Spring Security

Contents

[Getting Started 5](#_Toc69484271)

[Introduction to Security 5](#_Toc69484272)

[Spring Security Flow 5](#_Toc69484273)

[Changing the default Security Configuration 7](#_Toc69484274)

[Defining and Managing Users 8](#_Toc69484275)

[Configuring Users using inMemoryAuthentication 8](#_Toc69484276)

[Configuring Users using InMemoryUserDetailsManager 9](#_Toc69484277)

[Understanding User Management interfaces and classes 10](#_Toc69484278)

[Deep Dive of UserDetails interface 11](#_Toc69484279)

[Deep Dive of UserDetailsService interface 11](#_Toc69484280)

[Deep Dive of UserDetailsManager interface 11](#_Toc69484281)

[Deep Dive of UserDetailsManager implementations 11](#_Toc69484282)

[Creating Custom Implementation of UserDetailsService 12](#_Toc69484283)

[Notes 12](#_Toc69484284)

[Password Management with Password Encoders 13](#_Toc69484285)

[How our passwords are validated by default 13](#_Toc69484286)

[Steps 13](#_Toc69484287)

[Issues with this (default) approach 13](#_Toc69484288)

[Encoding Vs Encryption Vs Hashing 14](#_Toc69484289)

[How our Passwords will be validated with Hashing 15](#_Toc69484290)

[Steps 16](#_Toc69484291)

[Password Encoder 16](#_Toc69484292)

[NoOpPasswordEncoder 17](#_Toc69484293)

[StandardPasswordEncoder 18](#_Toc69484294)

[BCryptPasswordEncoder and SCryptPasswordEncoder 18](#_Toc69484295)

[Pbkdf2PasswordEncoder 19](#_Toc69484296)

[Using bcrypt password encoder 19](#_Toc69484297)

[Spring Security Crypto package 20](#_Toc69484298)

[Understanding Authentication Provider 21](#_Toc69484299)

[Role of AuthenticationProvider in the Spring Security flow 21](#_Toc69484300)

[Scenarios where we need to implement Authentication Provider 21](#_Toc69484301)

[Understanding AuthenticationProvider Definition 22](#_Toc69484302)

[Understanding Authentication and Principal interfaces 23](#_Toc69484303)

[Authentication vs UserDetails 23](#_Toc69484304)

[Implementing and Customizing the Authentication Provider 24](#_Toc69484305)

[Understanding CORS and CSRF 25](#_Toc69484306)

[Deep dive into CORS 25](#_Toc69484307)

[Resolving CORS issue 26](#_Toc69484308)

[Deep dive into CSRF 27](#_Toc69484309)

[Resolving CSRF Issue 28](#_Toc69484310)

[Notes 29](#_Toc69484311)

[Understanding & Implementing Authorization 30](#_Toc69484312)

[Authentication Vs Authorization 30](#_Toc69484313)

[Spring Security Internal flow for Authentication & Authorization 30](#_Toc69484314)

[How Authorities stored in Spring Security 31](#_Toc69484315)

[Configuring Authorities in Spring Security 32](#_Toc69484316)

[Authority Vs Role 33](#_Toc69484317)

[Configuring Roles in Spring Security 34](#_Toc69484318)

[Deep dive of Ant, MVC, Regex matchers for applying restrictions on the paths 35](#_Toc69484319)

[Notes 36](#_Toc69484320)

[Filters in Spring Security 37](#_Toc69484321)

[Introduction 37](#_Toc69484322)

[Inbuilt Filters provided by Spring Security 37](#_Toc69484323)

[Custom filters in Spring Security 38](#_Toc69484324)

[Implementing Custom Filter 38](#_Toc69484325)

[Adding Custom Filter in the filter chain 38](#_Toc69484326)

[Details about GenericFilterBean and OncePerRequestFilter 41](#_Toc69484327)

[GenericFilterBean 41](#_Toc69484328)

[OncePerRequestFilter 41](#_Toc69484329)

[Notes 42](#_Toc69484330)

[Token based Authentication using JSON Web Token (JWT) 43](#_Toc69484331)

[Introduction to Tokens in Authentication flow 43](#_Toc69484332)

[Advantages of Token based Authentication 43](#_Toc69484333)

[Exploring the JSESSIONID & CSRF Tokens inside our application 44](#_Toc69484334)

[Deep dive into JWT Tokens 45](#_Toc69484335)

[Using JWT in the Application 48](#_Toc69484336)

[Step 1: Making Configuration changes 48](#_Toc69484337)

[Step 2: Configure Filters to generate and validate JWT Tokens 50](#_Toc69484338)

[Step 3: Making Changes at Front end / Client side 51](#_Toc69484339)

[Notes 52](#_Toc69484340)

[Method Level Security 53](#_Toc69484341)

[Introduction 53](#_Toc69484342)

[Details about **method invocation authorization** in method level security 54](#_Toc69484343)

[Implementing method level security using preauthorize and postauthorize 55](#_Toc69484344)

[Details about **filtering authorization** in method level security 55](#_Toc69484345)

[Notes 56](#_Toc69484346)

[Deep dive of OAUTH2 57](#_Toc69484347)

[Problems that OAUTH2 framework trying to solve 57](#_Toc69484348)

[Introduction to OAUTH2 59](#_Toc69484349)

[Different Components involved in OAUTH2 flow 60](#_Toc69484350)

[Grant Types / Auth Flows 61](#_Toc69484351)

[Authorization Code Grant Type flow in OAUTH2 62](#_Toc69484352)

[Implicit Grant Type flow in OAUTH2 63](#_Toc69484353)

[Resource Owner Credentials Grant Type flow in OAUTH2 65](#_Toc69484354)

[Client Credentials Grant Type flow in OAUTH2 66](#_Toc69484355)

[Refresh Token Grant Type flow in OAUTH2 67](#_Toc69484356)

[How resource server validates the tokens issued by Auth server 69](#_Toc69484357)

[Ways of Validating Access Tokens 69](#_Toc69484358)

[Implementing OAUTH2 using Spring Security 72](#_Toc69484359)

[Registering the client details with the GitHub to use it's OAUTH2 Auth server 72](#_Toc69484360)

[Building client application that uses OAUTH2 72](#_Toc69484361)

[1. Add Dependencies in pom.xml 72](#_Toc69484362)

[2. Use OAUTH2 login in your WebSecurityConfigurerAdapter 72](#_Toc69484363)

[3. Side Note 73](#_Toc69484364)

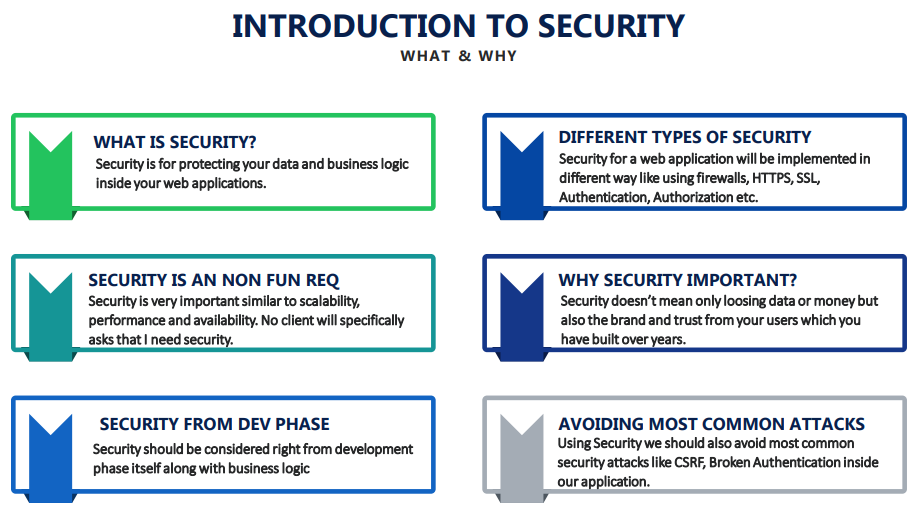
[4. Register the client in your WebSecurityConfigurerAdapter 73](#_Toc69484365)

[Tips and Tricks 75](#_Toc69484366)

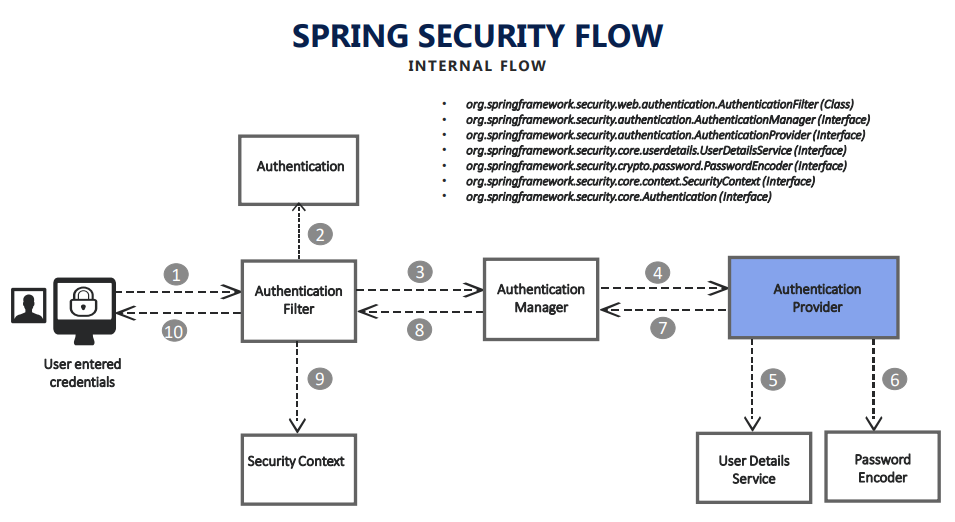
# Getting Started

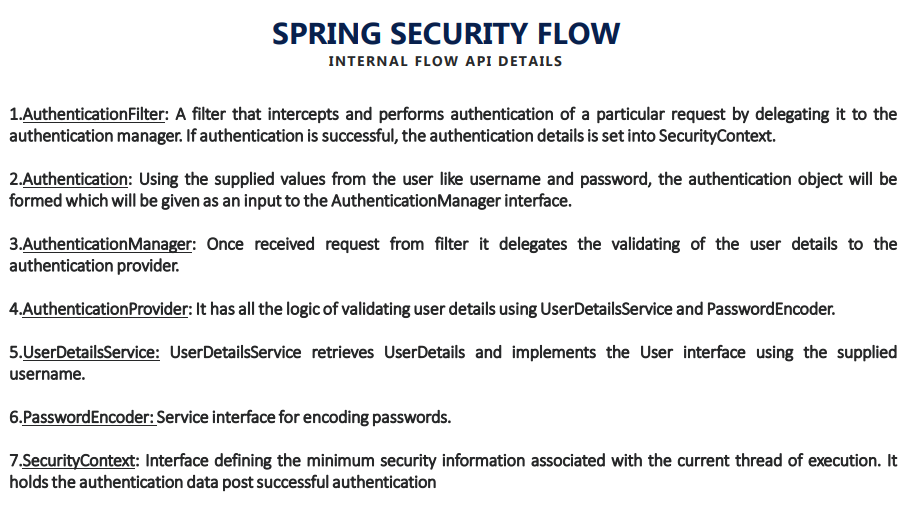
* IMP Instructor Notes – <https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf>

## Introduction to Security



## Spring Security Flow





* By default, spring security will try to secure all the services that we configure inside the project.
* One of the basic thing that we have to do whenever we try to customize the spring security as per our needs is we have to extend **WebSecurityConfigurerAdapter** class and override configure(HttpSecurity) method in our custom class as per our needs.
* **WebSecurityConfigurerAdapter** class has the default implementation of Spring Security. You can see different methods of this class to know the default behavior.
* Spring Security Flow – Refer slide no. 6, 7 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf).

# Changing the default Security Configuration

@Configuration  
public class ProjectSecurityConfig extends WebSecurityConfigurerAdapter {  
@Override  
 protected void configure(HttpSecurity http) throws Exception {  
  
 */\*\*  
 \* Default configurations which will secure all the requests  
 \*/*

/\*  
 \* http .authorizeRequests() .anyRequest().authenticated() .and()  
 \* .formLogin().and() .httpBasic();  
 \*/  
  
 */\*\*  
 \* Custom configurations as per our requirement*

*\* /myAccount - Secured /myBalance - Secured /myLoans - Secured /myCards -  
 \* Secured /notices - Not Secured /contact - Not Secured  
 \*/*

/\*  
 \* http .authorizeRequests() .antMatchers("/myAccount").authenticated()  
 \* .antMatchers("/myBalance").authenticated()  
 \* .antMatchers("/myLoans").authenticated()  
 \* .antMatchers("/myCards").authenticated() .antMatchers("/notices").permitAll()  
 \* .antMatchers("/contact").permitAll() .and() .formLogin().and() .httpBasic();  
 \*/  
  
 */\*\*  
 \* Configuration to deny all the requests  
 \*/*

/\*  
 \* http .authorizeRequests() .anyRequest().denyAll() .and() .formLogin().and()  
 \* .httpBasic();  
 \*/  
  
 */\*\*  
 \* Configuration to permit all the requests  
 \*/* http .authorizeRequests() .anyRequest().permitAll().and() .formLogin().and()  
 .httpBasic();   
  
 }  
}

# Defining and Managing Users

## Configuring Users using inMemoryAuthentication

* User Detail Service and Password Encoder are the important interfaces that helps in maintaining the user details and validating them.
* That means if someone provides some credentials, these two user detail services and password encoder take the responsibility of validating whether the username and password entered by user is correct or not.
* Override **WebSecurityConfigurerAdapter** .[configure](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/configuration/WebSecurityConfigurerAdapter.html" \l "configure-org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder-)([AuthenticationManagerBuilder](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/authentication/builders/AuthenticationManagerBuilder.html" \o "class in org.springframework.security.config.annotation.authentication.builders) auth)
* This is a method where if we want to customize your user, user detail, password encoders along with authentication providers.
* inMemoryAuthentication – That means all these users that we want to maintain will be stored inside memory of its Spring container, which will be leveraged by spring security while performing authentication and authorization details.
* With inMemoryAuthentication, you can add more users and for each user, we should pass username, password and authorities (roles).
* Also we should also user password encoder, otherwise spring will use default NoOpPasswordEncoder.
* When we setup inMemoryAuthentication, our application will no longer accept the credentials mentioned in the application.properties. We need to use the ones mentioned in inMemoryAuthentication.
* With application.properties, we can have only one user. However with inMemoryAuthentication, we can have multiple users with different roles/authorities.
* Of course we don’t use inMemoryAuthentication for production ready applications.
* Example –

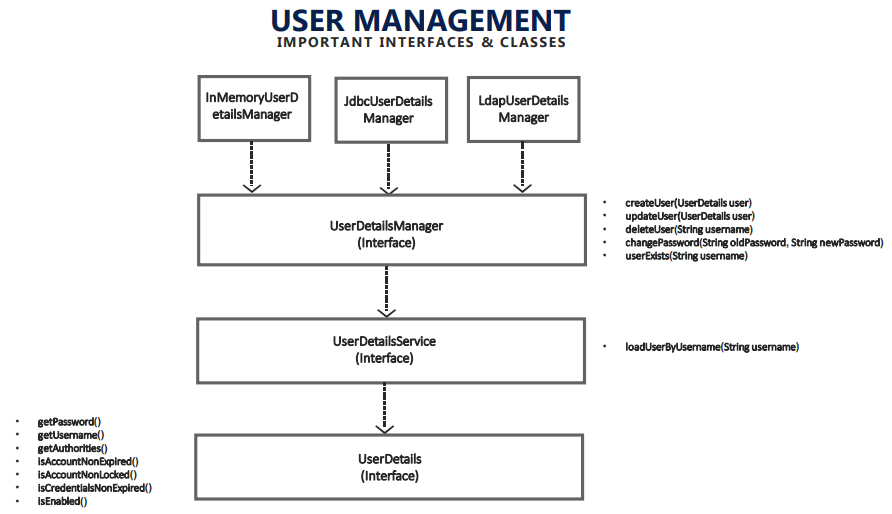
@Override   
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.inMemoryAuthentication()  
 .withUser("admin").password("12345").authorities("admin")  
 .and()  
 .withUser("user").password("12345").authorities("read")  
 .and()  
 .passwordEncoder(NoOpPasswordEncoder.getInstance());   
}

## Configuring Users using InMemoryUserDetailsManager

* Here is another way to configure in memory users using [InMemoryUserDetailsManager](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/provisioning/InMemoryUserDetailsManager.html).
* Override **WebSecurityConfigurerAdapter** .[configure](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/configuration/WebSecurityConfigurerAdapter.html" \l "configure-org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder-)([AuthenticationManagerBuilder](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/authentication/builders/AuthenticationManagerBuilder.html" \o "class in org.springframework.security.config.annotation.authentication.builders) auth)
* Note – In this case, we need to define a bean for Password Encoder, otherwise there will be exception – No Password Encoder mapped.
* Example –

@Override   
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {   
 InMemoryUserDetailsManager userDetailsService = new InMemoryUserDetailsManager();   
   
 UserDetails user = User.withUsername("admin").password("12345").authorities("admin").build();  
 UserDetails user1 = User.withUsername("user").password("12345").authorities("read").build();  
   
 userDetailsService.createUser(user);   
 userDetailsService.createUser(user1);  
 auth.userDetailsService(userDetailsService);   
 }  
   
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
 return NoOpPasswordEncoder.getInstance();  
 }

## Understanding User Management interfaces and classes



* Refer slide no. 10 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Whenever we're dealing with authentication and authorization in any framework, we need User details who want to access our application. That's way we call it as user management in spring security.
* **UserDetails** – is an interface which will define the schema or blueprint of the user that we are going to deal inside our application.
* **UserDetailsService** – is an interface which has a logic of fetching the user from the database or any other place. And the return type of loadUserByUsername is UserDetails, because whenever we are dealing with users, we decided to use UserDetails schema adhering to that spring security.
* **UserDetailsManager** – If your application has a requirement to also maintain the user details that is create user, update, delete, change passwords along with fetching the user details, then in all such scenarios we have to make sure we are customizing by implementing **UserDetailsManager**. If we are just fetching the users from a database or somewhere, then just **UserDetailsService** is sufficient.
* Spring Security has provided three inbuilt UserDetailsManager implementations. These are – InMemoryUserDetailsManager, JdbcUserDetailsManager, LdapUserDetailsManager.
  + **InMemoryUserDetailsManager** – maintains the users (create/update/delete the user) and also fetch the users from the memory of the spring container.
  + **JdbcUserDetailsManager** – use it when you have your users in some database tables.
  + **LdapUserDetailsManager** – use it when you have your users in an LDAP server.
* If these three inbuilt implementations are not matching with our requirements, we are free to go and implement UserDetailsManager and override all those methods as per our requirements.

