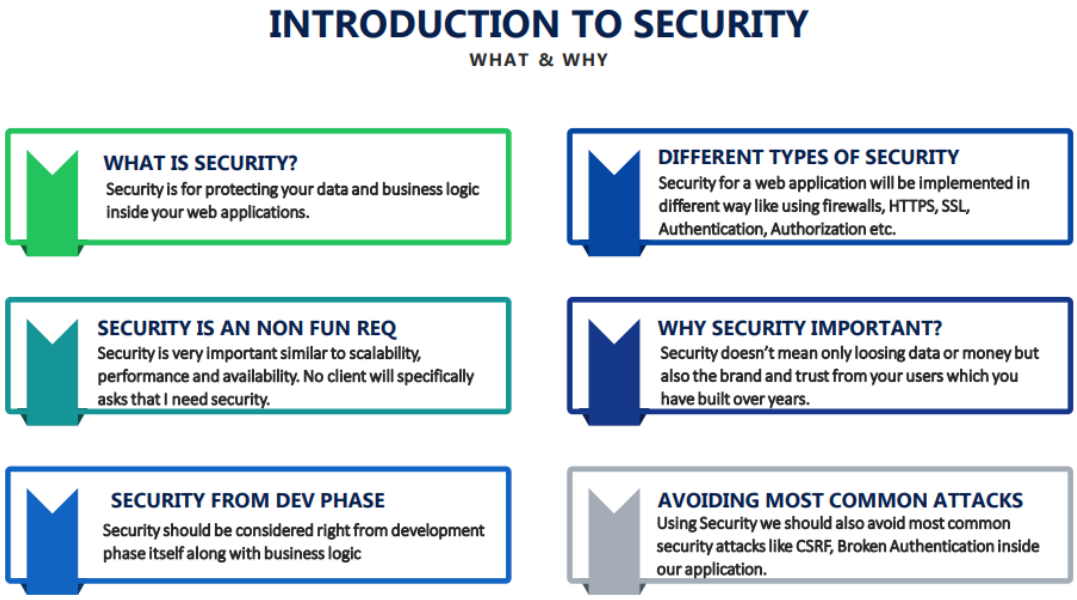
Spring Security

Contents

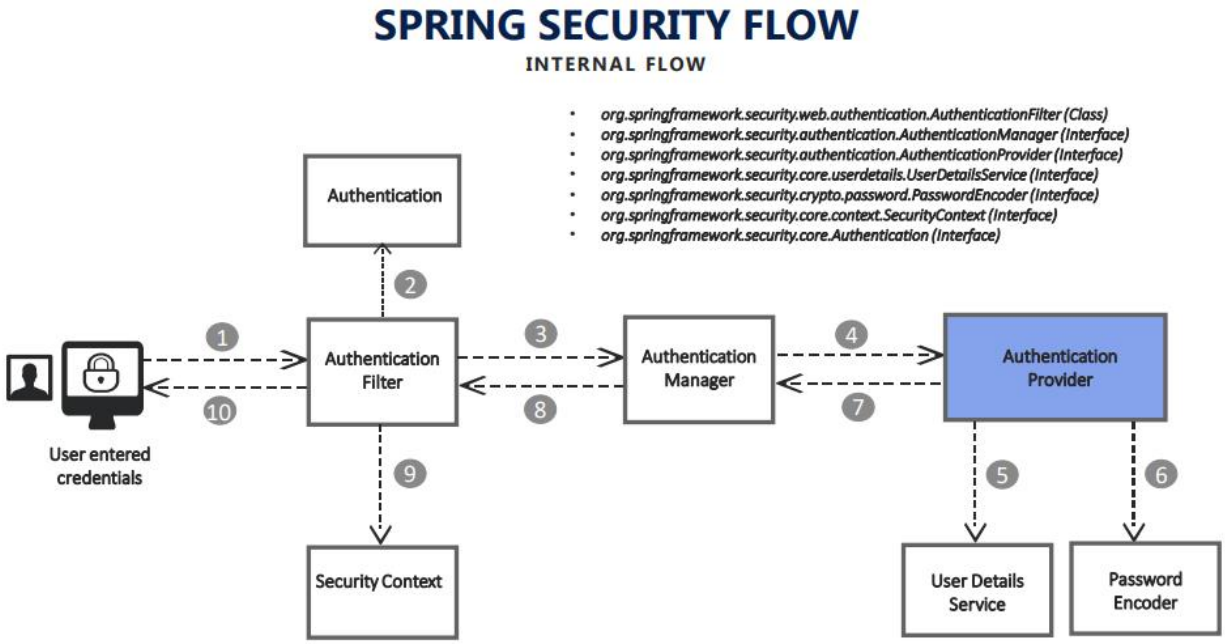
Getting Started

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

Introduction to Security



Spring Security Flow



SPRING SECURITY FLOW

INTERNAL FLOW API DETAILS

1.AuthenticationFilter: A filter that intercepts and performs authentication of a particular request by delegating it to the authentication manager. If authentication is successful, the authentication details is set into SecurityContext.

2.Authentication: Using the supplied values from the user like username and password, the authentication object will be formed which will be given as an input to the AuthenticationManager interface.

3.AuthenticationManager: Once received request from filter it delegates the validating of the user details to the authentication provider.

4.AuthenticationProvider: It has all the logic of validating user details using UserDetailsService and PasswordEncoder.

5.UserDetailsService: UserDetailsService retrieves UserDetails and implements the User interface using the supplied username.

6.PasswordEncoder: Service interface for encoding passwords.

7.SecurityContext: Interface defining the minimum security information associated with the current thread of execution. It holds the authentication data post successful authentication

- 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.

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(AuthenticationManagerBuilder 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 –

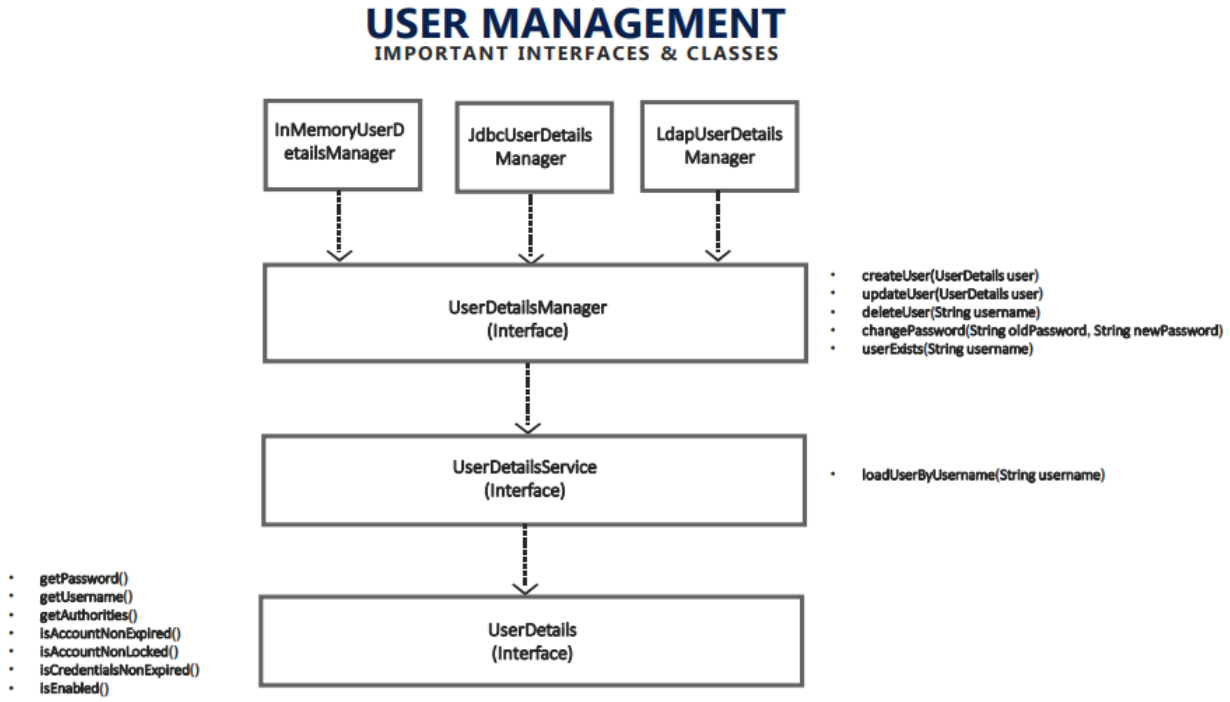
```
@Override
 protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
  auth.inMemoryAuthentication()
        .withUser("admin").password("12345").authorities("admin")
        .and()
        .withUser("user").password("12345").authorities("read")
        .and()
       .passwordEncoder(NoOpPasswordEncoder.getInstance());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

Configuring Users using InMemoryUserDetailsManager

- Here is another way to configure in memory users using InMemoryUserDetailsManager.
- Override **`WebSecurityConfigurerAdapter`**
  `.configure(AuthenticationManagerBuilder 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
- 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

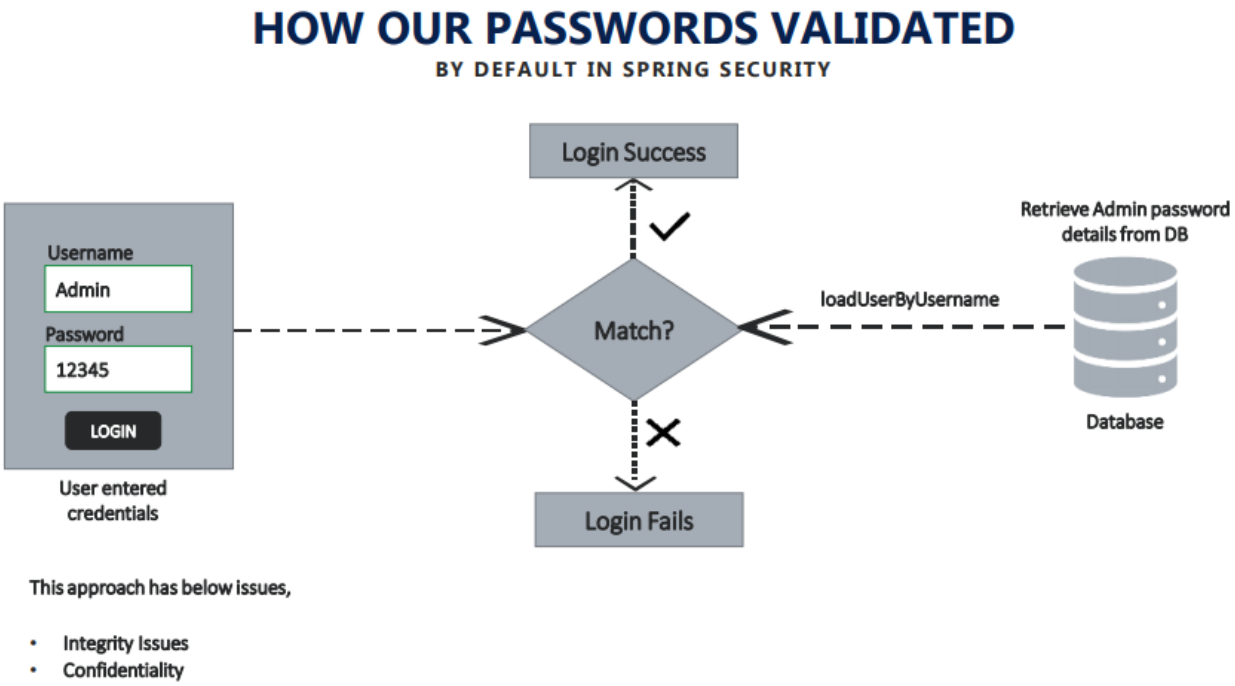
- If we have our own database tables which don't adhere to the requirement of built in `JdbcUserDetailsManager` class, we can create our own implementation of UserDetailsService.
- Use Spring Data JPA.
- Refer implementation of `JdbcUserDetailsManager` class.
- Refer Code – https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/01-ManagingUsers

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.

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

ENCODING

DETAILS

- Encoding is defined as the process of converting data from one form to another and has nothing to do with cryptography. It guarantees none of the 3 cryptographic properties of confidentiality, integrity, and authenticity because it involves no secret and is completely reversible.
- Encoding can be used for reducing the size of audio and video files. Each audio and video file format has a corresponding coder-decoder (codec) program that is used to code it into the appropriate format and then decodes for playback.
- It can't be used for securing data, various publicly available algorithms are used for encoding.

Example: ASCII, BASE64, UNICODE

ENCRYPTION

DETAILS

- Encryption is defined as the process of transforming data in such a way that guarantees confidentiality. To achieve that, encryption requires the use of a secret which, in cryptographic terms, we call a "key".
- Encryption is divided into two categories: symmetric and asymmetric, where the major difference is the number of keys needed.
- In symmetric encryption algorithms, a single secret (key) is used to both encrypt and decrypt data. Only those who are authorized to access the data should have the single shared key in their possession.

Example: file system encryption, database encryption e.g. credit card details

- On the other hand, in asymmetric encryption algorithms, there are two keys in use: one public and one private. As their names suggest, the private key must be kept secret, whereas the public can be known to everyone. When applying encryption, the public key is used, whereas decrypting requires the private key. Anyone should be able to send us encrypted data, but only we should be able to decrypt and read it!

Example: TLS, VPN, SSH

HASHING

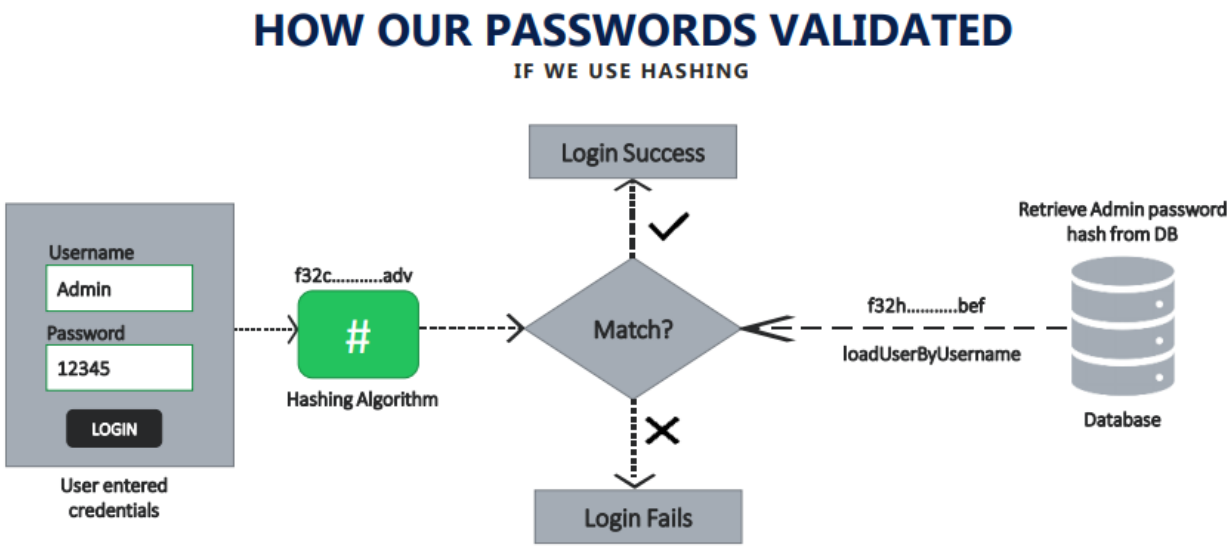
DETAILS

- In hashing, data is converted to the hash using some hashing function, which can be any number generated from string or text. Various hashing algorithms are MD5, SHA256. Data once hashed is non-reversible.
- One cannot determine the original data given only the output of a hashing algorithm.
- Given some arbitrary data along with the output of a hashing algorithm, one can verify whether this data matches the original input data without needing to see the original data
- You may have heard of hashing used in the context of passwords. Among many uses of hashing algorithms, this is one of the most well-known. When you sign up on a web app using a password, rather than storing your actual password, which would not only be a violation of your privacy but also a big risk for the web app owner, the web app hashes the password and stores only the hash.
- Then, the next time you log in, the web app again hashes your password and compares this hash with the hash stored earlier. If the hashes match, the web app can be confident that you know your password even though the web app doesn't have your actual password in storage.

Example: Password management, verify the integrity of the downloaded file

- Refer slide no. 13, 14, 15 from Course Notes

How our Passwords will be validated with Hashing



- Refer slide no. 16 from Course Notes

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

DEFINITION OF THE PASSWORDENCODER

DETAILS

