Day 2 — Customizing Authentication in Spring Security

Goal of the Day

Learn how to override Spring Security's default authentication mechanism by creating a custom security configuration. We will define our own usernames, passwords, and roles without relying on Spring Security's generated credentials.

1. Understanding Default Authentication

By default, Spring Security provides a randomly generated password and a fixed username 'user'. While this is useful for quick setups, production-grade applications need a custom configuration. We will override this behavior to define our own users and roles.

2. Today’s Deliverables

✔ Create a custom Security Configuration class. ✔ Define an in-memory user store with usernames, passwords, and roles. ✔ Configure access to endpoints based on roles.

3. Step-by-Step Work

Step 1 — Create a Security Configuration Class

Create a new package `config` and inside it, create a class named `SecurityConfig`. Annotate it with `@Configuration` and `@EnableWebSecurity` to tell Spring that this is our security configuration.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/user/**").hasRole("USER")
                .anyRequest().authenticated()
            )
            .formLogin(withDefaults())
            .httpBasic(withDefaults());
        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails admin = User.withDefaultPasswordEncoder()
            .username("admin")
            .password("admin123")
            .roles("ADMIN")
            .build();

        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user123")
            .roles("USER")
            .build();
        return new InMemoryUserDetailsManager(admin, user);
    }
}
```

Step 2 — Create Role-Based Endpoints

Create two controllers, one for ADMIN role and one for USER role, to test role-based access control.

```
@RestController
@RequestMapping("/admin")
public class AdminController {
    @GetMapping("/dashboard")
    public String dashboard() {
        return "Welcome Admin!";
    }
}

@RestController
@RequestMapping("/user")
public class UserController {
    @GetMapping("/profile")
    public String profile() {
        return "Welcome User!";
    }
}
```

Step 3 — Test the Configuration

1. Start the application. 2. Access `/admin/dashboard` → Login as `admin/admin123`. 3. Access `/user/profile` → Login as `user/user123`. 4. Attempt accessing `/admin/dashboard` with user credentials → Should be denied.

4. Key Notes for Day 2

- The `SecurityFilterChain` bean replaces the old `WebSecurityConfigurerAdapter`. - `InMemoryUserDetailsManager` is suitable for demos/testing but not for production. - We will integrate database-backed authentication in upcoming days.

5. Homework

1. Add a public `/home` endpoint that can be accessed without login. 2. Try creating a third role, e.g., `MANAGER`, and secure some endpoints with it. 3. Share a screenshot of your role-based access working successfully.

Created by Pravin Sonwane - https://youtube.com/@programmingwithpravin