### Deep Dive of UserDetails interface

* **UserDetails** is an interface which will define the schema or blueprint of the user that we are going to deal inside our application.
* UserDetails has lots of methods.
* Suppose if you don't have a very big application where there is no need to create our own user schema, in such scenarios we can go with the inbuilt implementation provided by the Spring Security, which is **User** class.
* This User class has constructors as well as handy inner class **UserBuilder** to build a user.

### Deep Dive of UserDetailsService interface

* <https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UserDetailsService.html>

### Deep Dive of UserDetailsManager interface

* <https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/provisioning/UserDetailsManager.html>

### Deep Dive of UserDetailsManager implementations

#### InMemoryUserDetailsManager

* Maintains the users (create/update/delete the user) and also fetch the users from the memory of the spring container.
* Useful for creating POCs or some demo applications.
* Apart from UserDetailsManager interface, InMemoryUserDetailsManager class also implements UserDetailsPasswordService interface.
* **UserDetailsPasswordService** interface has only one method updatePassword(UserDetails user, String newPassword) which is useful in the scenarios where a user doesn't know what his password is, but he still want to create a new password.

#### JdbcUserDetailsManager

* Use it when you have your users in some database tables.
* The most famous implementation provided by spring security.
* This is a production grade ready implementation. That means if you provide data source details of MySQL or Oracle or any database, this database user details manager has all the code related to loading the user details, maintaining them, creating them, deleting them, changing passwords, whatnot. It has all the implementations provided inside it.
* Refer implementation of JdbcUserDetailsManager class to see which tables/columns this class expects us to have. Otherwise it will not work.
* If you want to maintain authorities, it's always good to have a rules associated for the group and assign all the users to your group. For that, this class also implements GroupManager interface.
* <https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/provisioning/JdbcUserDetailsManager.html>
* To use in your application, just create bean as below in your implementation of WebSecurityConfigurerAdapter class.   
  The data source will be created by spring security based upon the credentials that we have given inside the application.properties

@Bean   
 public UserDetailsService userDetailsService(DataSource dataSource) {   
 return new JdbcUserDetailsManager(dataSource);   
}

#### LdapUserDetailsManager

* Use it when you have your users in an LDAP server.

### Creating Custom Implementation of UserDetailsService

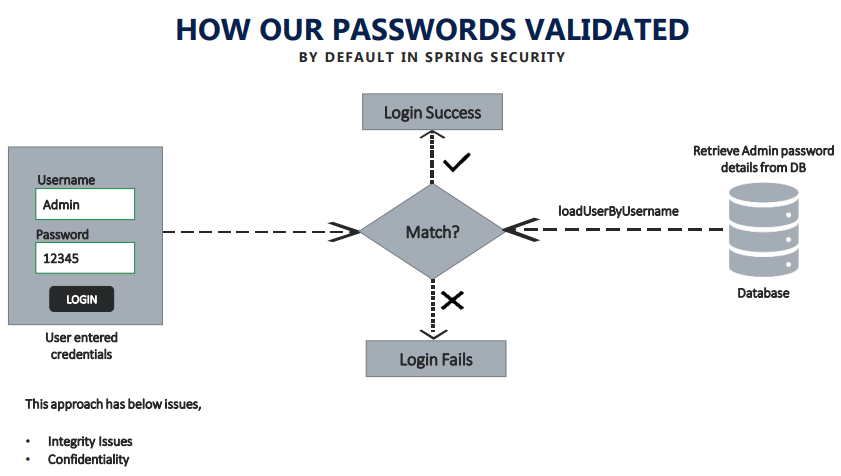
* If we have our own database tables which don’t adhere to the requirement of built in JdbcUserDetailsManager class, we can create our own implementation of UserDetailsService.
* Use Spring Data JPA.
* Refer implementation of JdbcUserDetailsManager class.
* Refer Code – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/01-ManagingUsers>

### Notes

* Refer code example – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/01-ManagingUsers>

# Password Management with Password Encoders

## How our passwords are validated by default



* Refer slide no. 12 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf).

### Steps

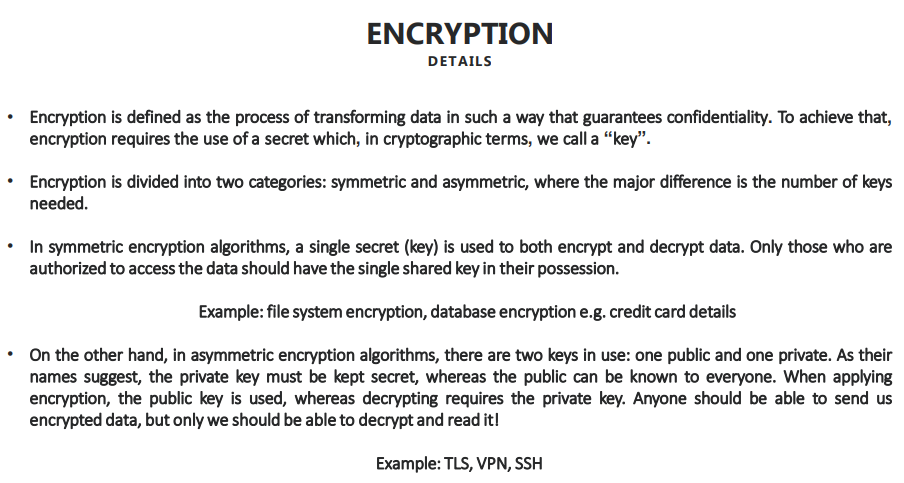
* First, the user enters credentials (username and password).
* As soon as the user enters login, spring takes the user name and call loadUserByUsername() to the database. And it will try to get all the user details, including the password.
* Once it receives the passwords from the database, it will try to compare the password received from the user with the one from database.
* If both are matched, it will allow further processing of the request.
* If not, it will throw 401 unauthorized error which indicate login failure.

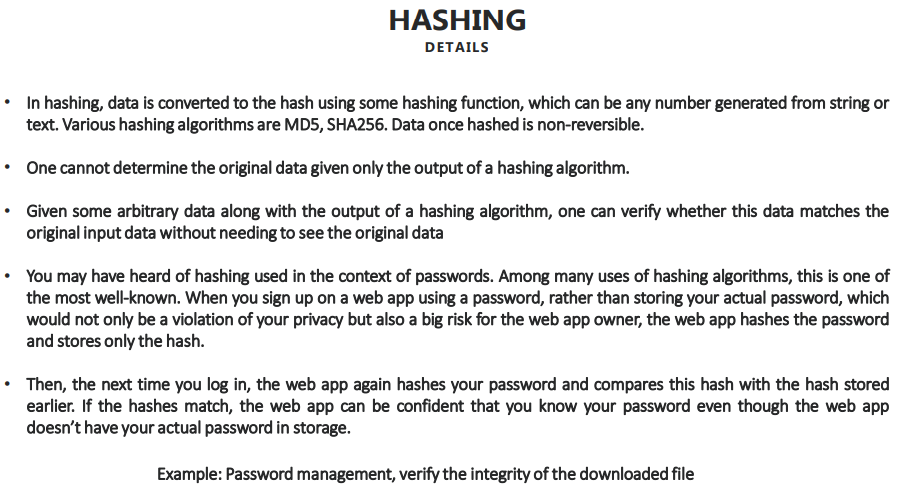
### Issues with this (default) approach

* The very first issue is you are storing user details like password, which is a very sensitive data inside your database without any encryption encoding or hash.
* The other issue is you are sending passwords in plain text or the network, like from my client browser to your server back. And so that means you are exposing the most sensitive data inside your application, which is password over the network, which result in two types of issues –
  + Integrity issues where your database administrators can look into the passwords
  + Confidentiality issues where your application is not maintaining confidentiality and it is taking the password in plain text and it is communicating with the network, using the plain text.

## Encoding Vs Encryption Vs Hashing

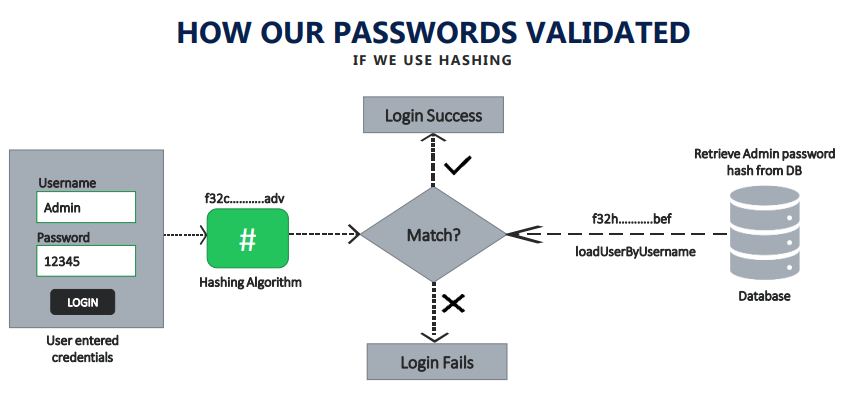






* Refer slide no. 13, 14, 15 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)

## How our Passwords will be validated with Hashing



* Refer slide no. 16 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)

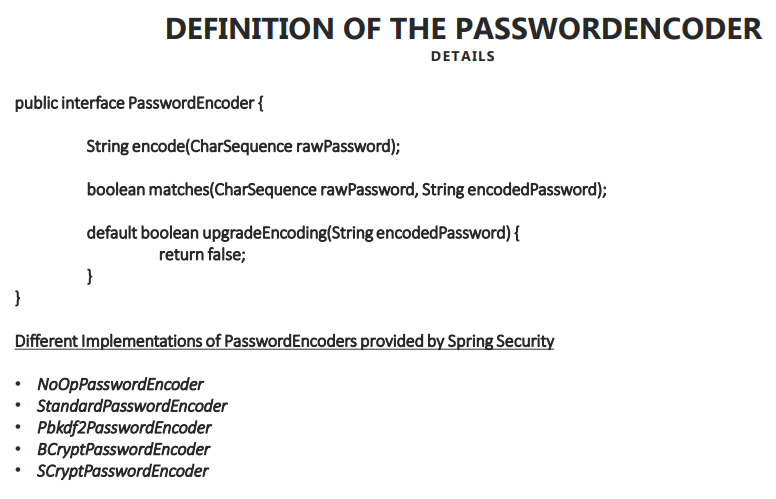
### Steps

* First, the user enters credentials (username and password).
* As soon as the user enters credentials, the password will be hashed either at the UI side or at the server side depending on requirement. This will convert the plain text password to a hashed value.
* Then Spring takes the username and call loadUserByUsername() to the database. And it will try to get all the user details, including the already hashed password (as we are only storing hashed passwords into database).
* Once it receives the passwords from the database, it will try to compare the hashed password received from the user with the one from database.
* The hash values of same text can be different but the underlying actual value will be same.
* So if underlying value of both hashes (from user and from database) are matched, it will allow further processing of the request.
* If not, it will throw 401 unauthorized error which indicate login failure.

#### How does spring security know which hashing algorithm is used

* You may have question that how does spring security know which hashing algorithm is used. The answer is the UserDetails and the PasswordEncoder are fully responsible for validating the credentials.
* So password encoder will tell spring security that the password has been hashed or encrypted using this mechanism and you have to follow the same mechanism while matching the passwords and accordingly try to do the authentication.

## Password Encoder



* Refer slide no. 17 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Password Encoder is the responsible inside spring security, which handle validating the password by leveraging the hashing, encryption or encoding.
* The term ‘Encoder’ here in PasswordEncoder is a generic term which can be used for encoding, encryption and hashing. It doesn’t mean it only supports encoding.
* PasswordEncoder is an interface in Spring security and it has primarily three methods.
  + String encode(CharSequence rawPassword);  
    The purpose of this method is whoever implementing this method should accept the raw password, which is coming from the framework, and they have to encode it as per our requirement (encode/ecrypt/hash).
  + boolean matches(CharSequence rawPassword, String encodedPassword);  
    Based upon this method, framework will decide whether I should authenticate the person or not.
  + default boolean upgradeEncoding(String encodedPassword) { return false; }  
    The purpose of this method is if you want to make your hashing algorithm or encoding or encryption algorithm to make it even more complex for the users to decode it. If you override this method and return true, which means spring security will try to do encoding on top of existing encoding. That means you will be doing that encoding, encryption or hashing two times, which gives you more security. But that doesn't mean you should always override this method and set it to true, because it will slow down your process because hashing, encryption took some good amount of time and processing time.
  + Refer [PasswordEncoder Javadoc](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/PasswordEncoder.html)
* Spring Security has provided different Implementations of PasswordEncoders
  + NoOpPasswordEncoder
  + StandardPasswordEncoder
  + Pbkdf2PasswordEncoder
  + BCryptPasswordEncoder
  + SCryptPasswordEncoder
* If these built-in password encoders do not fulfill your requirement, you are free to implement the PasswordEncoder interface.
* Bcrypt password encoder is the most famous in the industry.

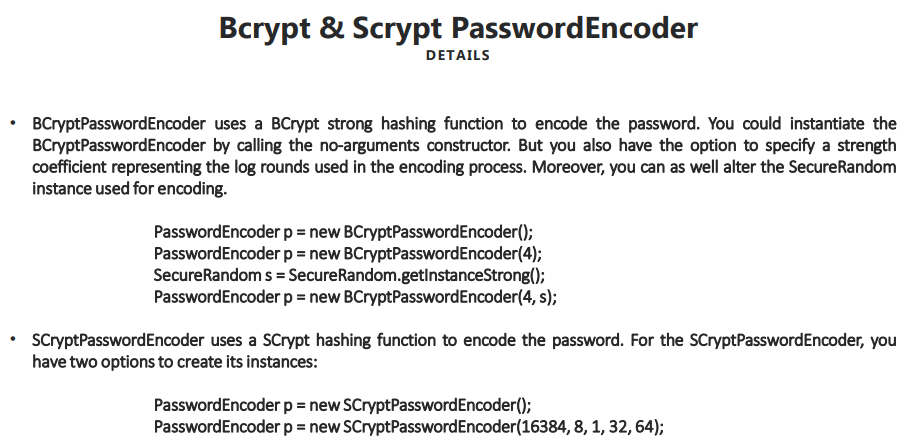
### NoOpPasswordEncoder

* Refer [NoOpPasswordEncoder Javadoc](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/NoOpPasswordEncoder.html) and also see implementation.
* It is **deprecated** to indicate that this is a legacy implementation and using it is considered insecure.

### StandardPasswordEncoder

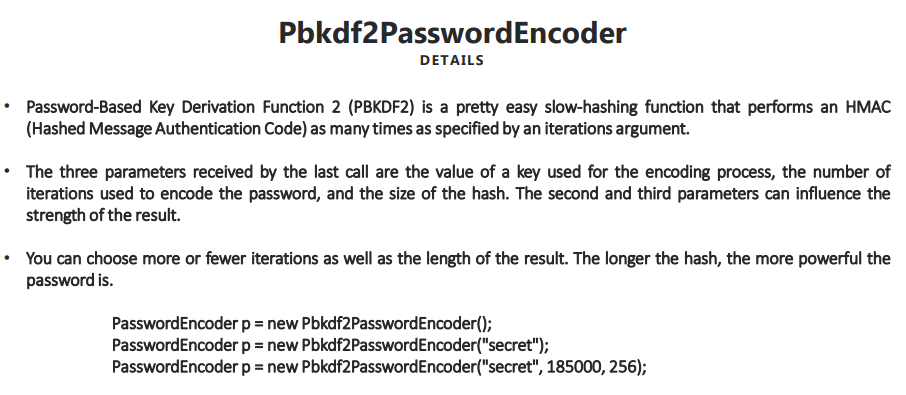
* Refer [StandardPasswordEncoder Javadoc](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/StandardPasswordEncoder.html)
* The purpose of secret or salt in hashing mechanism or encryption mechanism is, it will add more complexity to the hashing mechanism so that whatever attempts that hackers will do to decode your password will get delayed.
* Encoding/Decoding is possible here. Hence is it not considered as secured. Hence it is marked as Deprecated.

### BCryptPasswordEncoder and SCryptPasswordEncoder



* Refer slide no. 19 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Refer [BCryptPasswordEncoder Javadoc](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html)
* Refer [SCryptPasswordEncoder Javadoc](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/scrypt/SCryptPasswordEncoder.html)
* These are considered strong password encoders.
* SCrypt is the more powerful because it not only takes exponential time of the hacker, but also exponential memory of the hacker’s processor, CPU and GPU that hacker is using.

