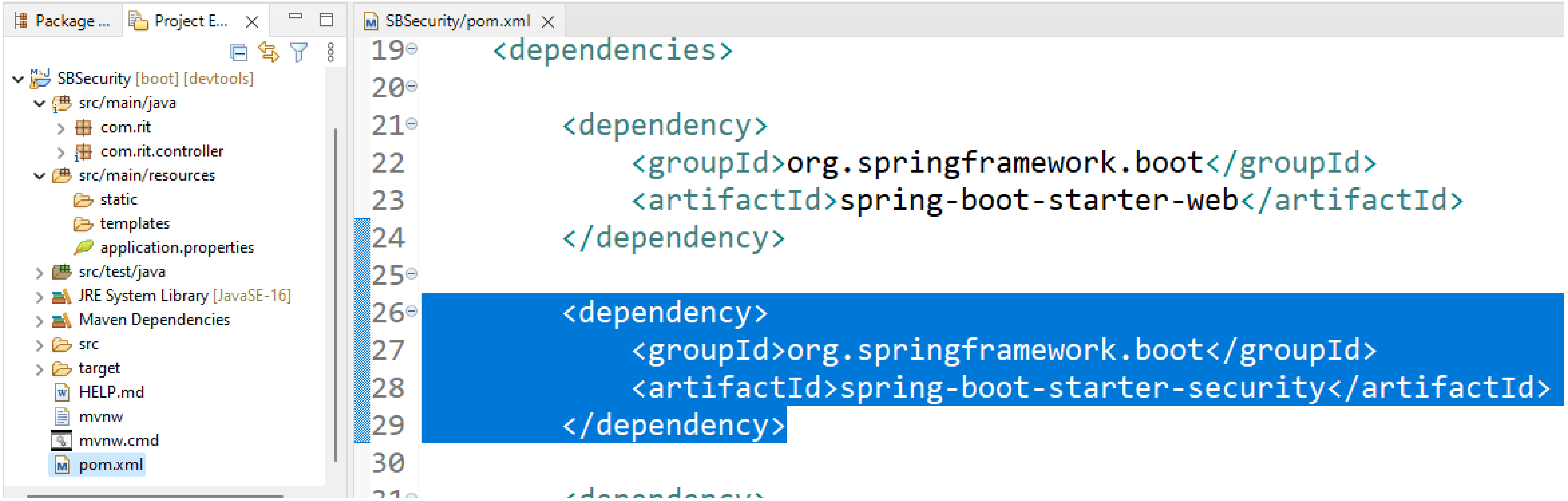
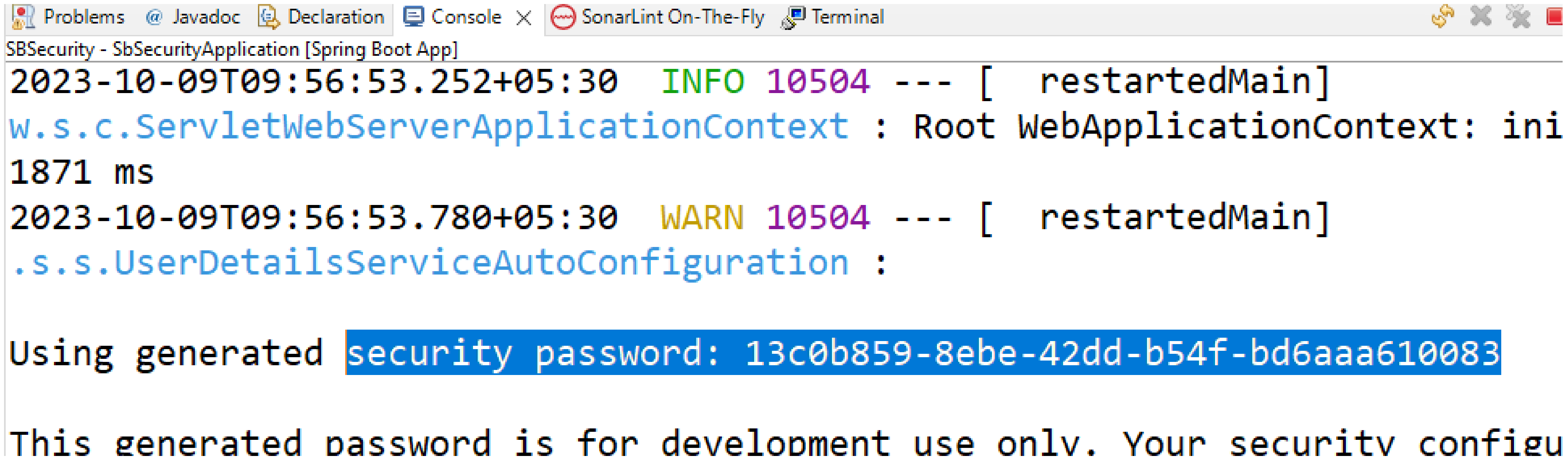
Spring Boot Security

