CS544 Enterprise Application Architecture

Lesson 10 – Spring Security

Securing Java Applications with Spring Framework

Payman Salek, M.S.

Original Material: Payman Salek – June 2024

SPRING SECURITY

Agenda

- Introduction to Spring Security
- Core Concepts
- Authentication and Authorization
- Configuration and Customization
- Advanced Features
- Best Practices
- Conclusion and Q&A

What is Spring Security?

- A powerful and customizable authentication and access control framework.
- Protects applications from various security threats.
- Seamlessly integrates with the Spring ecosystem.

Why Use Spring Security?

- Provides out-of-the-box security features.
- Simplifies complex security configurations.
- Works well with Spring Boot, Spring MVC, and other Spring projects.

Core Concepts

- Authentication: Verifying the identity of a user.
- Authorization: Determining what an authenticated user is allowed to do.
- Filters: Intercepts and processes HTTP requests.

Spring Security Architecture

- Components: Filters, Authentication Manager, Security Context, etc.
- Flow: Request -> Filter Chain -> Authentication -> Access Decision

Key Features

- Authentication: Supports various authentication mechanisms (form, basic, OAuth2).
- Authorization: Role-based, method-based, and URL-based access control.
- CSRF Protection: Prevents Cross-Site Request Forgery attacks.

Authentication Mechanisms

- Form-Based Authentication: Login form submission.
- Basic Authentication: HTTP Basic scheme.
- LDAP Authentication: Using LDAP for user authentication.
- OAuth2: Token-based authentication.

Authorization Methods

- Role-Based Access Control (RBAC): Access control based on user roles.

- Method Security: Securing methods with annotations like @PreAuthorize and @Secured.

- URL-Based Security: Securing URLs with antMatchers.

Configuration Basics

- Java Config vs XML Config: Modern approach with Java Config.
- Spring Boot Auto-Configuration: Simplifies configuration setup.

Example: Basic Configuration

- Java Configuration Example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin().loginPage("/login").permitAll();
    }
}
```

AuthenticationManager

- Manages the authentication process.
- Setup Example:

```
@Bean
public AuthenticationManager authManager() throws Exception
{
   return super.authenticationManagerBean();
}
```

Custom UserDetailsService

- Loads user-specific data.
- Example Implementation:

```
@Service
public class MyUserDetailsService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        return new User(username, "password", new ArrayList<>());
    }
}
```

Using JDBC for Authentication

- Configure DataSource, UserDetailsService.
- Example Configuration:

```
@Configuration
@EnableWebSecurity
public class JdbcSecurityConfig extends WebSecurityConfigurerAdapter {
   @Autowired
   private DataSource dataSource;

   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws
            Exception {
      auth.jdbcAuthentication().dataSource(dataSource);
   }
}
```

CSRF Protection

- Protects against CSRF attacks.
- Enable/Disable CSRF:

```
@Override
protected void configure(HttpSecurity http)
        throws Exception {
    http.csrf().disable(); // Disable for testing
}
```

Session Management

- Configuring Session Settings:

```
@Override
protected void configure(HttpSecurity http) throws
        Exception {
    http
        .sessionManagement()
                .sessionFixation().migrateSession()
                .maximumSessions(1);
}
```

OAuth2 Integration

- Supports OAuth2 for third-party logins.
- Example Dependencies:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

JWT Authentication

- JSON Web Tokens for stateless authentication.
- Setup Example:

```
@Configuration
@EnableWebSecurity
public class JwtSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private JwtAuthenticationProvider jwtAuthenticationProvider;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/api/**").authenticated()
            .anyRequest().permitAll()
            .and()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }
}
```

Advanced Configuration

- Custom Filters: Implementing custom security filters.
- Example Filter:

```
public class CustomAuthenticationFilter extends
        OncePerRequestFilter {
    @Override
    protected void doFilterInternal(HttpServletRequest request,
            HttpServletResponse response, FilterChain filterChain)
                    throws ServletException, IOException {
        // Custom authentication logic
        filterChain.doFilter(request, response);
    }
}
```

Method Security

- Annotations: @PreAuthorize, @Secured, @PostAuthorize.
- Example Use Case:

```
@PreAuthorize("hasRole('ADMIN')")
public void deleteUser(Long userId) {
    // Method implementation
}
```

Security Best Practices

- Keep Dependencies Updated
- Use HTTPS
- Regularly Review Access Controls
- Implement Logging and Monitoring

Testing Spring Security

- Tools: MockMvc, JUnit, Mockito.
- Example Test:

```
@RunWith(SpringRunner.class)
@WebMvcTest
public class SecurityTests {
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testLogin() throws Exception {
        mockMvc.perform(post("/login").param("username",
                "user").param("password", "password"))
                        .andExpect(status().isOk());
    }
}
```

Common Pitfalls

- Overlooking CSRF Protection
- Neglecting Secure Password Storage
- Improper Configuration of CORS (Cross-Origin Resource Sharing)
    - **Same-Origin Policy:** By default, web browsers enforce a same-origin policy, which restricts web pages from making requests to a different origin. This policy helps prevent cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks.
    - **Cross-Origin Requests:** Modern web applications often need to access APIs hosted on different domains, making CORS necessary to allow legitimate cross-origin requests.

Tools and Libraries

- Spring Security Test: For testing security configurations.
- Spring Security OAuth: For OAuth2 integration.
- Spring Security Kerberos: For Kerberos authentication.

Community and Support

- Spring Community: Forums, GitHub, and Spring.io.
- Documentation: Detailed guides and API docs on Spring Security.

Conclusion

- Spring Security:
  - Simplifies security implementation.
  - Offers robust authentication and authorization mechanisms.
  - Enhances application security with minimal setup.

Q&A

- Questions and Discussion:
- Open floor for any questions or clarifications.

References and Further Reading

- Books: "Spring Security in Action" by Laurentiu Spilca.
- Documentation: Spring Security Reference Guide.
- Online Resources: Official Spring Security GitHub repository, Tutorials on Baeldung.