Spring Security

Introduction to Spring Security

Overview of Spring Security

Spring Security is a powerful and customizable authentication and access-control framework for the Spring framework. It is widely used in the Java ecosystem to secure applications by providing mechanisms for authentication, authorization, and protection against common security vulnerabilities like CSRF attacks, session fixation, and more.

Key Features of Spring Security:

- **Authentication**: Verifies the identity of a user or system.
- **Authorization**: Determines the level of access or permissions granted to the authenticated user.
- **Protection against common threats**: CSRF, Clickjacking, and more.
- **Integration**: Seamlessly integrates with Spring-based applications.
- **Flexible configuration**: Can be configured through annotations, Java-based configuration, or XML.

How Spring Security Integrates with Spring Applications

Spring Security integrates into a Spring application by applying security at various layers, such as HTTP request handling, method invocation, and even directly at the data access layer if required. It operates through a series of filters that intercept requests and apply security checks before allowing the request to proceed.

Key Components:

1. **Security Filters**: A filter chain that intercepts HTTP requests and applies security logic. Examples include `UsernamePasswordAuthenticationFilter`, `BasicAuthenticationFilter`, etc.

2. **AuthenticationManager**: Manages the process of authenticating users. It delegates the authentication request to one or more `AuthenticationProvider`s.
3. **AuthenticationProvider**: Responsible for processing authentication requests. It checks the user credentials and returns an `Authentication` object if the user is authenticated.
4. **SecurityContextHolder**: Holds the security context of the current thread, which includes the authenticated user's details and authorities.
5. **GrantedAuthority**: Represents an authority granted to the user, typically in the form of roles (e.g., `ROLE_USER`, `ROLE_ADMIN`).
6. **UserDetailsService**: A core interface used by Spring Security to retrieve user-related data. It has a method to load a user based on their username, which is typically implemented to retrieve user details from a database.

Understanding the Authentication Flow in Spring Security (Latest Version)

In Spring Security, the authentication process is central to securing an application. Understanding the flow of authentication helps you grasp how the framework works under the hood. Let’s break down the authentication process step by step using the latest version of Spring Security.

1. HTTP Request Interception by Security Filters

- When a client sends an HTTP request to a protected resource, the request is intercepted by the Spring Security filter chain.
- The filter chain is a series of security filters that handle different security concerns. The core filter responsible for authentication is `UsernamePasswordAuthenticationFilter`.

2. Authentication Begins

- The `UsernamePasswordAuthenticationFilter` kicks in when a user submits their credentials (usually via a form submission).
- This filter extracts the username and password from the request and creates an `Authentication` object (commonly a `UsernamePasswordAuthenticationToken`).

3. Delegating to the AuthenticationManager

- The filter passes the `Authentication` object to the `AuthenticationManager`.
- In Spring Security, the `AuthenticationManager` is the primary interface for authenticating a user. The default implementation used is `ProviderManager`.

4. Authentication Providers

- The `AuthenticationManager` delegates the authentication request to one or more `AuthenticationProvider`s. Each `AuthenticationProvider` is responsible for a particular type of authentication (e.g., LDAP, JDBC, OAuth).
- The most commonly used `AuthenticationProvider` is `DaoAuthenticationProvider`, which is responsible for authentication using a database.

5. UserDetailsService and Password Encoding

- The `DaoAuthenticationProvider` uses a `UserDetailsService` to load user details from a data source (like a database).
- The `UserDetailsService` returns a `UserDetails` object that contains the user's username, password, and granted authorities.
- The `DaoAuthenticationProvider` then checks the provided password against the stored password. It uses a `PasswordEncoder` to encode the raw password and compare it with the stored hash.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.security.authentication.Authenti
cationManager;
import org.springframework.security.config.annotation.authe
ntication.configuration.AuthenticationConfiguration;
import org.springframework.security.core.userdetails.UserDe
tailsService;
import org.springframework.security.crypto.bcrypt.BCryptPas
swordEncoder;
import org.springframework.security.crypto.password.Passwor
dEncoder;

@Configuration
public class SecurityConfig {
```

```
    @Bean
    public UserDetailsService userDetailsService() {
        // Your custom UserDetailsService implementation he
re
        return username -> {
            // Fetch the user from the database and return
a UserDetails object
        };
    }

