



Bookstore



Model View Controller

2.View Generuje interfejs użytkownika. Widoki są tworzone na podstawie danych zbieranych przez komponent modelu, ale dane te nie są pobierane bezpośrednio, ale przekazywane za pośrednictwem kontrolera.

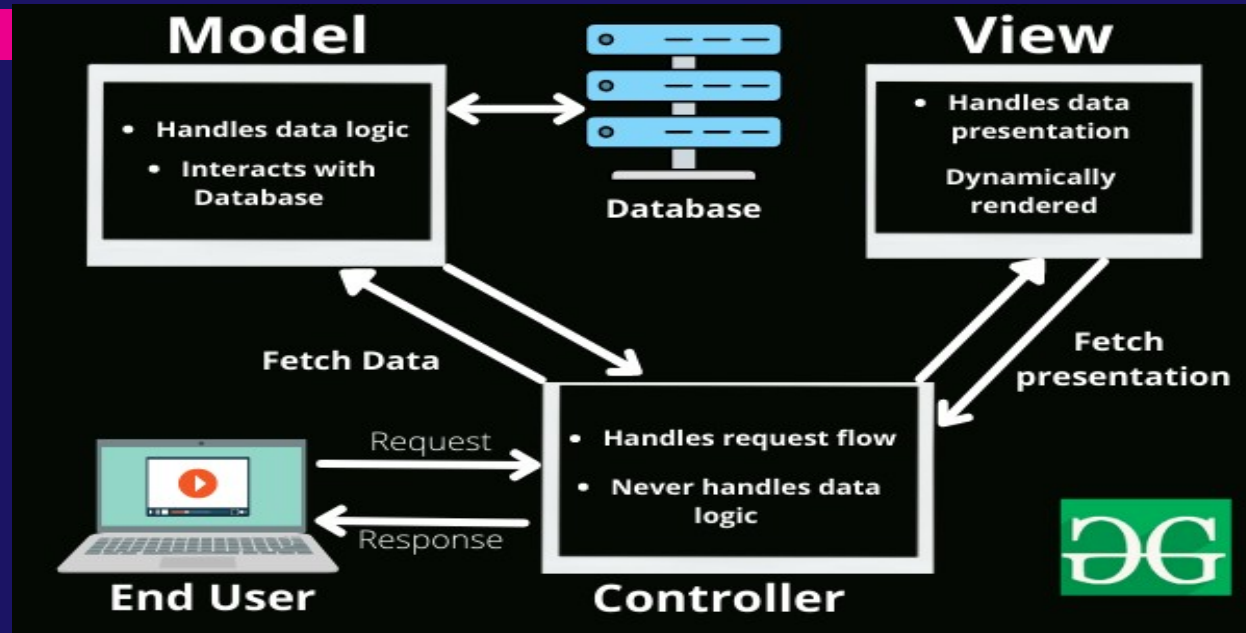
1.Model odpowiada całej logice związanej z danymi, z którą pracuje użytkownik.

Może reprezentować dane przesyłane pomiędzy komponentami View i Controller lub dowolne inne dane związane z logiką biznesową.

Może dodawać lub pobierać dane z bazy danych.

3.Kontroler jest komponentem umożliwiającym połączenie widoków z modelem, pełni więc rolę pośrednika. Kontroler nie musi się martwić obsługą logiki danych, po prostu mówi modelowi, co ma robić. Przetwarza całą logikę biznesową i przychodzące żądania, manipuluje danymi za pomocą komponentu Model i wchodzi w interakcję z widokiem w celu wyrenderowania końcowego wyniku.

Model View Controller



Model View Controller

Aby stworzyć prostą aplikację MVC w Spring boot wystarczą nam 2 pliki klas – BookstoreApp, HomeController, oraz plik html ze stroną internetową, która ma zostać wygenerowana.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.1.5</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

Model View Controller

Strony umieszczamy w katalogu projektu w:
src/main/resources/templates/home.html

Przy tworzeniu stron użyjemy Thymeleaf, jest to silnik szablonów. W przypadku prostej strony do testu nie użyjemy jeszcze funkcjonalności thymeleafa.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Strona główna</title>
</head>
<body>
<h1>Witaj na stronie głównej!</h1>
<p>Jesteś zalogowany!</p>
</body>
</html>
```

Model View Controller

Kontrolery będziemy umieszczać w katalogu z kodem:

src/main/java/...

W katalogu o nazwie controller. Np:

src/main/java/com/umcspro/controller/
HomeController.java