Authentication & Authorization

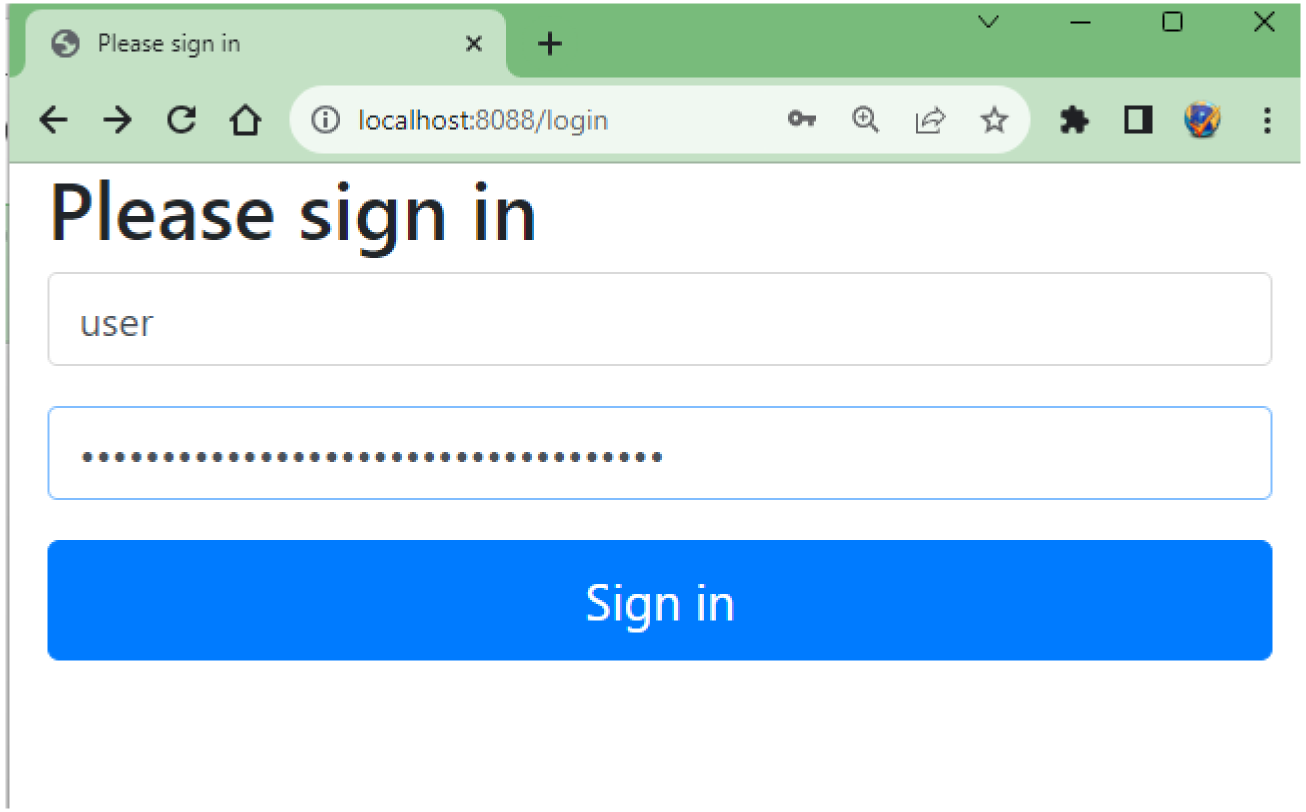
- Authentication
  - Who are you?
  - Username & password
- Authorization
  - What access should I grant to you?
  - Admin, Manager, Executive…



```
    <dependencies>

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

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

        <dependency>
```



```
SBSecurity - SbSecurityApplication [Spring Boot App]
2023-10-09T09:56:53.252+05:30  INFO 10504 --- [  restartedMain]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: ini
1871 ms
2023-10-09T09:56:53.780+05:30  WARN 10504 --- [  restartedMain]
.s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: 13c0b859-8ebe-42dd-b54f-bd6aaa610083

This generated password is for development use only. Your security configu
```