```
public interface PasswordEncoder {

        String encode(CharSequence rawPassword);

        boolean matches(CharSequence rawPassword, String encodedPassword);

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

Different Implementations of PasswordEncoders provided by Spring Security

- *NoOpPasswordEncoder*
- *StandardPasswordEncoder*
- *Pbkdf2PasswordEncoder*
- *BCryptPasswordEncoder*
- *SCryptPasswordEncoder*

- Refer slide no. 17 from Course Notes
- 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
- 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 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
- 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

Bcrypt & Scrypt PasswordEncoder

DETAILS

- BCryptPasswordEncoder uses a BCrypt strong hashing function to encode the password. You could instantiate the BCryptPasswordEncoder by calling the no-arguments constructor. But you also have the option to specify a strength coefficient representing the log rounds used in the encoding process. Moreover, you can as well alter the SecureRandom instance used for encoding.

```
PasswordEncoder p = new BCryptPasswordEncoder();
PasswordEncoder p = new BCryptPasswordEncoder(4);
SecureRandom s = SecureRandom.getInstanceStrong();
PasswordEncoder p = new BCryptPasswordEncoder(4, s);
```

- SCryptPasswordEncoder uses a SCrypt hashing function to encode the password. For the SCryptPasswordEncoder, you have two options to create its instances:

```
PasswordEncoder p = new SCryptPasswordEncoder();
PasswordEncoder p = new SCryptPasswordEncoder(16384, 8, 1, 32, 64);
```

- Refer slide no. 19 from Course Notes
- Refer BCryptPasswordEncoder Javadoc
- Refer SCryptPasswordEncoder Javadoc
- 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

Pbkdf2PasswordEncoder

DETAILS

- Password-Based Key Derivation Function 2 (PBKDF2) is a pretty easy slow-hashing function that performs an HMAC (Hashed Message Authentication Code) as many times as specified by an iterations argument.
- The three parameters received by the last call are the value of a key used for the encoding process, the number of iterations used to encode the password, and the size of the hash. The second and third parameters can influence the strength of the result.
- You can choose more or fewer iterations as well as the length of the result. The longer the hash, the more powerful the password is.

```
PasswordEncoder p = new Pbkdf2PasswordEncoder();
PasswordEncoder p = new Pbkdf2PasswordEncoder("secret");
PasswordEncoder p = new Pbkdf2PasswordEncoder("secret", 185000, 256);
```

- Refer slide no. 18 from Course Notes
- Refer Pbkdf2PasswordEncoder Javadoc
- 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().antM
atchers("/myBalance").authenticated()

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

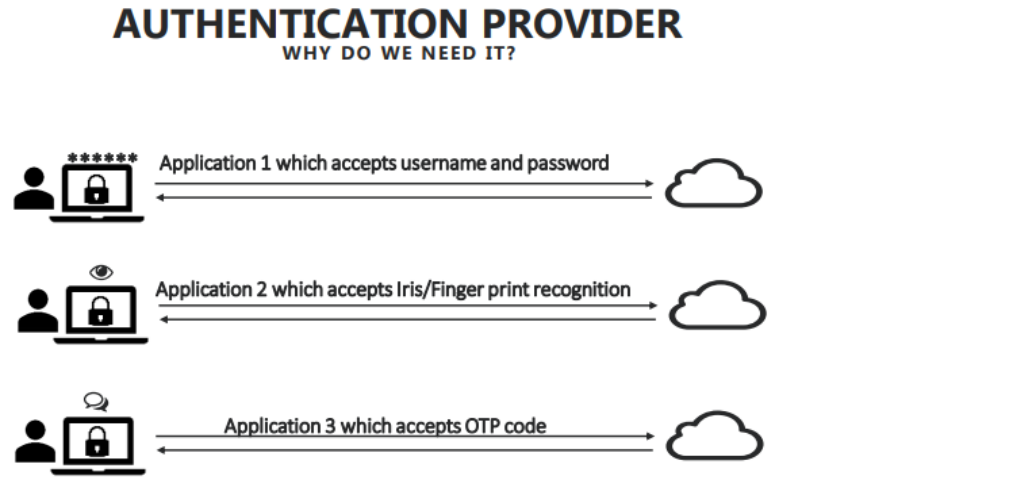
- In the Spring Security Crypt package we have many utility interfaces, classes, implementations provided by spring security for various scenarios that we face on day to day basis.
- Suppose you want to generate a random key salt then for such scenarios, we have leverage sample implementations from org.springframework.security.crypto.keygen package.
- If you want to encrypt and decrypt the data for such scenarios, also we have a package called org.springframework.security.crypto.encrypt
- 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
- 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

AUTHENTICATION PROVIDER DEFINITION

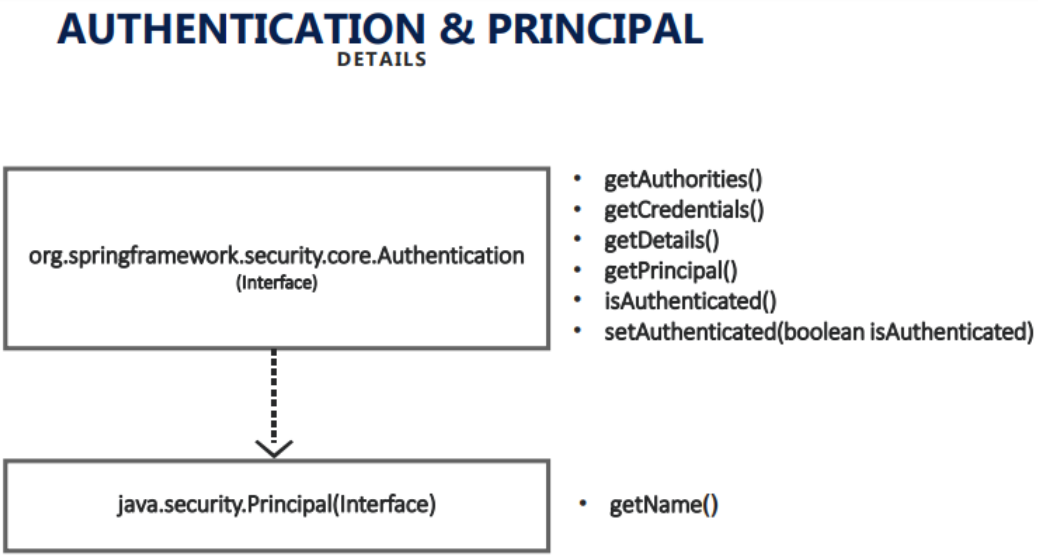
DETAILS

```
public interface AuthenticationProvider {

        Authentication authenticate(Authentication authentication)
                        throws AuthenticationException;

        boolean supports(Class<?> authentication);
}
```

- The authenticate() method receives an Authentication object as a parameter and returns an Authentication object as well. We implement the authenticate() method to define the authentication logic.
- The second method in the AuthenticationProvider interface is supports(Class<?> authentication). You'll implement this method to return true if the current AuthenticationProvider supports the type provided as the Authentication object.

- Refer slide no. 22 from Course Notes
- `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
- `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

CROSS-ORIGIN RESOURCE SHARING (CORS)

HOW TO HANDLE IT USING SPRING SECURITY

1. CORS is a protocol that enables scripts running on a browser client to interact with resources from a different origin.
2. For example, if a UI app wishes to make an API call running on a different domain, it would be blocked from doing so by default due to CORS. So CORS is not a security issue/attack but the default protection provided by browsers to stop sharing the data/communication between different origins.
3. "other origins" means the URL being accessed differs from the location that the JavaScript is running from, by having:
    - a different scheme (HTTP or HTTPS)
    - a different domain
    - a different port

CROSS-ORIGIN RESOURCE SHARING (CORS)

HOW TO HANDLE IT USING SPRING SECURITY

4. But what if there is a legitimate scenario where cross-origin access is desirable or even necessary. For example, in our EazyBank application where the UI and backend are hosted on two different ports.
5. When a server has been configured correctly to allow cross-origin resource sharing, some special headers will be included. Their presence can be used to determine that a request supports CORS. Web browsers can use these headers to determine whether a request should continue or fail.
6. First the browser sends a pre-flight request to the backend server to determine whether it supports CORS or not. The server can then respond to the pre-flight request with a collection of headers:
    - *Access-Control-Allow-Origin: Defines which origins may have access to the resource. A '*' represents any origin*
    - *Access-Control-Allow-Methods: Indicates the allowed HTTP methods for cross-origin requests*
    - *Access-Control-Allow-Headers: Indicates the allowed request headers for cross-origin requests*
    - *Access-Control-Allow-Credentials : Indicates whether or not the response to the request can be exposed when the credentials flag is true.*
    - *Access-Control-Max-Age: Defines the expiration time of the result of the cached preflight request*

- 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
- 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:42
00"));
       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

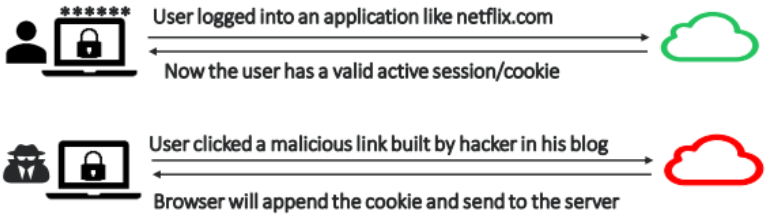
CROSS-SITE REQUEST FORGERY (CSRF)
HOW TO HANDLE IT USING SPRING SECURITY

1. A typical Cross-Site Request Forgery (CSRF or XSRF) attack aims to perform an operation in a web application on behalf of a user without their 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.