```
package com.umcspro.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {
    @GetMapping({"/home", "/"})
    public String home() {
        return "home";
    }
}
```

Model View Controller

```
package com.umcspro;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

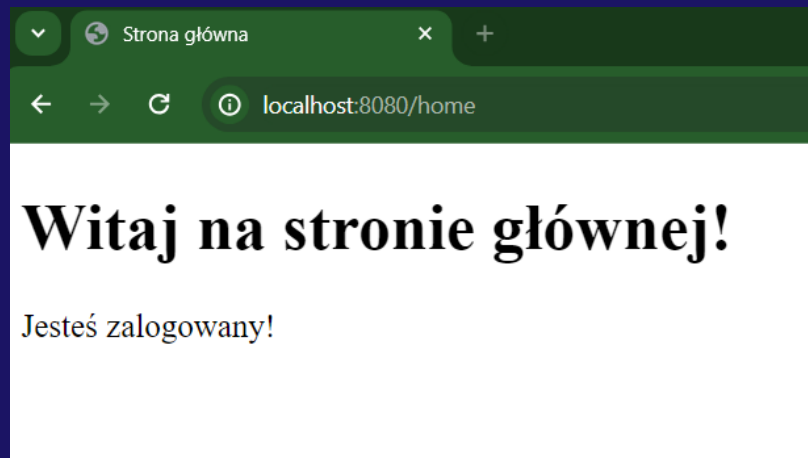
@SpringBootApplication
public class BookstoreApp {
    public static void main(String[] args) {
        SpringApplication.run(BookstoreApp.class, args);
    }
}
```

Kod odpalający aplikację SpringBoot wygląda tak jak w poprzednim projekcie. Znajduje się w katalogu projektu np:

src/main/com/umcspro/BookstoreApp.java

Po uruchomieniu i wpisaniu adresu w przeglądarce localhost:8080/home

powiniśmy uzyskać widok prostej strony internetowej.



Logowanie

Do zrealizowania logowania i rejestracji będziemy potrzebowali następujących modułów:

- Spring Web
- Spring Security
- Thymeleaf
- Spring Data JPA
- PostgreSQL/MySQL Driver



Logowanie

Struktura projektu i zależności

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
  </dependency>

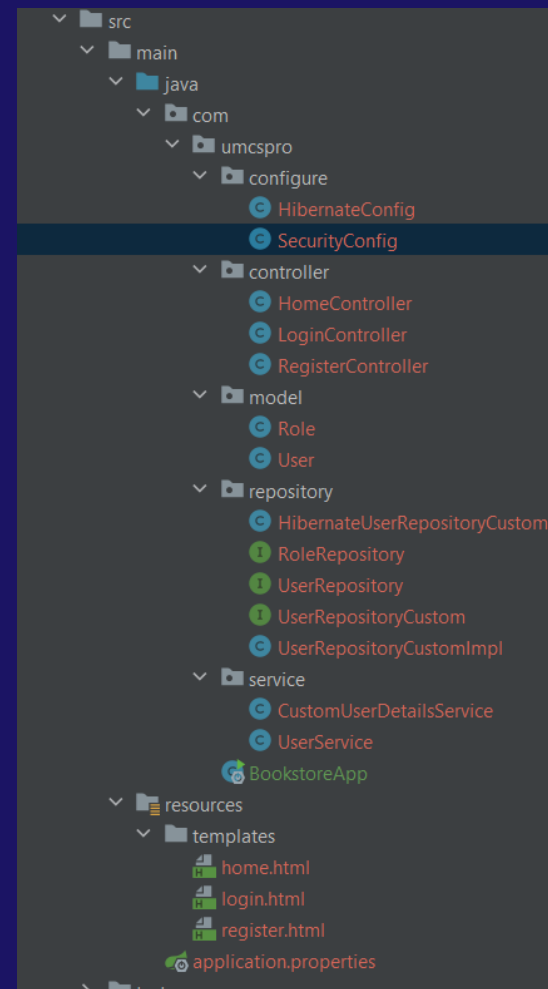
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>

  <dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
  </dependency>
</dependencies>
```