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

    @Bean
    public AuthenticationManager authenticationManager(Auth
enticationConfiguration authenticationConfiguration) throws
Exception {
        return authenticationConfiguration.getAuthenticatio
nManager();
    }
}
```

6. Successful Authentication

- If the credentials are valid, the `AuthenticationProvider` returns a fully populated `Authentication` object, indicating the user is successfully authenticated.
- The `SecurityContextHolder` is populated with this `Authentication` object, making the user's identity and authorities available throughout the application.

7. Security Context and Session

- The `SecurityContextHolder` holds the security context, which includes the `Authentication` object. This context is either stored in the HTTP session or managed through stateless sessions (like JWT in REST APIs).

- If the application is configured for session management, the security context is stored in the user's session, allowing for persistent authentication across multiple requests.

8. Redirect or Access Granted

- After successful authentication, the user is redirected to their original requested resource or a default page (like a dashboard).
- The application continues processing the request with the user's authorities considered for authorization decisions.

9. Handling Authentication Failure

- If authentication fails (e.g., wrong password), the `AuthenticationFailureHandler` is invoked.
- Typically, the user is redirected back to the login page with an error message indicating the failure.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.security.config.annotation.web.b
uilders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChai
n;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecu
rity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/login", "/register").per
mitAll()
                .anyRequest().authenticated()
            )
            .formLogin(form -> form
```

```
                .loginPage("/login")
                .failureHandler(customAuthenticationFailure
Handler())
                .defaultSuccessUrl("/home")
                .permitAll()
            )
            .logout(logout -> logout
                .logoutUrl("/logout")
                .logoutSuccessUrl("/login?logout")
                .permitAll()
            );

        return http.build();
    }

    @Bean
    public CustomAuthenticationFailureHandler customAuthent
icationFailureHandler() {
        return new CustomAuthenticationFailureHandler(); //
Replace with your custom failure handler logic
    }
}
```

10. Logout Process

- When the user logs out, Spring Security clears the `SecurityContextHolder`, invalidates the session, and redirects the user to a logout confirmation page or the login page.

Custom AuthenticationProvider Example

You can implement a custom `AuthenticationProvider` to handle specific authentication logic. Here's an example:

```
import org.springframework.security.authentication.Authenti
cationProvider;
```

```
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationExce
ption;
import org.springframework.security.core.userdetails.UserDe
tails;
import org.springframework.security.core.userdetails.UserDe
tailsService;
import org.springframework.security.crypto.password.Passwor
dEncoder;
import org.springframework.stereotype.Component;

@Component
public class CustomAuthenticationProvider implements Authen
ticationProvider {

    private final UserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;

    public CustomAuthenticationProvider(UserDetailsService
userDetailsService, PasswordEncoder passwordEncoder) {
        this.userDetailsService = userDetailsService;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    public Authentication authenticate(Authentication authe
ntication) throws AuthenticationException {
        String username = authentication.getName();
        String password = (String) authentication.getCreden
tials();

        UserDetails user = userDetailsService.loadUserByUse
rname(username);

        if (user != null && passwordEncoder.matches(passwor
d, user.getPassword())) {
            return new UsernamePasswordAuthenticationToken
(user, password, user.getAuthorities());
```

```
        } else {
            throw new AuthenticationException("Invalid user
name or password") {};
        }
    }

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

Configuring ProviderManager

In Spring Security 6, you don't directly configure `ProviderManager` as you did in previous versions. Instead, Spring automatically wires the `AuthenticationProvider`s into the `ProviderManager` when they are defined as beans. Here's how you can include your custom `AuthenticationProvider`:

```
javaCopy code
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.security.authentication.Authenti
cationManager;
import org.springframework.security.config.annotation.authe
ntication.configuration.AuthenticationConfiguration;

@Configuration
public class SecurityConfig {

    private final CustomAuthenticationProvider customAuthen
ticationProvider;

    public SecurityConfig(CustomAuthenticationProvider cust
omAuthenticationProvider) {
        this.customAuthenticationProvider = customAuthentic
```