### Pbkdf2PasswordEncoder



* Refer slide no. 18 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Refer [Pbkdf2PasswordEncoder Javadoc](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/Pbkdf2PasswordEncoder.html)
* PBKDF2 means Password-Based Key Derivation Function 2

## Using bcrypt password encoder

* Example

@Configuration  
public class ProjectSecurityConfig extends WebSecurityConfigurerAdapter {  
  
@Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http.authorizeRequests().antMatchers("/myAccount").authenticated().antMatchers("/myBalance").authenticated()  
 .antMatchers("/myLoans").authenticated().antMatchers("/myCards").authenticated().antMatchers("/notices")  
 .permitAll().antMatchers("/contact").permitAll().and().formLogin().and().httpBasic();  
 }  
  
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
}

* Complete example – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/02-BCryptPasswordEncoder>

## Spring Security Crypto package

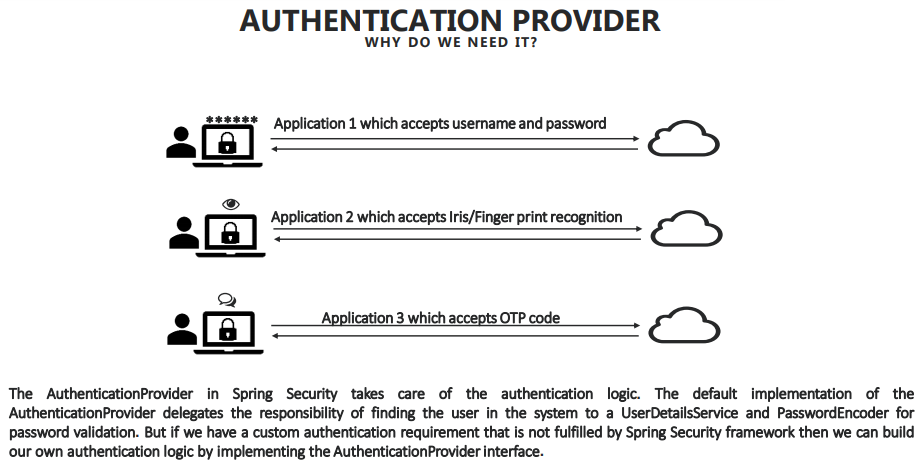
* In the Spring Security Crypt package we have many utility interfaces, classes, implementations provided by spring security for various scenarios that we face on day to day basis.
* Suppose you want to generate a random key salt then for such scenarios, we have leverage sample implementations from [org.springframework.security.crypto.keygen](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/keygen/package-summary.html) package.
* If you want to encrypt and decrypt the data for such scenarios, also we have a package called [org.springframework.security.crypto.encrypt](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/encrypt/package-summary.html)
* And many more….

# Understanding Authentication Provider

## Role of AuthenticationProvider in the Spring Security flow

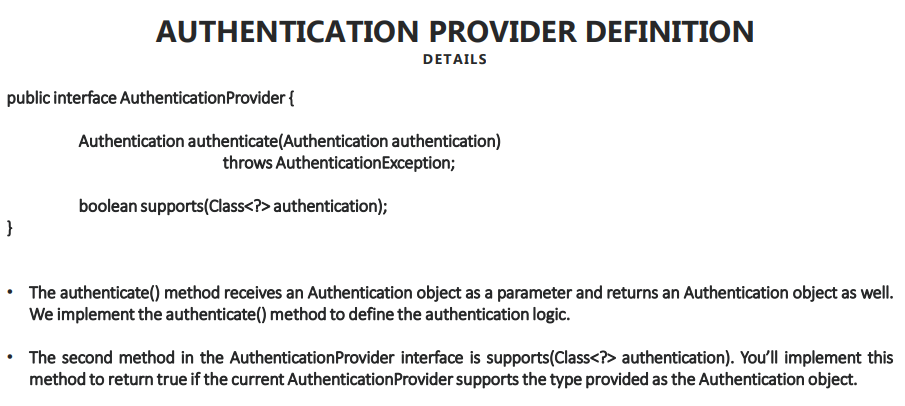
* Authentication provider is the component that leverages UserDetailService and PasswordEncoder to perform the authentication.
* If you don’t want to leverage spring security contract by using its Authentication Provider, User Details, Password Encoders, etc., in such scenarios we can customize Authentication Provider component in the Spring Security architecture.

## Scenarios where we need to implement Authentication Provider



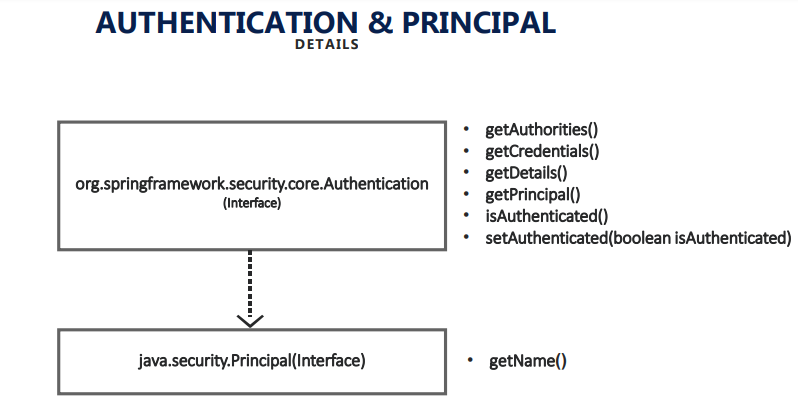
* Refer slide no. 21 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Applications doing authentication based on fingerprint/iris scan or authentication using OTPs. In such applications, we need to implement our own Authentication Provider.
* Spring Security allows us to maintain as many Authentication Providers as we want. So in an application, we can have multiple authentication providers which will be used based on inputs received. E.g. one doing authentication based on username/password, other doing authentication based on fingerprint scanner, some other using authentication based on OTPs.
* Since Authentication Manager calls Authentication Provider, so it’s responsibility of the Authentication Manager to call the right Authentication Provider.

## Understanding AuthenticationProvider Definition



* Refer slide no. 22 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* AuthenticationProvider interface has 2 methods
  + Authentication authenticate(Authentication authentication) throws AuthenticationException;
  + boolean supports(Class authentication);  
    The purpose of this method is, your organization may have multiple ways of authentication e.g. username credentials, fingerprint face recognition, etc. So this support method will help which type of authentication format that this provider supports.
* Authentication Manager’s job is only to identify and call all the providers present inside my application one by one, till the user is authenticated successfully or once we tried all the providers, then we should throw an exception.
* AuthenticationManager has only one method –  
  Authentication authenticate(Authentication authentication) throws AuthenticationException  
  This method is same as that of AuthenticationProvider’s authenticate method.
* ProviderManager class is the implementation of AuthenticationManager interface.
* There are many implementations of AuthenticationProvider interface which we can use depending on the requirement like LDAP, OAuth, etc. However DaoAuthenticationProvider is the default implementation of AuthenticationProvider.

## Understanding Authentication and Principal interfaces



* Refer slide no. 23 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Authentication object is very crucial to the Spring framework whenever it wants to perform any authentication of a user.
* Spring security Authentication interface implements java.security.Principal interface.
* Principle is an interface from Java Security API and it has only one method which indicates the name of the user.
* Authentication interface has many useful methods.
* There are many implementations of Authentication interface which we can use depending on the requirement like LDAP, OAuth, etc. e.g. UsernamePasswordAuthenticationToken class is for username/password based authentication. And so on.

### Authentication vs UserDetails

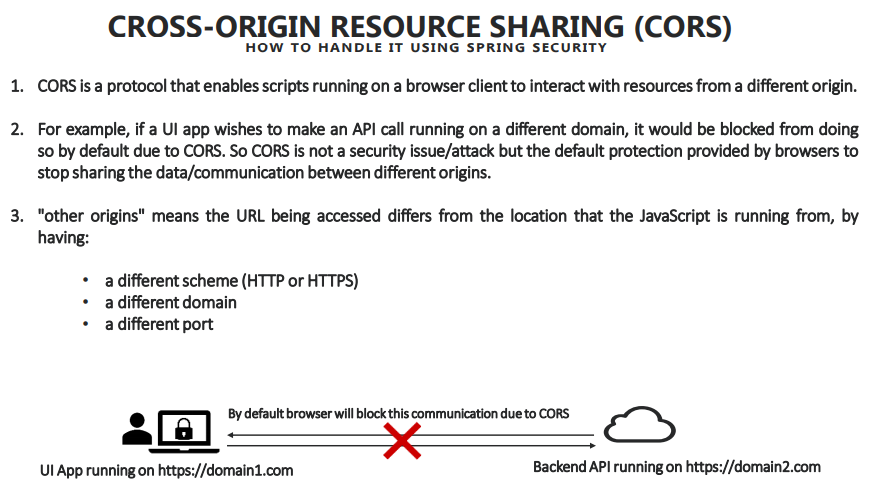
* If you remember inside our application, we are dealing with UserDetails object insider UserDetailService.loadUserbyUsername() implementation.
* And we know by default, spring security leverages DaoAuthenticationProvider which deals with Authentication object.
* **How the conversion between this UserDetails and Authentication is happening**?
* The answer is the conversion happens inside AbstractUserDetailsAuthenticationProvider class which is super class of DaoAuthenticationProvider.
* You may ask, **what is the need of Authentication and UserDetails – two different mechanisms to maintain the similar user details?**
* The answer is spring security want to give you that flexibility at each and every layer. So my UserDetailsManager and UserDetailService will leverage my UserDetails, whereas my AuthenticationManager and the AuthenticationProvider always leverage Authentication interface.
* So that's why spring security always give you the flexibility of maintaining as per your requirement. But if you don't want UserDetails to be maintained, you can always free to customize the AuthenticationProvider as per your requirement and free to use directly the Authentication object.

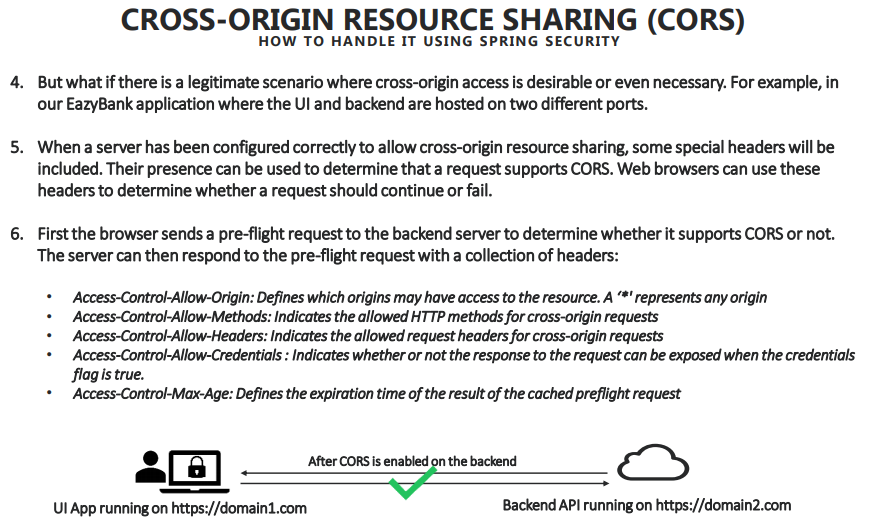
## Implementing and Customizing the Authentication Provider

* If I don't want to follow the UserDetailService and I don't want to tightly coupled with this spring security mechanism of UserDetailService and UserDetails, I just want to have my own implementation ignoring this default provider interfaces, user contract, everything.
* For that, we just have to create a class which implements AuthenticationProvider and annotate the class with @Component. And override the authenticate() and supports() methods.
* If we have multiple authentication mechanisms (like username/password, fingerprint, OTP, etc.), we can have multiple such concrete implementations and each will have proper authenticate() and supports() methods implementations.
* Just annotate those Authentication Providers with @Component annotation, then Spring is smart enough to automatically detect those providers and will take care of calling those providers one by one from AuthenticationManager.
* Complete code example – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/03-CustomizedAuthenticationProvider>

# Understanding CORS and CSRF

## Deep dive into CORS





* CORS – Cross Origin Resource Sharing
* MDN Article – <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
* Refer slide no. 26, 27 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* CORS is default protection provided by browsers so that applications which are hosted in two different hosts/post/domains will be stopped communicating with each other.

### Resolving CORS issue

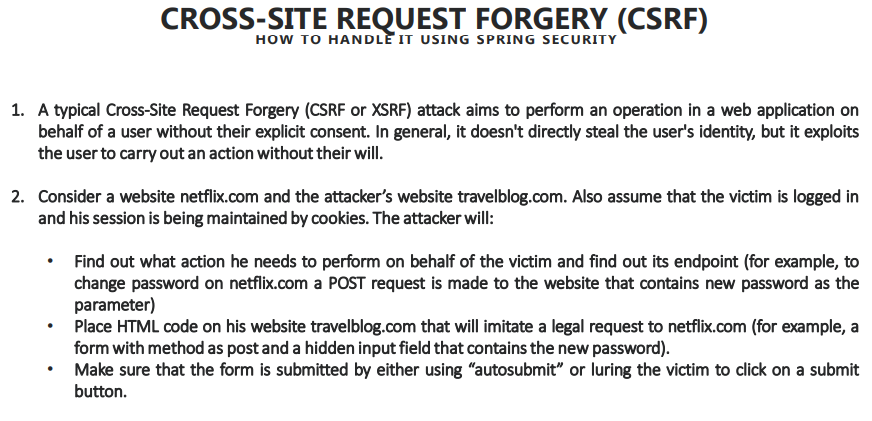
protected void configure(HttpSecurity http) throws Exception {  
  
 http  
 .cors().configurationSource(new CorsConfigurationSource() {  
 @Override  
 public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {  
 CorsConfiguration config = new CorsConfiguration();  
 config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));  
 config.setAllowedMethods(Collections.singletonList("\*"));  
 config.setAllowCredentials(true);  
 config.setAllowedHeaders(Collections.singletonList("\*"));  
 config.setMaxAge(3600L);  
 return config;  
 }  
 })  
 .and()  
 .authorizeRequests()  
 .antMatchers("/myAccount").authenticated()  
 .antMatchers("/myBalance").authenticated()

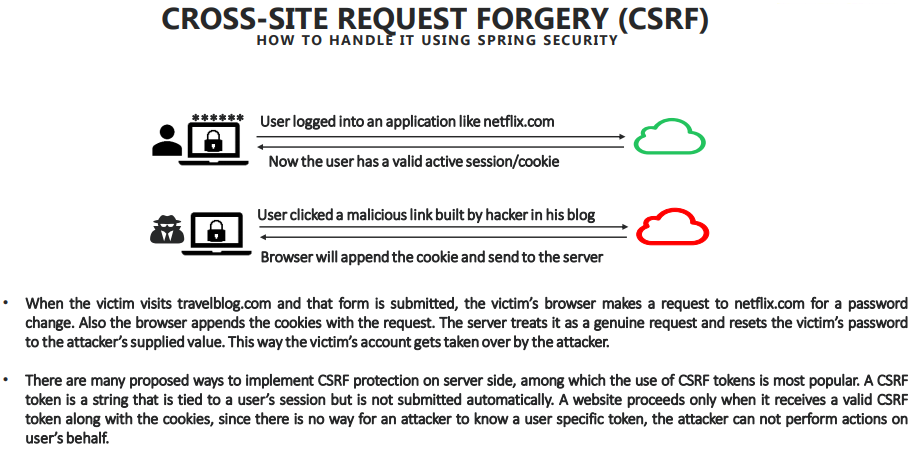
.antMatchers("/myLoans").authenticated()  
 .antMatchers("/myCards").authenticated()

.antMatchers("/user").authenticated()  
 .antMatchers("/notices").permitAll()

.antMatchers("/contact").permitAll()  
 .and().httpBasic();  
}

## Deep dive into CSRF





* CSRF – Cross Site Request Forgery
* CSRF issue will never be thrown in GET request. It will be thrown when you have a POST, DELETE, PUT, etc. request.
* Refer slide no. 28, 29 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* CSRF can happen in both same origin or cross origin applications. It is nothing to do with CORS. The reason is a typical CSRF attack, aims to perform an operation in a Web application on behalf of a logged in user without his explicit consent.
* In general, it doesn't directly steal the user's identity, but it exploits the user to carry out an action without their will.

### Resolving CSRF Issue

#### Resolving CSRF error by disabling it

* One of the simplest approach is by disabling the CSRF itself.
* As we discussed, any Web application or any Web framework will stop the communication if someone is using POST, PUT, DELETE, etc. methods (non GET), which will potentially alter the data causing CSRF issue.
* There might be some valid reasons where you don't want to handle all CSR tokens and everything because you have all the firewalls inside your organization where outside hackers cannot access your links (Intranet applications). So in such scenarios, we just have to simply disable CSRF inside our spring security framework.
* When we disable CSRF, it clearly indicates that I don't want to have any CSRF tokens to be handled and all operations including GET or DELETE, POST, PUT has to be accepted by my application.
* Disabling is not a solution always and it's not recommended also when your application is open to the outside world.
* Example –

protected void configure(HttpSecurity http) throws Exception {  
  
 http.csrf().disable()  
 .and()  
 .authorizeRequests()  
 .antMatchers("/myAccount").authenticated()  
 .antMatchers("/myBalance").authenticated()

.antMatchers("/myLoans").authenticated()  
 .antMatchers("/myCards").authenticated()

.antMatchers("/user").authenticated()  
 .antMatchers("/notices").permitAll()

.antMatchers("/contact").permitAll()  
 .and().httpBasic();  
}

#### Resolving CSRF error by generating CSRF token

* What we can do to avoid CSRF attack inside our application is by maintaining CSRF token. So CSRF token is a token which is issued by your backend application when the **first** request happens from the user to your application, like during the login of the user.
* Now that token will never be stored into the browser cache or browser cookies, but it will be given to whoever calling our application or to the browser. They have to make sure that they are passing the same token value whenever they want to communicate with the backend.
* If we don’t pass this CSRF token with each request or if the CSRF token passed with the request is not valid, then it will block the communication and will throw 403 error.
* So that means on top of your **authentication token**, which you are maintaining inside your cookie, we are also maintaining a **CSRF token** that is tied to the user session, but it is not submitted automatically like cookies. And you have to take enough majors to send that token every time when we are making a request to the backend.
* There might be some pages which are open to all (meaning logging into application is not required for those pages) e.g. contacts or news pages. For such pages, we should not have CSRF protection otherwise non-loggedin users won’t be able to use such pages if those pages are making POST, PUT, DELETE requests. With Spring security, we can easily do that using .csrf().ignoringAntMatchers() method.
* Example –

protected void configure(HttpSecurity http) throws Exception {  
  
 http

.csrf()

.ignoringAntMatchers("/contact")

.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())  
 .and()  
 .authorizeRequests()  
 .antMatchers("/myAccount").authenticated()  
 .antMatchers("/myBalance").authenticated()

.antMatchers("/myLoans").authenticated()  
 .antMatchers("/myCards").authenticated()

.antMatchers("/user").authenticated()  
 .antMatchers("/notices").permitAll()

.antMatchers("/contact").permitAll()  
 .and().httpBasic();  
}

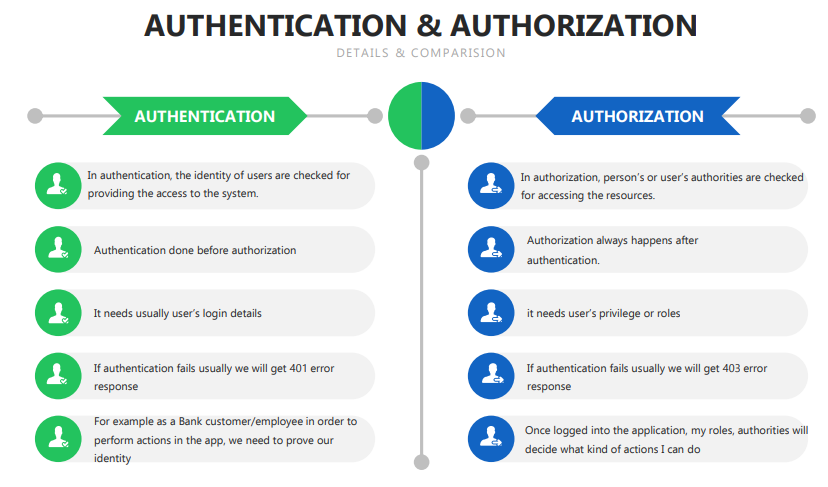
Here we are using Spring Security’ built in implementation for CSRF which is CookieCsrfTokenRepository. CookieCsrfTokenRepository persists (sends to browser) the CSRF token in a cookie named "XSRF-TOKEN" and reads (from browser) from the header "X-XSRF-TOKEN" following the conventions of AngularJS. When using with AngularJS be sure to use withHttpOnlyFalse().