Please sign in

localhost:8088/login

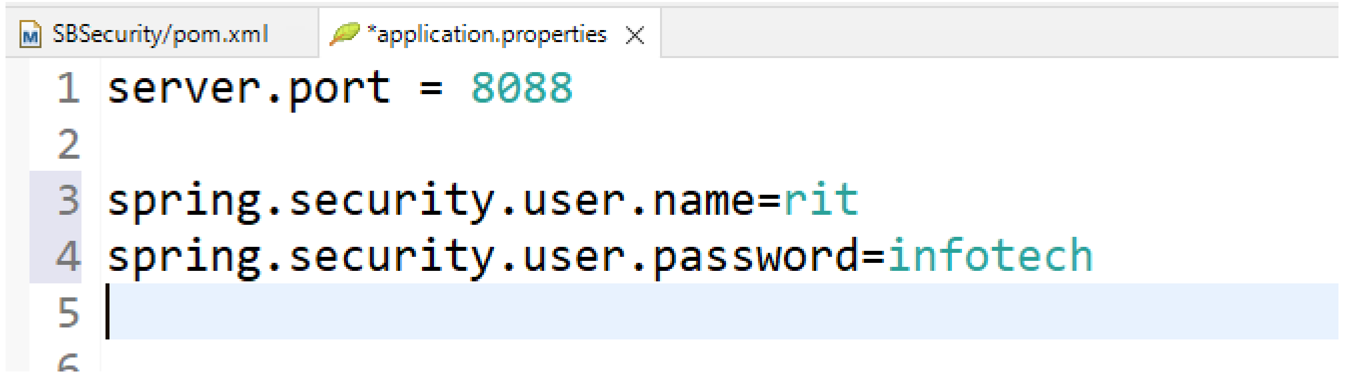
Please sign in

user

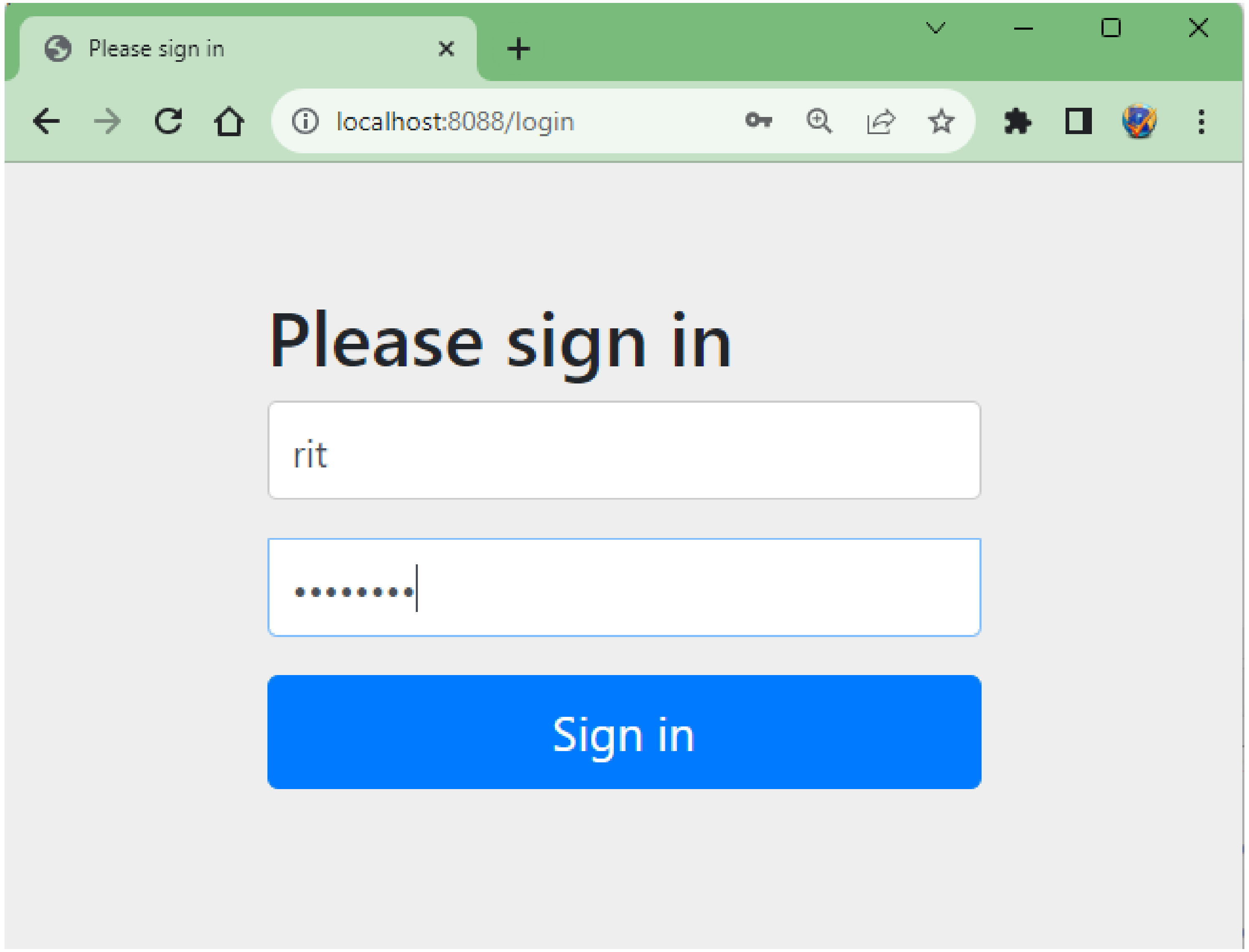
••••••••••••••••••••••••••••••••••••

Sign in

Some predefined properties



```
server.port = 8088

spring.security.user.name=rit
spring.security.user.password=infotech
```