2. Consider a website netflix.com and the attacker's website travelblog.com. Also assume that the victim is logged in and his session is being maintained by cookies. The attacker will:

   - Find out what action he needs to perform on behalf of the victim and find out its endpoint (for example, to change password on netflix.com a POST request is made to the website that contains new password as the parameter)
   - Place HTML code on his website travelblog.com that will imitate a legal request to netflix.com (for example, a form with method as post and a hidden input field that contains the new password).
   - Make sure that the form is submitted by either using "autosubmit" or luring the victim to click on a submit button.

CROSS-SITE REQUEST FORGERY (CSRF)
HOW TO HANDLE IT USING SPRING SECURITY



- When the victim visits travelblog.com and that form is submitted, the victim's browser makes a request to netflix.com for a password change. Also the browser appends the cookies with the request. The server treats it as a genuine request and resets the victim's password to the attacker's supplied value. This way the victim's account gets taken over by the attacker.

- There are many proposed ways to implement CSRF protection on server side, among which the use of CSRF tokens is most popular. A CSRF token is a string that is tied to a user's session but is not submitted automatically. A website proceeds only when it receives a valid CSRF token along with the cookies, since there is no way for an attacker to know a user specific token, the attacker can not perform actions on user's behalf.

- 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

- 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();
}
```

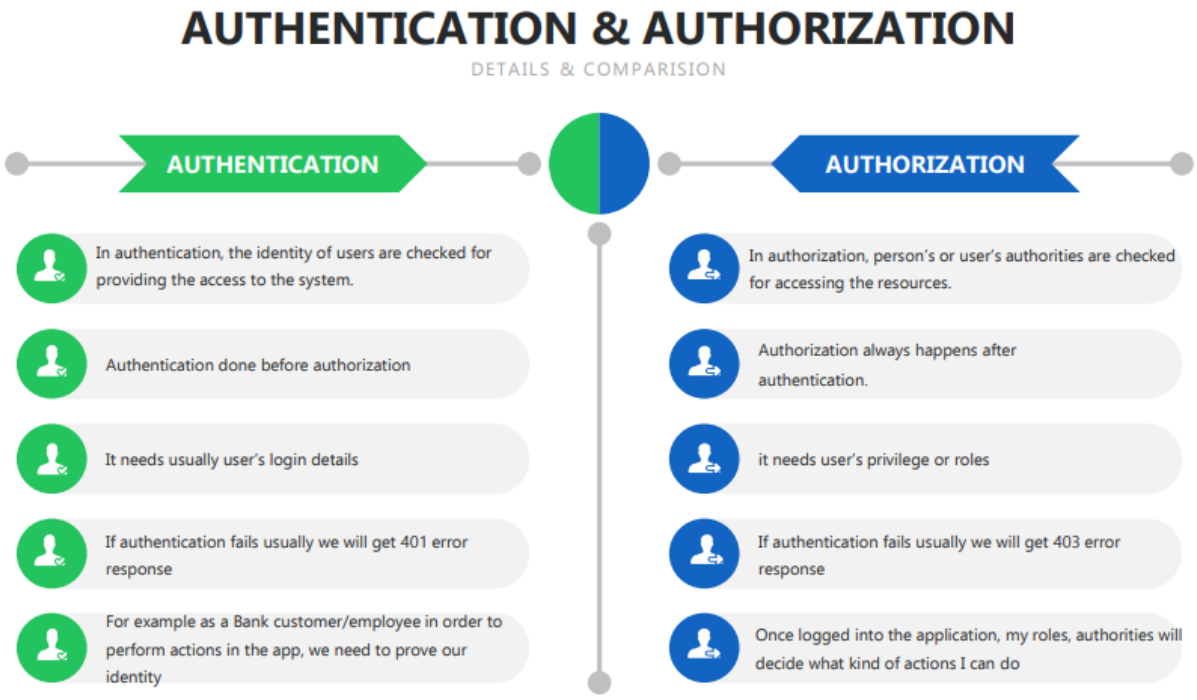
Here we are using Spring Security' built in implementation for CSRF which is `CookieCsrfTokenRepository. CookieCsrfTokenRepository` persists (sends to browser) the CSRF token in a cookie named "XSRF-TOKEN" and reads (from browser) from the header "X-XSRF-TOKEN" following the conventions of AngularJS.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

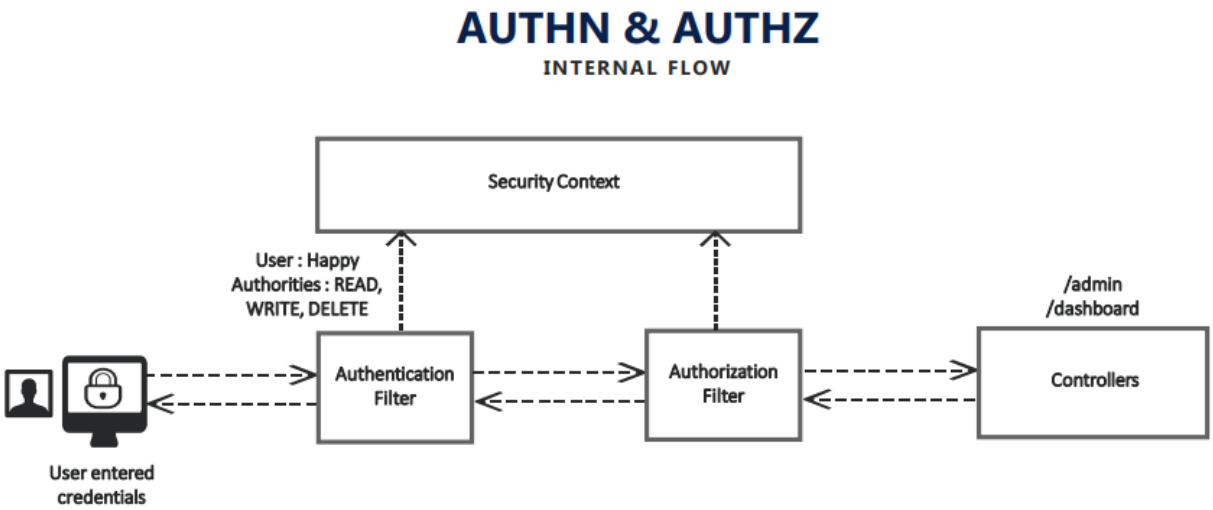
Understanding & Implementing Authorization

Authentication Vs Authorization



- Refer slide no. 31 from Course Notes

Spring Security Internal flow for Authentication & Authorization



When the Client makes a request with the credentials, the authentication filter will intercept the request and validate if the person is valid and is he/she the same person whom they are claiming. Post authentication the filter stores the UserDetails in the SecurityContext. The UserDetails will have his username, authorities etc. Now the authorization filter will intercept and decide whether the person has access to the given path based on this authorities stored in the SecurityContext. If authorized the request will be forwarded to the applicable controllers.

- Refer slide no. 32 from Course Notes
- 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

HOW AUTHORITIES STORED?

IN SPRING SECURITY

Inside UserDetails which is a contract of the User inside the Spring Security, the authorities will be stored in the form of Collection of GrantedAuthority. These authorities can be fetched using the method getAuthorities()

```
public interface UserDetails {
        Collection<? extends GrantedAuthority> getAuthorities();
}
```

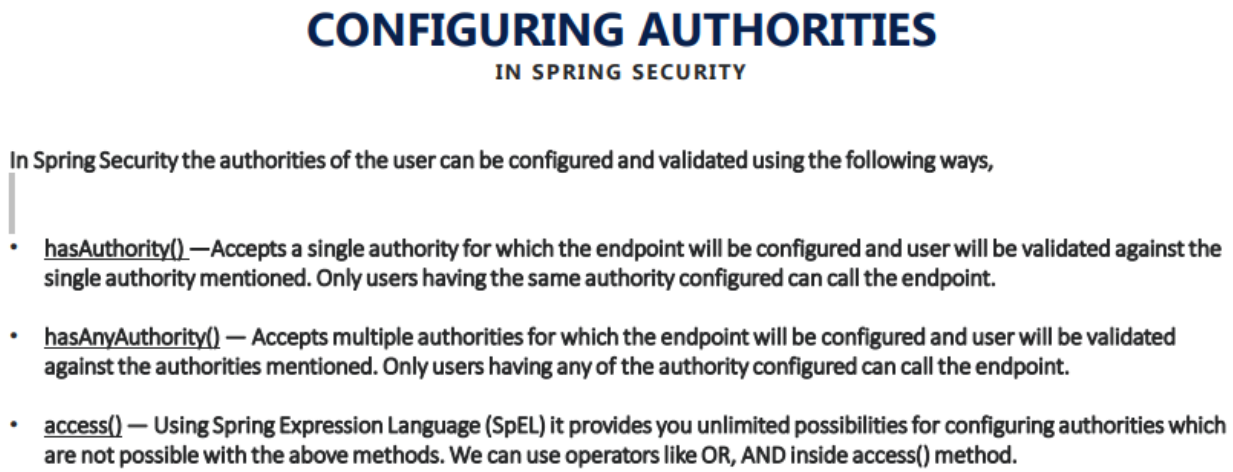
Inside GrantedAuthority interface we have a getAuthority() method which will return the authority/role name in the form of a string. Using this value the framework will try to validate the authorities of the user with the implementation of the application.

```
public interface GrantedAuthority {
        String getAuthority();
}
```

- Refer slide no. 33 from Course Notes
- 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

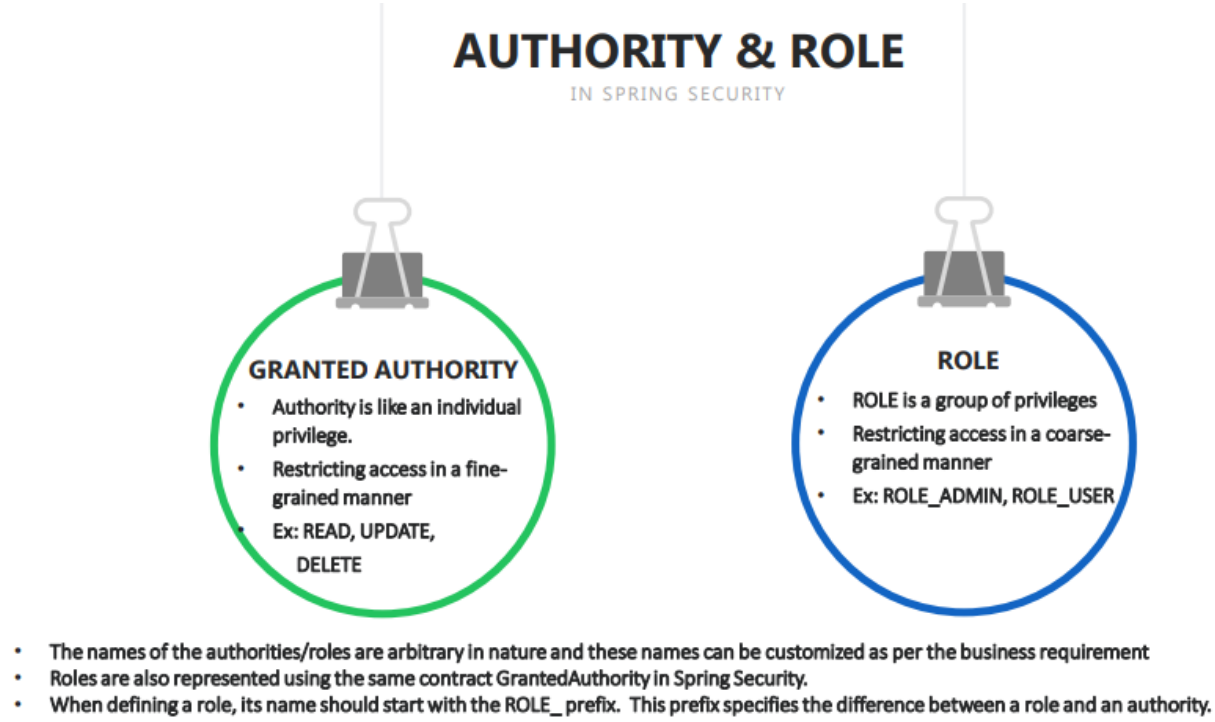
Configuring Authorities in Spring Security



- Refer slide no. 34 from Course Notes
- 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
- 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

CONFIGURING ROLES
IN SPRING SECURITY

In Spring Security the roles of the user can be configured and validated using the following ways,

- hasRole() —Accepts a single role name for which the endpoint will be configured and user will be validated against the single role mentioned. Only users having the same role configured can call the endpoint.
- hasAnyRole() — Accepts multiple roles for which the endpoint will be configured and user will be validated against the roles mentioned. Only users having any of the role configured can call the endpoint.
- access() — Using Spring Expression Language (SpEL) it provides you unlimited possibilities for configuring roles which are not possible with the above methods. We can use operators like OR, AND inside access() method.

Note :

- *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.*
- *access() method can be used not only for configuring authorization based on authority or role but also with any special requirements that we have. For example we can configure access based on the country of the user or current time/date.*

- Refer slide no. 36 from Course Notes
- 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

MATCHERS METHODS

IN SPRING SECURITY

Spring Security offers three types of matchers methods to configure endpoints security,

1) MVC matchers, 2) Ant matchers, 3) Regex matchers

- MVC matchers

MvcMatcher() uses Spring MVC's HandlerMappingIntrospector to match the path and extract variables.

- *mvcMatchers(HttpMethod method, String... patterns)— We can specify both the HTTP method and path pattern to configure restrictions*