## Notes

* Complete Example – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/04-cors-and-csrf>

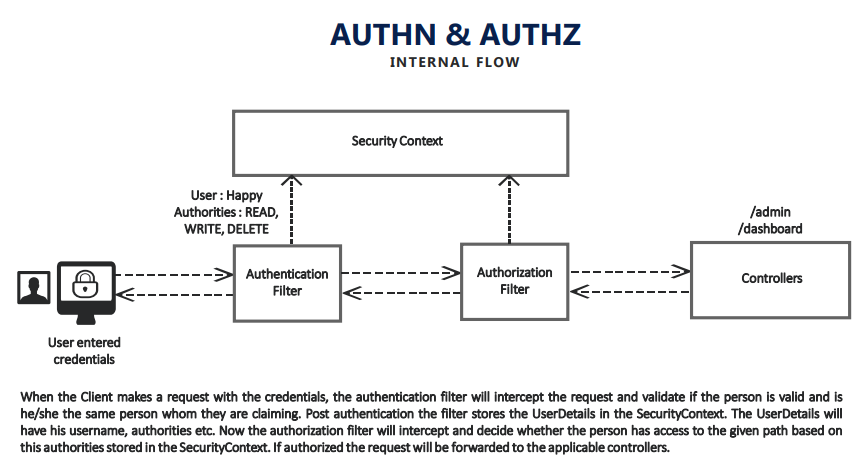
# Understanding & Implementing Authorization

## Authentication Vs Authorization



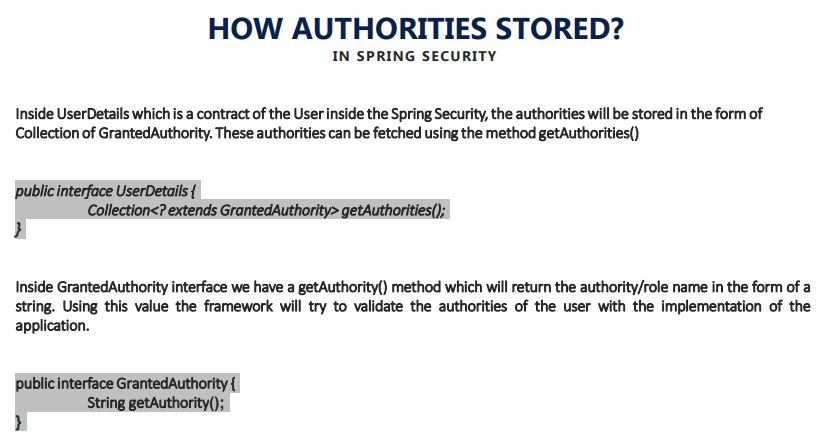
* Refer slide no. 31 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)

## Spring Security Internal flow for Authentication & Authorization



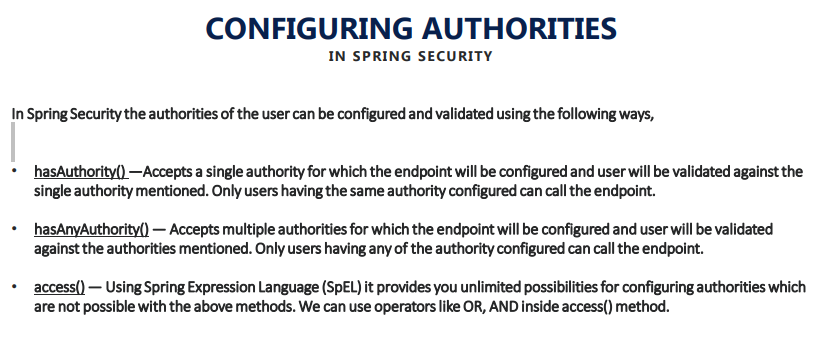
* Refer slide no. 32 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Authentication – AUTHN
* Authorization – AUTHZ
* Authorization is optional, you may not need it depending on type of your application. But majority of applications need it.
* Spring Security Internal flow –
  + After successful authentication, all the user details (name, role, authorities, etc.) excluding password will be stored in the UserDetails schema and it will be stored inside the security context
  + Once it is stored inside the security context and if you have configured authorization inside your application (roles/authorities), AuthorizationFilter also will come into picture and it will try to load the UserDetails from the security context.
  + Then AuthorizationFilter will try to compare whether this particular user has access to perform given action or not. Only if he has access, then it will allow the business execution to the associated API/URL.
* We know after successful authentication Spring Security Framework stores that user details inside security context, which we can access via SecurityContextHolder.getContext().getAuthentication()

## How Authorities stored in Spring Security



* Refer slide no. 33 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* The authorities/roles are stored inside UserDetails.
* UserDetails is a contract which is maintained by Spring Security, where all the details associated to a user will be stored. It includes is authentication details, username, email, password along with the authorities. (Depending on the stage of the application, the password will be removed from UserDetails object).
* GrantedAuthority is the interface used for maintaining authorities.
* A single user can have multiple authorities/roles since UserDetails maintains GrantedAuthority as a collection.
* There are many implementations of GrantedAuthority. One of those is SimpleGrantedAuthority. SimpleGrantedAuthority stores a String representation of an authority granted to the Authentication object

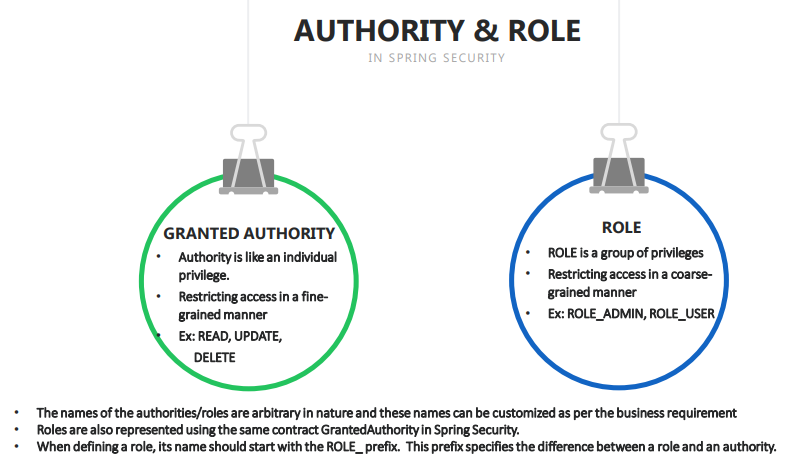
## Configuring Authorities in Spring Security



* Refer slide no. 34 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* When there is no authentication, there is no authorization we could configure.
* We can configure authorization for secured urls only (urls which needs authentication).
* Example –

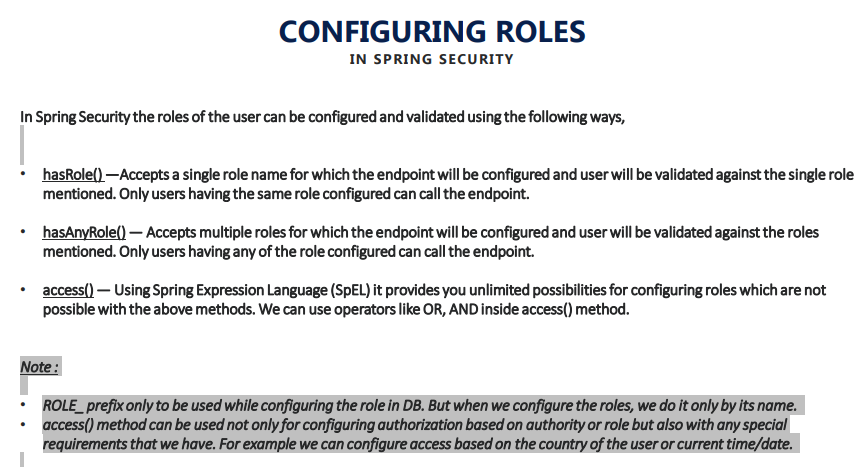
protected void configure(HttpSecurity http) throws Exception {  
  
 http.authorizeRequests()  
 .antMatchers("/myAccount").hasAuthority("READ")  
 .antMatchers("/myBalance").hasAnyAuthority("READ","WRITE")  
 .antMatchers("/myLoans").hasAuthority("DELETE")  
 .antMatchers("/myCards").authenticated()  
 .antMatchers("/user").authenticated()  
 .antMatchers("/notices").permitAll()  
 .antMatchers("/contact").permitAll().and().httpBasic();  
}

## Authority Vs Role



* Refer slide no. 35 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* An **authority** is a single individual privilege/permission.
* A user can have multiple authorities.
* We can group multiple authorities into a **role**. And then provide r**ole-based authorization**.
* E.g. We can think of operations like READ, WRITE, DELETE as authorities and ROLE**\_**ADMIN, ROLE**\_**USER as roles where ROLE \_ADMIN role has authorities READ, WRITE, DELETE etc.

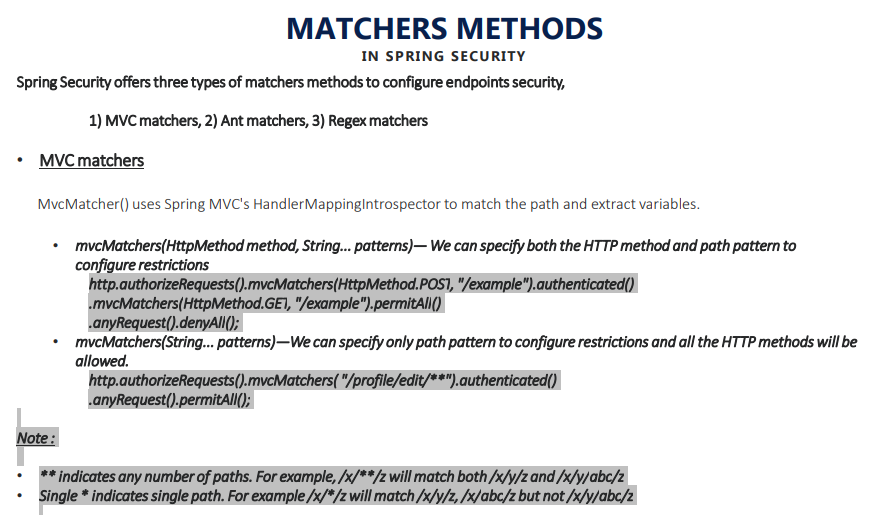
## Configuring Roles in Spring Security

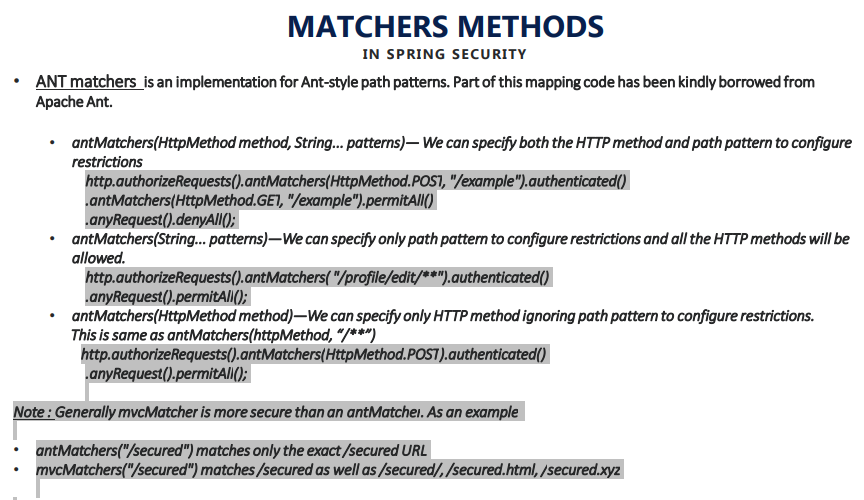


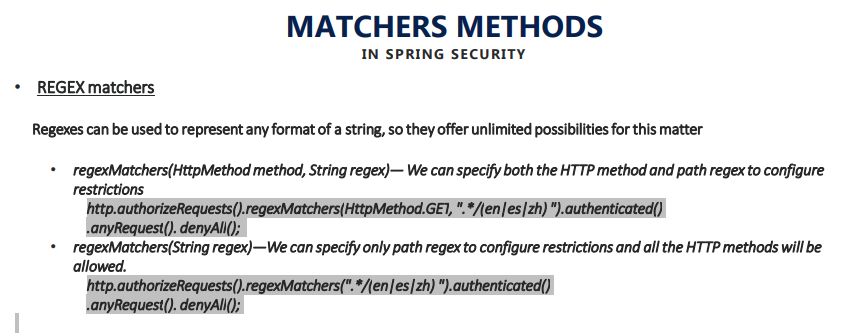
* Refer slide no. 36 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* When defining a role, its name should start with the **ROLE\_** prefix in the database. This prefix specifies Spring Security the difference between a role and an authority.
* ROLE\_ prefix only to be used while configuring the role in DB. But when we configure the roles, we do it only by its name. The reason is, while performing these comparison checks, spring security automatically will add ROLE\_ prefix to the value that you provide.
* Example –

protected void configure(HttpSecurity http) throws Exception {  
  
 http.authorizeRequests()  
 .antMatchers("/myAccount").hasRole("USER") // in DB, the role name must be ROLE\_USER  
 .antMatchers("/myBalance").hasAnyRole("USER","ADMIN") // in DB, the role name must be ROLE\_ADMIN  
 .antMatchers("/myLoans").hasRole("ROOT") // in DB, the role name must be ROLE\_ROOT  
 .antMatchers("/myCards").authenticated()  
 .antMatchers("/user").authenticated()  
 .antMatchers("/notices").permitAll()  
 .antMatchers("/contact").permitAll().and().httpBasic();  
}

## Deep dive of Ant, MVC, Regex matchers for applying restrictions on the paths







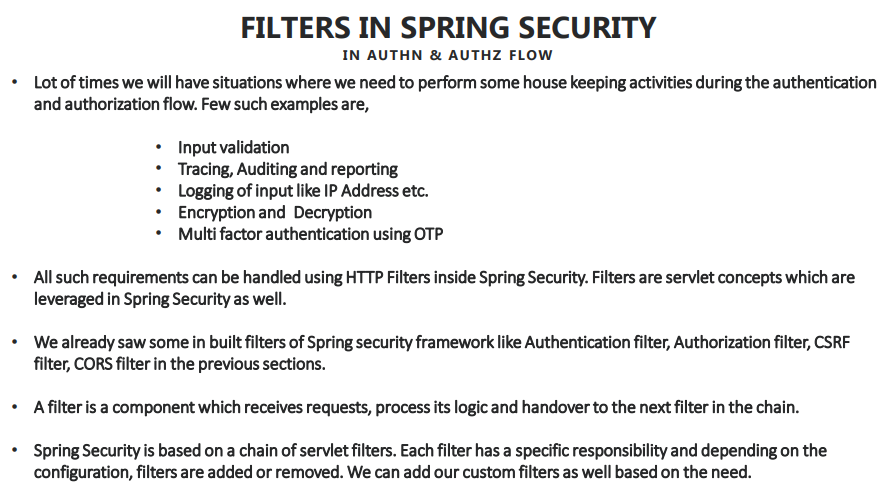
* Refer slide no. 37, 38, 39 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Spring security of three important types of matchers – Ant Matchers, MVC matchers, Regex matchers.
* Ant matcher recognizes ANT based path patterns.
* MVC matcher recognizes Spring MVC path patterns.
* Regex matchers uses regular expressions for path patterns.
* Generally mvcMatcher is more secure than an antMatcher.  
  E.g.   
  antMatchers("/secured") matches only the exact /secured URL  
  mvcMatchers("/secured") matches /secured as well as /secured/, /secured.html, /secured.xyz
* \*\* indicates any number of paths.   
  For example, /x/\*\*/z will match both /x/y/z and /x/y/abc/z
* Single \* indicates single path.   
  For example /x/\*/z will match /x/y/z, /x/abc/z but not /x/y/abc/
* Both Ant Matchers and MVC matchers support \* and \*\* in path patterns.