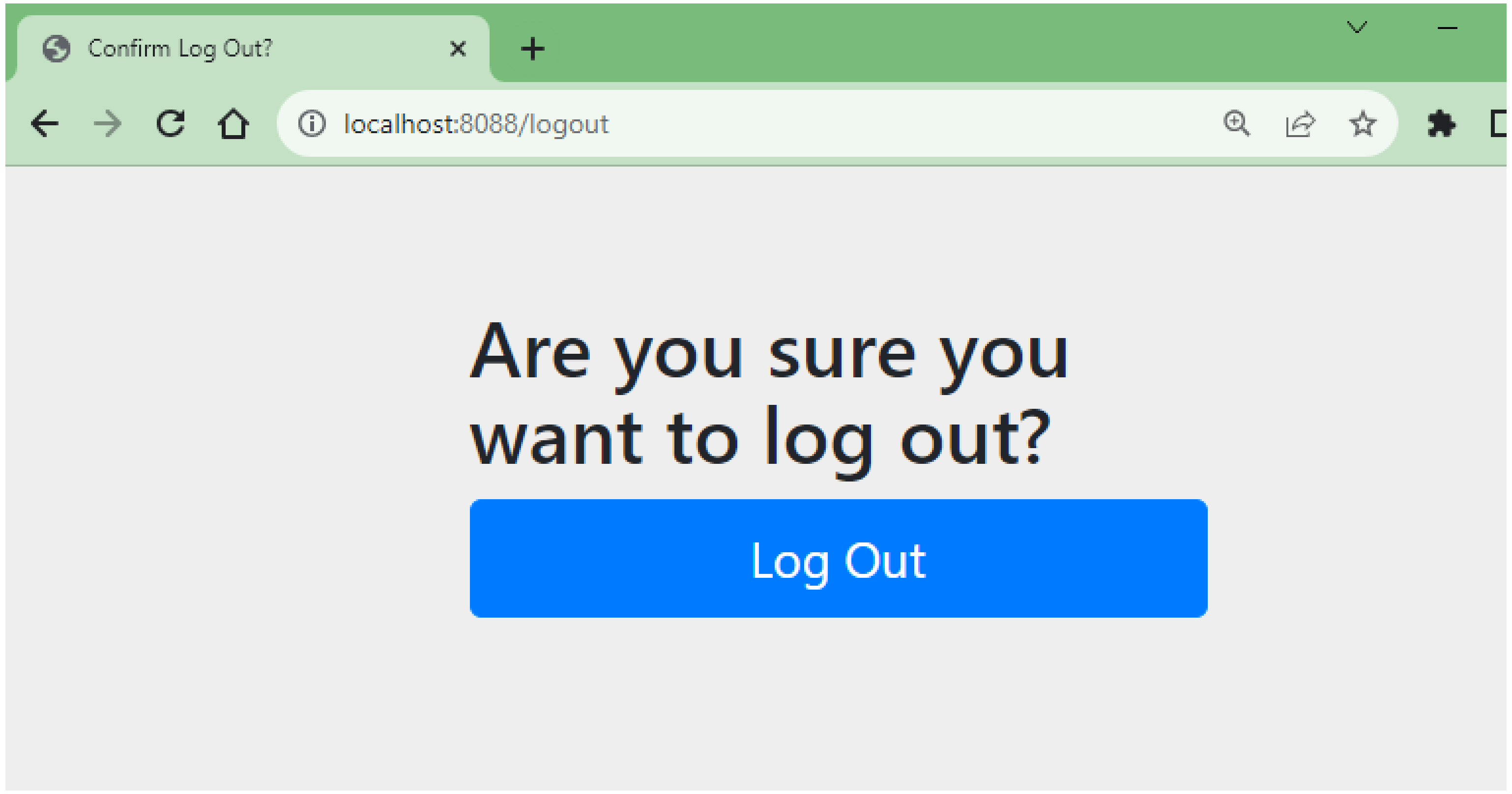


Please sign in

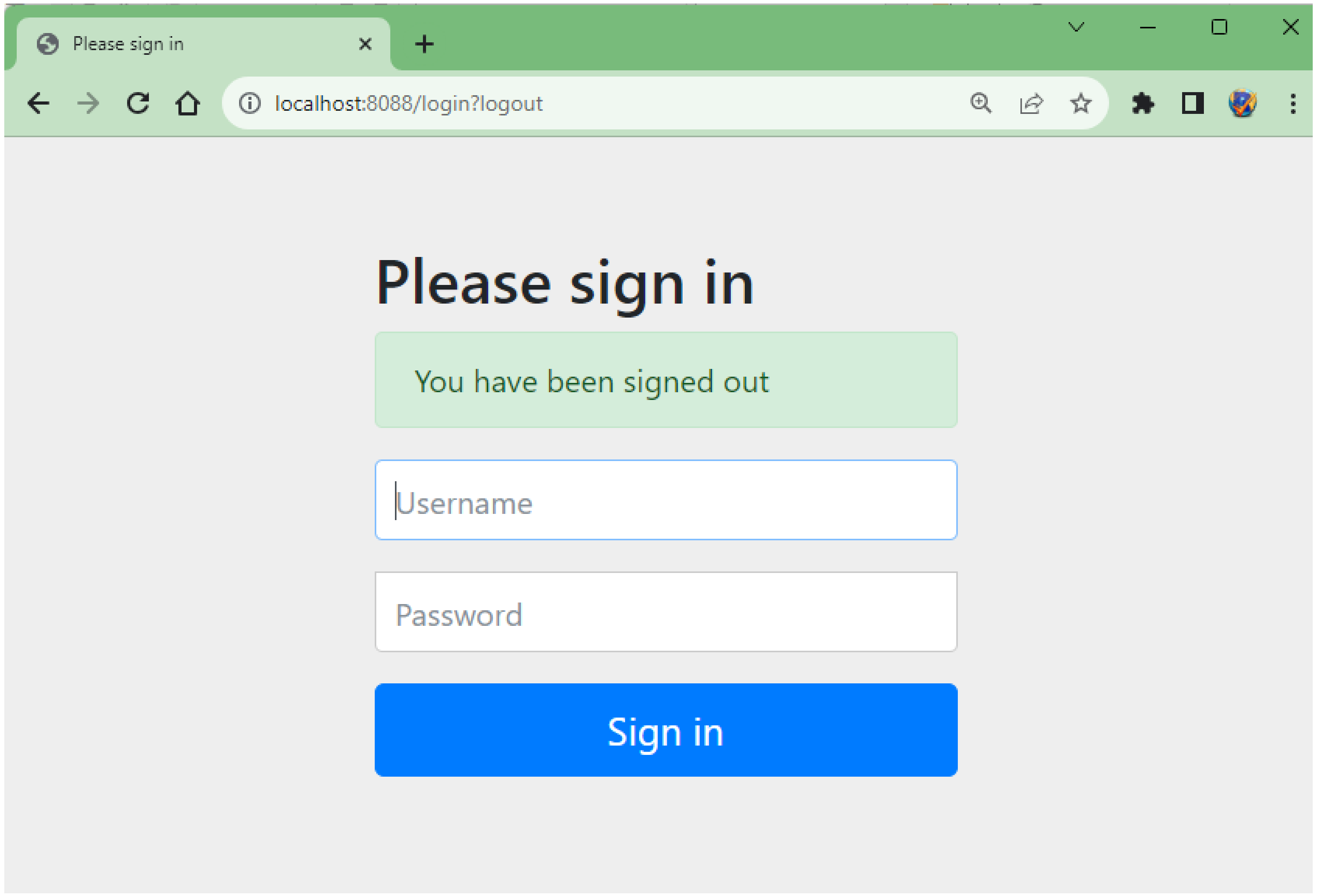
rit

••••••••

Sign in



Are you sure you want to log out?