```
http.authorizeRequests().mvcMatchers(HttpMethod.POST, "/example").authenticated()
.mvcMatchers(HttpMethod.GET, "/example").permitAll()
.anyRequest().denyAll();
```

- *mvcMatchers(String... patterns)—We can specify only path pattern to configure restrictions and all the HTTP methods will be allowed.*

```
http.authorizeRequests().mvcMatchers( "/profile/edit/**").authenticated()
.anyRequest().permitAll();
```

Note :

- *** 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/z*

MATCHERS METHODS

IN SPRING SECURITY

- ANT matchers is an implementation for Ant-style path patterns. Part of this mapping code has been kindly borrowed from Apache Ant.

- *antMatchers(HttpMethod method, String... patterns)— We can specify both the HTTP method and path pattern to configure restrictions*

```
http.authorizeRequests().antMatchers(HttpMethod.POST, "/example").authenticated()
.antMatchers(HttpMethod.GET, "/example").permitAll()
.anyRequest().denyAll();
```

- *antMatchers(String... patterns)—We can specify only path pattern to configure restrictions and all the HTTP methods will be allowed.*

```
http.authorizeRequests().antMatchers( "/profile/edit/**").authenticated()
.anyRequest().permitAll();
```

- *antMatchers(HttpMethod method)—We can specify only HTTP method ignoring path pattern to configure restrictions. This is same as antMatchers(httpMethod, "/**")*

```
http.authorizeRequests().antMatchers(HttpMethod.POST).authenticated()
.anyRequest().permitAll();
```

Note : Generally mvcMatcher is more secure than an antMatcher. As an example

- *antMatchers("/secured") matches only the exact /secured URL*
- *mvcMatchers("/secured") matches /secured as well as /secured/, /secured.html, /secured.xyz*

MATCHERS METHODS

IN SPRING SECURITY

- REGEX matchers

Regexes can be used to represent any format of a string, so they offer unlimited possibilities for this matter

- *regexMatchers(HttpMethod method, String regex)— We can specify both the HTTP method and path regex to configure restrictions*

```
http.authorizeRequests().regexMatchers(HttpMethod.GET, ".*/(en|es|zh) ").authenticated()
.anyRequest(). denyAll();
```

- *regexMatchers(String regex)—We can specify only path regex to configure restrictions and all the HTTP methods will be allowed.*

```
http.authorizeRequests().regexMatchers(".*/(en|es|zh) ").authenticated()
.anyRequest(). denyAll();
```

- Refer slide no. 37, 38, 39 from Course Notes
- 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

FILTERS IN SPRING SECURITY

IN AUTHN & AUTHZ FLOW

- Lot of times we will have situations where we need to perform some house keeping activities during the authentication and authorization flow. Few such examples are,
    - Input validation
    - Tracing, Auditing and reporting
    - Logging of input like IP Address etc.
    - Encryption and Decryption
    - Multi factor authentication using OTP
- All such requirements can be handled using HTTP Filters inside Spring Security. Filters are servlet concepts which are leveraged in Spring Security as well.
- We already saw some in built filters of Spring security framework like Authentication filter, Authorization filter, CSRF filter, CORS filter in the previous sections.
- A filter is a component which receives requests, process its logic and handover to the next filter in the chain.
- Spring Security is based on a chain of servlet filters. Each filter has a specific responsibility and depending on the configuration, filters are added or removed. We can add our custom filters as well based on the need.
    - Refer slide no. 41 from Course Notes

Inbuilt Filters provided by Spring Security

FILTERS IN SPRING SECURITY

IN AUTHN & AUTHZ FLOW

- We can always check the registered filters inside Spring Security with the below configurations,
    1. @EnableWebSecurity(debug = true) – We need to enable the debugging of the security details
    2. Enable logging of the details by adding the below property in application.properties
       logging.level.org.springframework.security.web.FilterChainProxy=DEBUG
- Below are the some of the internal filters of Spring Security that gets executed in the authentication flow,