Logowanie

Spring Initializr

start.spring.io

Project

☐ Gradle - Groovy

☐ Gradle - Kotlin

☒ Maven

Language

☒ Java

☐ Kotlin

☐ Groovy

Spring Boot

☐ 3.3.0 (SNAPSHOT)

☐ 3.3.0 (RC1)

☐ 3.2.6 (SNAPSHOT)

☒ 3.2.5

☐ 3.1.12 (SNAPSHOT)

☐ 3.1.11

Project Metadata

Group

com.example

Artifact

demo

Name

demo

Description

Demo project for Spring Boot

Package name

com.example.demo

Packaging

☒ Jar

☐ War

Java

☐ 22

☐ 21

☒ 17

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf

TEMPLATE ENGINES

A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Spring Security

SECURITY

Highly customizable authentication and access-control framework for Spring applications.

PostgreSQL Driver

SQL

A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

Spring Data JPA

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...

The image shows the Spring logo, which is a stylized green leaf. The leaf is composed of two main parts: a larger, rounded upper section and a smaller, pointed lower section, both in a vibrant green color. The leaf is set against a plain white background.

A solid black rectangular box used to redact information, likely a username or password, from the login form.

Logowanie

Spring Security jest frameworkiem zabezpieczeń, który zarządza:

- procesem uwierzytelniania, czyli procesem weryfikacji tożsamości użytkownika.
- autoryzacją, czyli procesem decydowania, czy dany użytkownik ma dostęp do zasobu, który próbuje uzyskać.
- ochroną przed atakami takimi jak
Cross-Site Scripting (XSS),
Cross-Site Request Forgery (CSRF), SessionFixation



Logowanie

Po dodaniu zależności SpringSecurity umożliwienie logowania jest bardzo proste, musimy napisać:

- Klasy Encji dla Usera i Roli
- implementację UserDetailsService,
- interfejsy, które zostaną zaimplementowane przez Spring Data JPA: UserRepository, RoleRepository
- Bean do hashowania hasła.

-w katalogu resources – plik z ustawieniami naszej aplikacji: application.properties



Logowanie

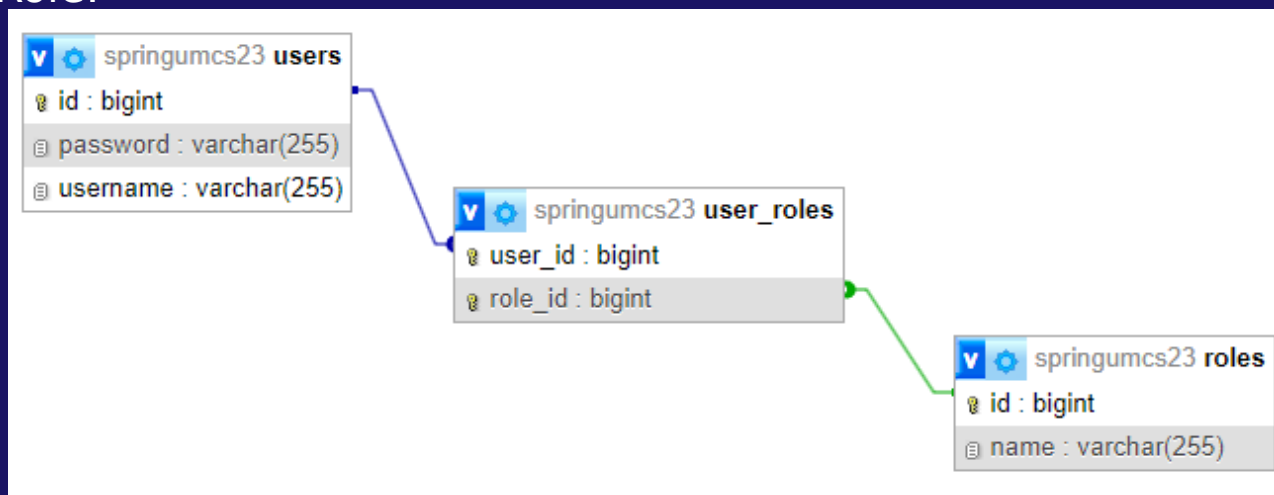
application.properties

```
spring.datasource.url=jdbc:mysql://...  
spring.datasource.username=...  
spring.datasource.password=...  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.show-sql=true
```