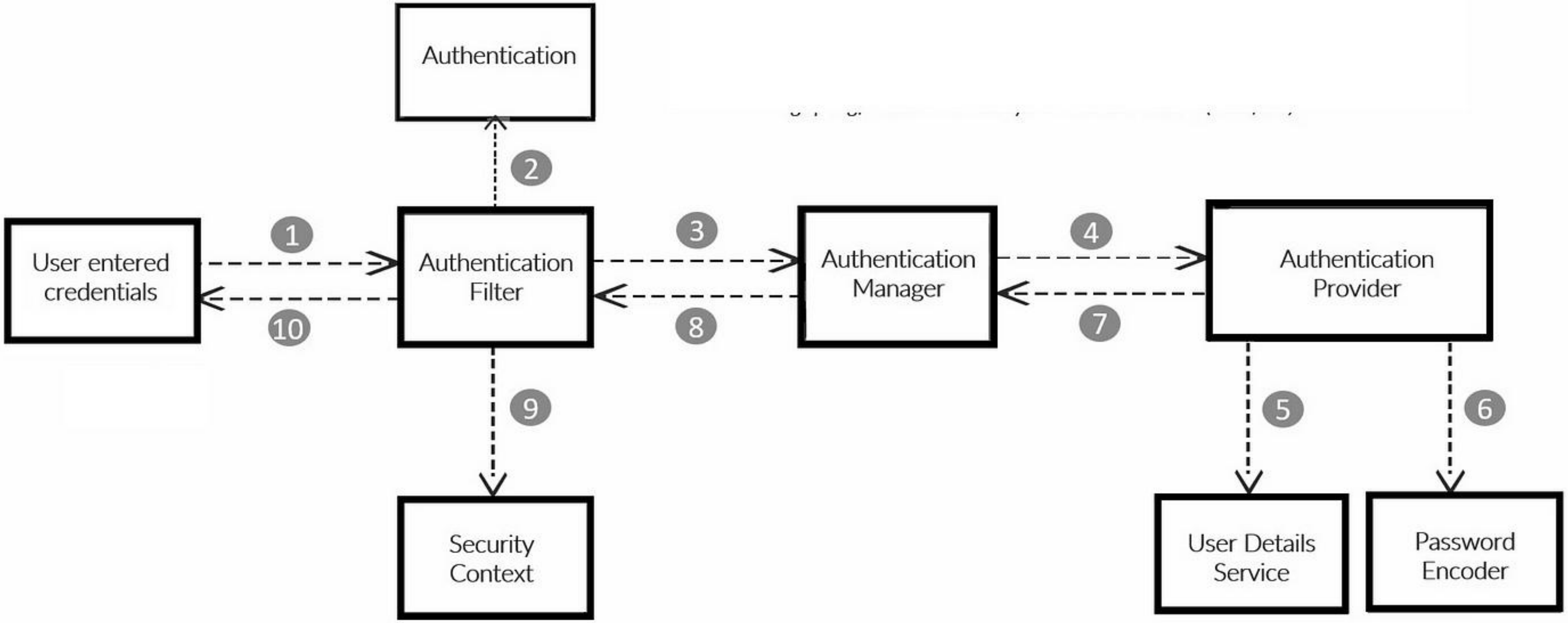
Please sign in

You have been signed out

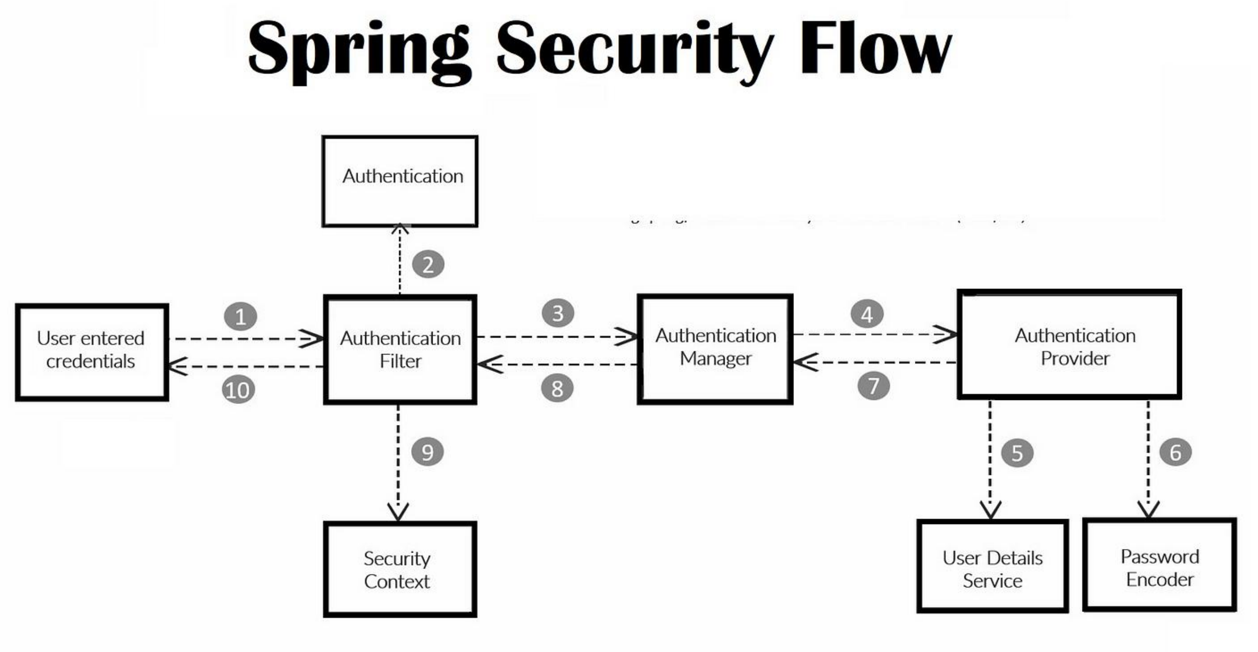
Spring Security

Internal Work flow

Spring Security Flow



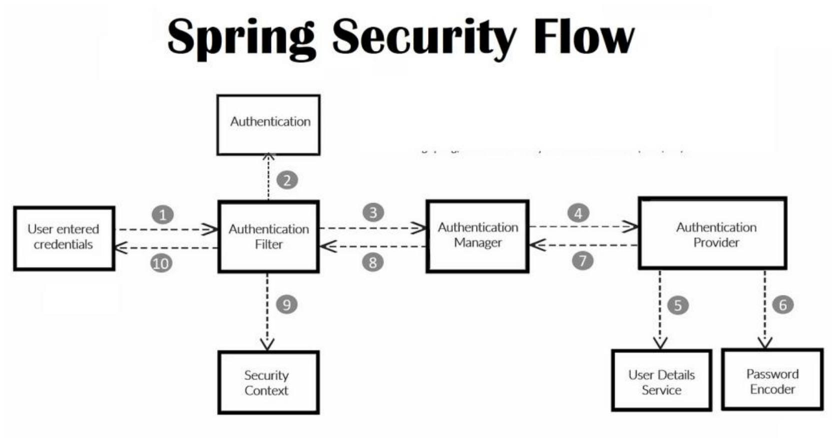
Spring Security Flow



1. User Submits Login Form
   - This triggers an HTTP POST request.

2. Authentication Filter Triggers
   - There are multiple authentication filters available like BasicAuthenticationFilter, BearerTokenAuthenticationFilter, OAuth2LoginAuthenticationFilter, UsernamePasswordAuthenticationFilter and more.
   - In this UsernamePasswordAuthenticationFilter is specifically used to intercept the form-based login request.
   - It extracts the credentials and creates an Authentication object (UsernamePasswordAuthenticationToken).

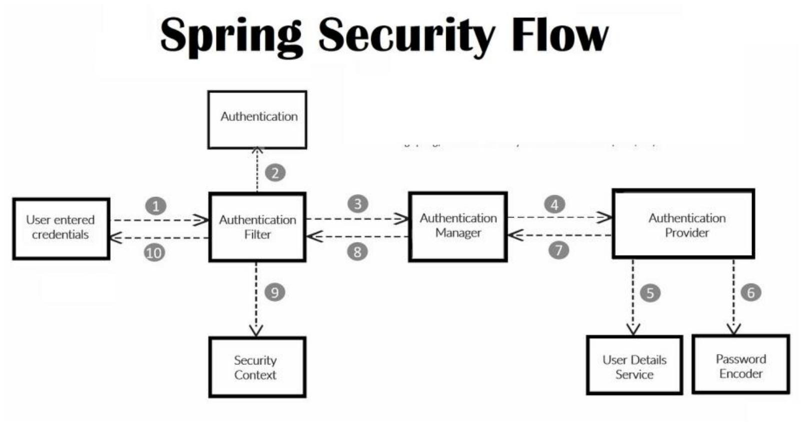
Spring Security Flow



3. Authentication Object Sent to Manager
   - The AuthenticationManager receives the authentication token.
   - Its job is to validate the token using configured authentication providers.

4. Authentication Provider Takes Over
   - There are different authentication providers includes LdapAuthenticationProvider, JwtAuthenticationProvider, OAuth2LoginAuthenticationProvider, RememberMeAuthenticationProvider, DaoAuthenticationProvider and more
   - In this DaoAuthenticationProvider is most commonly used.
   - Authenticates using a UserDetailsService and a PasswordEncoder.
   - Used in traditional form-based login.

Spring Security Flow



5. UserDetailsService Loads User
   - Retrieves user info like username, password, and roles from the data source.
   - Returns a UserDetails object with these credentials.

6. Password Verified by Encoder
   - The entered password is encoded (e.g., using BCrypt). Compared against the stored encoded password in UserDetails.

7. Authentication Success → Security Context
   - If credentials are valid, a new Authentication object (authenticated = true) is created.
   - Stored in SecurityContextHolder, allowing access throughout the app.
   - The authenticated UserDetails (principal) is now available.
   - Access to protected routes and resources is granted based on roles/authorities.