```
Security filter chain: [
  WebAsyncManagerIntegrationFilter
  SecurityContextPersistenceFilter
  HeaderWriterFilter
  CorsFilter
  CsrfFilter
  LogoutFilter
  BasicAuthenticationFilter
  RequestCacheAwareFilter
  SecurityContextHolderAwareRequestFilter
  AnonymousAuthenticationFilter
  SessionManagementFilter
  ExceptionTranslationFilter
  FilterSecurityInterceptor
]
```

- Refer slide no. 42 from Course Notes
- 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

IMPLEMENTING FILTERS

IN SPRING SECURITY

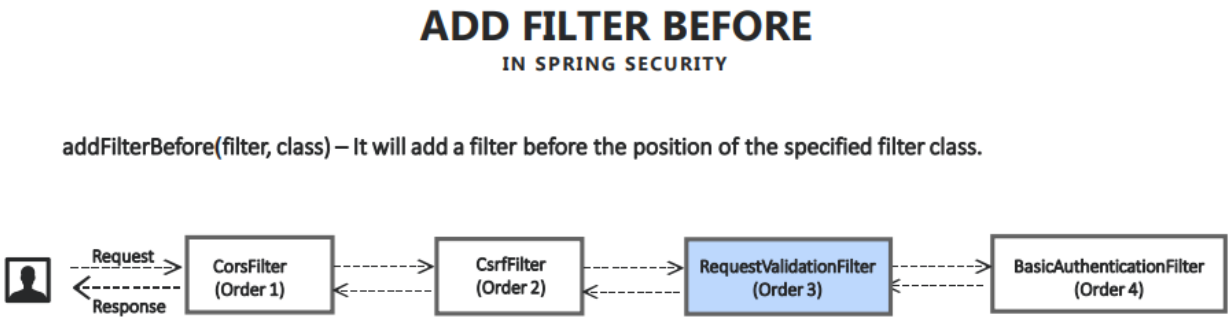
- We can create our own filters by implementing the Filter interface from the javax.servlet package. Post that we need to override the doFilter() method to have our own custom logic. This method receives as parameters the ServletRequest, ServletResponse and FilterChain.
  - *ServletRequest—It represents the HTTP request. We use the ServletRequest object to retrieve details about the request from the client.*
  - *ServletResponse—It represents the HTTP response. We use the ServletResponse object to modify the response before sending it back to the client or further along the filter chain.*
  - *FilterChain—The filter chain represents a collection of filters with a defined order in which they act. We use the FilterChain object to forward the request to the next filter in the chain.*
- You can add a new filter to the spring security chain either before, after, or at the position of a known one. Each position of the filter is an index (a number), and you might find it also referred to as "the order."
- Below are the methods available to configure a custom filter in the spring security flow,
  - *addFilterBefore(filter, class) – adds a filter before the position of the specified filter class*
  - *addFilterAfter(filter, class) – adds a filter after the position of the specified filter class*
  - *addFilterAt(filter, class) – adds a filter at the location of the specified filter class*

- Refer slide no. 43 from Course Notes

Adding Custom Filter in the filter chain

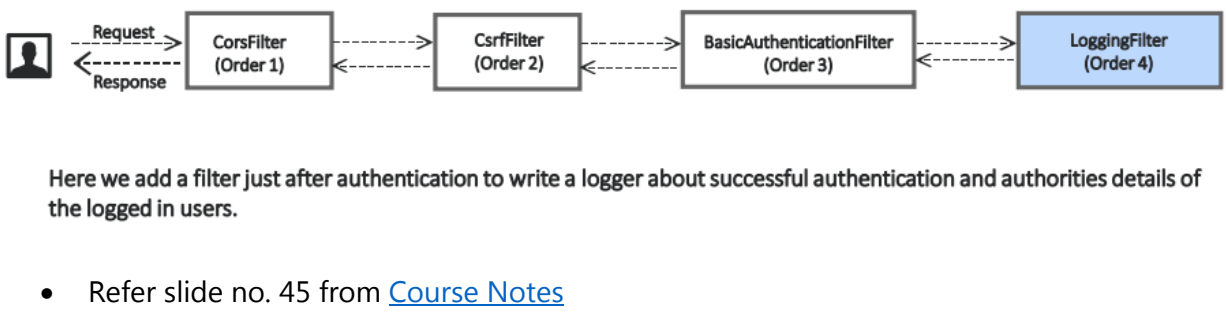
- Imp Note – You must add your custom filters in the Spring Security filter chain at the appropriate place. E.g. If your custom filter depends on say successful authentication, then it must be added after the `BasicAuthenticationFilter`.

addFilterBefore()



- Refer slide no. 44 from Course Notes

addFilterAfter()

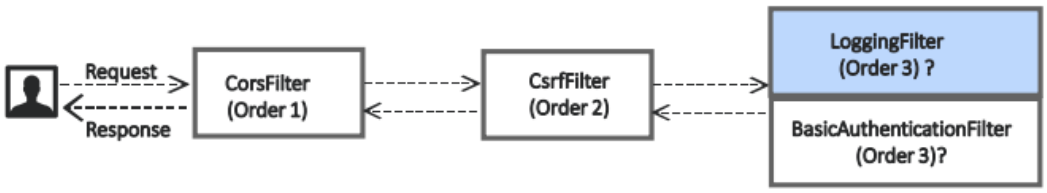


- Refer slide no. 45 from Course Notes

addFilterAt()

ADD FILTER AT
IN SPRING SECURITY

- addFilterAt(filter, class) – Adds a filter at the location of the specified filter class. But the order of the execution can't be guaranteed. This will not replace the filters already present at the same order. Since we will not have control on the order of the filters and it is random in nature we should avoid providing the filters at same order.

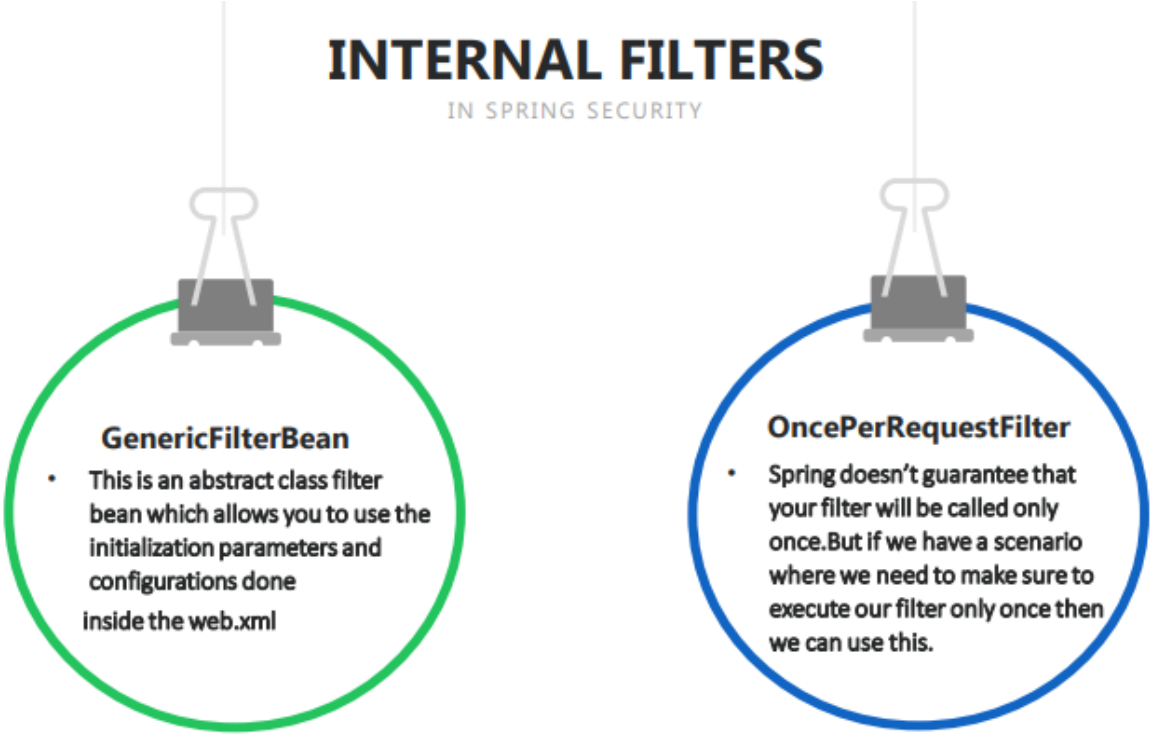


- Refer slide no. 46 from Course Notes
- `addFilterAt()` doesn't mean it will replace the spring security filter. And at the same time, we can't guarantee the order of those filters which are positioned at the same place and it is up to the spring security to decide randomly to execute which filter before 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(CookieCsrfTokenRe
pository.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
- 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.

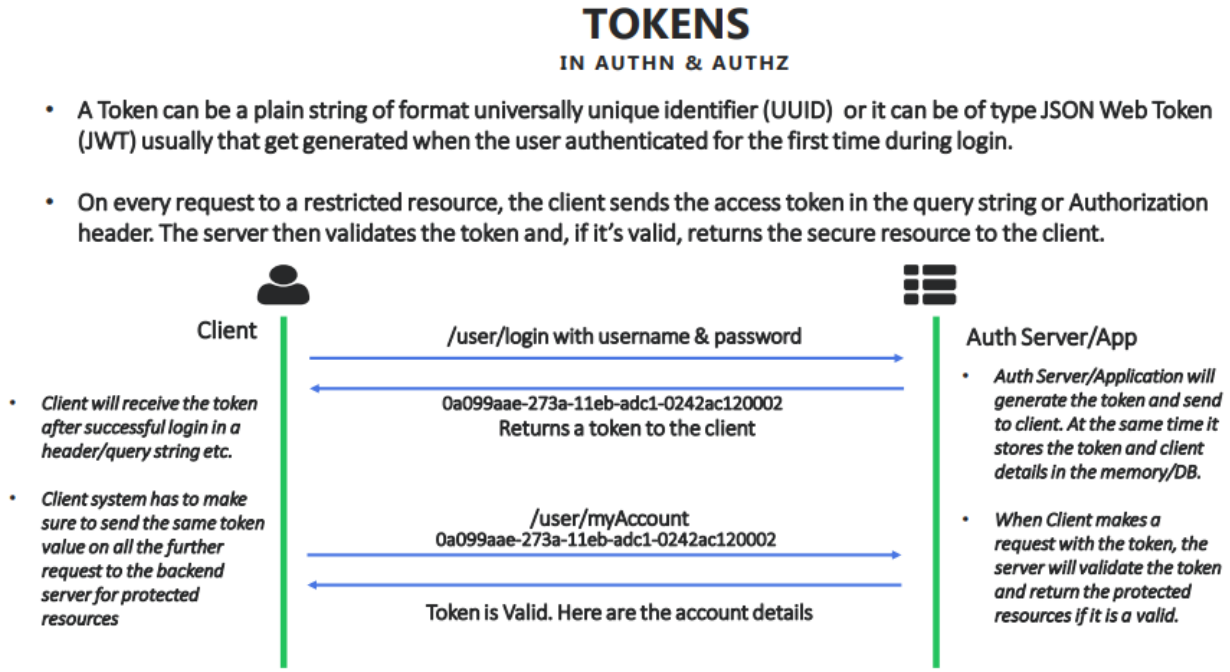
- But if we have a scenario where you want your own custom logic to be executed only once per request, then you can always extend `OncePerRequestFilter` class, which will give you the logic of making sure that your custom filter will be executable only and once for every request coming from the client.
- `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

Advantages of Token based Authentication

TOKENS

IN AUTHN & AUTHZ

Advantages of Token based Authentication

- Security - Since we are not using cookies, we don't have to protect against cross-site request forgery (CSRF) attacks.
- Stateless, easier to scale. The token contains all the information to identify the user, eliminating the need for the session state. If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on.
- We already used tokens in the previous sections in the form of CSRF and JSESSIONID tokens.
    1. CSRF Token protected our application from CSRF attacks.
    2. JSESSIONID is the default token generated by the Spring Security which helped us not to share the credentials to the backend every time.

- Refer slide no. 50, 51 from Course Notes

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

JWT

TOKEN DETAILS

- JWT means JSON Web Token. It is a token implementation which will be in the JSON format and designed to use for the web requests.
- JWT is the most common and favorite token type that many systems use these days due to its special features and advantages.
- JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the server side.
- A JWT token has 3 parts each separated by a dot(.). Below is a sample JWT token,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

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

JWT

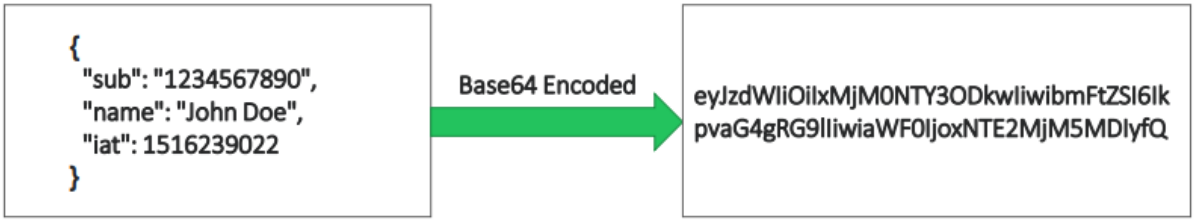
TOKEN DETAILS

- JWTs have three parts: a header, a payload, and a signature.
- In the header, we store metadata/info related to the token. If I chose to sign the token, the header contains the name of the algorithm that generates the signature.

JWT

TOKEN DETAILS

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



JWT

TOKEN DETAILS

- The last part of the token is the digital signature. This part can be optional if the party that you share the JWT token is internal and that someone who you can trust but not open in the web.
- But if you are sharing this token to the client applications which will be used by all the users in the open web then we need to make sure that no one changed the header and body values like Authorities, username etc.
- To make sure that no one tampered the data on the network, we can send the signature of the content when initially the token is generated. To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.
- For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

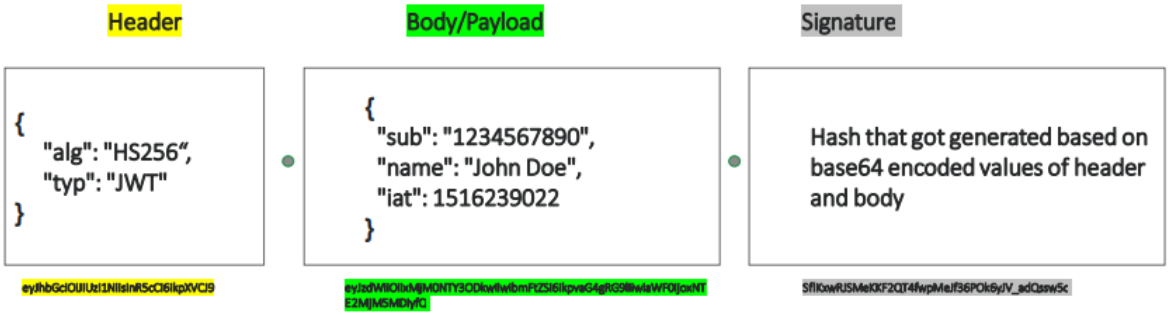
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

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

JWT

TOKEN DETAILS

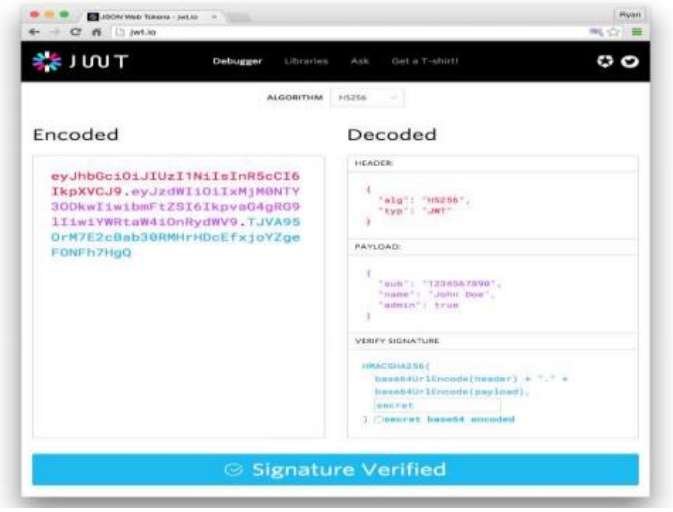
- Putting all together the JWT token is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.



JWT

TOKEN DETAILS

- If you want to play with JWT and put these concepts into practice, you can use jwt.io Debugger to decode, verify, and generate JWTs.



- Refer slide no. 52 to 57 from Course Notes
- 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.ST
ATELESS).and().
 cors().configurationSource(new CorsConfigurationSource() {
  @Override
  public CorsConfiguration getCorsConfiguration(HttpServletRequest
request) {
   CorsConfiguration config = new CorsConfiguration();

config.setAllowedOrigins(Collections.singletonList("http://localhost:42
00"));
   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

METHOD LEVEL SECURITY
IN SPRING SECURITY

- As of now we have applied authorization rules on the API paths/URLs using spring security but method level security allows to apply the authorization rules at any layer of an application like in service layer or repository layer etc. Method level security can be enabled using the annotation @EnableGlobalMethodSecurity on the configuration class.
- Method level security will also helps authorization rules even in the non-web applications where we will not have any endpoints.
- Method level security provides the below approaches to apply the authorization rules and executing your business logic,
  - ✓ Invocation authorization – Validates if someone can invoke a method or not based on their roles/authorities.
  - ✓ Filtering authorization – Validates what a method can receive through its parameters and what the invoker can receive back from the method post business logic execution.

METHOD LEVEL SECURITY
IN SPRING SECURITY

- Spring security will use the aspects from the AOP module and have the interceptors in between the method invocation to apply the authorization rules configured.
- Method level security offers below 3 different styles for configuring the authorization rules on top of the methods,
  - ✓ The prePostEnabled property enables Spring Security @PreAuthorize & @PostAuthorize annotations
  - ✓ The securedEnabled property enables @Secured annotation
  - ✓ The jsr250Enabled property enables @RoleAllowed annotation

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true,  securedEnabled = true,  jsr250Enabled = true)
public class MethodSecurityConfig {
        ...
}
```

- @Secured and @RoleAllowed are less powerful compared to @PreAuthorize and @PostAuthorize

- Refer slide no. 59, 60 from Course Notes
- 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

METHOD LEVEL SECURITY

USING INVOCATION AUTHORIZATION IN SPRING SECURITY

- Using invocation authorization we can decide if a user is authorized to invoke a method before the method executes (preauthorization) or after the method execution is completed (postauthorization).
- For filtering the parameters before calling the method we can use Prefiltering,

```
@Service
public class LoanService {

        @PreAuthorize("hasAuthority('admin')")
        @PreAuthorize("hasRole('admin')")
        @PreAuthorize("hasAnyRole('admin')")
        @PreAuthorize("# username == authentication.principal.username")
        @PreAuthorize("hasPermission(returnObject, 'admin')")
        public Loan getLoanDetails(String username) {
                return loanRepository.loadLoanByUserName(username);
        }
}
```

METHOD LEVEL SECURITY

USING INVOCATION AUTHORIZATION IN SPRING SECURITY

- For applying postauthorization rules below is the sample configuration,

```
@Service
public class LoanService {

        @PostAuthorize ("returnObject.username == authentication.principal.username")
        @PostAuthorize("hasPermission(returnObject, 'admin')")
        public Loan getLoanDetails(String username) {
          return loanRepository.loadLoanByUserName(username);
        }
}
```

- When implementing complex authorization logic, we can separate the logic using a separate class that implements PermissionEvaluator and overwrite the method hasPermission() inside it which can be leveraged inside the hasPermission configurations.

  - Refer slide no. 61, 62 from Course Notes

- 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

METHOD LEVEL SECURITY

USING FILTERING AUTHORIZATION IN SPRING SECURITY

- If we have a scenario where we don't want to control the invocation of the method but we want to make sure that the parameters sent and received to/from the method need to follow authorization rules, then we can consider filtering.

- For filtering the parameters before calling the method we can use Prefiltering,

```
@Service
public class LoanService {

        @PreFilter("filterObject.username == authentication.principal.username")
        public Loan updateLoanDetails(Loan loan) {
                // business logic
                return loan;
        }
}
```

METHOD LEVEL SECURITY

USING FILTERING AUTHORIZATION IN SPRING SECURITY

- For filtering the parameters after executing the method we can use Postfiltering,

```
@Service
public class LoanService {

        @PostFilter("filterObject.username == authentication.principal.username")
        public Loan getLoanDetails() {
                // business logic
                return loans;
        }
}
```

- We can use the @PostFilter on the Spring Data repository methods as well to filter any unwanted data coming from the database.

