

# Spring Security

# Spring Security

- ▶ Spring Security is a framework which provides various security features like:
- ▶ **authentication**, **authorization** to create secure Java Enterprise Applications.
- ▶ It is a sub-project of Spring framework which was started in 2003 by Ben Alex.
- ▶ It overcomes all the problems that come during creating non spring security applications
- ▶ **Authorization** is the process to allow authority to perform actions in the application.
- ▶ We can apply authorization to authorize web request, methods and access to individual domain.

# Advantages

- ▶ Spring Security has numerous advantages. Some of that are given below.
- ▶ Comprehensive support for authentication and authorization.
- ▶ Protection against common tasks
- ▶ Servlet API integration
- ▶ Integration with Spring MVC
- ▶ Portability
- ▶ CSRF protection
- ▶ Java Configuration support

# Spring Security Features

- ▶ LDAP (Lightweight Directory Access Protocol)
  - ▶ It is an open application protocol for maintaining and accessing distributed directory information services over an Internet Protocol.
- ▶ Single sign-on
  - ▶ This feature allows a user to access multiple applications with the help of single account(user name and password).
- ▶ JAAS (Java Authentication and Authorization Service) LoginModule
  - ▶ It is a Pluggable Authentication Module implemented in Java. Spring Security supports it for its authentication process.
- ▶ Basic Access Authentication
  - ▶ Spring Security supports Basic Access Authentication that is used to provide user name and password while making request over the network.
- ▶ Digest Access Authentication
  - ▶ This feature allows us to make authentication process more secure than Basic Access Authentication. It asks to the browser to confirm the identity of the user before sending sensitive data over the network.

# Spring Security Features

- ▶ Remember-me
  - ▶ Spring Security supports this feature with the help of HTTP Cookies. It remember to the user and avoid login again from the same machine until the user logout.
- ▶ Web Form Authentication
  - ▶ In this process, web form collect and authenticate user credentials from the web browser. Spring Security supports it while we want to implement web form authentication.
- ▶ Authorization
  - ▶ Spring Security provides the this feature to authorize the user before accessing resources. It allows developers to define access policies against the resources.
- ▶ Software Localization
  - ▶ This feature allows us to make application user interface in any language.
- ▶ HTTP Authorization
  - ▶ Spring provides this feature for HTTP authorization of web request URLs using Apache Ant paths or regular expressions.

# Features added in Spring Security 5.0

- ▶ OAuth 2.0 Login
  - ▶ This feature provides the facility to the user to login into the application by using their existing account at GitHub or Google. This feature is implemented by using the Authorization Code Grant that is specified in the OAuth 2.0 Authorization Framework.
- ▶ Reactive Support
  - ▶ From version Spring Security 5.0, it provides reactive programming and reactive web runtime support and even, we can integrate with Spring WebFlux
- ▶ Modernized Password Encoding
  - ▶ Spring Security 5.0 introduced new Password encoder **DelegatingPasswordEncoder** which is more modernize and solve all the problems of previous encoder **NoOpPasswordEncoder**.

# Spring Project Modules

- ▶ In Spring Security 3.0, the Security module is divided into separate jar files. The purpose was to divide jar files based on their functionalities, so, the developer can integrate according to their requirement.
- ▶ It also helps to set required dependency into pom.xml file of maven project.

# Spring Project Modules

## ► Core - spring-security-core.jar

- This is core jar file and required for every application that wants to use Spring Security. This jar file includes core access-control and core authentication classes and interfaces. We can use it in standalone applications or remote clients applications.
- It contains top level packages:
  - `org.springframework.security.core`
  - `org.springframework.security.access`
  - `org.springframework.security.authentication`
  - `org.springframework.security.provisioning`

## ► Remoting - spring-security-remoting.jar

- This jar is used to integrate security feature into the Spring remote application. We don't need it until or unless we are creating remote application. All the classes and interfaces are located into **`org.springframework.security.remoting`** package.