```
ationProvider;
    }

    @Bean
    public AuthenticationManager authenticationManager(Auth
enticationConfiguration authenticationConfiguration) throws
Exception {
        return authenticationConfiguration.getAuthenticatio
nManager();
    }
}
```

Lets understand in more detail with classname and method class

1. User Submits Credentials

- **Class**: `AbstractAuthenticationProcessingFilter` (e.g., `UsernamePasswordAuthenticationFilter`)
- **Method**: `attemptAuthentication(HttpServletRequest request, HttpServletResponse response)`
- **Description**: When the user submits their login form, the request is intercepted by `UsernamePasswordAuthenticationFilter`. The `attemptAuthentication()` method extracts the username and password from the request and creates an `Authentication` object (typically a `UsernamePasswordAuthenticationToken`).

2. Delegating to `AuthenticationManager`

- **Class**: `AuthenticationManager` (commonly implemented by `ProviderManager`)
- **Method**: `authenticate(Authentication authentication)`
- **Description**: The `AuthenticationManager` is called to authenticate the credentials. This method delegates the authentication process to one or more `AuthenticationProvider`s. The `authenticate()` method is responsible for finding a provider that supports the type of `Authentication` object and processing it.

3. Authentication by `AuthenticationProvider`

- **Class**: `AuthenticationProvider` (e.g., `DaoAuthenticationProvider`)
- **Method**: `authenticate(Authentication authentication)`
- **Description**: The `AuthenticationProvider` is where the actual authentication logic happens. For example, in `DaoAuthenticationProvider`, it will retrieve the user details using a `UserDetailsService`, compare the provided password with the stored one using a `PasswordEncoder`, and then return an authenticated `Authentication` object if successful.

4. Returning the Authenticated Object

- **Class**: `ProviderManager`
- **Method**: `authenticate(Authentication authentication)`
- **Description**: Once an `AuthenticationProvider` successfully authenticates the user, it returns a fully populated `Authentication` object to the `ProviderManager`, which then returns it to the original caller (`AbstractAuthenticationProcessingFilter`).

5. Setting the `Authentication` in `SecurityContextHolder`

- **Class**: `AbstractAuthenticationProcessingFilter` (e.g., `UsernamePasswordAuthenticationFilter`)
- **Method**: `successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication authResult)`
- **Description**: After receiving the authenticated `Authentication` object from the `AuthenticationManager`, this method sets the `Authentication` object into the `SecurityContext` via the `SecurityContextHolder`. This step is crucial as it marks the user as authenticated within the security context of the application.

```
javaCopy code
SecurityContextHolder.getContext().setAuthentication(authResult);
```

6. Security Context Persistence

- **Class**: `SecurityContextPersistenceFilter`
- **Method**: `doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain)`

- **Description**: The `SecurityContextPersistenceFilter` is responsible for storing the `SecurityContext` containing the `Authentication` object into the HTTP session or other persistence mechanisms. This ensures that the user remains authenticated across multiple requests.

```
javaCopy code
SecurityContext contextBeforeChainExecution = repo.loadCont
ext(holder);
SecurityContextHolder.setContext(contextBeforeChainExecutio
n);
```

7. Redirecting the User

- **Class**: `AbstractAuthenticationProcessingFilter` (e.g., `UsernamePasswordAuthenticationFilter`)
- **Method**: `successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication authResult)`
- **Description**: After setting the `Authentication` in the `SecurityContextHolder`, the user is typically redirected to the originally requested URL or a default success URL (e.g., `/home`). This method is also where any post-authentication logic can be executed.

```
response.sendRedirect(targetUrl);
```

8. Handling Subsequent Requests

- **Class**: `SecurityContextPersistenceFilter`
- **Method**: `doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain)`
- **Description**: For subsequent requests, the `SecurityContextPersistenceFilter` retrieves the `SecurityContext` from the session (or other storage) and sets it in the `SecurityContextHolder`, ensuring that the user's authentication state is maintained throughout their session.

9. Accessing the `Authentication` Object

- **Class**: Application Code (any part of the application)
- **Method**: `SecurityContextHolder.getContext().getAuthentication()`
- **Description**: At any point in the application, you can retrieve the currently authenticated user's details by accessing the `SecurityContextHolder`. This allows you to perform authorization checks, get user-specific data, etc.

```
Authentication authentication = SecurityContextHolder.getCo
ntext().getAuthentication();
```

debugging:

logging.level.org.springframework.security=DEBUG

`@EnableWebSecurity(debug = true)`

Managing users

In-Memory User Management

Before transitioning to database-backed user management, let's briefly recap how in-memory user management is typically configured:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.security.config.annotation.authe
ntication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.b
uilders.HttpSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDe
tails;
import org.springframework.security.core.userdetails.UserDe
tailsService;
```

```
import org.springframework.security.provisioning.InMemoryUs
erDetailsManager;
import org.springframework.security.web.SecurityFilterChai
n;

@Configuration
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager manager = new InMemoryUs
erDetailsManager();
        manager.createUser(User.withDefaultPasswordEncoder
()
            .username("user")
            .password("password")
            .roles("USER")
            .build());
        manager.createUser(User.withDefaultPasswordEncoder
()
            .username("admin")
            .password("admin")
            .roles("ADMIN")
            .build());
        return manager;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecu
rity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .formLogin(withDefaults());

        return http.build();
    }
```

```
}
```

2. Transition to Database-Backed User Management

To transition to database-backed user management, you'll typically follow these steps:

Step 1: Set Up Your Database

You need a database (e.g., MySQL, PostgreSQL, H2) to store user information. Ensure that your database is configured and that the necessary tables for storing user details and roles are created.