Spring Security

Legacy using **WebSecurityConfigurerAdapter**

Step to develop

- Create a Security Configuration class (in a package security)
- Annotate it with @Configuration
- Extends from WebSecurityConfigurerAdapter
- Press Ctrl+space within the class scope, it will list 3 **configure** methods
  - Authentication (via **AuthenticationManagerBuilder**):
    - Configure which users are allowed to authenticate, where they come from, and how (in-memory, JDBC, LDAP, custom authentication providers, etc.)
  - Authorization (via **HttpSecurity**):
    - Set up rules on which users can access certain URLs and resources.
    - Can control things like form login, logout, HTTP basic authentication, CSRF protection, session management, URL access restrictions, etc.
  - Global Security Settings (via **WebSecurity**):
    - To apply general web security configuration for the whole application.
    - Like exclude certain resources (like CSS, JS, Images..) from security filters.

Imports & Class

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

Authentication

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    // In-memory authentication with plain text passwords
    auth.inMemoryAuthentication()
        .withUser("admin")
            .password("admin123")  // Plain text password
            .roles("ADMIN")
        .and()
        .withUser("user")
            .password("user123")
            .roles("USER")
        .and()
        .withUser("guest")
            .password("guest123")
            .roles("GUEST");
}
```

Authentication with Encoder

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .passwordEncoder(passwordEncoder())  // Set password encoder
        .withUser("admin")
        .password(passwordEncoder().encode("admin123"))  // Encoded password
        .roles("ADMIN")
        .and()
        .withUser("user")
        .password(passwordEncoder().encode("user123"))  // Encoded password
        .roles("USER");
}

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

Authorization

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .antMatchers("/user/**").hasRole("USER")
            .antMatchers("/guest/**").hasRole("GUEST")
            .antMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
        .and()
        .formLogin()  // Enable form login
            .loginPage("/login")  // Custom login page
            .permitAll()
        .and()
        .logout()  // Enable logout
            .permitAll();
}
```

Web security

```
    @Override
    public void configure(WebSecurity web) throws Exception {
        // Ignoring static resources like CSS, JS, images
        web.ignoring().antMatchers("/resources/**", "/static/**", "/css/**", "/js/**", "/images/**");
    }
}
```

Authentication Types

Authentication Type	Use Case
In-Memory Authentication	Simple apps, development, testing
JDBC Authentication	Persistent user data in a relational DB
LDAP Authentication	Use of LDAP/Active Directory for user management
Custom Authentication Provider	Custom logic, integration with external systems
Form-Based Authentication	Traditional web apps with username/password login
HTTP Basic Authentication	Simple APIs, stateless HTTP communication
OAuth2 / OpenID Connect	SSO, third-party authentication (e.g., Google, GitHub login)
JWT Authentication	Stateless APIs, microservices

Spring Security

Modern using **SecurityFilterChain**

In-Memory Authentication

In Memory Authentication

Steps to develop

- Create a project with following dependencies
  - Web, Security.
- Create a controller with end points
  - /admin/{id} for admin
  - /user/{id} for both admin and user
  - /home for any role
- Create a configuration class to override username and password
- Run the application
  - /admin/123 for admin,
  - /user/123 for both admin and user,
  - /home for any role

Dependencies

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

Controller Class

```
@RestController
public class AccountController {
		@GetMapping("/admin/{id}")
		public String admin(@PathVariable int id) {
			return "Welcome Admin, Id is "+id;
		}
		@GetMapping("/user/{id}")
		public String user(@PathVariable int id) {
			return "Welcome User, Id is "+id;
		}
		@GetMapping("/home")
		public String home() {
			return "Welcome Home ";
		}
}
```

Configuration Class

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
```

Authentication

```
@Bean
public UserDetailsService userDetailsService() {
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

    manager.createUser(User.withUsername("admin")
            .password(passwordEncoder().encode("admin")) // encoded password
            .roles("ADMIN").build());
    manager.createUser(User.withUsername("user")
            .password(passwordEncoder().encode("user"))
            .roles("USER").build());
    return manager;
}

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

Authorization

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

                // http.authorizeRequests() is deprecated from Spring Security 6.1
                http.authorizeHttpRequests(requests -> requests
                                .requestMatchers("/admin/**").hasRole("ADMIN")
                                .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
                                .anyRequest().authenticated())
                        .formLogin(login -> login.permitAll())
                        .logout( logout -> logout.permitAll());

                        // Custom login page
                        //.formLogin(login -> login.loginPage("/login").permitAll())

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

JDBC Authentication

JDBC Authentication

Steps to Develop

- Create a project with following dependencies Web, Security, MySQL and JPA.
- Update application.properties with datasource details

- Create 2 tables users & authorities with couple of records
- Credentials : admin & admin, user & user

- Modify configuration class for JDBC authentication
- Run the application

```
use springsecurityjdbcauth;

CREATE TABLE users (
    username VARCHAR(50) PRIMARY KEY,
    password VARCHAR(255) NOT NULL,
    enabled BOOLEAN NOT NULL
);
CREATE TABLE authorities (
    username VARCHAR(50),
    authority VARCHAR(50),
    FOREIGN KEY (username) REFERENCES users(username)
);
INSERT INTO authorities (username, authority) VALUES ('admin', 'ROLE_ADMIN');
INSERT INTO authorities (username, authority) VALUES ('user', 'ROLE_USER');
INSERT INTO users (username, password, enabled) VALUES ('user',
'$2a$10$k7LQQ9hZ8g8g7O9KlZzZcOsNzjps9kwf7sgWmcnDZ2yygdWVhdU6y', true);
INSERT INTO users (username, password, enabled) VALUES ('admin',
'$2a$10$Xf9n8RiVFEFTejp.AgXOUyN3ccVwprXjV6a9dcNxk1XgwvEphYBei', true);
```

Configuration Class

```
import javax.sql.DataSource;
 …
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl;
…
@Configuration
@EnableWebSecurity
public class SecurityConfig {


        @Bean
        public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{
                //Same as InMemory Code
        }
         @Bean
         public PasswordEncoder passwordEncoder() {
             return new BCryptPasswordEncoder();
         }
```

```
  @Bean
  public UserDetailsService userDetailsService(DataSource dataSource) {
    JdbcDaoImpl jdbcDao = new JdbcDaoImpl();
    jdbcDao.setDataSource(dataSource);
    jdbcDao.setUsersByUsernameQuery("SELECT username, password, enabled FROM users WHERE username = ?");
    jdbcDao.setAuthoritiesByUsernameQuery("SELECT username, authority FROM authorities WHERE username = ?");
    return jdbcDao;
  }

     //AuthenticationConfiguration, picks up the UserDetailsService and PasswordEncoder automatically if they are
     defined as beans.
     @Bean
     public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration)
     throws Exception {
            return authenticationConfiguration.getAuthenticationManager();
     }
}
```