# Spring Project Modules

- ▶ Web - spring-security-web.jar
  - ▶ This jar is useful for Spring Security web authentication and URL-based access control. It includes filters and web-security infrastructure.
  - ▶ All the classes and interfaces are located into the **org.springframework.security.web** package.
- ▶ Config - spring-security-config.jar
  - ▶ This jar file is required for Spring Security configuration using XML and Java both. It includes Java configuration code and security namespace parsing code. All the classes and interfaces are stored in **org.springframework.security.config** package.
- ▶ LDAP - spring-security-ldap.jar
  - ▶ This jar file is required only if we want to use LDAP (Lightweight Directory Access Protocol). It includes authentication and provisioning code. All the classes and interfaces are stored into **org.springframework.security.ldap** package.
- ▶ OAuth 2.0 Core - spring-security-oauth2-core.jar
  - ▶ This jar is required to integrate Oauth 2.0 Authorization Framework and OpenID Connect Core 1.0 into the application. This jar file includes the core classes for OAuth 2.0 and classes are stored into the **org.springframework.security.oauth2.core** package.

# Spring Project Modules

- ▶ OpenID - spring-security-openid.jar
  - ▶ This jar is used for OpenID web authentication support. We can use it to authenticate users against an external OpenID server. It requires OpenID4Java and top level package is **org.springframework.security.openid**.
- ▶ Test - spring-security-test.jar
  - ▶ This jar provides support for testing Spring Security application.
- ▶

# Spring Security Filters Chain

- ▶ When you add the Spring Security framework to your application, it automatically registers a filters chain that intercepts all incoming requests. This chain consists of various filters, and each of them handles a particular use case.
- ▶ Check if the requested URL is publicly accessible, based on configuration.
- ▶ In case of session-based authentication, check if the user is already authenticated in the current session
- ▶ Check if the user is authorized to perform the requested action, and so on
- ▶ Spring Security filters are registered with the lowest order and are the first filters invoked.
- ▶ For some use cases, if you want to put your custom filter in front of them, you will need to add padding to their order. This can be done with the following configuration:
  - ▶ `spring.security.filter.order=10`

# Important

- ▶ **Authentication** refers to the process of verifying the identity of a user, based on provided credentials. A common example is entering a username and a password when you log in to a website. You can think of it as an answer to the question *Who are you?*.
- ▶ **Authorization** refers to the process of determining if a user has proper permission to perform a particular action or read particular data, assuming that the user is successfully authenticated.
- ▶ **Principle** refers to the currently authenticated user.
- ▶ **Granted authority** refers to the permission of the authenticated user.
- ▶ **Role** refers to a group of permissions of the authenticated user.

# Important

- ▶ AuthenticationManager

- ▶ You can think of AuthenticationManager as a coordinator where you can register multiple providers, and based on the request type, it will deliver an authentication request to the correct provider.

- ▶ AuthenticationProvider

AuthenticationProvider processes specific types of authentication. Its interface exposes only two functions:

- ▶ authenticate performs authentication with the request.
  - ▶ supports checks if this provider supports the indicated authentication type.

- ▶ UserDetailsService

UserDetailsService is described as a core interface that loads user-specific data in the Spring documentation.

- ▶ loadUserByUsername accepts username as a parameter and returns the user identity object.

- ▶ as

# Spring Boot - Spring Security

- ▶ When working with Spring Boot, the [spring-boot-starter-security](#) starter will automatically include all dependencies, such as *spring-security-core*, *spring-security-web*, and *spring-security-config* among others:

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
  <version>2.5.0</version>  
</dependency>
```

- ▶ Check in localhost:8080  
(spring default login form)

# Spring Security Java Configuration

- By adding **@EnableWebSecurity**, we get Spring Security and MVC integration support:

```
@Configuration
```

```
@EnableWebSecurity
```

```
public class SecSecurityConfig extends WebSecurityConfigurerAdapter {
```

```
    @Override
```

```
    protected void configure(final AuthenticationManagerBuilder auth) throws Exception {
```

```
        // authentication manager
```

```
    }
```

```
    @Override
```

```
    protected void configure(final HttpSecurity http) throws Exception {
```

```
        // http builder configurations for authorize requests and form login
```