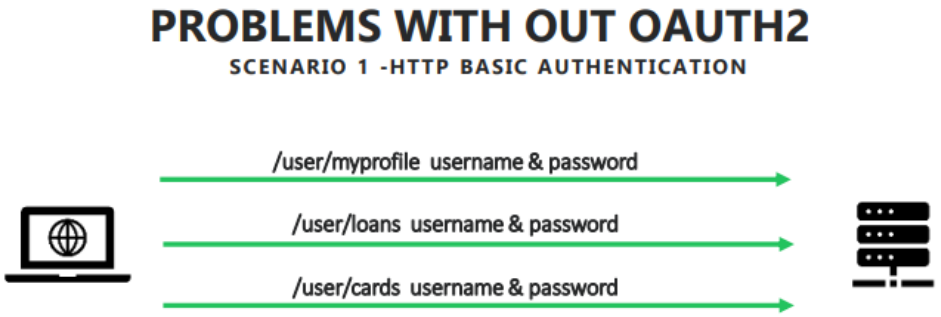
- Refer slide no. 63, 64 from Course Notes
- This has 2 approaches – @PreFilter and @PostFilter

Notes

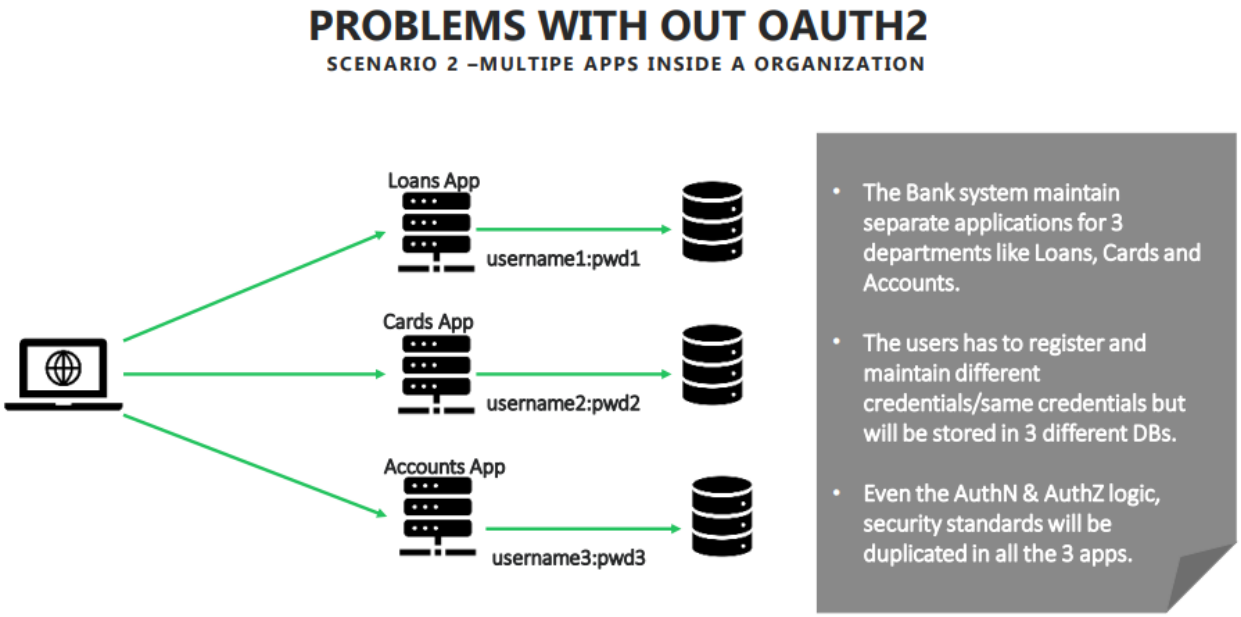
- Complete example – https://github.com/sameerbhilare/Spring-Security/tree/main/Workspace/08-Method-Level-Security

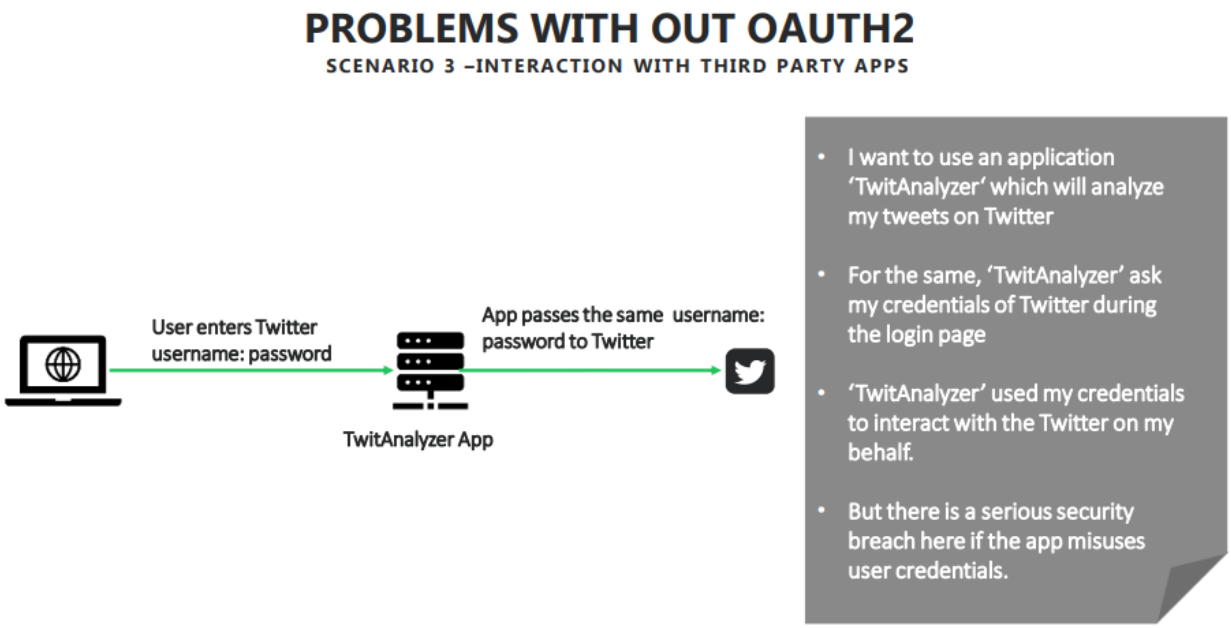
Deep dive of OAUTH2

Problems that OAUTH2 framework trying to solve



With HTTP Basic authentication, the client need to send the user credentials every time and authentication logic has to be executed every time with all the requests. With this approach we ended up sharing the credentials often over the network.





- Refer slide no. 66, 67, 68 from Course Notes
- 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

OAUTH2
INTRODUCTION

- OAuth stands for Open Authorization. It's a free and open protocol, built on IETF standards and licenses from the Open Web Foundation.
- OAuth 2.0 is a delegation protocol, which means letting someone who controls a resource allow a software application to access that resource on their behalf without impersonating them.
    - *For example in our EazyBank application instead of maintain both Auth and Business logic inside a same application/server, it will allow other application to handle authorization before allowing the client to access protected resources. This will happen mostly with the help of tokens.*
    - *The other example is, I have an application 'TwitAnalyzer' where it can analyze the tweets that somebody made on the Twitter. But in order to work we should allow this application to pull the tweets from the Twitter. So here Twitter exposes a Authorization server to the 'TwitAnalyzer'. So this application now will have a login page where it will redirect the user to the Twitter login and his credentials will be validated by Twitter. Post that Twitter provides a token to the 'TwitAnalyzer' which will be used to pull the tweets of the user.*

OAUTH2
INTRODUCTION

- According to the OAuth 2.0 specification it enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.
- In the OAuth world, a client application wants to gain access to a protected resource on behalf of a resource owner (usually an end user). For this client application will interact with the Authorization server to obtain the token.
- In many ways, you can think of the OAuth token as a "access card" at any office/hotel. These tokens provides limited access to someone, without handing over full control in the form of the master key.

- Refer slide no. 69, 70 from Course Notes
- 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

OAUTH2

INTRODUCTION

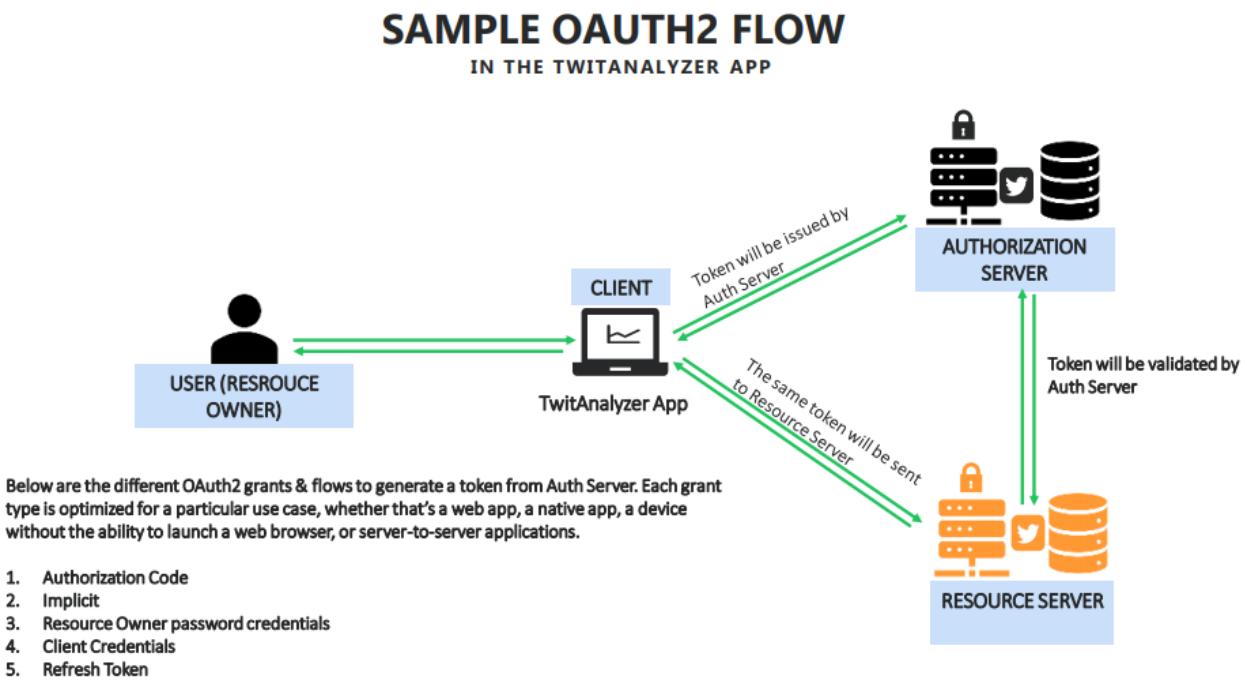
- OAuth 2.0 has the following components,

✓ The Resource Server – where the protected resources owned by user is present like photos, personal information, transactions etc.

✓ The user (also known as resource owner) – The person who owns resources exposed by the resource server. Usually the user will prove his identity with the help of username and password.

✓ The client – The application that want to access the resources owned by the user on their behalf. The client uses a client id and secret to identify itself. But these are not same as user credentials.

✓ The authorization server - The server that authorizes the client to access the user resources in the resource server. When the authorization server identifies that a client is authorized to access a resource on behalf of the user, it issues a token. The client application uses this token to prove to the resource server that it was authorized by the authorization server. The resource server allows the client to access the resource it requested if it has a valid token after validating the same with Auth server.

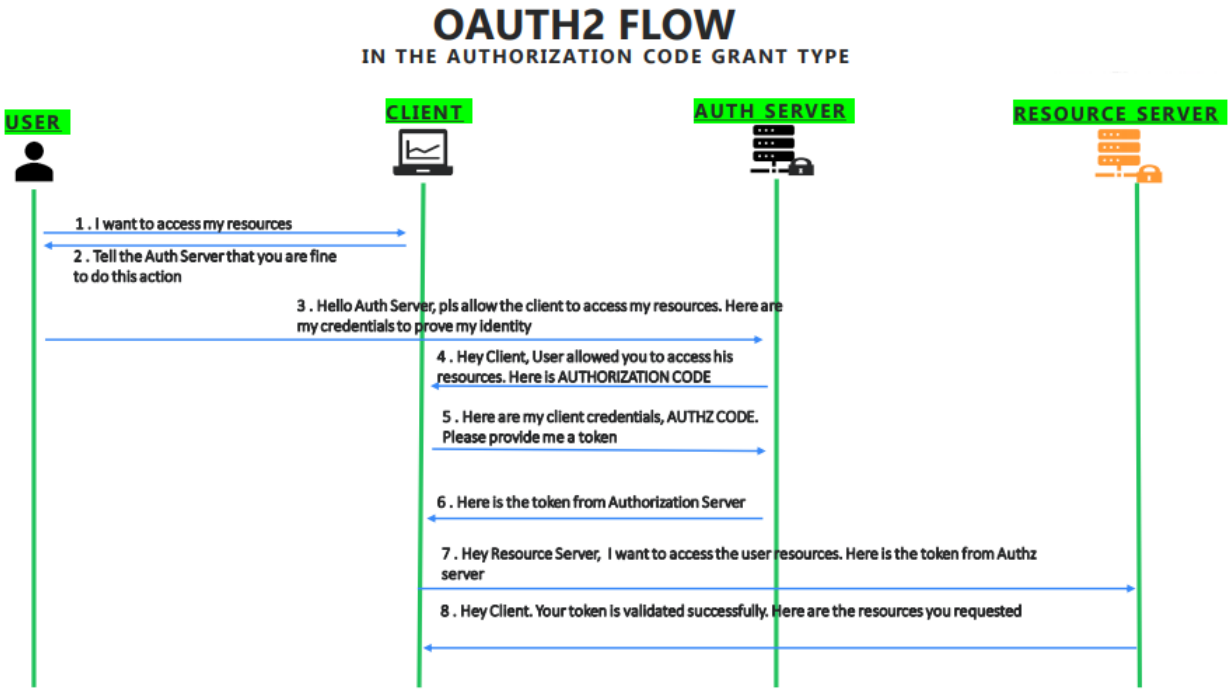


- Refer slide no. 71, 72 from Course Notes
- Whenever some application (stack overflow) wants to use the Google Auth and Resource servers and it also wants to get the basic details, first of all they have to register themselves with Google.
- So first Stack Overflow has to reach out to the Google developer 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



OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE

- In the steps 2 & 3, where client is making a request to Auth Server endpoint have to send the below important details,
  - ✓ client_id – the id which identifies the client application by the Auth Server. This will be granted when the client register first time with the Auth server.
  - ✓ redirect_uri – the URI value which the Auth server needs to redirect post successful authentication. If a default value is provided during the registration then this value is optional
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ state – CSRF token value to protect from CSRF attacks
  - ✓ response_type – With the value 'code' which indicates that we want to follow authorization code grant

- In the step 5 where client after received a authorization code from Auth server, it will again make a request to Auth server for a token with the below values,
  - ✓ code – the authorization code received from the above steps
  - ✓ client_id & client_secret – the client credentials which are registered with the auth server. Please note that these are user credentials
  - ✓ grant_type – With the value 'authorization_code' which identifies the kind of grant type is used
  - ✓ redirect_uri

OAUTH2 FLOW

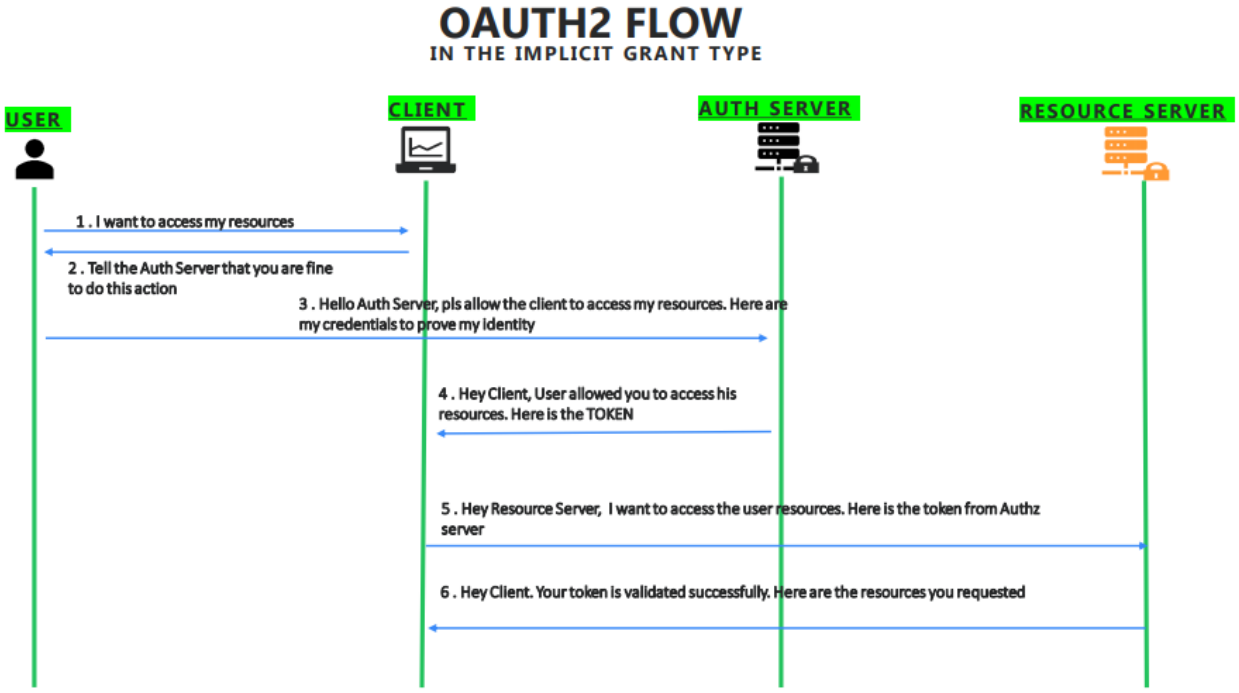
IN THE AUTHORIZATION CODE GRANT TYPE

- We may wonder that why in the Authorization Code grant type client is making request 2 times to Auth server for authorization code and access token.
  - ✓ In the first step, authorization server will make sure that user directly interacted with it along with the credentials. If the details are correct, auth server send the authorization code to client
  - ✓ Once it receives the authorization code, in this step client has to prove it's identity along with the authorization code, client credentials to get the access token.

- Well you may ask why can't Auth server directly club both the steps together and provide the token in a single step. The answer we used to have that grant type as well which is called as 'implicit grant type'. But this grant type is not recommended to use due to it's less secure.

- Refer slide no. 73, 74, 75 from Course Notes
- 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



OAUTH2 FLOW

IN THE IMPLICIT GRANT TYPE

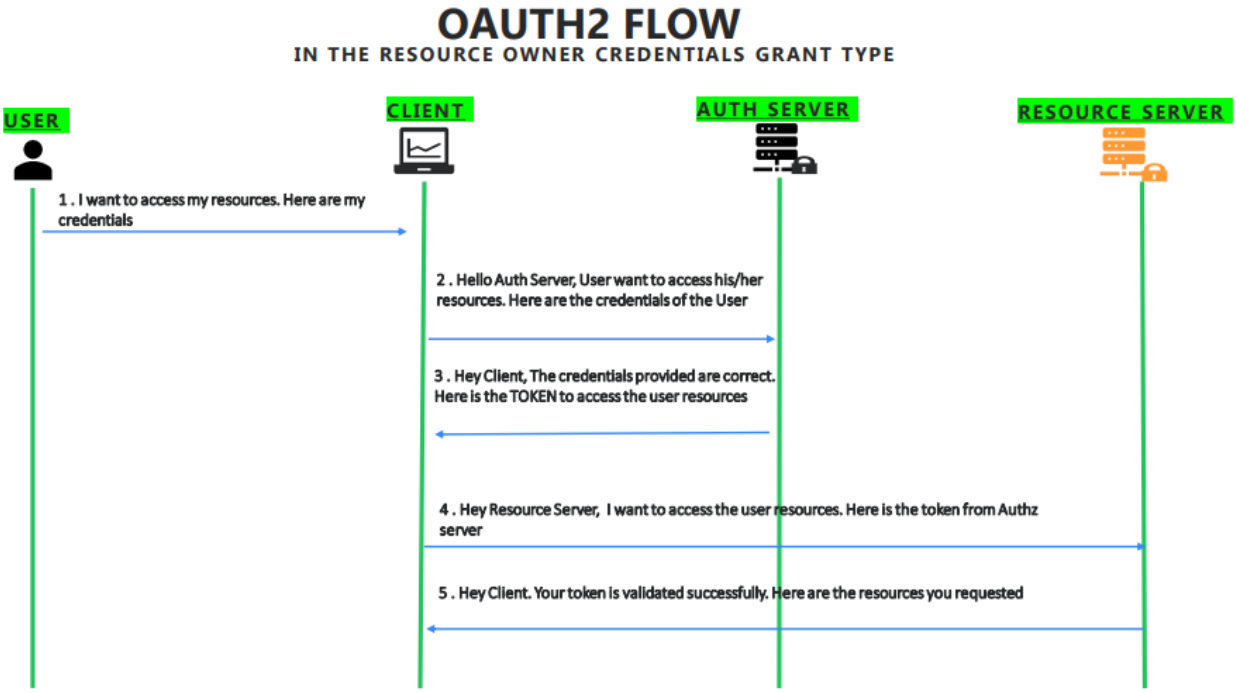
- In the step 3, where client is making a request to Auth Server endpoint have to send the below important details,
  - ✓ client_id – the id which identifies the client application by the Auth Server. This will be granted when the client register first time with the Auth server.
  - ✓ redirect_uri – the URI value which the Auth server needs to redirect post successful authentication. If a default value is provided during the registration then this value is optional
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ state – CSRF token value to protect from CSRF attacks
  - ✓ response_type – With the value 'token' which indicates that we want to follow implicit grant type

- If the user approves the request, the authorization server will redirect the browser back to the redirect_uri specified by the application, adding a token and state to the fragment part of the URL. For example, the user will be redirected back to a URL such as

https://example-app.com/redirect
#access_token=xMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiY&token_type=Bearer&expires_in=600
&state=80bvln4pUfdSxr6UTjtay8Yzg9ZiAkCzKNwy

- Refer slide no. 76, 77 from Course Notes
- 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



OAUTH2 FLOW

IN THE RESOURCE OWNER CREDENTIALS GRANT TYPE

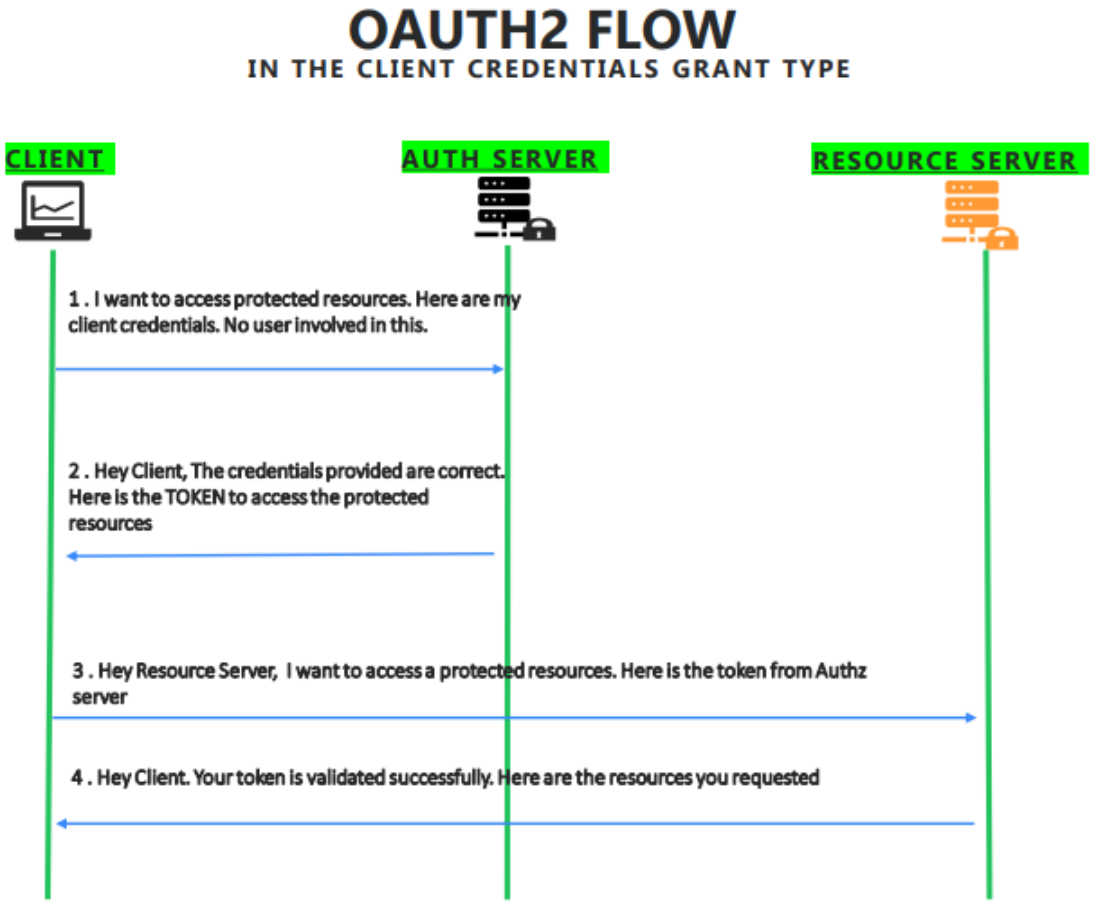
- In the step 2, where client is making a request to Auth Server endpoint have to send the below important details,
  - ✓ client_id & client_secret – the credentials of the client to authenticate itself.
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ username & password – Credentials provided by the user in the login flow
  - ✓ grant_type – With the value 'password' which indicates that we want to follow password grant type
- We use this authentication flow only if the client, authorization server and resource servers are maintained by the same organization.
- This flow will be usually followed by the enterprise applications who want to separate the Auth flow and business flow. Once the Auth flow is separated different applications in the same organization can leverage it.
- We can't use the Authorization code grant type since it won't look nice for the user to redirect multiple pages inside your organization for authentication.