Logowanie

User i Role:



Logowanie

User i Role:

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Nazwa użytkownika jest wymagana")
    private String username;
    @NotBlank(message = "Hasło jest wymagane")
    private String password;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id")
    )
    private Set<Role> roles = new HashSet<>();
    //...
}
```

```
@Entity
@Table(name = "roles")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    //...
}
```

Logowanie



User i Role:

-prosta walidacja pól Usera w formularzu będzie używała:

```
<dependency>  
  <groupId>org.hibernate.validator</groupId>  
  <artifactId>hibernate-validator</artifactId>  
</dependency>
```


Logowanie

```
public interface RoleRepository extends JpaRepository<Role, Long> {  
    Optional<Role> findByName(String name);  
}
```

Repozytoria:

Spring Data JPA korzysta z mechanizmu zwane "Repository Proxies" do automatycznego generowania implementacji repozytorium w czasie wykonywania aplikacji.

```
public interface UserRepository extends JpaRepository<User, Long> {  
    Optional<User> findByUsername(String username);  
    User save(User user);  
}
```

W przykładzie, UserRepository dziedziczy standardowe metody CRUD oraz dodaje niestandardową metodę findByUsername, która również jest automatycznie implementowana przez Spring Data JPA na podstawie konwencji nazewnictwa.

Logowanie

UserDetails:

UserDetailsService jest prostym interfejsem, który ma tylko jedną metodę:

```
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
```

Metoda ta służy do ładowania obiektu UserDetails, jest używany przez Spring Security do celów takich jak weryfikacja danych uwierzytniających użytkownika, przypisywanie ról czy uprawnień.

```
@Service
public class CustomUserDetailsService implements UserDetailsService
{
    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException(
                "User not found with username: " + username));
        return new org.springframework.security.core.userdetails.User(user.getUsername(),
            user.getPassword(),
            mapRolesToAuthorities(user.getRoles()));
    }

    private Collection<? extends GrantedAuthority> mapRolesToAuthorities(Collection<Role> roles){
        return roles.stream()
            .map(role -> new SimpleGrantedAuthority(role.getName()))
            .collect(Collectors.toList());
    }
}
```

Logowanie

Aby móc się zalogować należy dodać użytkownika w bazie danych, rolę, oraz przypisać rolę do użytkownika:

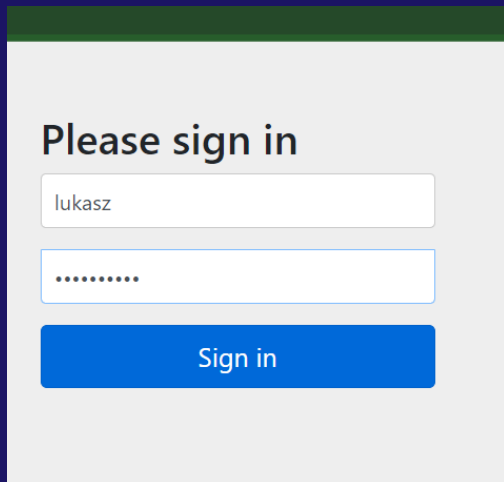
		id	password	username
<input type="checkbox"/>	Edit Copy Delete	1	\$2a\$10\$cTm9gLRzwtQ7u6RIj/yvuWKOpmBMzCTuYtXeX3msf9...	lukasz
<input type="checkbox"/>	Edit Copy Delete	4	\$2a\$10\$TvnuShxgFoO7V4xGWI4umOf6/Ad8A8WsTsiGl.PFUzo...	nowy

		id	name
<input type="checkbox"/>	Edit Copy Delete	1	USER
<input type="checkbox"/>	Edit Copy Delete	2	ADMIN

		user_id	role_id
<input type="checkbox"/>	Edit Copy Delete	4	1
<input type="checkbox"/>	Edit Copy Delete	1	2

Logowanie

SpringSecurity sam tworzy stronę do logowania, oraz kontroler.

A screenshot of the Spring Security default login page. It features a light gray background with a dark green header bar at the top. The main content area is white and contains the text "Please sign in" in a bold, dark font. Below this text are two input fields: the first is for the username, containing the text "lukasz", and the second is for the password, filled with dots. At the bottom of the form is a blue button with the text "Sign in" in white.

Bean do hashowania hasła:

```
@Configuration
public class SecurityConfig {

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

Logowanie



Kolejnymi elementami w projekcie będą:
własna strona do logowania,
zabezpieczenie endpointów,
oraz obsługa wylogowania.