```
    }
```

```
}
```

# WebSecurityConfigurerAdapter

- ▶ Provides a convenient base class for creating a [WebSecurityConfigurer](#) instance.
- ▶ The implementation allows customization by overriding methods.
- ▶ **AuthenticationManagerBuilder:** Allows for easily building in memory authentication, LDAP authentication, JDBC based authentication, adding [UserDetailsService](#), and adding [AuthenticationProvider](#)'s.



# Spring Security - In memory users

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .inMemoryAuthentication()
        .withUser("admin").password("{noop}adminPswd").roles("admin")
        .and()
        .withUser("user").password("{noop}userPswd").roles("user");
}
```

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests() // starting implementation
        .anyRequest().authenticated()
        .and().httpBasic();

    http.csrf().disable();
}
```

# Spring Security - In memory users

- ▶ **authorizeRequests()** Allows restricting access based upon the `HttpServletRequest` using `RequestMatcher` implementations.
- ▶ **permitAll()** This will allow the public access that is anyone can access endpoint *PUBLIC\_URL* *without authentication*.
- ▶ **anyRequest().authenticated()** will restrict the access for any other endpoint other than *PUBLIC\_URL*, and the *user must be authenticated*.
- ▶ **httpBasic()** - Calling this method on `HttpSecurity` will enable [Http Basic Authentication](#) for your application with some "reasonable" defaults.
- ▶ **.authenticated()** - checks whether is user authorized
- ▶ **.fullyAuthenticated()** - checks whether is user authorized or remember user.
- ▶ **http.csrf().disable()** - CSRF protection is **enabled by default** in the Java configuration. We can still disable it if we need to:

# Spring Security

- ▶ Please note that in the following approach each password has a prefix of `{xxxx}` which specifies what password encoder should be used to encode the provided password. Below are different values you can use as password prefix:
- ▶ Use `{bcrypt}` for *BCryptPasswordEncoder*,
- ▶ Use `{noop}` for *NoOpPasswordEncoder*,
- ▶ Use `{pbkdf2}` for *Pbkdf2PasswordEncoder*,
- ▶ Use `{scrypt}` for *SCryptPasswordEncoder*,
- ▶ Use `{sha256}` for *StandardPasswordEncoder*.

# Spring Security

- ▶ `http.antMatchers("...")` - tells Spring to only configure `HttpSecurity` if the path matches this pattern.
  - ▶ `.antMatchers("/init/**").permitAll()`
  - ▶ `.antMatchers("/user/**").hasRole("USER")`
  - ▶ `.antMatchers(HttpMethod.GET, "/restricgted/get/**", "/restricgted2/get/**").`
- ▶ `hasAnyRole(.....)`

# Spring Security

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .inMemoryAuthentication()
        .withUser("admin").password("{noop}adminPswd").roles("admin")
        .and()
        .withUser("user").password("{noop}userPswd").roles("user");
}
```

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/init/**").permitAll()
        .antMatchers("/user/**").hasRole("user")
        .antMatchers("/admin/**").hasRole("admin")
        .anyRequest()
        .authenticated()
        .and().httpBasic();

    http.csrf().disable();
}
```

# Retrieve User Information in Spring Security

- ▶ The currently authenticated user is available through a number of different mechanisms in Spring.
- ▶ 1. Get the User in a Bean
- ▶ 2. Get the User in a Controller
- ▶ All codes in `spring_security_04_get_user` project

Can be test by Brouser of PostMan

GET /user/test HTTP/1.1

Host: localhost:8080

Authorization: Basic YWRtaW46YWRtaW5Qc3dk (Basic Auth)

# Send HttpBasic Request

- ▶ In .http file

```
// Basic authentication  
GET http://example.com  
Authorization: Basic username password
```

- ▶ In java :
- ▶ [https://docs.agora.io/en/All/faq/restful\\_authentication](https://docs.agora.io/en/All/faq/restful_authentication)

# 1 - Get the User in a Bean.

►

```
Authentication authentication =  
    SecurityContextHolder.getContext().getAuthentication();  
String currentPrincipalName = authentication.getName();  
  
