Spring Security

Autorzy:
Sylwia Rusek
Kinga Wrona
Klaudia Kromołowska
Marcin Teofil Poręba
Mateusz Sitek
Mateusz Krasiński
Robert Kozik
Adam Machaj

Czym jest Spring Security

Podprojekt Spring Security był pierwotnie osobnym projektem rozpoczętym w 2003 roku o nazwie Acegi Security System. Po 3 latach niezależnego rozwoju, został oficjalnie przyjęty w 2007 roku przez Spring i przemianowany na Spring Security.

Najważniejsze moduły:

- spring-security-core
- spring-security-config
- spring-security-web
- spring-security-test
- spring-security-acl
- spring-security-ldap

Włączenie obsługi Spring Security

Dla projektów wykorzystujących Gradle

```
implementation group: 'org.springframework.boot', name:
'spring-boot-starter-security'
```

Dla projektów wykorzystujących Mavena

```
<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-security</artifactId>

</dependency>
```

Konfigurowanie Spring Security

Na początku należy utworzyć podstawową klasę konfiguracyjną

```
import org.springframework.context.annotation.Bean;

import
org.springframework.security.config.annotation.web.configuration.EnableWe
bSecurity;

import
org.springframework.security.config.annotation.web.configuration.WebSecur
ityConfigurerAdapter;



@EnableWebSecurity

public class GatewaySecurityConfig extends WebSecurityConfigurerAdapter {

}
```

Adnotacja @EnableWebSecurity poinformuje Spring Security o użyciu tej klasy do konfiguracji.

Klasa rozszerza WebSecuirtyConfigurationAdapter, nadpisanie metod z tej klasy pozwala dostosować konfigurację zabezpieczeń.

Definiowanie użytkowników w magazynie danych w pamięci

```
@Override

@Bean

public UserDetailsService userDetailsService() {

InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

UserDetails adminUser = User

         .withUsername("admin")

         .password(encoder().encode("admin123"))

         .authorities("FULL_PRIVILEGES")

         .roles("ADMIN")

         .build();

manager.createUser(adminUser);

return manager;

}
```

Metoda userDetailsService() konfiguruje magazyn użytkowników w pamięci z jednym użytkownikiem. Ten użytkownik otrzymuje nazwę użytkownika **admin**, hasło **admin123** i rolę **ADMIN**.

Przechowujemy zakodowane hasło i identyfikujemy je, udostępniając koder w metodzie kodera, który używa **BCrypt**.

Konfiguracja zabezpieczeń

```
@Override

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

	.antMatchers("/home").permitAll()

	.antMatchers("/dashboard").authenticated()

	.and()

	.formLogin();

}
```

Metoda configure(HttpSecurity) określa, które ścieżki URL powinny być zabezpieczone, a które nie.

Ścieżka “/home” skonfigurowana jest tak, aby nie wymagała żadnego uwierzytelnienia.

Ścieżka “/dashboard” skonfigurowana jest tak aby wymagała uwierzytelnienia.

Opcję konfiguracji

Metoda	Opis
and	Przydatne do łączenia metod, ponieważ zwracany jest objeckt SecurityBuilder
anonymous	Umożliwia uzyskanie dostępu użytkownikom anonimowym
authenticated	Umożliwia uzyskanie dostępu zweryfikowanym użytkownikom
authorizeRequests	Ogranicza dostęp na podstawie HTTPServletRequest
antMatchers	Zezwala lub ogranicza dostęp, używane w połączeniu z wywołaniem authorizeRequests
formLogin	Zapewnia, że używamy logowania za pomocą formularza
httpBasic	Wykorzystuje prostą metodę uwierzytelniania HTTP Basic
hasRole	Zezwala na dostęp jeśli użytkownik ma określoną rolę
permitAll	Bezwarunkowo umożliwia uzyskanie dostępu
loginPage	Określa adres URL strony logowania, jeśli wymagane jest logowanie.
rememberMe	Umożliwia uzyskanie dostępu użytkownikom, którzy zostali uwierzytelnieni za pomocą funkcji “zapamiętaj mnie”

Dostosuj swoje bezpieczeństwo

Spring Security umożliwia dużą konfigurowalność funkcji.