Logowanie

Strona do logowania:

Do dynamicznego tworzenia stron użyjemy podstawowych dyrektyw thymeleaf:

Warunkowe wyświetlenie elementu gdy istnieje/nie jest pusty(będzie przekazany przez kontroler w modelu):

```
<div th:if="${errorMessage}" th:text="${errorMessage}"></div>
```

Ustawienie gdzie mają być wysłane dane formularza:

```
<form th:action="@{/login}" method="post">
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Login</title>
</head>
<body>
<h1>Logowanie</h1>
<div th:if="${errorMessage}" th:text="${errorMessage}"></div>
<div th:if="${logoutMessage}" th:text="${logoutMessage}"></div>
<form th:action="@{/login}" method="post">
  <div>
    <label for="username">Nazwa użytkownika:</label>
    <input type="text" id="username" name="username" required />
  </div>
  <div>
    <label for="password">Hasło:</label>
    <input type="password" id="password" name="password" required />
  </div>
  <div>
    <button type="submit">Zaloguj się</button>
  </div>
</form>
</body>
</html>
```

Logowanie

Kontroler dla logowania:

Parametry metody z adnotacjami `@RequestParam` - te parametry służą do przechwytywania dodatkowych danych przekazywanych w URL. Parametry `error` i `logout` są opcjonalne.

Atrybut `error` pojawia się wtedy, gdy proces logowania nie powiedzie się.

Logout pojawia się, gdy użytkownik wyloguje się z aplikacji.

```
@Controller
public class LoginController {

    @GetMapping("/login")
    public String login(
        @RequestParam(value = "error", required = false) String error,
        @RequestParam(value = "logout", required = false) String logout,
        Model model) {
        if (error != null) {
            model.addAttribute("errorMessage", "Nieprawidłowa nazwa
użytkownika lub hasło!");
        }
        if (logout != null) {
            model.addAttribute("logoutMessage", "Pomyślnie
wylogowano!");
        }
        return "login";
    }
}
```

Logowanie

Zabezpieczenie endpointów:

Kiedy żądanie HTTP przychodzi do aplikacji, przechodzi przez serię filtrów zdefiniowanych w SecurityFilterChain.

Konfiguracja Autoryzacji (authorizeHttpRequests):

authorizeHttpRequests, przekazując obiekt anonimowej klasy implementującej Customizer<AuthorizeHttpRequestsConfigurer<HttpSecurity>.AuthorizationManagerRequestMatcherRegistry>

W tej klasie, metoda customize definiuje zasady autoryzacji.

Konfiguracja Logowania (formLogin): konfiguracja logowania przez formularz jest realizowana poprzez przekazanie anonimowej klasy implementującej Customizer<FormLoginConfigurer<HttpSecurity>>

Metoda customize pozwala na dostosowanie procesu logowania.

Konfiguracja Wylogowania (logout): Używa anonimowej klasy implementującej Customizer<LogoutConfigurer<HttpSecurity>>

metoda customize pozwala na dostosowanie procesu wylogowania.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/login", "/register").permitAll()
                .anyRequest().authenticated()
            )
            .formLogin(form -> form
                .loginPage("/login")
                .defaultSuccessUrl("/home", true)
                .permitAll()
            )
            .logout(logout -> logout.permitAll());
        return http.build();
    }