// authentication.getPrincipal();
```



# Get the User in a Controller



```
@RequestMapping(value = "/principle")  
public String userPrinciple(Principal principal) {  
    String userName = principal.getName();  
    return "User Principle method";  
}
```

# Get the User in a Controller

►

```
@RequestMapping(value = "/authentication")
public String currentUserName(Authentication authentication) {
    UserDetails userDetails = (UserDetails) authentication.getPrincipal();
    System.out.println("User has authorities: " + userDetails.getAuthorities());
    return authentication.getName();
}
```

# Get the User in a Controller



```
@RequestMapping(value = "/http")  
public String currentUserSimple(HttpServletRequest request) {  
    Principal userPrincipal = request.getUserPrincipal();  
    String userName = userPrincipal.getName();  
    return "User http method: " + userName;  
}
```

# Get the User in a Controller



```
@RequestMapping(value = "/detail")
public String userDetailsM(@AuthenticationPrincipal final UserDetails userDetails) {
    String userName = userDetails.getUsername();
    String userPassword = userDetails.getPassword();

    Collection roles = userDetails.getAuthorities();
    roles.forEach(System.out::println);
    String s = "userName: " + userName + "\n" + "UserPassword: " + userPassword + "\n"
+ " roles: " + roles;
    return "User Detail method.  " + s;
}
```

# Spring Security + JPA

- UserDetails Spring Security ni ichida aylanib yuradigan user Object

SpringSecurity UserDetailsService orqali UserDetails ni oladi.

UserDetailsService dan implementatsiya olgan class foydalaniladi.

Bunda UserDetailsService interfacesining metodlari implementatsiya qilinishi kerak.

- UserDetailsService ga userni logini orqali topib UserDetails ga o'rab berib yuboramiz. Password ni tekshirishni o'ziga qo'yib beramiz. Yaniy config qilamiz. Encodinglarni inobatga olgan xolda.

It compares the password you submit to the password returned by the UserDetails object returned by your UserDetailsService. Please post your config and your UserDetailsService if you need more help.

# Spring Security + JPA Example



```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
  
    @Autowired  
    private CustomUserDetailsService customUserDetailsService;  
  
    @Override  
    protected void configure(AuthenticationManagerBuilder auth) throws.. {  
        auth.userDetailsService(customUserDetailsService)  
            .passwordEncoder(getPasswordEncoder());  
    }  
}
```

# Spring Security + JPA Example



```
@Component  
public class CustomUserDetailsService implements UserDetailsService {
```

```
    @Autowired  
    private ProfileRepository profileRepository;
```

```
    @Override  
    public UserDetails loadUserByUsername(String s) ... {
```




```
        // get user from db and return it by wrapping
```

```
        return new CustomUserDetails(profile);
```

```
    }
```

```
}
```



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



# Spring Security + JWT

- ▶ Which steps we should perform ?

# Adding JWT dependency

```
► <dependency>  
  <groupId>io.jsonwebtoken</groupId>  
  <artifactId>jjwt</artifactId>  
  <version>0.9.1</version>  
</dependency>
```

# Adding Jwt Util

- ▶ @Component  
public class JwtTokenUtil {
  - 1. Generate Jwt Token From some User information
  - 2. Valdiate Jwt token and get user info from jwt token}
- ▶ // full code in spring\_security\_06\_jpa\_jwt project

# Adding JWT Filter

► @Component  
public class JwtTokenFilter extends OncePerRequestFilter {

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

// 1. get jwt token from http request  
// 2. validate jwt token  
// 3. get user detail from jwt  
// 4. create authentication  
// 5. set authentication to spring context.  
}  
  
} // full code in spring\_security\_06\_jpa\_jwt project

# Adding JWT filter to Spring Security

- ▶ `@Autowired`  
`private JwtTokenFilter jwtTokenFilter;`  
`.`  
`protected void configure(HttpSecurity http) throws Exception {`  
    `// Add JWT token filter`  
    `http.addFilterBefore(`  
        `jwtTokenFilter,`  
        `UsernamePasswordAuthenticationFilter.class`  
    `);`  
`}`
- ▶ `// full code in spring_security_06_jpa_jwt project`

# Adding Not Authorized Case. 1

- ▶ If user not authorized spring security send as response the authorization from.
- ▶ We can change it by changing NotAuthorized response.

