Working with Multiple Users

Overview:

When using Spring Security, if no specific UserDetailsService is defined, the framework uses a default configuration. However, you can define your own UserDetailsService to manage custom users with different roles and authorities.

Code Example for Custom In-Memory UserDetailsService:

```
@Bean
public UserDetailsService userDetailsService() {
    UserDetails user1 = User.withDefaultPasswordEncoder()
            .username("muskan")
            .password("root")
            .roles("USER")
            .build();

    UserDetails user2 = User.withDefaultPasswordEncoder()
            .username("admin")
            .password("root")
            .roles("ADMIN")
            .build();

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

Explanation:

- **UserDetailsService Bean**:
  - A UserDetailsService bean is created to define custom users.
  - The User class is a convenient way to create UserDetails instances.
- **User.withDefaultPasswordEncoder()**:
  - Creates a user with a password encoder that stores passwords in plain text (not recommended for production).
- **User Definitions**:
  - Two users are defined:
    - muskan with the role USER.
    - admin with the role ADMIN.

- **InMemoryUserDetailsManager**:
    - An in-memory implementation of UserDetailsService that takes the UserDetails objects (user1 and user2) and stores them.

Complete SecurityConfig.java Example:

```
package com.spring.security.config;

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.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.configurers.CsrfConfigurer;
import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(Customizer.withDefaults())
            .authorizeHttpRequests(request -> request.anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .sessionManagement(session ->
                session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user1 = User.withDefaultPasswordEncoder()
                .username("muskan")
                .password("root")
                .roles("USER")
                .build();
        UserDetails user2 = User.withDefaultPasswordEncoder()
                .username("admin")
                .password("root")
                .roles("ADMIN")
                .build();

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

Notes:

- **@EnableWebSecurity**: Enables web security configuration.
- **UserDetailsService**: Interface to provide user details to Spring Security.
- **Security Filter Chain**:
    - Configures HTTP security settings like CSRF, basic authentication, and session management.
- **In-Memory Users**:
    - Ideal for quick setups, testing, or demos. For production, consider using a database-backed UserDetailsService.