OAuth2 Authentication

OAuth2 Authentication

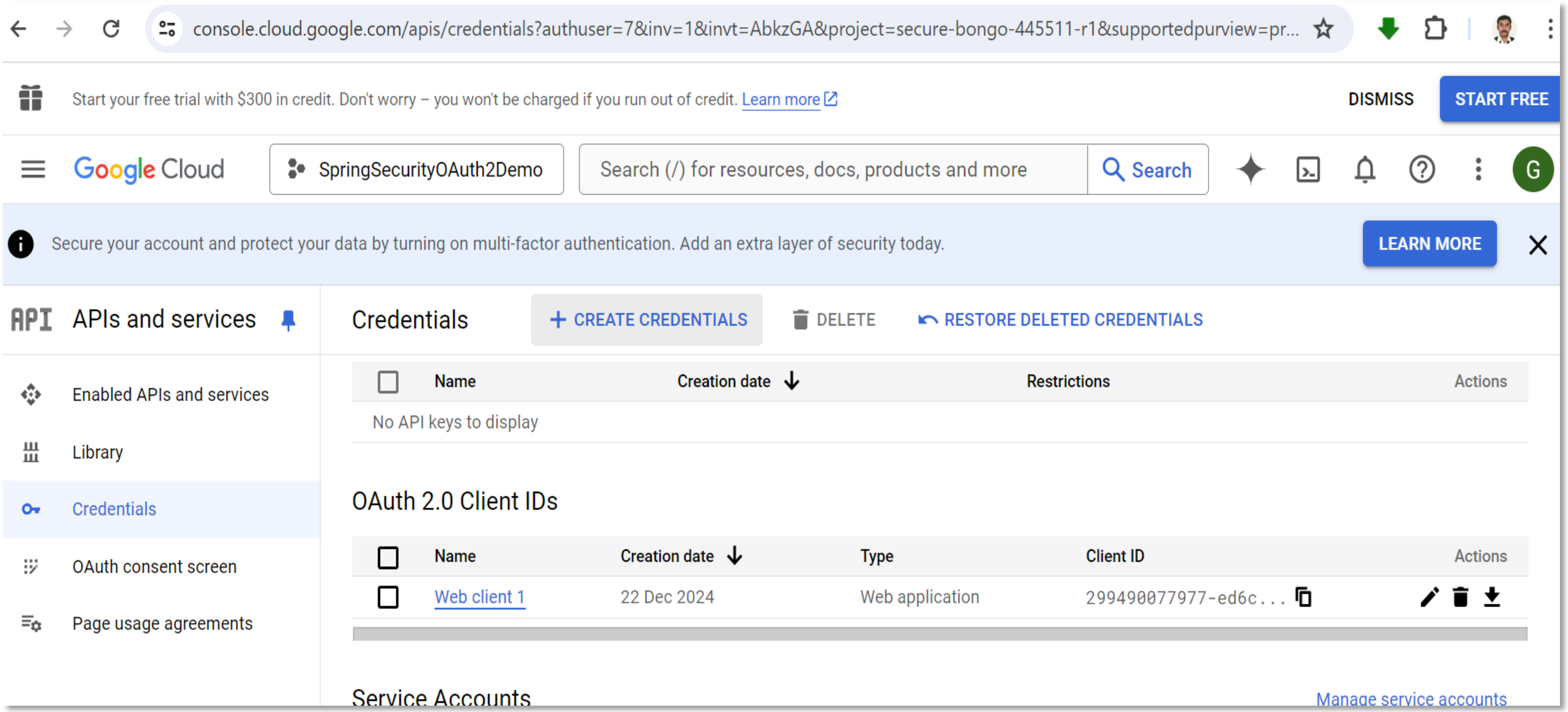
Steps to Develop

- Create Oauth Client Id with Google/Facebook…
- Create a project with following dependencies Web, Security, MySQL, JPA & OAuth2 Client.
- Create application.yml with OAuth2 details
- Modify configuration class for OAuth & JDBC authentication
- Update endpoint in controller to display returned user details
- Run the application

OAuth Client ID

- Go to Google Developer Console: https://console.developers.google.com/
- Create a new project.
- Navigate to APIs & Services > Credentials> + CREATE CREDENTIALS.
- Create OAuth2 credentials by selecting "OAuth client ID".
- Configure the consent screen
  - Javascript Origin: http://localhost:8080
  - Redirect URI: http://localhost:8080/login/oauth2/code/google.
- Copy the Client ID and Client Secret and paste them into your application.yml.

OAuth Client ID



console.cloud.google.com/apis/credentials?authuser=7&inv=1&invt=AbkzGA&project=secure-bongo-445511-r1&supportedpurview=pr...

Start your free trial with $300 in credit. Don't worry – you won't be charged if you run out of credit. Learn more

DISMISS START FREE

Google Cloud SpringSecurityOAuth2Demo Search (/) for resources, docs, products and more Search

Secure your account and protect your data by turning on multi-factor authentication. Add an extra layer of security today. LEARN MORE

APIs and services

- Enabled APIs and services
- Library
- Credentials
- OAuth consent screen
- Page usage agreements

Credentials + CREATE CREDENTIALS DELETE RESTORE DELETED CREDENTIALS

Name	Creation date ↓	Restrictions	Actions
No API keys to display			

OAuth 2.0 Client IDs

Name	Creation date ↓	Type	Client ID	Actions
Web client 1	22 Dec 2024	Web application	299490077977-ed6c...	