```
▶ @Component
public class AuthEntryPointJwt implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
        AuthenticationException authException) throws IOException {

        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized");
    }
}
```

# Adding Not Authorized Case. 2

- Adding to Spring config.

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint);  
}
```

# AuthenticationManager

- By default, it's not publicly accessible, and we need to explicitly expose it as a bean in our configuration class.

```
@EnableWebSecurity
```

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
    // Details omitted for brevity
```

```
    @Override
```

```
    @Bean
```

```
    public AuthenticationManager authenticationManagerBean() throws Exception {  
        return super.authenticationManagerBean();
```

```
    }
```

```
}
```



# .hasRole()

```
► http.authorizeRequests()  
  // Our public endpoints  
  .antMatchers("/api/public/**").permitAll()  
  
  .antMatchers(HttpMethod.GET, "/api/author/**").permitAll()  
  .antMatchers(HttpMethod.POST, "/api/author/search").permitAll()  
  
  // Our private endpoints  
  .antMatchers("/api/admin/user/**").hasRole("USER_ADMIN")  
  .antMatchers("/api/author/**").hasRole("AUTHOR_ADMIN")  
  .antMatchers("/api/book/**").hasRole("BOOK_ADMIN")  
  .anyRequest().authenticated();  
  
// example in spring_security_07_jpa_jwt project
```

# Method Security 1

- ▶ The Spring Security framework defines the following annotations for web security:
- ▶ @PreAuthorize supports [Spring Expression Language](#) and is used to provide expression-based access control *before* executing the method.
- ▶ @PostAuthorize supports [Spring Expression Language](#) and is used to provide expression-based access control *after* executing the method (provides the ability to access the method result).