Example User Table Structure:

```
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(100) NOT NULL,
    enabled BOOLEAN NOT NULL
);

CREATE TABLE authorities (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    authority VARCHAR(50) NOT NULL,
    FOREIGN KEY (username) REFERENCES users(username)
);
```

- `users` **table**: Stores user details, including the username, password, and whether the user is enabled.
- `authorities` **table**: Stores the roles/authorities for each user.

Step 2: Configure a DataSource

You need to configure a `DataSource` bean to connect to your database. This typically involves setting up your `application.properties` or `application.yml` with

the necessary database connection details.

`application.properties` Example:

```
spring.datasource.url=jdbc:mysql://localhost:3306/yourdatab
ase
spring.datasource.username=yourusername
spring.datasource.password=yourpassword
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Drive
r
spring.jpa.hibernate.ddl-auto=update
```

Step 3: Implement `UserDetailsService`

You need to implement `UserDetailsService` to load user details from the database. This involves querying the `users` and `authorities` tables to build a `UserDetails` object.

```
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGr
antedAuthority;
import org.springframework.security.core.userdetails.UserDe
tails;
import org.springframework.security.core.userdetails.UserDe
tailsService;
import org.springframework.security.core.userdetails.Userna
meNotFoundException;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;
import java.util.ArrayList;
import java.util.List;

@Service
public class CustomUserDetailsService implements UserDetail
sService {
```

```
    private final DataSource dataSource;

    public CustomUserDetailsService(DataSource dataSource)
{
        this.dataSource = dataSource;
    }

    @Override
    public UserDetails loadUserByUsername(String username)
throws UsernameNotFoundException {
        // Load user from the 'users' table
        User user = findUserByUsername(username);

        if (user == null) {
            throw new UsernameNotFoundException("User not f
ound with username: " + username);
        }

        // Load user roles/authorities from the 'authoritie
s' table
        List<GrantedAuthority> authorities = findAuthoritie
sByUsername(username);

        return new org.springframework.security.core.userde
tails.User(
                user.getUsername(),
                user.getPassword(),
                user.isEnabled(),
                true, // accountNonExpired
                true, // credentialsNonExpired
                true, // accountNonLocked
                authorities
        );
    }

    private User findUserByUsername(String username) {
        // Implement query logic here using JDBC or ORM (e.
```

```
g., JPA)
        // Example: SELECT * FROM users WHERE username = ?
    }

    private List<GrantedAuthority> findAuthoritiesByUsernam
e(String username) {
        // Implement query logic here using JDBC or ORM (e.
g., JPA)
        // Example: SELECT authority FROM authorities WHERE
username = ?
        List<GrantedAuthority> authorities = new ArrayList<
>();
        // Example: authorities.add(new SimpleGrantedAuthor
ity("ROLE_USER"));
        return authorities;
    }
}
```

Step 4: Update Security Configuration

Update your security configuration to use your custom `UserDetailsService`:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.security.config.annotation.authe
ntication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.b
uilders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPas
swordEncoder;
import org.springframework.security.crypto.password.Passwor
dEncoder;
import org.springframework.security.web.SecurityFilterChai
n;

@Configuration
```

```
public class SecurityConfig {

    private final CustomUserDetailsService customUserDetail
sService;

    public SecurityConfig(CustomUserDetailsService customUs
erDetailsService) {
        this.customUserDetailsService = customUserDetailsSe
rvice;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecu
rity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .formLogin(withDefaults());

        return http.build();
    }

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

    @Bean
    public AuthenticationManager authenticationManager(Auth
enticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(customUserDetailsService)
            .passwordEncoder(passwordEncoder());
        return auth.build();
    }
}
```

Role management

Managing roles and authorization in Spring Security involves several steps to ensure that users have the correct access rights within your application. This includes defining roles, assigning roles to users, and enforcing role-based access control throughout your application.

1. Define Roles and Authorities

In Spring Security, roles and authorities are typically represented as `GrantedAuthority` objects. Roles usually start with a `ROLE_` prefix (e.g., `ROLE_USER`, `ROLE_ADMIN`), and these roles can be assigned to users.

2. Database Structure for Roles

You should have a database structure that supports roles and authorities. Here's an example structure:

```
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(100) NOT NULL,
    enabled BOOLEAN NOT NULL
);