- Refer slide no. 78, 79 from Course Notes
- 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

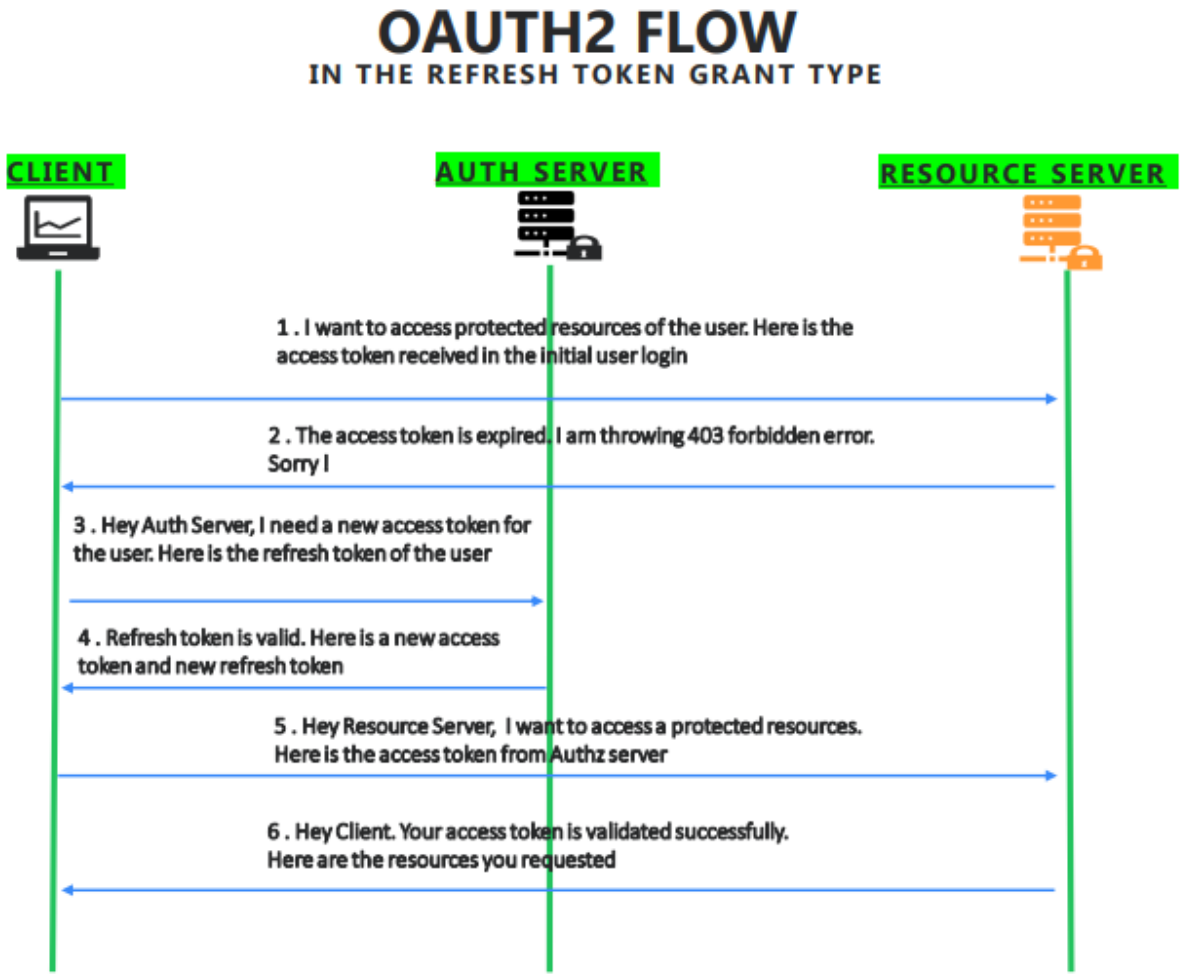


OAUTH2 FLOW
IN THE CLIENT CREDENTIALS GRANT TYPE

- In the step 1, where client is making a request to Auth Server endpoint have to send the below important details,
  - ✓ client_id & client_secret – the credentials of the client to authenticate itself.
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ grant_type – With the value 'client_credentials' which indicates that we want to follow client credentials grant type
- This is the most simplest grant type flow in OAUTH2.
- We use this authentication flow only if there is no user and UI involved. Like in the scenarios where 2 different applications want to share data between them using backend APIs.

- Refer slide no. 80, 81 from Course Notes
- 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



OAUTH2 FLOW

IN THE REFRESH TOKEN GRANT TYPE

- In the step 3, where client is making a request to Auth Server endpoint have to send the below important details,
  - ✓ client_id & client_secret – the credentials of the client to authenticate itself.
  - ✓ refresh_token – the value of the refresh token received initially
  - ✓ scope – similar to authorities. Specifies level of access that client is requesting like READ
  - ✓ grant_type – With the value 'refresh_token' which indicates that we want to follow refresh token grant type
- This flow will be used in the scenarios where the access token of the user is expired. Instead of asking the user to login again and again, we can use the refresh token which originally provided by the Authz server to reauthenticate the user.
- Though we can make our access tokens to never expire but it is not recommended considering scenarios where the tokens can be stole if we always use the same token
- Even in the resource owner credentials grant types we should not store the user credentials for reauthentication purpose instead we should reply on the refresh tokens.

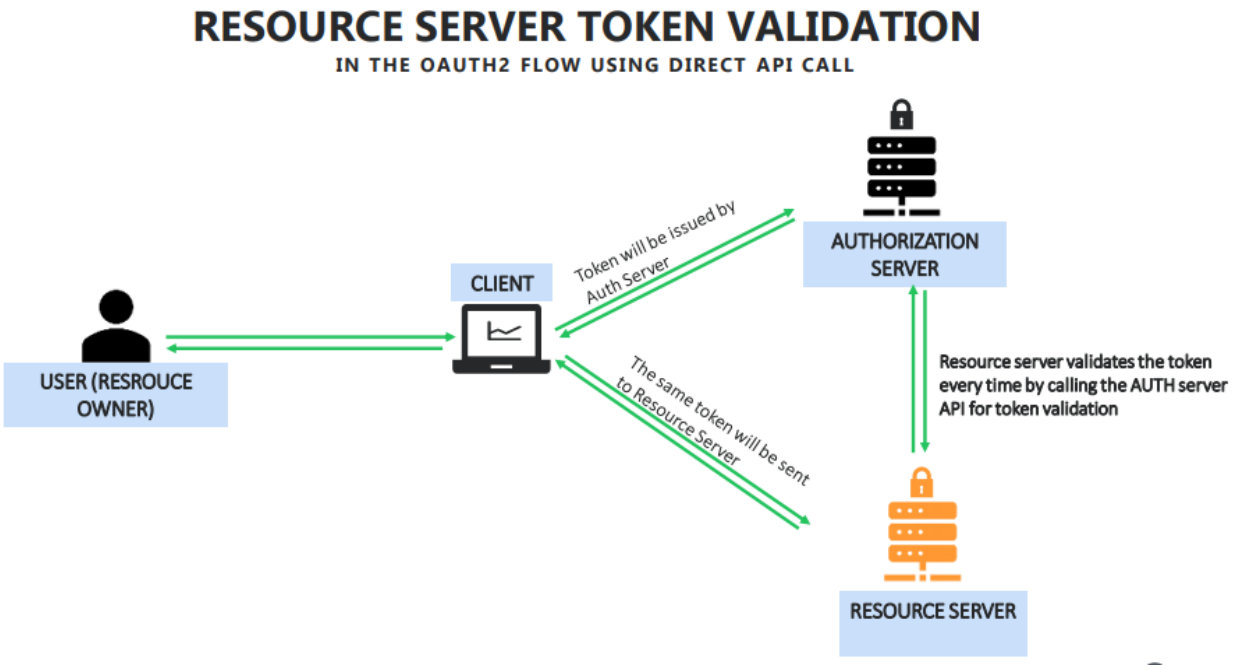
- Refer slide no. 82, 83 from Course Notes
- Whenever we interact with auth server initially during the authrorize code grant type or implicit grant, along with the access token, you also get the refresh token from the auth server. You save that refresh token and access token in the database.
- And whenever 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.

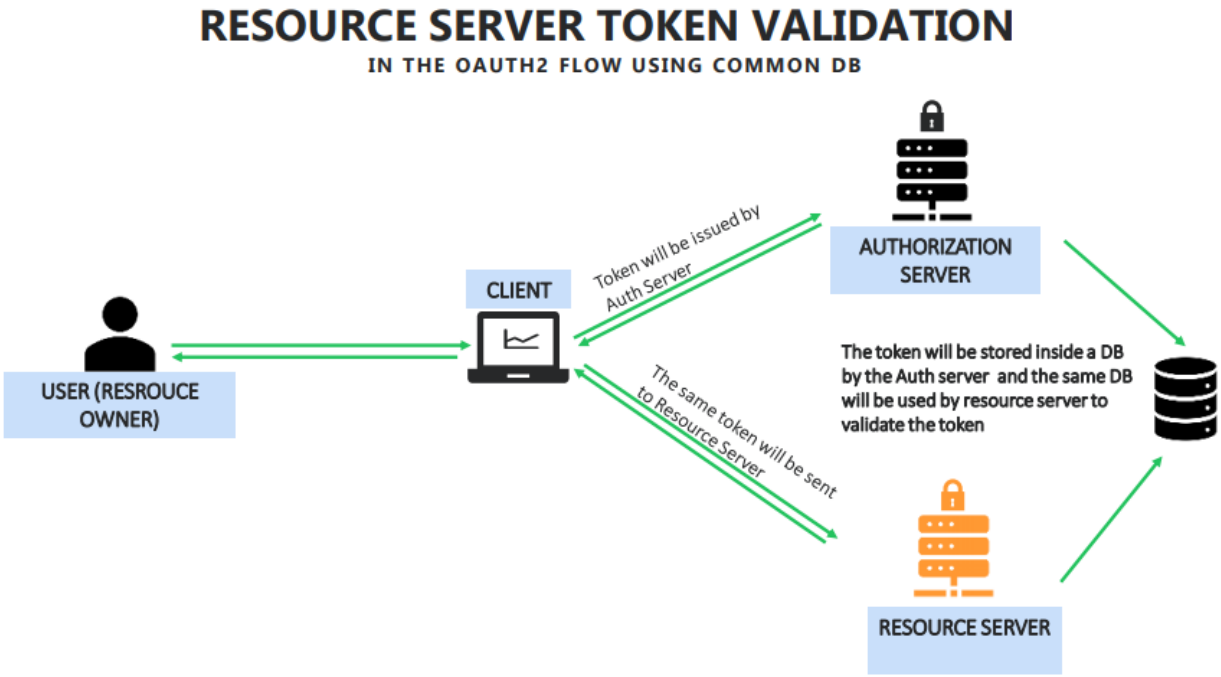
Ways of Validating Access Tokens

By making direct API calls



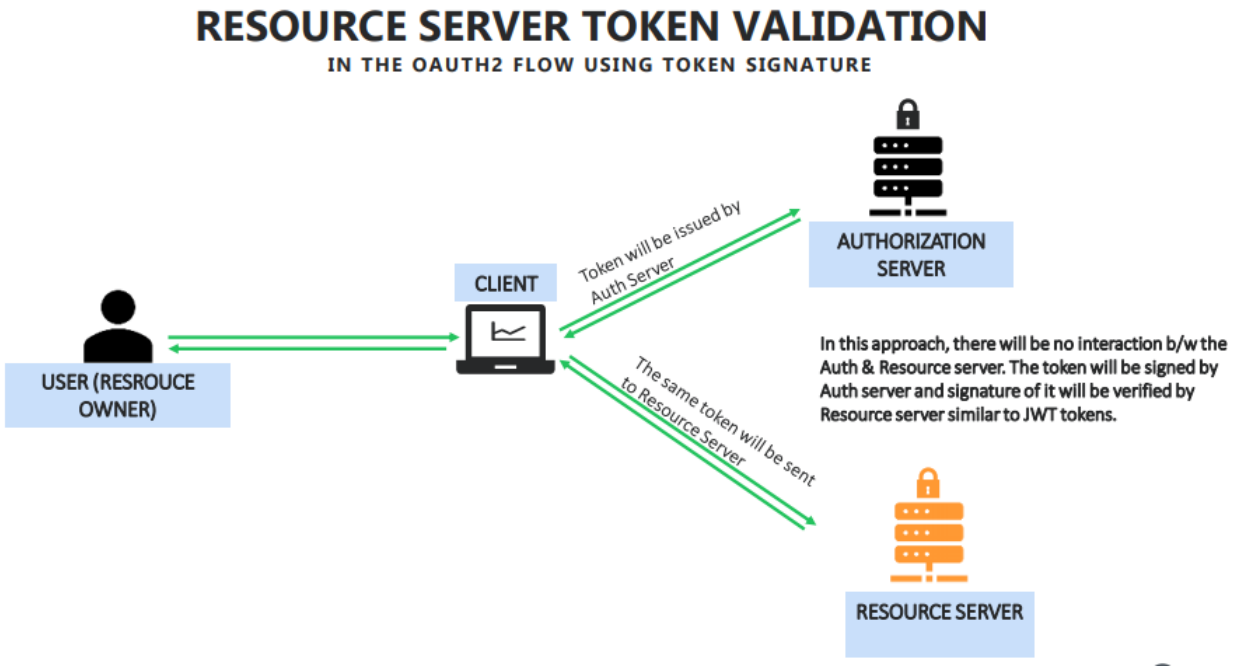
- Refer slide no. 84 from Course Notes
- The very first and basic way of validating the access token by a resource server with the authentication server is by making a REST API call in the backend, so whenever a resource server receives the access token from the client, it can simply make a REST API 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
- 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
- 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/{registratio
nId}")
     .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/