# Spring Security – Java

O **Spring Security** é um framework robusto para garantir a segurança de aplicações Spring, oferecendo mecanismos de autenticação, autorização e proteção contra ataques comuns.

Com uma vasta gama de funcionalidades, ele é flexível o suficiente para atender às necessidades de segurança de aplicações empresariais, microserviços e APIs.

Neste artigo, vamos discutir as melhores práticas para configuração do Spring Security, considerando as versões LTS (Long Term Support) do **Java** e **Spring Boot**. Apresentaremos exemplos comentados para facilitar a compreensão e a aplicação dos conceitos em projetos reais.

## 1. EVersões: Java, Spring Boot e Spring Security

A compatibilidade entre as versões de **Java**, **Spring Boot** e **Spring Security** é crucial para o sucesso da implementação.

Para garantir que seu projeto esteja atualizado e estável, recomenda-se o uso de versões LTS, que oferecem suporte de longo prazo e manutenção estendida.

### 1.1. Versões LTS do Java

As versões LTS do Java garantem suporte estendido, recebendo atualizações de segurança e correções críticas. Aqui estão as versões LTS recomendadas:

- **Java 8** (LTS): Lançada em 2014, com suporte até 2030.
- **Java 11** (LTS): Lançada em 2018, com suporte até 2026.
- **Java 17** (LTS): Lançada em 2021, com suporte até 2029.

### 1.2. Versões do Spring Boot

O Spring Boot evolui constantemente para oferecer uma base sólida para o desenvolvimento de aplicações modernas. Para garantir compatibilidade, escolha uma versão do Spring Boot que suporte as versões LTS do Java:

- **Spring Boot 2.5+**: Suporte para Java 8 e 11.
- **Spring Boot 2.7+**: Suporte para Java 11 e 17.

### 1.3. Versões do Spring Security

Com base nas versões do Spring Boot e do Java, as versões recomendadas do **Spring Security** são:

- **Spring Boot 2.5 a 2.6**: Utilize o **Spring Security 5.5.x**.
- **Spring Boot 2.7+**: Utilize o **Spring Security 5.7.x** ou superior.

A combinação correta dessas versões garante a estabilidade e a segurança da aplicação, além de acesso contínuo a melhorias e correções de vulnerabilidades.

## 2. Configurando Spring Security:

A configuração do Spring Security é flexível e pode ser feita de forma declarativa ou programática.

Vamos explorar as formas recomendadas de configurar a segurança para diferentes cenários de aplicação.

### 2.1. Dependências no Maven e Gradle

Antes de configurar o Spring Security, é essencial adicionar as dependências corretas ao projeto. Abaixo estão as dependências mínimas necessárias:

Maven:

```
<dependencies>
    <!-- Dependência principal do Spring Security -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>

    <!-- Dependência para testes de segurança -->
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Gradle:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-security'
    testImplementation 'org.springframework.security:spring-security-test'
}
```

### 2.2. Configuração Programática

A configuração do Spring Security pode ser feita de maneira declarativa através do WebSecurityConfigurerAdapter (para versões até 5.7) ou por meio da API programática com SecurityFilterChain (a partir da versão 5.7+). A abordagem programática é mais moderna e oferece maior flexibilidade.

Exemplo com WebSecurityConfigurerAdapter (para Spring Security ≤ 5.7):

```
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.config.annotation.web.configuration.WebSecurityConfigurerAdapt
er;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                // Permite acesso público ao endpoint "/public"
                .antMatchers("/public").permitAll()
                // Exige autenticação para qualquer outro endpoint
                .anyRequest().authenticated()
            .and()
            // Configura login customizado
            .formLogin()
                .loginPage("/login")
                .permitAll()
            .and()
            // Configura logout
            .logout()
                .permitAll();
    }
}
```

Exemplo com SecurityFilterChain (para Spring Security ≥ 5.7):

```
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.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {
```

```
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    // Permite acesso público ao endpoint "/public"
                    .antMatchers("/public").permitAll()
                    // Exige autenticação para qualquer outro endpoint
                    .anyRequest().authenticated()
            )
            .formLogin(formLogin ->
                formLogin
                    // Configura página de login customizada
                    .loginPage("/login")
                    .permitAll()
            )
            .logout(logout ->
                logout.permitAll()
            );

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

### 2.3. Considerações sobre Proteção CSRF

Por padrão, o Spring Security habilita a proteção contra ataques **Cross-Site Request Forgery (CSRF)**. Essa proteção é essencial para aplicações que utilizam sessões e formulários web. No entanto, em APIs REST, onde a autenticação por tokens é comumente utilizada, essa proteção pode ser desativada.

```
http.csrf().disable();
```

Desabilitar o CSRF é uma prática comum em APIs, mas deve ser feito com cuidado, garantindo que outras medidas de segurança, como tokens JWT, estejam devidamente configuradas.

## 3. Autenticação com Banco de Dados: Integração com JDBC e JPA

Uma das funcionalidades mais poderosas do Spring Security é sua capacidade de integrar autenticação com bancos de dados. Isso permite a autenticação de usuários a partir de uma base de dados relacional.

### Exemplo: Usando JdbcUserDetailsManager com DataSource

Aqui está um exemplo simples de como integrar um banco de dados relacional usando o JdbcUserDetailsManager:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.JdbcUserDetailsManager;

import javax.sql.DataSource;

@Configuration
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService(DataSource dataSource) {
        return new JdbcUserDetailsManager(dataSource);
    }
}
```

Esse exemplo utiliza o DataSource configurado na aplicação para gerenciar a autenticação. O JdbcUserDetailsManager espera que as tabelas sigam um esquema padrão, mas pode ser customizado conforme necessário.

## 4. Testes de Segurança: Validação e Cobertura de Código

O **Spring Security Test** oferece ferramentas poderosas para testar a configuração de segurança, validando o comportamento da aplicação em cenários de autenticação e autorização. Abaixo estão exemplos de como testar a segurança de endpoints:

```
import static
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors
.httpBasic;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@Test
public void testUnauthorizedAccess() throws Exception {
    mockMvc.perform(get("/private"))
            .andExpect(status().isUnauthorized());
}

@Test
public void testAuthorizedAccess() throws Exception {
    mockMvc.perform(get("/private").with(httpBasic("user", "password")))
            .andExpect(status().isOk());
}
```

Os testes garantem que a configuração está funcionando conforme o esperado, simulando diferentes cenários de acesso (autorizado e não autorizado).

## Cconclusão

O **Spring Security** é um dos frameworks mais completos para garantir a segurança de aplicações Java. Seguir boas práticas como a escolha de versões LTS, configurar autenticação robusta, utilizar bancos de dados e realizar testes automatizados são fundamentais para criar soluções escaláveis e seguras.

Referências recomendadas para maior aprofundamento:

- Spring Security Documentation
- Spring Boot Security

***EducaCiência FastCode para a comunidade***