## Notes

* Complete example – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/05-Authorization>

# Filters in Spring Security

## Introduction



* Refer slide no. 41 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)

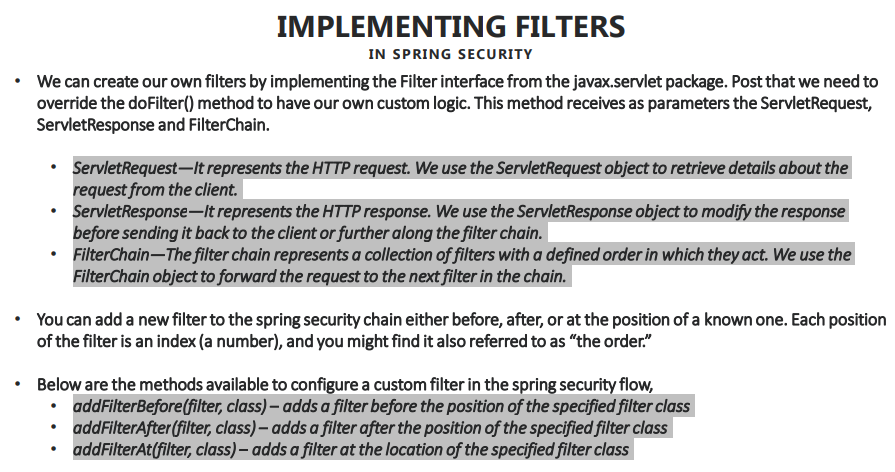
## Inbuilt Filters provided by Spring Security



* Refer slide no. 42 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Spring security framework is built based upon a series of filters which are maintained in a chain manner, and each filter has its own rules and responsibilities and wants it done with its execution. It will handle the logic of execution to the next filter inside the chain.
* And based upon the configurations that we do inside our application, these filters will be enabled or disabled inside spring security flow while starting our application itself.
* E.g. If you make cors and csrf configurations inside your application, then the filters associated to csrf and cors will be activated and they will be added to the filter chain that maintained by the spring security.
* Also there are mandatory filters also present inside spring security flow, which will be always executed regardless of what configurations you are making inside your application.

## Custom filters in Spring Security

### Implementing Custom Filter

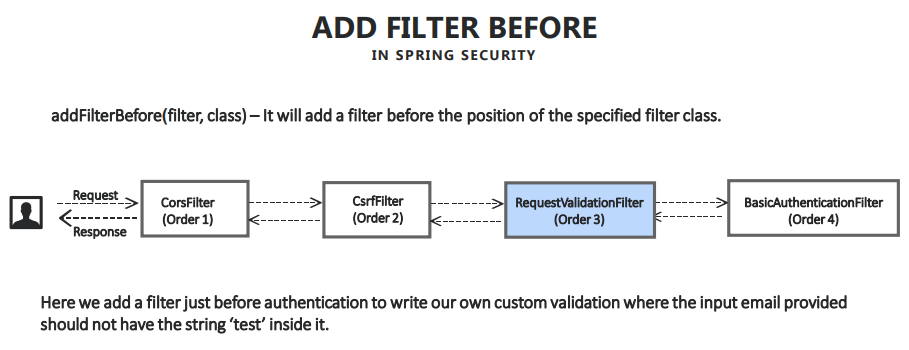


* Refer slide no. 43 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)

### Adding Custom Filter in the filter chain

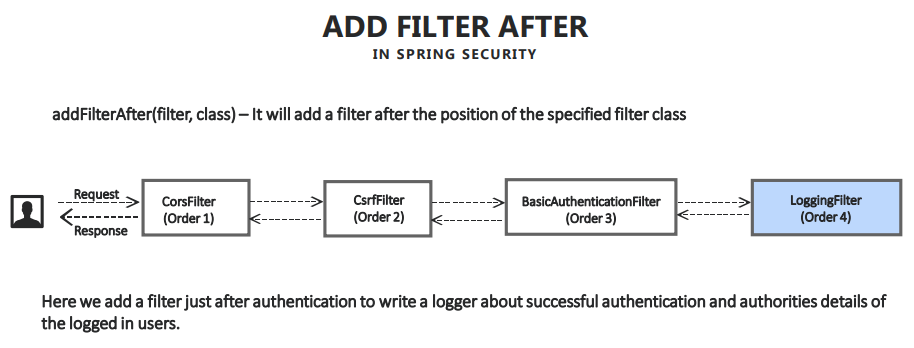
* Imp Note – You must add your custom filters in the Spring Security filter chain at the appropriate place. E.g. If your custom filter depends on say successful authentication, then it must be added after the BasicAuthenticationFilter.

#### addFilterBefore()



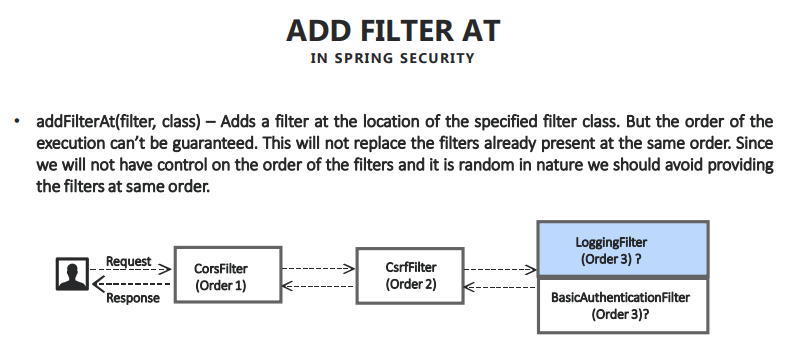
* Refer slide no. 44 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)

#### addFilterAfter()



* Refer slide no. 45 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)

#### addFilterAt()



* Refer slide no. 46 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* addFilterAt() doesn't mean it will replace the spring security filter. And at the same time, we can't guarantee the order of those filters which are positioned at the same place and it is up to the spring security to decide randomly to execute which filter before and after which are located at the same position based upon our configurations.
* Not recommended to use because of random order of execution of those filters.

#### Example

@Override  
protected void configure(HttpSecurity http) throws Exception {  
  
 http

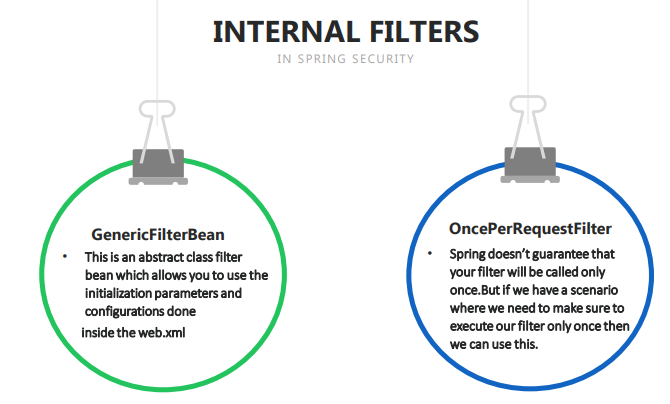
.cors().disable()

.and()

.csrf().ignoringAntMatchers("/contact").csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())  
 .and()

.addFilterBefore(new RequestValidationBeforeFilter(), BasicAuthenticationFilter.class)  
 .addFilterAfter(new AuthoritiesLoggingAfterFilter(), BasicAuthenticationFilter.class)  
 .addFilterAt(new AuthoritiesLoggingAtFilter(), BasicAuthenticationFilter.class)  
 .authorizeRequests()  
 .antMatchers("/myAccount").hasRole("USER")  
 .antMatchers("/myBalance").hasAnyRole("USER","ADMIN")  
 .antMatchers("/myLoans").hasRole("ROOT")  
 .antMatchers("/myCards").hasAnyRole("USER","ADMIN")  
 .antMatchers("/user").authenticated()  
 .antMatchers("/notices").permitAll()  
 .antMatchers("/contact").permitAll().and().httpBasic();  
}

## Details about GenericFilterBean and OncePerRequestFilter



* Generally we create filter class by extending the javax.servlet.Filter interface. But for certain scenarios, Spring Security Framework also provided certain interfaces and classes which your filter can extend and implement similar to filter interface.
* Refer slide no. 47 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Creating filter by implementing javax.servlet.Filter interface is pretty basic way of creating your own custom filter because everything you have to build on your wound.
* Spring security also provided two important filters which can be used for certain scenarios – GenericFilterBean and OncePerRequestFilter.

### GenericFilterBean

* It wraps all the properties and configuration from web.xml and servlet context details for you. And whenever you implement this filter, all such properties will also can be leveraged and used by your own custom filter.
* Your custom filter needs to override doFilter() function.

### OncePerRequestFilter

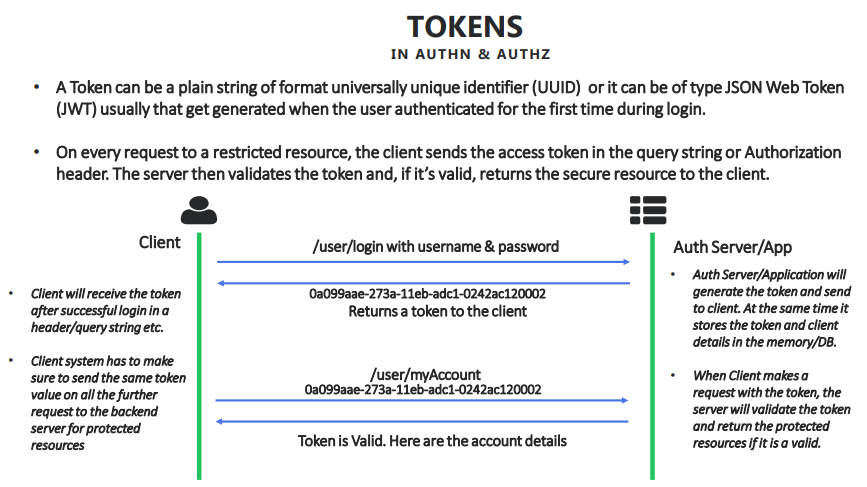
* Whenever we write a custom filter or any spring security filter present inside the framework, Spring does not guarantee that that filter will be invoked only once by request. It can be invoked multiple times.
* But if we have a scenario where you want your own custom logic to be executed only once per request, then you can always extend OncePerRequestFilter class, which will give you the logic of making sure that your custom filter will be executable only and once for every request coming from the client.
* OncePerRequestFilter extends GenericFilterBean
* Your custom filter needs to override doFilterInternal() function.
* At the same time you may have certain requirements where your filter should not be executed for certain scenarios. For all such scenarios, you can leverage shouldNotFilter() method by overriding it. By default, this method always return false.
* The Spring Security’s BasicAuthenticationFilter itself extends OncePerRequestFilter

## Notes

* Complete example – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/06-Filters>

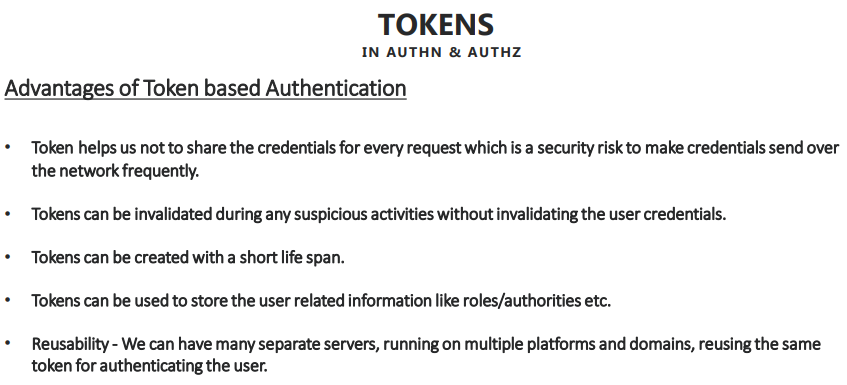
# Token based Authentication using JSON Web Token (JWT)

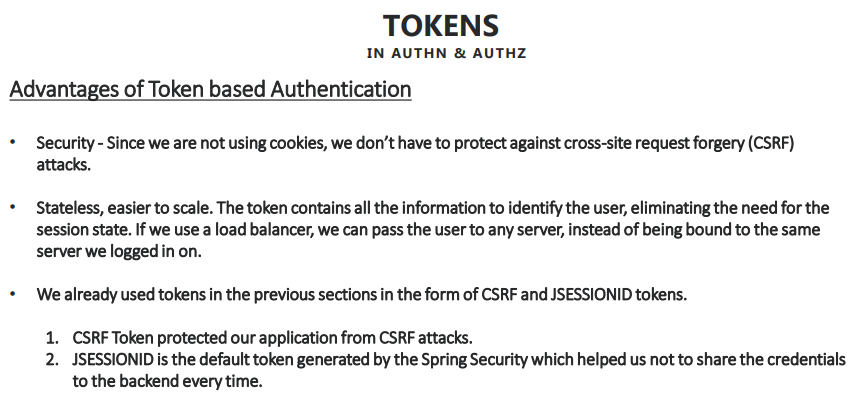
## Introduction to Tokens in Authentication flow



* Refer slide no. 49 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)

## Advantages of Token based Authentication



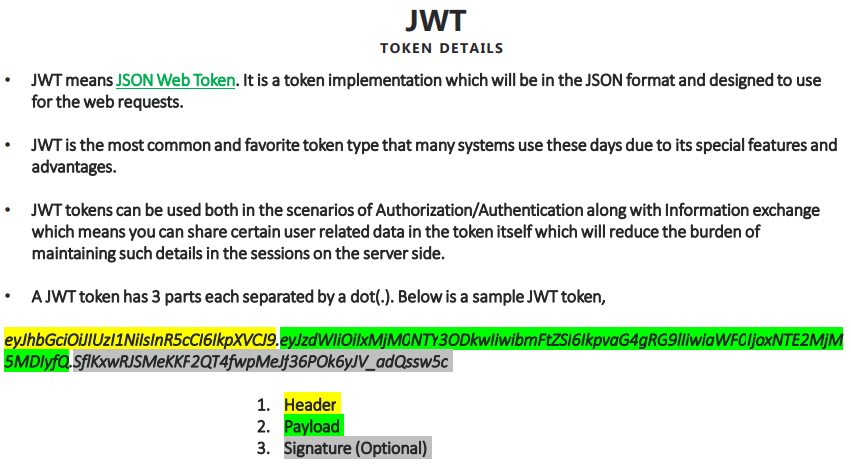


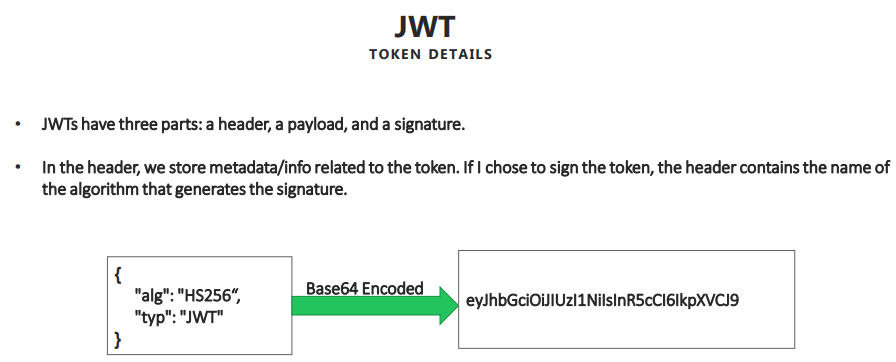
* Refer slide no. 50, 51 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)

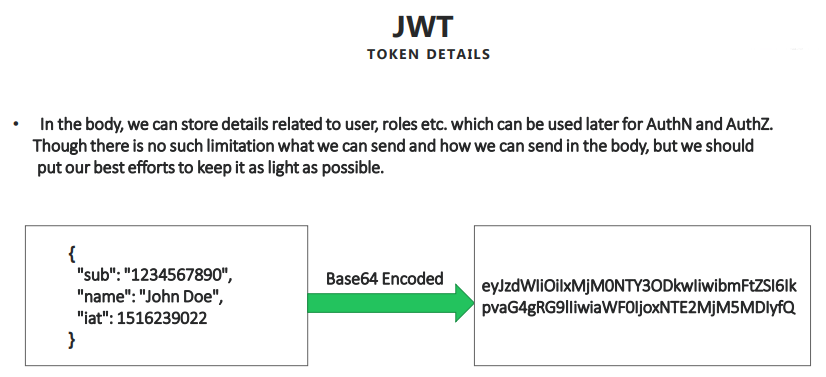
## Exploring the JSESSIONID & CSRF Tokens inside our application

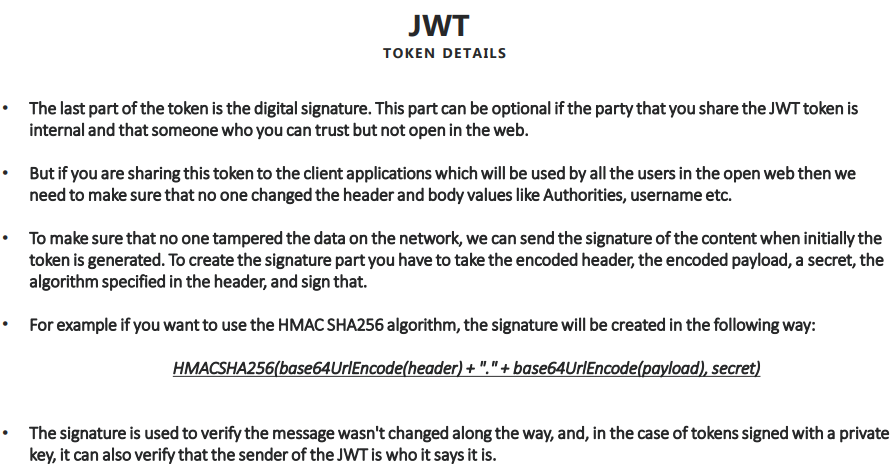
* If we have a multiple micro services scenario where you have a lot of servers in those scenarios, we will go with the separate authentication server, which will help all our application servers in generating and validating the tokens.
* These tokens (JSESSIONID & CSRF) are very simple in nature and they don't support sharing the user related information to the client side.
* And at the same time, these tokens (JSESSIONID & CSRF) are not supporting any encryption and encoding mechanisms to apply extra layer of security to your tokens.