Service Accounts Manage service accounts

OAuth2 Dependencies

```xml
<dependencies>

    <!-- Spring Security OAuth2 Login -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-client</artifactId>
    </dependency>
    <!-- Spring Security OAuth2 Ends -->
    <!—- web, mysql, jpa, security, test -->
</dependencies>
```

Application.yml (with OAuth Details)

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          google:
#            client-id: YOUR_GOOGLE_CLIENT_ID
#            client-secret: YOUR_GOOGLE_CLIENT_SECRET
            scope: profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "http://localhost:8080/login/oauth2/code/google"
            client-name: Google
#             login-page: /login
        provider:
          google:
            authorization-uri: https://accounts.google.com/o/oauth2/auth
            token-uri: https://oauth2.googleapis.com/token
            user-info-uri: https://www.googleapis.com/oauth2/v3/userinfo
            user-name-attribute: sub
```

```
import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
```

```
//Authentication with AuthenticationManager using PasswordEncoder, UserService
   @Bean
    public PasswordEncoder passwordEncoder() {
              return new BCryptPasswordEncoder();
   }

   @Bean
   public UserDetailsService userDetailsService(DataSource dataSource) {
      JdbcDaoImpl jdbcDao = new JdbcDaoImpl();
      jdbcDao.setDataSource(dataSource);
      jdbcDao.setUsersByUsernameQuery("SELECT username, password, enabled FROM users WHERE username = ?");
      jdbcDao.setAuthoritiesByUsernameQuery("SELECT username, authority FROM authorities WHERE username = ?");
      return jdbcDao;
   }

   @Bean
   public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {
      return authenticationConfiguration.getAuthenticationManager();
   }
```

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

            http.authorizeHttpRequests(requests ->
                requests.requestMatchers("/login", "/login/oauth2/**").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
                .anyRequest().authenticated())
                .oauth2Login(oauth2 -> {}) // using default OAuth2UserService internally
                .formLogin(login -> login.permitAll())
                .logout(logout -> logout.permitAll());

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

Update End point in controller

```
@GetMapping("/home")
public OAuth2User home(@AuthenticationPrincipal OAuth2User principal, Model model) {

        if (principal != null) {
            // Access user details from the OAuth2User object
            // String name = principal.getAttribute("name");
        }
        return principal;
}
```

JWT Authentication

Add Dependencies

include web, security, jpa, mysql dependencies

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.11.5</version>
</dependency>

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

Configuration class

```
import com.example.demo.filter.JwtAuthenticationFilter;
import com.example.demo.filter.JwtAuthorizationFilter;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
…
@Configuration
public class SecurityConfig {

    private final UserDetailsService userDetailsService;

    public SecurityConfig(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }
  @Bean     public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
@Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
      http.csrf().disable()
         .authorizeRequests()
            .antMatchers("/login", "/register").permitAll()
            .antMatchers("/api/**").authenticated()
         .and()
         .addFilter(new JwtAuthenticationFilter(authenticationManager()))
         .addFilter(new JwtAuthorizationFilter(authenticationManager(), userDetailsService));
      return http.build();
   }
   @Bean
   public AuthenticationManager authenticationManager(HttpSecurity http) throws Exception {
      return http.getSharedObject(AuthenticationManagerBuilder.class)
                .userDetailsService(userDetailsService)
                .passwordEncoder(passwordEncoder())
                .and().build();
   }
 }
```

About the configuration code

- **SecurityFilterChain**: Defines the security rules for URL patterns, such as allowing public access to /login and /register, while requiring authentication for /api/**.
- **JwtAuthenticationFilter**: Custom filter that handles authentication based on the login credentials.
- **JwtAuthorizationFilter**: Custom filter that checks and validates JWT tokens for protected endpoints.

JWT Utility class (Generating & validating JWT tokens)

```
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import java.util.Date;

public class JwtUtil {

    private static final String SECRET_KEY = "secret";  // You can use a more secure key in production

    public static String generateToken(String username) {
        return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + 86400000))  // 1 day expiration
            .signWith(SignatureAlgorithm.HS512, SECRET_KEY)
            .compact();
    }
    public static boolean validateToken(String token) {
        try {
            Jwts.parser()
                                .setSigningKey(SECRET_KEY)
                                .parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

  public static String getUsernameFromToken(String token) {
        return Jwts.parser()
                                .setSigningKey(SECRET_KEY)
                                .parseClaimsJws(token)
                                .getBody()
                                .getSubject();
    }
}
```

Custom Authentication Filter

```
import com.example.demo.util.JwtUtil;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
    private final AuthenticationManager authenticationManager;

    public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    @Override     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
throws IOException {
        String username = request.getParameter("username");
        String password = request.getParameter("password");
         UsernamePasswordAuthenticationToken authenticationToken =  new
                        UsernamePasswordAuthenticationToken(username, password);
         return authenticationManager.authenticate(authenticationToken);
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
                                          FilterChain chain, Authentication authResult) throws IOException, ServletException {
        String token = JwtUtil.generateToken(authResult.getName());
        response.setHeader("Authorization", "Bearer " + token);
    }
}
```

Custom Authorization Filter

```
import com.example.demo.util.JwtUtil;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.authentication.www.BasicAuthenticationFilter;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import javax.servlet.Filter;
…
import java.io.IOException;
import java.util.Collections;

public class JwtAuthorizationFilter extends BasicAuthenticationFilter {
    private final UserDetailsService userDetailsService;
    public JwtAuthorizationFilter(AuthenticationManager authenticationManager, UserDetailsService
userDetailsService) {
        super(authenticationManager);
        this.userDetailsService = userDetailsService;
    }
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {

        String token = ((HttpServletRequest) request).getHeader("Authorization");

        if (token != null && token.startsWith("Bearer ")) {
            token = token.substring(7);
            if (JwtUtil.validateToken(token)) {
                String username = JwtUtil.getUsernameFromToken(token);
                var user = userDetailsService.loadUserByUsername(username);
                var authentication = new UsernamePasswordAuthenticationToken(user, null, Collections.singleton(new
SimpleGrantedAuthority("USER")));
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        }
        chain.doFilter(request, response);
    }
}
```

About the filters

- **JwtAuthenticationFilter**: This filter processes user login requests, generates the JWT token on successful login, and adds it to the response header.
- **JwtAuthorizationFilter**: This filter checks if the incoming request contains a valid JWT token in the Authorization header, validates it, and sets the authentication in the security context.

Authentication Controller

```java
import com.example.demo.util.JwtUtil;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api")
public class AuthenticationController {

    private final AuthenticationManager authenticationManager;

    public AuthenticationController(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }
    @PostMapping("/login")
    public String login(@RequestParam String username, @RequestParam String password) {
        UsernamePasswordAuthenticationToken authenticationToken = new
UsernamePasswordAuthenticationToken(username, password);
    authenticationManager.authenticate(authenticationToken);

    // Generate and return JWT token after successful authentication
    return JwtUtil.generateToken(username);
    }
}
```

Test the api

****Login**:** Call `POST /api/login` with `username` and `password` parameters to obtain a JWT token.

Example request:

```
POST /api/login?username=john&password=password123
Response:
{
  "Authorization": "Bearer <JWT_TOKEN>"
}
```

- **Access Protected Resource**: Use the generated JWT token to access protected resources.
- Example request:

```
GET /api/protected
Authorization: Bearer <JWT_TOKEN>
Response:
{
  "message": "Success! You are authorized."
}
```

Thank you