CREATE TABLE roles (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE user_roles (
    user_id BIGINT NOT NULL,
    role_id BIGINT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (role_id) REFERENCES roles(id)
);
```

- `users` **table**: Stores basic user information.

- `roles` **table**: Stores the different roles available in the system.
- `user_roles` **table**: Links users to their assigned roles.

3. Implement `UserDetailsService` for Role Management

When implementing your custom `UserDetailsService`, you should load both user details and roles from the database.

Here's an updated example of `CustomUserDetailsService` that includes roles:

```
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGr
antedAuthority;
import org.springframework.security.core.userdetails.UserDe
tails;
import org.springframework.security.core.userdetails.UserDe
tailsService;
import org.springframework.security.core.userdetails.Userna
meNotFoundException;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;

@Service
public class CustomUserDetailsService implements UserDetail
sService {

    private final DataSource dataSource;

    public CustomUserDetailsService(DataSource dataSource)
{
        this.dataSource = dataSource;
    }
```

```
    @Override
    public UserDetails loadUserByUsername(String username)
throws UsernameNotFoundException {
        User user = findUserByUsername(username);

        if (user == null) {
            throw new UsernameNotFoundException("User not f
ound with username: " + username);
        }

        List<GrantedAuthority> authorities = findAuthoritie
sByUsername(user.getId());

        return new org.springframework.security.core.userde
tails.User(
                user.getUsername(),
                user.getPassword(),
                user.isEnabled(),
                true, // accountNonExpired
                true, // credentialsNonExpired
                true, // accountNonLocked
                authorities
        );
    }

    private User findUserByUsername(String username) {
        // Implement query logic here using JDBC or ORM (e.
g., JPA)
        // Example: SELECT * FROM users WHERE username = ?
    }

    private List<GrantedAuthority> findAuthoritiesByUsernam
e(Long userId) {
        List<GrantedAuthority> authorities = new ArrayList<
>();
        try (Connection connection = dataSource.getConnecti
on()) {
```

```
            PreparedStatement statement = connection.prepar
eStatement(
                "SELECT r.name FROM roles r INNER JOIN user
_roles ur ON r.id = ur.role_id WHERE ur.user_id = ?"
            );
            statement.setLong(1, userId);
            ResultSet rs = statement.executeQuery();

            while (rs.next()) {
                authorities.add(new SimpleGrantedAuthority
(rs.getString("name")));
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return authorities;
    }
}
```

4. Assign Roles to Users

You can assign roles to users by inserting records into the `user_roles` table. For example, if you want to assign the `ROLE_ADMIN` role to a user with `user_id` 1:

```
INSERT INTO user_roles (user_id, role_id)
VALUES (1, (SELECT id FROM roles WHERE name = 'ROLE_ADMI
N'));
```

5. Enforce Role-Based Access Control in Spring Security

You can enforce role-based access control using the following methods:

a) Securing URLs

You can specify which roles are allowed to access specific URLs in your `SecurityConfig`:

```
javaCopy code
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.security.config.annotation.web.b
uilders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChai
n;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecu
rity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .antMatchers("/admin/**").hasRole("ADMIN")
                .antMatchers("/user/**").hasAnyRole("USER",
"ADMIN")
                .antMatchers("/", "/home").permitAll()
                .anyRequest().authenticated()
            )
            .formLogin(withDefaults());

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

- `antMatchers("/admin/**").hasRole("ADMIN")`: Only users with the `ROLE_ADMIN` role can access URLs that start with `/admin/`.
- `antMatchers("/user/**").hasAnyRole("USER", "ADMIN")`: Users with either `ROLE_USER` or `ROLE_ADMIN` can access URLs that start with `/user/`.
- `permitAll()`: Allows access to anyone, whether authenticated or not.

b) Method-Level Security

You can also enforce role-based access at the method level using annotations:

- `@Secured` **Annotation**:

```
import org.springframework.security.access.annotation.Se
cured;

@Service
public class AdminService {

    @Secured("ROLE_ADMIN")
    public void performAdminTask() {
        // Only admins can perform this task
    }
}
```

- `@PreAuthorize` **and** `@PostAuthorize` **Annotations**:

```
import org.springframework.security.access.prepost.PreAu
thorize;

@Service
public class UserService {

    @PreAuthorize("hasRole('ADMIN') or hasRole('USER')")
    public void performUserTask() {
        // Both admins and users can perform this task
    }
}
```

To enable method-level security annotations, you need to add `@EnableGlobalMethodSecurity` in your security configuration:

```
import org.springframework.context.annotation.Configuratio
```

```
n;
import org.springframework.security.config.annotation.metho
d.configuration.EnableGlobalMethodSecurity;

@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true, prePostE
nabled = true)
public class MethodSecurityConfig {
    // Other configurations if needed
}
```