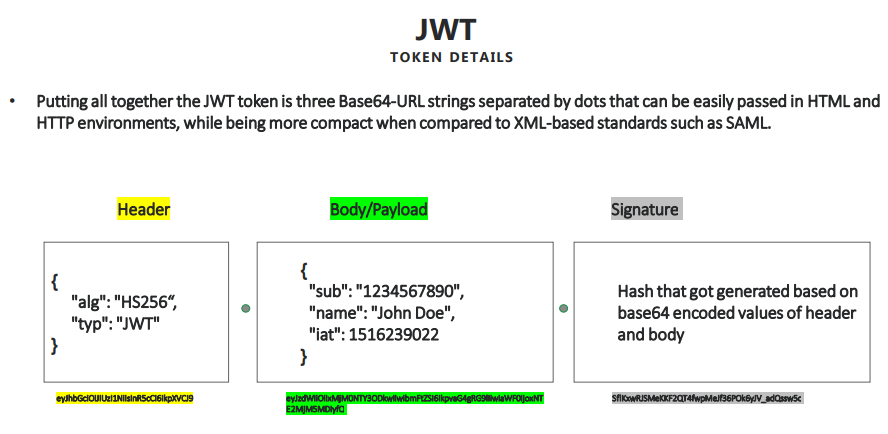
## Deep dive into JWT Tokens

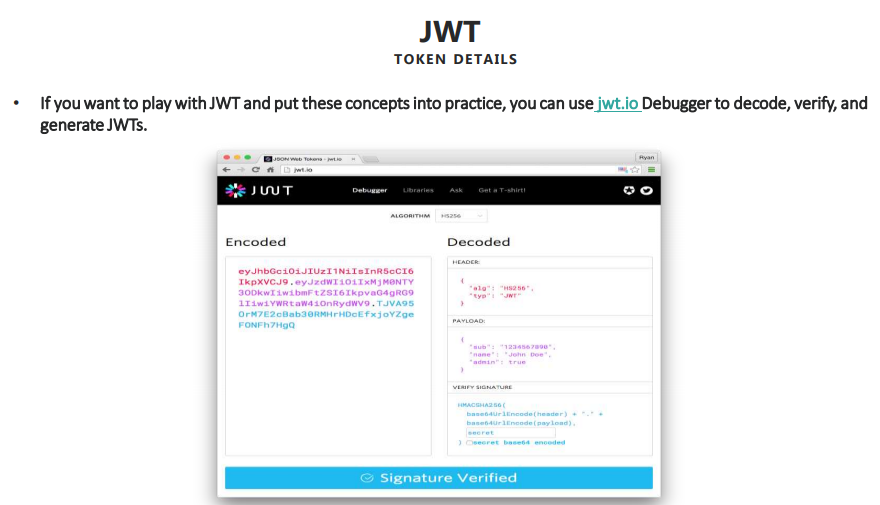












* Refer slide no. 52 to 57 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* JWT tokens can be used beyond authentication and authorization. For instance, if I want to share certain information or if I want to store certain information inside my token itself, rather than storing inside the server memory, which will degrade the performance of the server over a period of time. What I can do is I can store all such information that I want to communicate to the other system or for my future reference inside my token itself in an encrypted format.
* JWT has three parts – header, payload and signature.
* Signature is an optional part. We use it only if we are going for the encryption and if we want to make sure that no one tampers with your header and body parts of your token, then only you will use this last part. Use case can be if your application is intranet application only.
* Unlike JSESSIONID & CSRF tokens, we don’t store JWT token at server side.
* Since JWT’s header, body and signature are in base64 format, they can be decoded easily. You can also add encryption on top of base64 encoding to secure your header and body. But remember, whenever you introduce encryption and decryption unnecessarily, it will use a lot of your server computations. So by default you can always send your header and body in base64 encoded value, since that is very easy to decode and you can protect those values by generating a signature and hold it in the signature part.

## Using JWT in the Application

### Step 1: Making Configuration changes

#### 1. Add Dependencies

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-api</artifactId>  
 <version>0.11.1</version>  
</dependency>  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-impl</artifactId>  
 <version>0.11.1</version>  
 <scope>runtime</scope>  
</dependency>  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-jackson</artifactId>  
 <version>0.11.1</version>  
 <scope>runtime</scope>  
</dependency>

#### 2. Disable CSRF Token

* The reason is JWT itself is a token that we can use to make sure that whoever calling my backend is a valid person because for every request we are going to validate the token that frontend is going to provide to us.
* E.g. In the configure(HttpSecurity http) method, .csrf().disable()

#### 3. Disable Default JSESSIONID generation/ Make Services as Stateless

* By default, JSESSIONID is getting created by the spring security framework and it is getting stored inside the HTTP Session of our server. So first we have to disable this default behavior.
* We can disable it by making it as Stateless as by default HTTP sessions are Stateful.
* Example –

protected void configure(HttpSecurity http) throws Exception {  
 http.sessionManagement()  
.sessionCreationPolicy(SessionCreationPolicy.STATELESS)  
.and()  
. . . // some more code  
.and().httpBasic();  
}

#### 4. Make CORS header changes

* When we generate a JWT token, we are going to send that inside the header with name “Authorization”. For the same, inside our CORS configuration, we have to allow this header to go to my (frontend) application which is consuming my backend services.
* Example –

protected void configure(HttpSecurity http) throws Exception {  
 http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and().  
 cors().configurationSource(new CorsConfigurationSource() {  
 @Override  
 public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {  
 CorsConfiguration config = new CorsConfiguration();  
 config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));  
 config.setAllowedMethods(Collections.singletonList("\*"));  
 config.setAllowCredentials(true);  
 config.setAllowedHeaders(Collections.singletonList("\*"));  
 config.setExposedHeaders(Arrays.asList("Authorization"));  
 config.setMaxAge(3600L);  
 return config;  
 }  
 }).and().csrf().disable()  
 . . . // some more code  
 .and().httpBasic();  
}

### Step 2: Configure Filters to generate and validate JWT Tokens

* Now during the initial login into my application, I have to generate a JWT token and for any other scenarios (already logged in) where the user want to access a protected resource, I want to validate whether the JWT token that got generated by my server initially is really a valid one or it got tampered or not.

#### Creating JWT Generator and Validator filters

* Create 2 filters – one to generate the JWT token and other one is to validate the JWT token for each and every request that is coming from the client.
* Create a Filter to **generate** the JWT token
  + E.g. Refer JWTTokenGeneratorFilter from our example.
  + This filter should be executed only once per request.
  + This filter should only execute for login requests. (Override shouldNotFilter() method to restrict all other paths.)
  + This filter will have logic of generating the JWT token.
  + And finally it will save the generated JWT token inside “Authorization” header in the response.
  + Finally configure this filter **after** authentication is successful.
* Create another filter to **validate** incoming JWT token
  + E.g. Refer JWTTokenValidatorFilter from our example.
  + This filter should be executed only once per request.
  + This filter should not execute for login requests. (Override shouldNotFilter() method to restrict all other paths.) But for all other paths, it should.
  + The logic in this filter is –
    - UI needs to send the JWT token.
    - Then this filter will get JWT token from the Authorization header
    - And we will use the same secret key and algorithm to generate new hash value and compare it with hash value of incoming JWT token.
    - If doesn’t match, we will throw unauthorized error.
    - If matches, we will create new Authentication object and set it into Spring Security Context.
    - So here we are not storing anything at the backend server. We are validating and generating Authentication object for each request.
  + Configure this filter **before** authentication filter because since already authentication has happened (via login route), we have to make sure that if the incoming token is compromised, we need to invalidate the request and send it to the UI application.

#### Configure JWT Generator and Validator filters

protected void configure(HttpSecurity http) throws Exception {  
 http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and().  
 cors().  
 . . . // some code  
 .and().csrf().disable()  
 .addFilterBefore(new JWTTokenValidatorFilter(), BasicAuthenticationFilter.class)  
 .addFilterAfter(new JWTTokenGeneratorFilter(), BasicAuthenticationFilter.class)  
 .authorizeRequests()  
 . . . // some code  
 .and().httpBasic();  
}

## Step 3: Making Changes at Front end / Client side

* On the front end, we have to capture the JWT token from the Authorization header and we need to make sure that we pass this JWT token in each future requests to backend.
* Example –

// Storing the JWT token  
 validateUser(loginForm: NgForm) {

    this.loginService.validateLoginDetails(this.model).subscribe(

      responseData => {

        window.sessionStorage.setItem("Authorization",responseData.headers.get('Authorization'));

        // some code

      }, error => { console.log(error); });

  }

// Passing the JWT token for future requests  
@Injectable()

export class XhrInterceptor implements HttpInterceptor {

// some code

  intercept(req: HttpRequest<any>, next: HttpHandler) {

    let httpHeaders = new HttpHeaders();

    this.user = JSON.parse(sessionStorage.getItem('userdetails'));

    if(this.user && this.user.password && this.user.email){

      httpHeaders = httpHeaders.append('Authorization', 'Basic ' + btoa(this.user.email + ':' + this.user.password));

    }

    let authorization = sessionStorage.getItem('Authorization');

    if(authorization){

      httpHeaders = httpHeaders.append('Authorization', authorization);

    }

    httpHeaders = httpHeaders.append('X-Requested-With', 'XMLHttpRequest');

    const xhr = req.clone({

      headers: httpHeaders

    });

   // some more code

  }

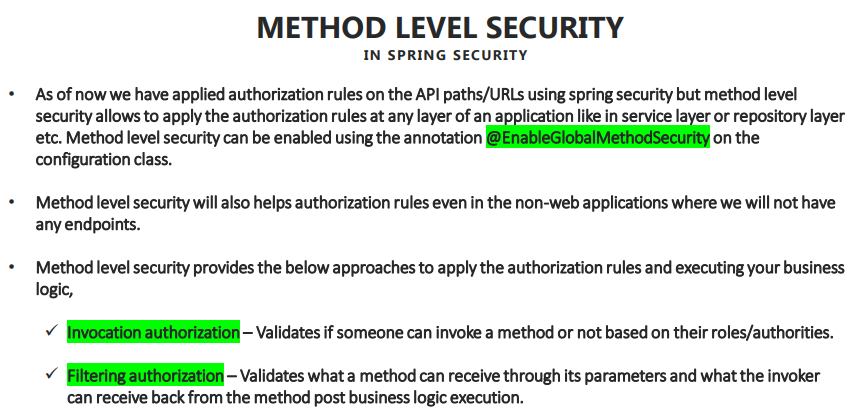
}

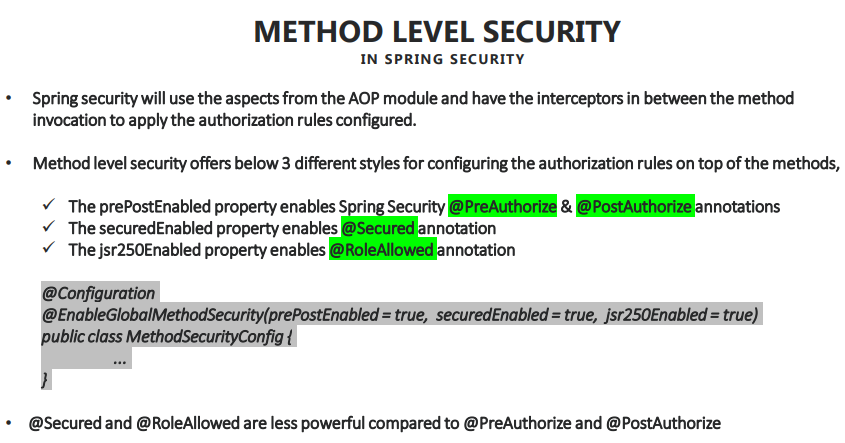
## Notes

* Complete example – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/07-JWT>

# Method Level Security

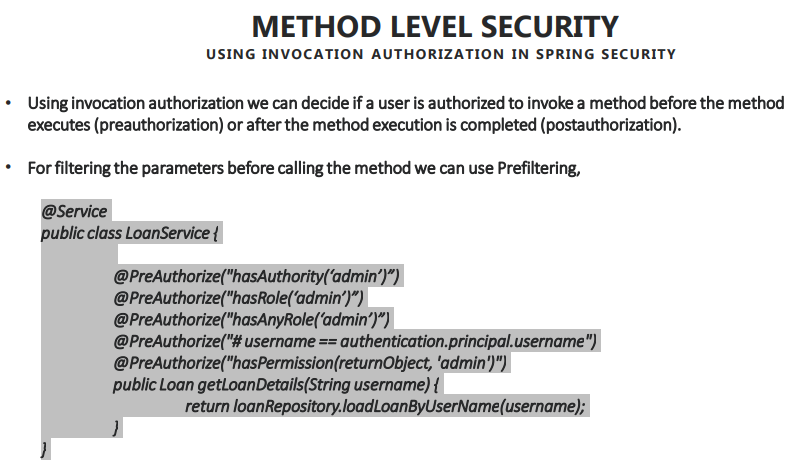
## Introduction

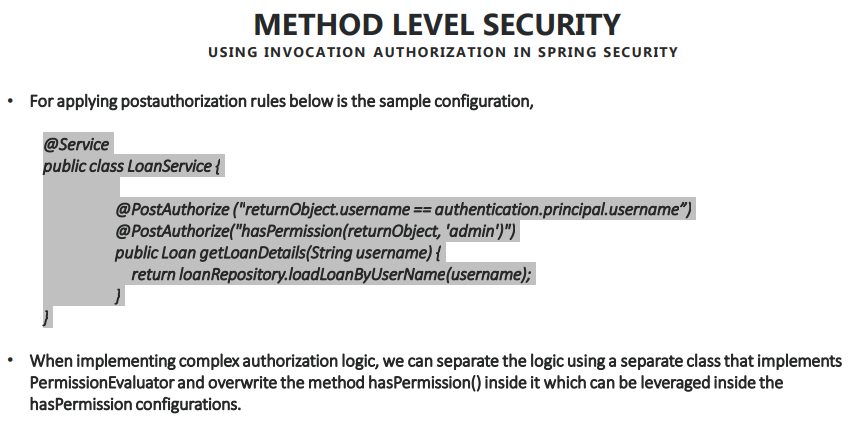




* Refer slide no. 59, 60 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Method Level Security make more sense for non-web applications.
* There are 2 approaches to implement method level security –
  + Invocation authorization
  + Filtering authorization

## Details about **method invocation authorization** in method level security



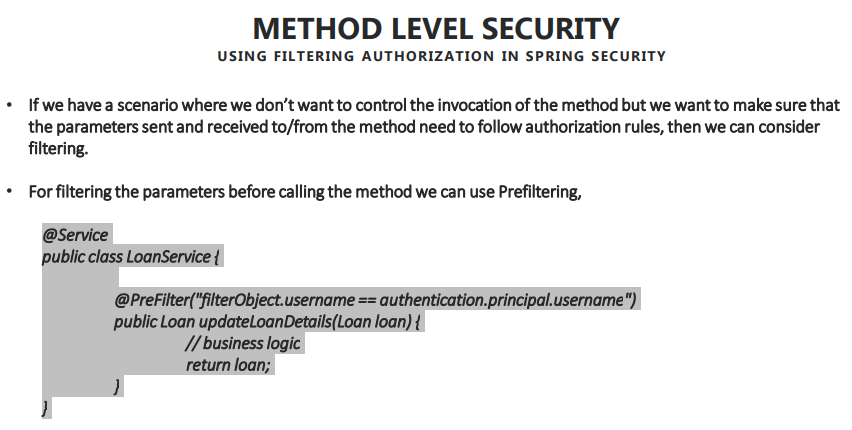


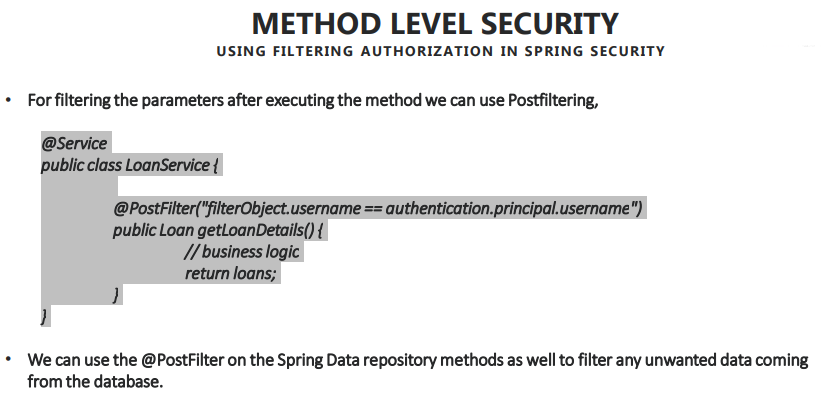
* Refer slide no. 61, 62 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Think of a scenario where inside your application you have a method at the service layer class LoanService and there is a method called getLoanDetails(). So this method will return all the loans of a specific user. So for such methods, if you have a requirement where you want to configure authorization rules like only an admin has to invoke this method or only the person who logged in can get his own loan details, but not other’s details. In such scenarios, we can use preauthorization annotation on top of the method.
* @PreAuthorize and @PostAuthorize are more powerful.
* @PreAuthorize is the most common.
* @PostAuthorize is dangerous to use. Be cautious.

## Implementing method level security using preauthorize and postauthorize

* You can keep the method level security on any layer like repository layer, controller layer or service layer.