- ▶ @PreFilter supports [Spring Expression Language](#) and is used to filter the collection or arrays *before* executing the method based on custom security rules we define.
- ▶ @PostFilter supports [Spring Expression Language](#) and is used to filter the returned collection or arrays *after* executing the method based on custom security rules we define (provides the ability to access the method result).
- ▶ @Secured doesn't support [Spring Expression Language](#) and is used to specify a list of roles on a method.
- ▶ @RolesAllowed doesn't support [Spring Expression Language](#) and is the [JSR 250](#)'s equivalent annotation of the @Secured annotation.

# Method Security 2

- ▶ These annotations are disabled by default and can be enabled in our application as follows:
- ▶ **@EnableWebSecurity**  
**@EnableGlobalMethodSecurity**(  
    securedEnabled = true,  
    jsr250Enabled = true,  
    prePostEnabled = true )  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
    *// Details omitted for brevity*  
}

## Method Security 2

- ▶ `securedEnabled = true` enables `@Secured` annotation.
- ▶ `jsr250Enabled = true` enables `@RolesAllowed` annotation.
- ▶ `prePostEnabled = true`  
enables `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, `@PostFilter` annotations.
- ▶ After enabling them, we can enforce role-based access policies on our API endpoints
- ▶ `@RolesAllowed("USER_ADMIN")`
- ▶ OR
- ▶ `@PreAuthorize("hasRole('MODERATOR')")`

`@PreAuthorize("hasAnyRole('bank','admin')")`

# Spring Security + Swagger

## ► In configuration

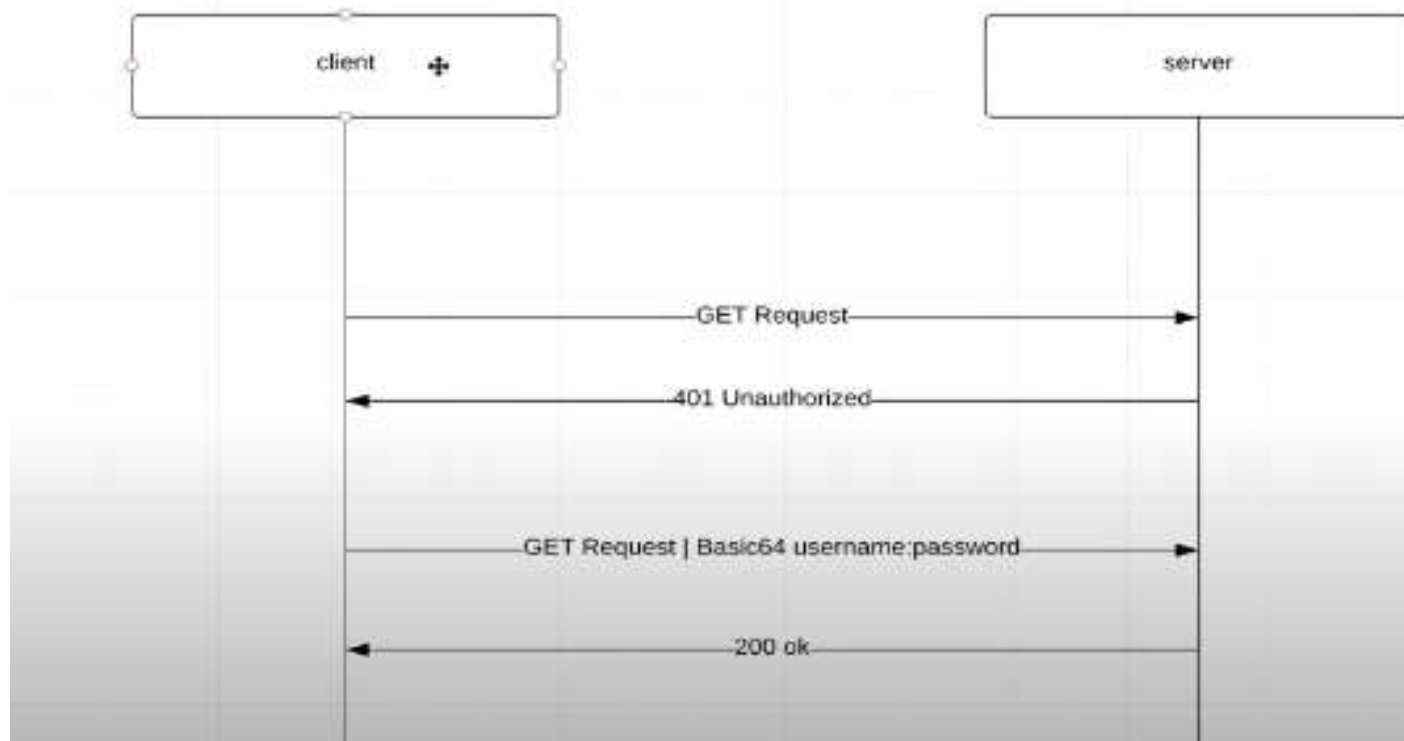
```
private static final String[] AUTH_WHITELIST = {  
    "/v2/api-docs",  
    "/configuration/ui",  
    "/configuration/security",  
    "/swagger-ui.html",  
    "/webjars/**",  
    "/v3/api-docs/**",  
    "/swagger-ui/**",  
    "/swagger-resources",  
    "/swagger-resources/**"  
};
```

## ► And Then