Jedną z nich jest np. definiowanie własnych tras logowania i wylogowywania. Możemy to zrobić w bardzo łatwy sposób - za pomocą aktualizacji naszej obecnej konfiguracji. W ten sposób możemy szybko zdefiniować niestandardową stronę logowania, która za pomocą loginu i odpowiedniego url będzie przekierowywać do odpowiedniej zakładki.

Zabezpieczenie Aplikacji REST

```
@Configuration
public class GatewaySecurityConfig extends WebSecurityConfigurerAdapter {

   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws
   Exception {
      auth.inMemoryAuthentication()
         .passwordEncoder(NoOpPasswordEncoder.getInstance())
         .withUser("user").password("user123")
         .roles("USER").and()
         .withUser("admin").password("admin123")
         .roles("USER", "ADMIN");

   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http
       .httpBasic()
       .and()
       .authorizeRequests()
       .antMatchers("/songs/**")
       .hasRole("USER")
       .antMatchers("/**")
       .hasRole("ADMIN")
       .and()
       .csrf().disable()
       .headers().frameOptions().disable();
}}
```

To jest REST nie boimy się cross-site request forgery

Wyłączamy X-frame header umożliwiając osadzenie w iFrame

Nowości w Spring Security 5

OAuth 2.0 Login

- otwarty standard autoryzacji dostępu do strony internetowej lub aplikacji bez podawania haseł
- powszechnie używany przez Google, Facebook, Twitter i wiele innych
- do skorzystania z spring-security-oauth2 w klasie trzeba uwzględnić elementy:
  - compile group: 'org.springframework.security',
  - name: 'spring-securityoauth2-client',
  - version:

Konfiguracja loginy Google i Facebook za pomocą Spring

- poświadczenie klienta dla każdego z nich
- Spring Boot skonfiguruje przekierowanie identyfikatora URI w następującym formacie
  - /login/oauth2/code/{registrationId},
- Plik application.properties
  - spring.security.oauth2.client.registration.google.client-id= *<your client id>*
  - spring.security.oauth2.client.registration.google.client-secret= *<your client secret>*
  - spring.security.oauth2.client.registration.facebook.client-id= *<your client id>*
  - spring.security.oauth2.client.registration.facebook.client-secret=*<your client secret>*

Podstawowa konfiguracja

```
@Configuration
public class OAuthSecurityConfig extends WebSecurityConfigurerAdapter {
        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http.authorizeRequests()
                .anyRequest().authenticated()
                .and()
                .oauth2Login();
        }
}
```

Metody dostępne w oauth2Login()

- loginPage("/my_login")
- defaultSuccessUrl()
- failureUrl()
- successHandler()
- failureHandler()

Wsparcie reaktywne

- Asynchroniczne strumienie danych
- Różnią się od tradycyjnych strumieni
- Wzrost wydajności

DelegatingPasswordEncoder
Domyślny enkoder

- {bcrypt} zaszyfrowane_hasło
- PasswordEncoder passwordEncoder =PasswordEncoderFactories.createDelegatingPasswordEncoder();
- klucze rozszyfrowujące: ldap, MD4, MD5, noop, pbkdf2, scrypt, SHA-1, SHA-256, sha256

Implementacja customowego DelegatingPasswordEncoder

```
@SuppressWarnings("deprecation")

static PasswordEncoder createDelegatingPasswordEncoder() {
	String idForEncode = "bcrypt";
	PasswordEncoder defaultEncoder = NoOpPasswordEncoder.getInstance();
	Map<String, PasswordEncoder> encoders = new HashMap<>();
	encoders.put("bcrypt", new BCryptPasswordEncoder());
	encoders.put("noop", defaultEncoder);
	encoders.put("SHA-256", new MessageDigestPasswordEncoder("SHA-256"));
	DelegatingPasswordEncoder delegatingPasswordEncoder =new
		DelegatingPasswordEncoder(idForEncode, encoders);
	delegatingPasswordEncoder.setDefaultPasswordEncoderForMatches (defaultEncoder);

	return delegatingPasswordEncoder;
}
```

DelegatingPasswordEncoder sposób użycia

```
@Bean
public PasswordEncoder passwordEncoder() {

return DefaultPasswordEncoderFactories.createDelegatingPasswordEncoder();

}
```

Dziękujemy za uwagę!