## Details about **filtering authorization** in method level security





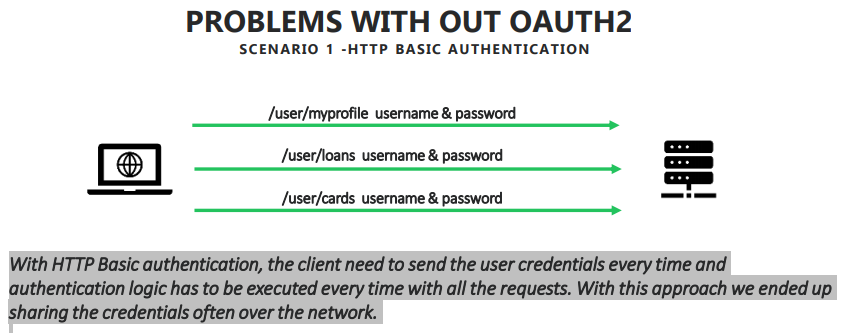
* Refer slide no. 63, 64 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* This has 2 approaches – @PreFilter and @PostFilter

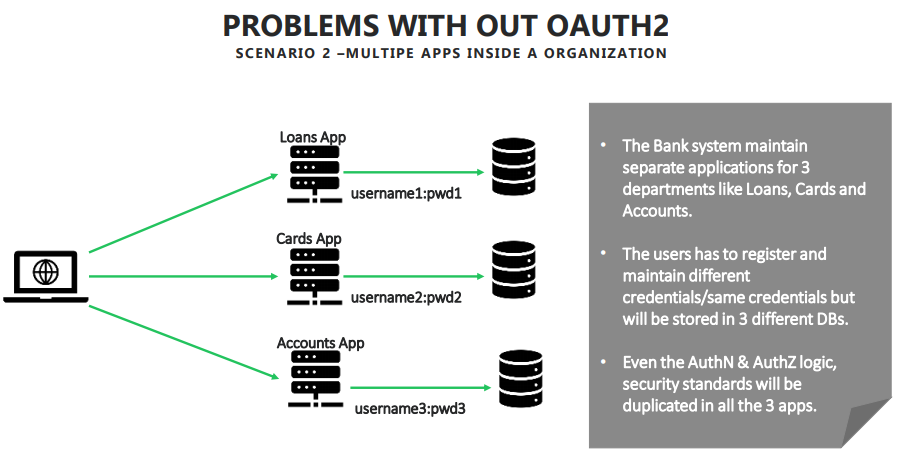
## Notes

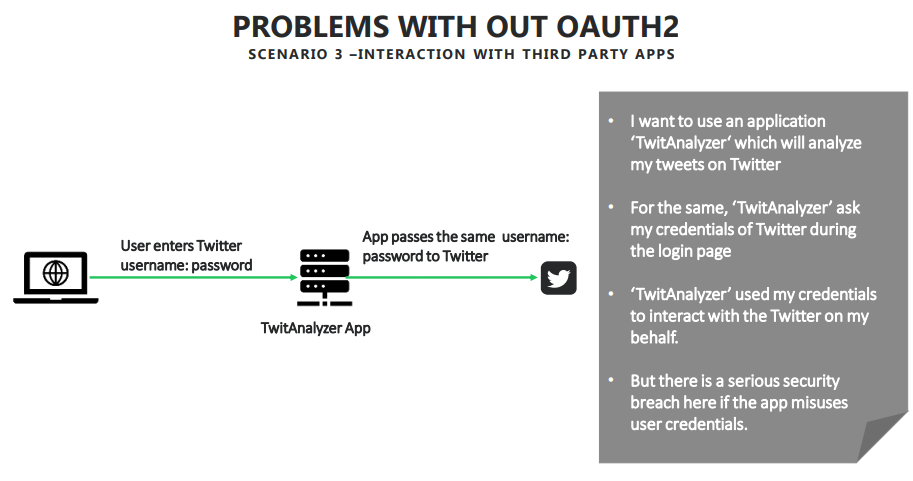
* Complete example – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/08-Method-Level-Security>

# Deep dive of OAUTH2

## Problems that OAUTH2 framework trying to solve

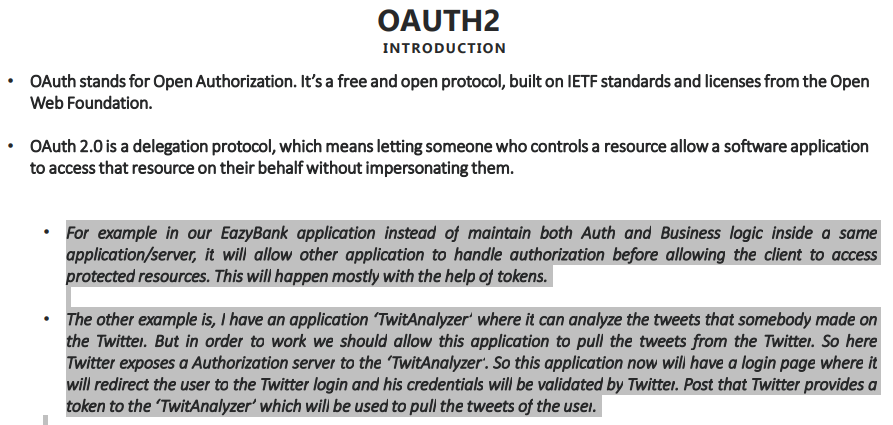


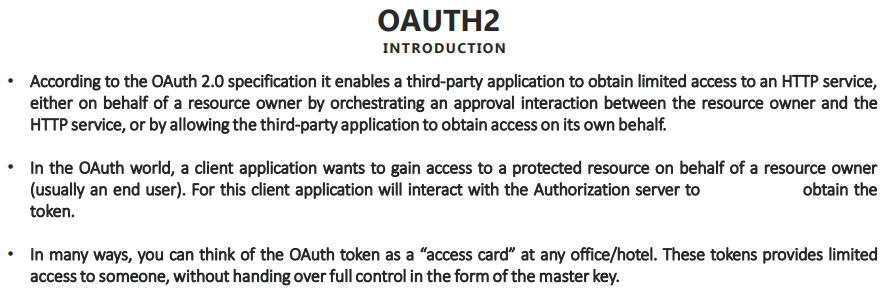




* Refer slide no. 66, 67, 68 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* OAUTH2 is the most standard and common authentication and authorization framework followed by many organizations.
* The most common scenario that OAUTH2 framework is trying to solve is the sharing of the credentials of the user over the network unnecessarily can be stopped by using OAUTH2 framework efficiently.
* OAUTH2 framework will leverage the token based authentication and authorization to make sure that we don't have to share the credentials every time between the client and the server.
* In case of multi-apps inside an organization, OAUTH2 framework can adopt a common authorization server inside your applications. That means all your authentication and authorization flows will be kept separately in a different server and that server will be leveraged whenever we try to authenticate and authorize inside any of the application used by the organization.
* OAUTH2 framework is also very useful while interacting with third party apps.

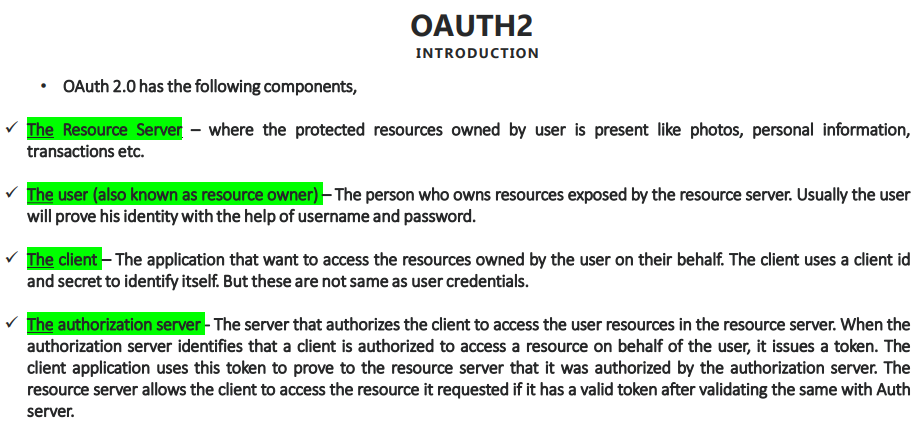
## Introduction to OAUTH2

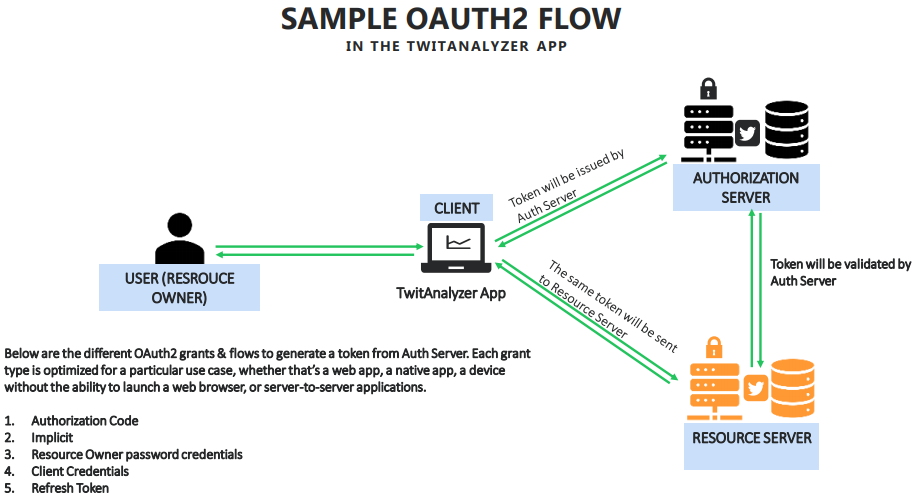




* Refer slide no. 69, 70 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* OAUTH2 stands for Open Authentication version 2.
* OAUTH / OAUTH1 has some limitations hence an upgrade to OAUTH2.
* The reason why OAUTH2 framework is so famous and everyone adopting it is that – OAUTH2 framework is a **delegation protocol** that means it will delegate the authentication and authorization of the user to something else so that we don't have to tie both authentication and authorization with the business logic that we maintain.
* OAUTH2 will allow you to decouple your authentication and authorization flow to another server **authorization server**. And at the same time, it will encourage you to maintain all your protected resources separately, like my accounts, my loans, my cards inside a separate server called **resource server**.
* So which means we are clearly drawing a boundary between authentication and actual protected resources so that authorization server can take of authentication and authorization, whereas the resource server will hold the resources. And if someone asks to give the resources, it will ask for the tokens from the auth server. If the token is valid, then only it will give the resources that users are requesting.
* If we are interacting with **third party applications** like in many websites you might have seen, like if you want to sign up very first time, you don't have to enter your last name, first name, email, mobile, because there are faster ways to achieve that by using Google, GitHub, Twitter, Facebook, because already these organizations have a basic information about me like my last name, my first name, my email. So instead of entering all those credentials again in the new application that I want to sign up, what I can do is I can tell the new application that I have my protected resources inside Facebook, like my last name, first name, email, so go and get from the Facebook. So as soon as a user clicks on the Facebook login inside that application, that application will redirect to Facebook login page where I can enter my facebook credentials. Once my authentication is successful there, Facebook will share a token to this new application to get basic details about me so that the registration form will be fast and efficient.
* The OAUTH2 token is kind of temporary access card. Temporary because it will have expiry or user can manually invalidate it.

## Different Components involved in OAUTH2 flow





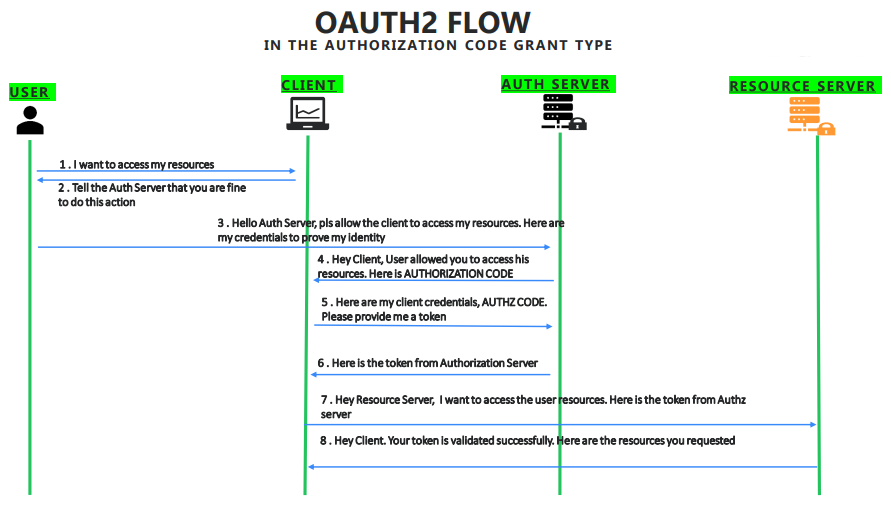
* Refer slide no. 71, 72 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Whenever some application (stack overflow) wants to use the Google Auth and Resource servers and it also wants to get the basic details, first of all they have to register themselves with Google.
* So first Stack Overflow has to reach out to the Google developer website and register themselves where they'll get to clientId and client secret, which they can use over the OAUTH2 flow.

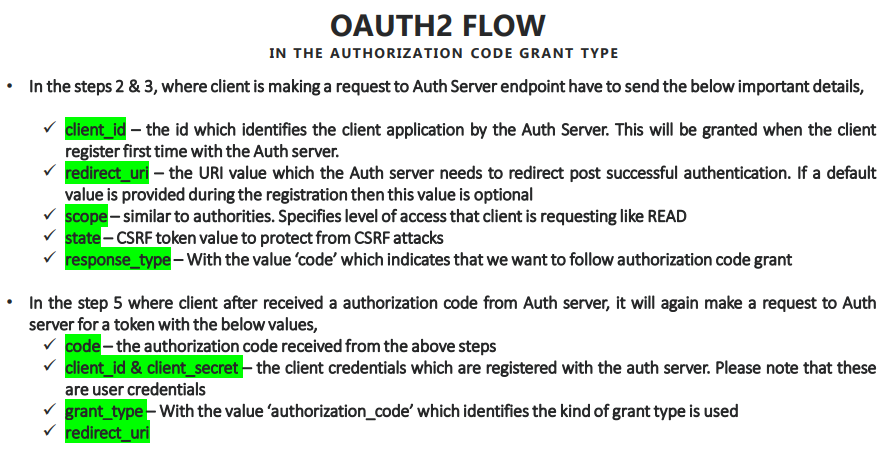
## Grant Types / Auth Flows

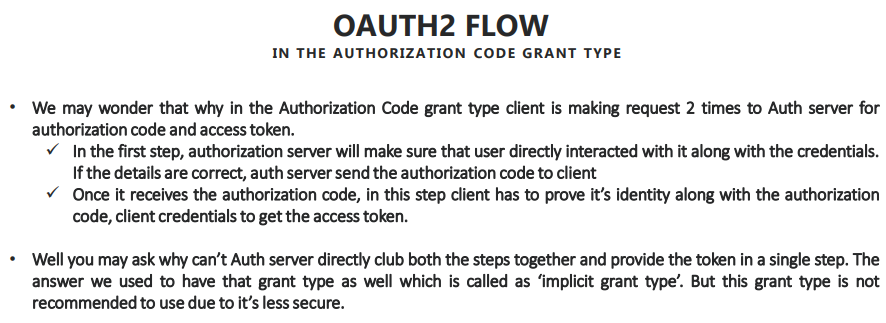
* There are five different flavors (Grant Types / Auth Flows) where OAUTH2 will follow to issue a token and validate the tokens during the authentication and authorization flow. Depending on the scenario that you are into, you can leverage any of these flows and all these flows have their own advantages and disadvantages. And obviously you have to be wise enough to choose the most optimum one for your application.
* Grant Types / Auth Flows means these are the flows, that authorization servers, client and resource servers follow in order to generate tokens, valid tokens and interact with each other.
* Those are –

1. Authorization Code
2. Implicit
3. Resource Owner password credentials
4. Client Credentials
5. Refresh Token

### Authorization Code Grant Type flow in OAUTH2

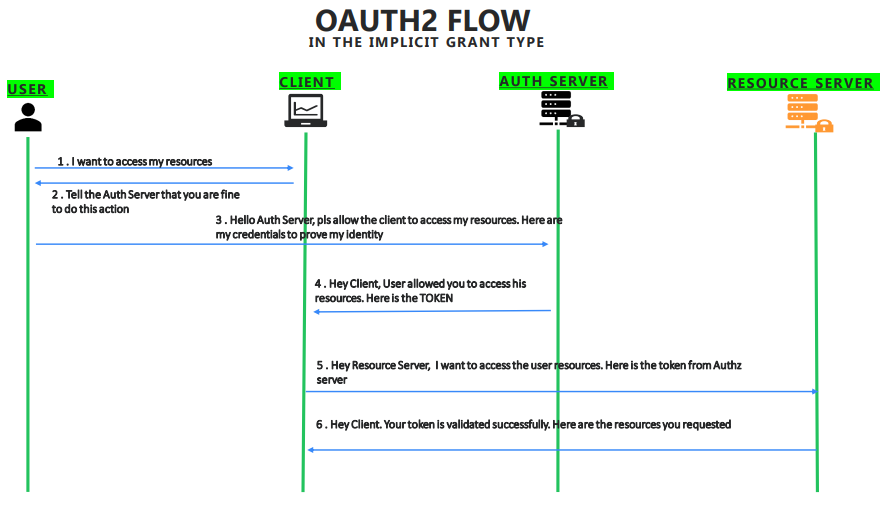






* Refer slide no. 73, 74, 75 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* E.g. Similar to login to stackoverflow via Google account.
* As shown in the slide 73, the AUTHORIZATION CODE is NOT the access token.
* “Authorization Code” Grant Type is the most commonly used.

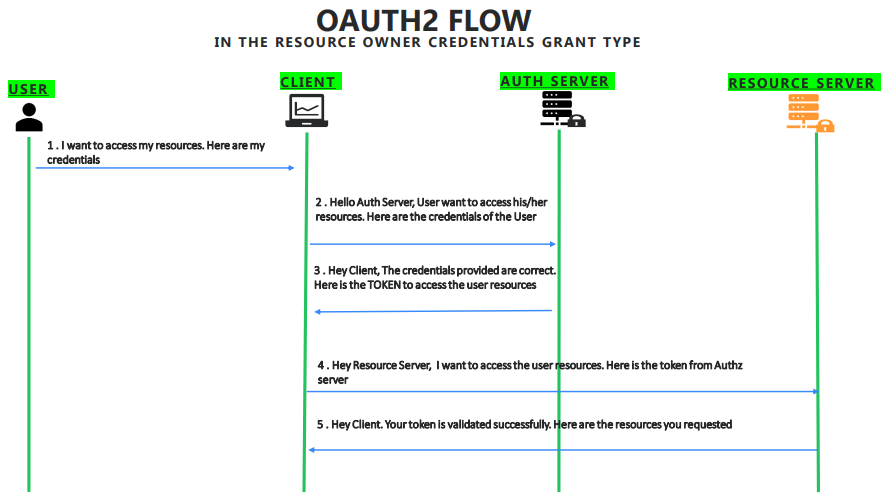
### Implicit Grant Type flow in OAUTH2

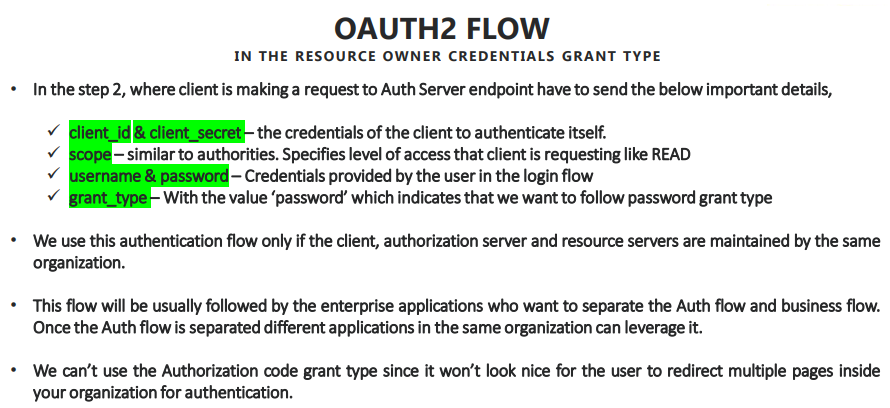




* Refer slide no. 76, 77 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Implicit Grant Type is not much used as it is less secured compared to Authorization Code Grant Type.
* The primary difference is there is no client secret involved here that means anyone who identifies the clientId of say stack overflow in the browser address bar, they can take it and they can make a request to the authorization server of say Google.
* Implicit Grant Type is a savior for the applications where they have only UI like they have only HTML and JavaScript code, but they do not have any backend where they can save the client secret on some backend server.