```
.antMatchers(AUTH_WHITELIST).permitAll()
```

# How it works

## Basic Auth



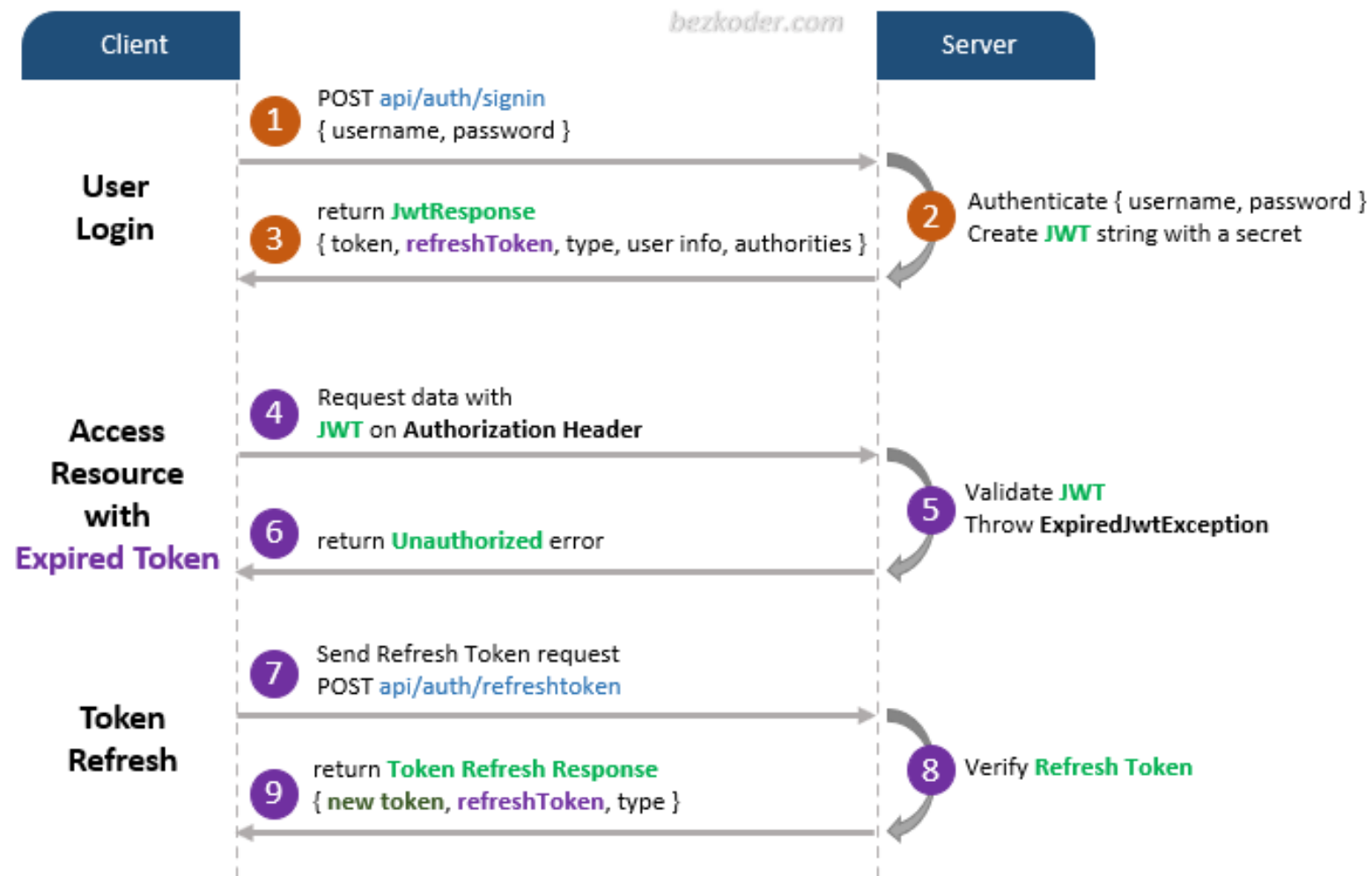
# JWT + Refresh Token

# RefreshToken vs AccessToken

- ▶ AccessToken is a main token and. AccessToken expired time will be short. User always sends Accesstoken in header.
- ▶ RefreshToken will be used when AccessToken expired and needs to be updated. So RefreshToken used only fo update AccessToken.
- ▶ AccessToken bu asosiy token bo'ladi. User har doim uni request da ishlatadi. AccessToken ni amal qilish muddati qisqa bo'ladi.
- ▶ AccessToken amal qilish muddati tugaganidan keyin user qaytadan Login qilib o'tirmasligi uchun u RefreshToken orqali yangi AccessToken oladi.
- ▶ RefreshToken faqat shu AccessToken ni update qilish uchun kerak holos.



# How it works



# AccessToken - Implementation

- ▶ Write during lesson. Mazgios

# Links

- ▶ <https://www.bezkoder.com/spring-boot-refresh-token-jwt/>
- ▶ <https://github.com/bezkoder/spring-boot-refresh-token-jwt>
- ▶ <https://www.youtube.com/watch?v=VVn9OG9nfH0>