The filter chain represents a collection of filters with a defined order in which they act. We use the FilterChain object to forward the request to the next filter in the chain.

We can add your own filter to the spring security filter chain either before, or after by using the below methods.

- *addFilterBefore(filter, class)* adds a *filter* before the position of the specified filter *class*.
- *addFilterAfter(filter, class)* adds a *filter* after the position of the specified filter *class*.
- *addFilterAt(filter, class)* adds a *filter* at the location of the specified filter class.

```
http.authorizeHttpRequests( (auth)->auth
.requestMatchers("/masai/welcomeP").authenticated()
.requestMatchers("/masai/employee/admin").hasRole("admin")
.requestMatchers("/masai/welcome","/masai/employee/register").permitAll()
).addFilterBefore(new MyCustomFilterBeforeBasicAuthenticationFilter(),
BasicAuthenticationFilter.class) .csrf().disable() .httpBasic();
```

Security filter chain: [
DisableEncodeUrlFilter
WebAsyncManagerIntegrationFilter
SecurityContextHolderFilter
HeaderWriterFilter
LogoutFilter
MyCustomFilterBeforeBasicAuthenticationFilter
BasicAuthenticationFilter
RequestCacheAwareFilter
SecurityContextHolderAwareRequestFilter
AnonymousAuthenticationFilter
ExceptionTranslationFilter
AuthorizationFilter
]

Now, you can see it in the console. we have added **MyCustomFilterBeforeBasicAuthenticationFilter** before the BasicAuthenticationFilter.

JWT(JSON Web Token)

JWT or JSON Web Token is an open standard used to share security information between two parties — a client and a server. Each JWT contains encoded JSON objects, including a set of claims. JWTs are signed using a cryptographic algorithm to ensure that the claims cannot be altered after the token is issued.

The token is mainly composed of a **header, payload, and signature**. These three parts are separated by period(.).

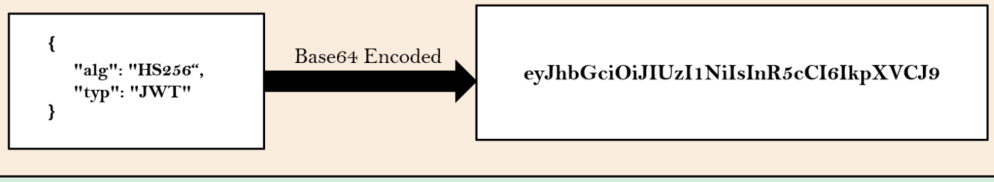
A JWT token has 3 parts each separated by a period(.) Below is a sample JWT token,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

1. Header
2. Payload
3. Signature (Optional)

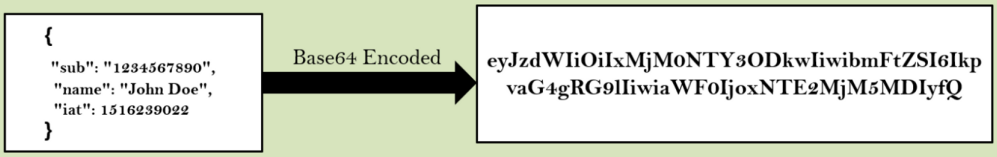
Header

✓ Inside the JWT header, we store metadata/info related to the token. If I chose to sign the token, the header contains the name of the algorithm that generates the signature.



Payload

✓ In the body, we can store details related to user, roles etc. which can be used later for AuthN and AuthZ. Though there is no such limitation what we can send and how much we can send in the body, but we should put our best efforts to keep it as light as possible.



Signature

To make sure no one tempered the data on the network, we can add the signature of the content when initially token is generated. To create the signature part you have to take the encoded header,the encoded payload,a secret key, the algorithm specified in the header, and sign that.

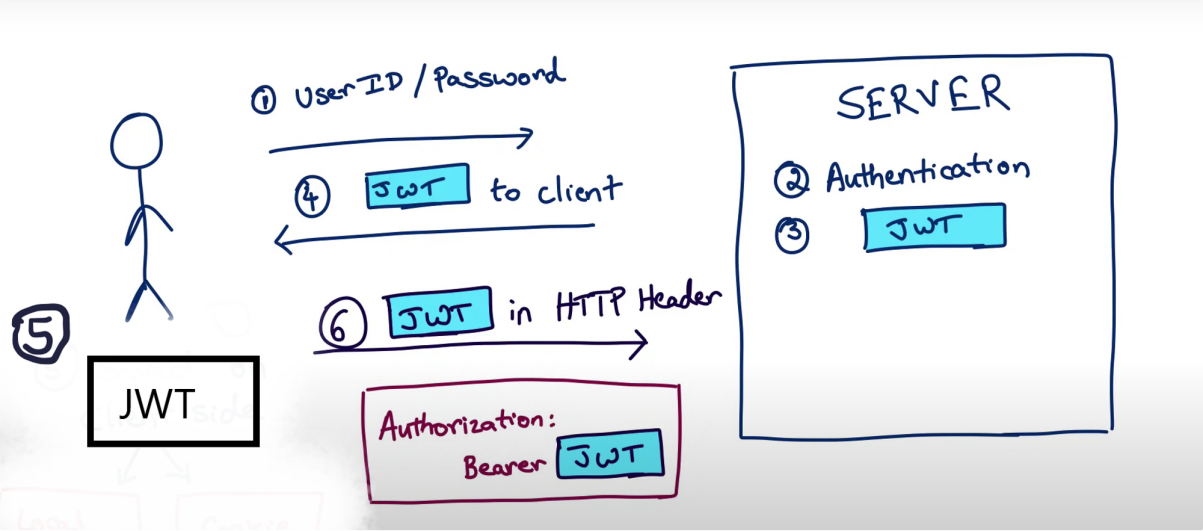
- ✓ For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

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

- ✓ The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

Working of JWT

- User sign-in using username and password.
- Authentication server verifies the credentials and issues a jwt signed using a private key.
- Client uses the JWT to access protected resources by passing the JWT in HTTP Authorization header.
- Resource server then verifies the authenticity of the token using the secret key.



Jwt Token

You can refer to this website to understand how JWT works internally - https://jwt.io/

This is the JWT token generated when we logged in to the system and we have analyzed the same from the above website.