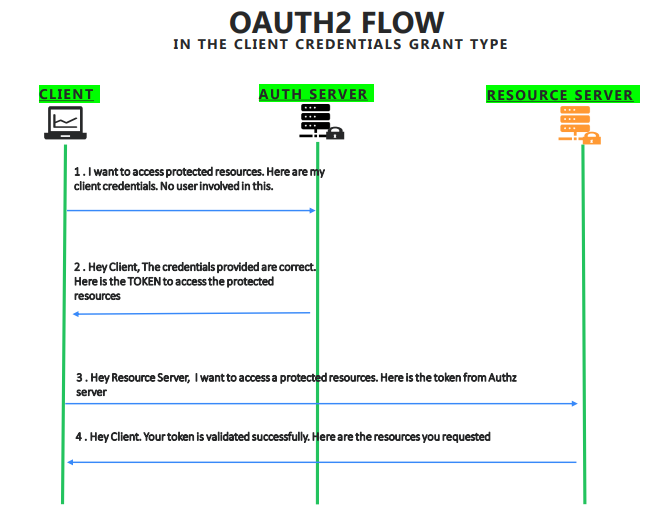
### Resource Owner Credentials Grant Type flow in OAUTH2

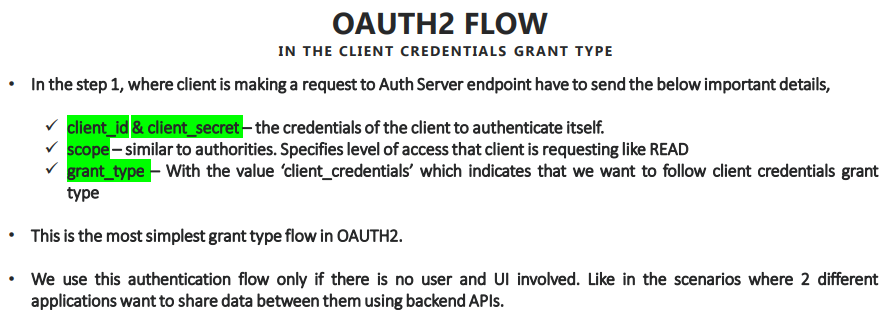




* Refer slide no. 78, 79 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Here resource server, auth server and client application all belong to the same organization. In such scenario, we avoid multiple redirects and provide some good experience to the user, we don't redirect the user to the auth server.
* So due to this reason, whenever user want to access some application like loan application, there'll be a login page and there I enter my credentials on the loans application login page. My client application will capture my username and password and it will send that request to the auth server in the backend without redirect experience to the user.
* With Resource Owner credentials Grant Type, we give a better experience to the user by avoiding multiple redirects since all these applications belong to the same organization.

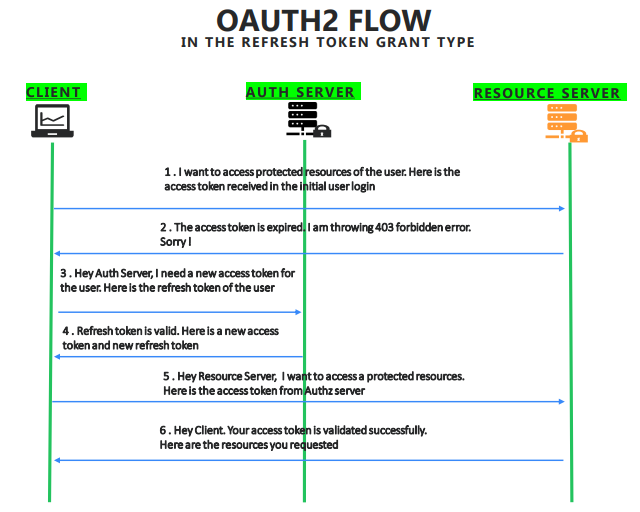
### Client Credentials Grant Type flow in OAUTH2

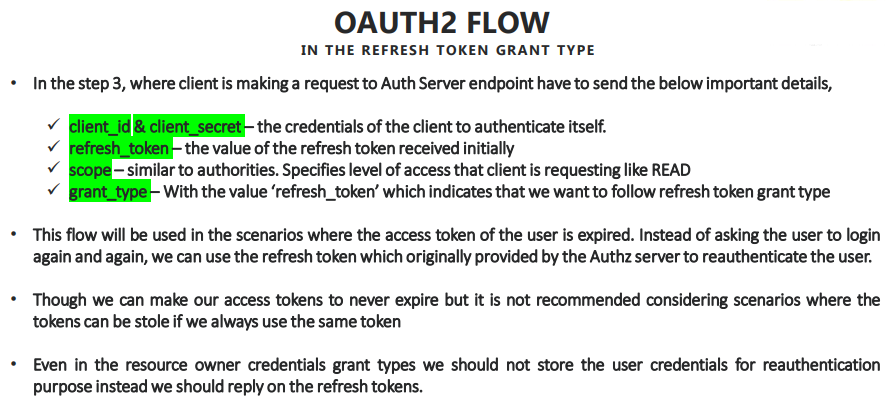




* Refer slide no. 80, 81 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Think of a scenario where you don't have any UI application, where you don't have any resource or user involved. You have multiple backend applications, which they share data between them, like in the scenario of micro services, where hundreds of applications trying to interact with each other and share some secured information between them.
* So in such scenarios, instead of maintaining the authentication and authorization servers inside each and every application, you decide to keep that all authentication and authorization logic in a separate server called auth server and every application before it tries to communicate with the other application inside your organization, they have to prove their identity and get the access token from the auth server, which will be eventually shared to the other application to get the resources from it.
* There is no involvement of resource owner or user this auth flow.

### Refresh Token Grant Type flow in OAUTH2





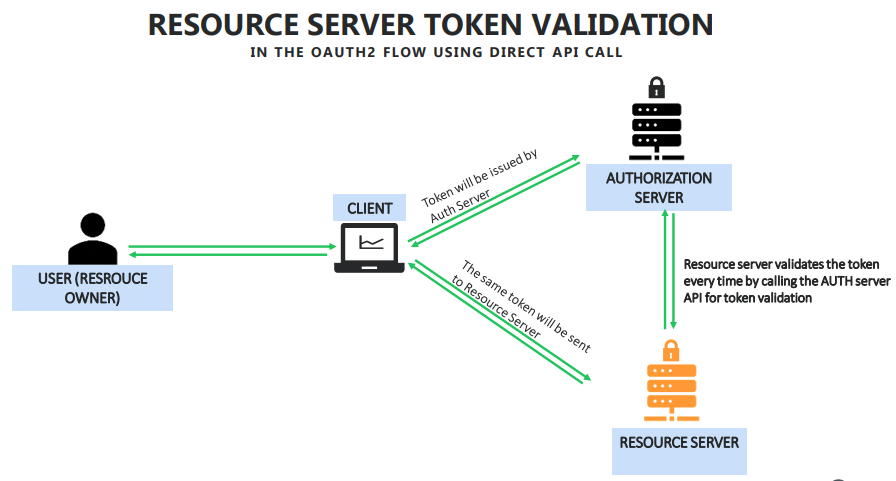
* Refer slide no. 82, 83 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* Whenever we interact with auth server initially during the authrorize code grant type or implicit grant, along with the access token, you also get the refresh token from the auth server. You save that refresh token and access token in the database.
* And whenever you get an error from the auth server saying that you are access token is expired, instead of you asking the resource owner or user to enter the credentials again, what you will be doing is you will be sending the saved refresh token to the auth server with the grant type as “refresh”, indicating that my original access token which you shared is expired and now I want to get the new access token.
* So in such scenarios, auth Server will validate the refresh token that it originally issued. If the refresh token is valid and if it belongs to the same user it initially issued, now the auth server will issue again a new access token with the new expiration time along with the new refresh token. Please do remember we also get a new refresh token. We can't use the same refresh token again and again. That's the purpose of the refresh token as a name indicates.
* So here there is no user involved or resource owner involved because my client don't want to ask the user for his credentials in the Gmail application instead, it can rely on the refresh token, which is originally issued from the Google auth server.
* Even in case of Resource Owner Credentials Grant Type also for re-authentication and re-authorization purpose, we should never store the resource owner credentials, instead, we should leverage the refresh tokens in order to get the new access token every time.

## How resource server validates the tokens issued by Auth server

* Every time resource server receives access token from the client, it has to validate that with the auth server because these are two different servers, are two different applications handling different responsibilities.
* Resource Server is not aware of the access tokens issued by the auth server to all the clients. So there should be some way for the resource server to validate that access token with the auth server.
* For the same, we have three different ways that most of the organizations follow, and you can also follow any one of them based upon your requirements.

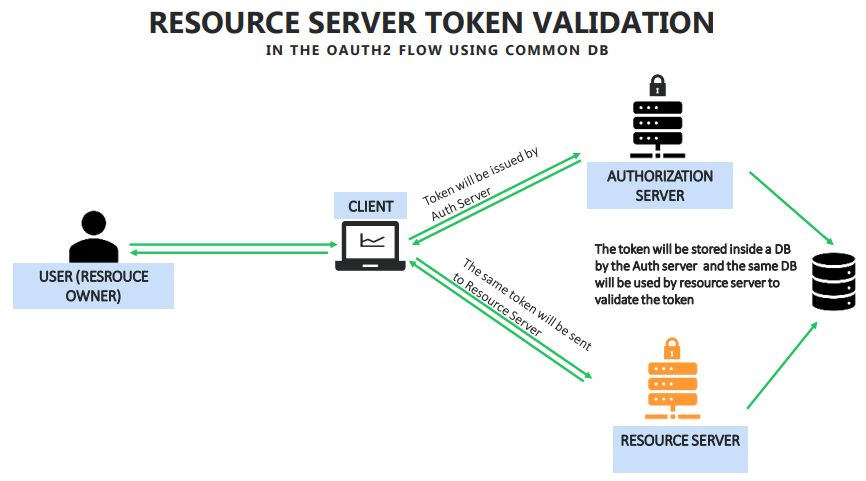
### Ways of Validating Access Tokens

#### By making direct API calls



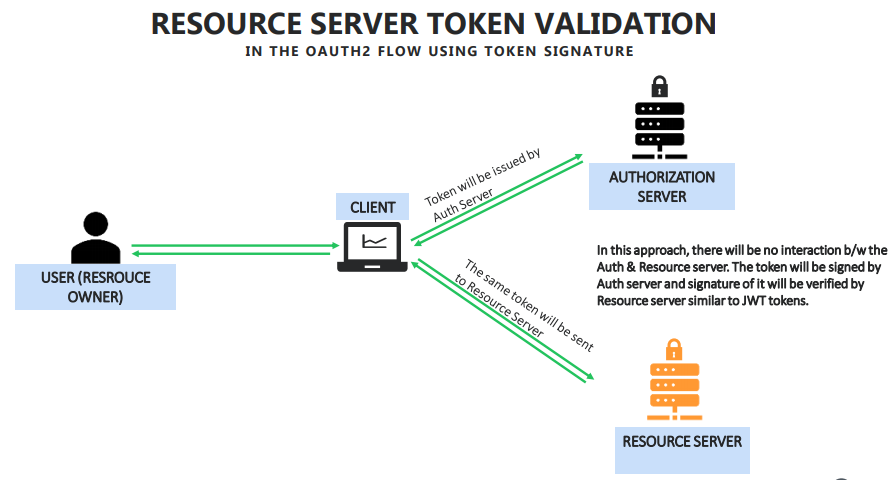
* Refer slide no. 84 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* The very first and basic way of validating the access token by a resource server with the authentication server is by making a REST API call in the backend, so whenever a resource server receives the access token from the client, it can simply make a REST API call to the auth server by sending I received this access token from this clientId, do you approve it or not?

#### Using Common DB



* Refer slide no. 85 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* The second way is by having a common database, like whenever an auth server issuing an access token, it can write into the common database and the same database can be pointed by the resource server also. So in such scenarios, resource server doesn't have to rely on the network to interact with the auth server. It can directly check in the database whether the access token received from the client is really issued by the auth server or not and what time it issued, whether it is expired or not, all sorts of details can be validated by using a common database between auth server and resource server.

#### Using Token Signature



* Refer slide no. 86 from [Course Notes](https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf)
* The last approach is that resource server can validate the signature of the token.
* So similar to JWT tokens, the signature of the tokens can be validated by having some secret keys maintained by the both parties to make sure that no one tampered it.
* So following the same approach auth server can generate an access token by using some encryption algorithm with the secret. And the resource server can simply check the signature or hash value of the token generated with the secret key that it maintains to understand if the token is valid or not.

# Implementing OAUTH2 using Spring Security

## Registering the client details with the GitHub to use it's OAUTH2 Auth server

* Anyone (client applications) who want to use the auth servers of organizations like Google, Github, Facebook, etc. first they have to approach them and register themselves with their own details.
* For instance for Github, you can register for using Github auth servers by logging into github.com -> click on profile -> Settings -> Developer Settings -> OAuth Apps -> Register a New App.
* After registration, you will get clientId and client secret which you need to use from your client application.

## Building client application that uses OAUTH2

* Complete example – <https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/09-OAUTH2-GitHub>

### 1. Add Dependencies in pom.xml

* Add spring-boot-starter-oauth2-client dependency in pom.xml as we are the consumer of OAUTH2.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-oauth2-client**</artifactId>

</dependency>

* Apart from this, we should also have spring-boot-starter-security and spring-boot-starter-web dependencies.

### 2. Use OAUTH2 login in your WebSecurityConfigurerAdapter

* Next, in your implementation of **WebSecurityConfigurerAdapter’s configure** method, make sure you use oauth2 authentication.

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http.authorizeRequests().anyRequest().authenticated()  
 .and().oauth2Login();  
}

* This will add **OAuth2LoginAuthenticationFilter** in the spring filter chain.

### 3. Side Note

* There are different components involved – user, client, auth server and resource server.
* Here the resource server is this application itself which we are building and resource is secure.html page, resource owner is the one who is trying to access secure.html

### 4. Register the client in your WebSecurityConfigurerAdapter

* Register the client inside our application in your implementation of **WebSecurityConfigurerAdapter**.
* There are 3 ways to do so.

#### Way 1: Complete configuration

* This approach is useful whenever you are building the auth server inside your organization only. So it involves lot of configurations.
* Example –

private ClientRegistration clientRegistration() {   
 ClientRegistration cr = ClientRegistration  
 .withRegistrationId("github")  
 .clientId( "3c9be97074f067e78e75")

.clientSecret("ab313f7ade3d79e06c192ca80cf152c43cb5d916")  
 .scope(new String[]{ "read:user" })

.authorizationUri("https://github.com/login/oauth/authorize") .tokenUri("https://github.com/login/oauth/access\_token")  
 .userInfoUri("https://api.github.com/user") .userNameAttributeName("id")

.clientName("GitHub")

.authorizationGrantType(AuthorizationGrantType.AUTHORIZATION\_CODE) .redirectUriTemplate("{baseUrl}/{action}/oauth2/code/{registrationId}")  
 .build();   
 return cr;   
}

#### Way 1: Simple Approach

* Use this if you have a scenario where you are using the most common auth server available in the industry, like GitHub, Google, Facebook.
* Here we can leverage the CommonOAuth2Provider from spring security, which has inbuilt support for most common oauth2 servers like Google, Github, etc.
* Basically this CommonOAuth2Provider has all the configuration that we have to manually add like in Way 1 above.
* So implement the clientRegistration() method in the WebSecurityConfigurerAdapter.  
  Example –

private ClientRegistration clientRegistration() {   
 return CommonOAuth2Provider.GITHUB.getBuilder("github")  
 .clientId("e482d40474aaaec77980")  
 .clientSecret("dcd7d4f3b2fabeaf8a8646b0d1d653a4378170e9")  
 .build();   
}

* Once we have this client registration values available, you should also create a client registration repository.

E.g.

@Bean  
 public ClientRegistrationRepository clientRepository() {  
 ClientRegistration clientReg = clientRegistration();   
 return new InMemoryClientRegistrationRepository(clientReg);   
}

Mark it as @Bean so that Spring will take care of using it.

* Client registration repository is same as our UserDetailService in normal authentication flow. In the same way, in OAUTH2 flow, we have a ClientRegistrationRepository which has a method findByRegistrationId(). So this method will use the registration Id to get the client registration details that we configured inside clientRegistration() method.
* You can use InMemoryClientRegistrationRepository or JdbcClientRegistrationRepository.

#### Way 3: Simplest approach

* You can leverage Spring boot auto-configuration to do client registration as well as ClientRegistrationRepository.
* Simply go to application.properties and mention below values if you are going to use GitHub auth server –

spring.security.oauth2.client.registration.**github**.client-id=<clientId>

spring.security.oauth2.client.registration.**github**.client-secret=<secret>

* Similarly you can use other auth server providers like Google, Facebook, etc.

# Tips and Tricks

* Repository – <https://github.com/sameerbhilare/Spring-Security>
* Instructor Notes – <https://github.com/sameerbhilare/Spring-Security/blob/main/Resources/Spring%20Security%20Notes.pdf>
* By default, spring security will try to secure all the services that we configure inside the project.
* Use @EnableJpaRepository annotation on Spring Boot Application main class if your repositories are not in the same or child packages of @SpringBootApplication class.
* Use @EntityScan annotation on Spring Boot Application main class if your entities are not in the same or child packages of @SpringBootApplication class.
* The term ‘Encoder’ here in PasswordEncoder is a generic term which can be used for encoding, encryption and hashing. It doesn’t mean it only supports encoding.
* If you want to generate hashed bcrypt password, you can use online utilities like <https://bcrypt-generator.com/>