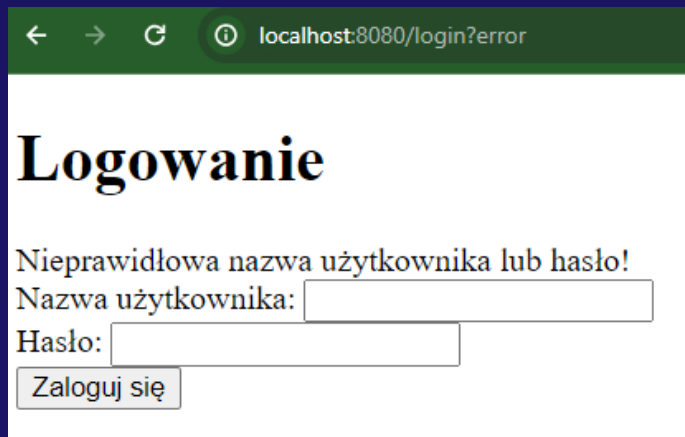
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```


Logowanie

Przykład anonimowej klasy implementującej interfejs Customizera:

```
http
    .authorizeHttpRequests(
        new Customizer<AuthorizeHttpRequestsConfigurer<HttpSecurity>.AuthorizationManagerRequestMatcherRegistry>(){
            @Override
            public void customize(AuthorizeHttpRequestsConfigurer<HttpSecurity>.
AuthorizationManagerRequestMatcherRegistry authorizationManagerRequestMatcherRegistry) {
                authorizationManagerRequestMatcherRegistry
                    .requestMatchers("/login", "/register").permitAll()
                    .anyRequest().authenticated();
            }
        }
    )
```

Logowanie



A screenshot of a web browser window showing a login page. The address bar displays 'localhost:8080/login?error'. The page title is 'Logowanie'. Below the title, there is an error message: 'Nieprawidłowa nazwa użytkownika lub hasło!'. This is followed by two input fields: 'Nazwa użytkownika:' and 'Hasło:'. At the bottom, there is a button labeled 'Zaloguj się'.

← → ↻ ⓘ localhost:8080/login?error

Logowanie

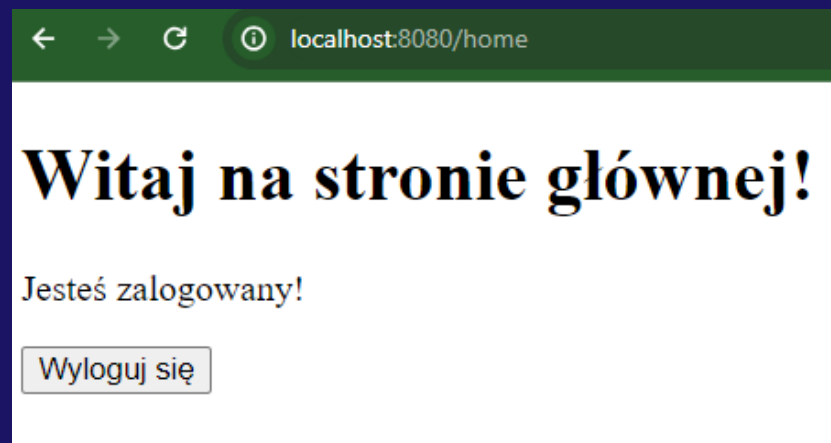
Nieprawidłowa nazwa użytkownika lub hasło!

Nazwa użytkownika:

Hasło:

Zaloguj się

VS



A screenshot of a web browser window showing a home page. The address bar displays 'localhost:8080/home'. The page title is 'Witaj na stronie głównej!'. Below the title, there is a message: 'Jesteś zalogowany!'. At the bottom, there is a button labeled 'Wyloguj się'.

← → ↻ ⓘ localhost:8080/home

Witaj na stronie głównej!

Jesteś zalogowany!

Wyloguj się

Logowanie

Generowanie sesji:

Spring Security generuje unikalny identyfikator sesji dla użytkownika.

Identyfikator sesji jest zapisywany jako wartość pliku cookie JSESSIONID i wysyłany do przeglądarki użytkownika.

Przechowywanie sesji: Serwer Spring Security przechowuje informacje o sesji użytkownika, w tym związane z nią dane, na serwerze.

Dzięki temu, kiedy użytkownik wykonuje kolejne żądania do aplikacji, serwer jest w stanie zidentyfikować sesję użytkownika na podstawie JSESSIONID.

Dane o zalogowanym użytkowniku oraz jego rolach są przechowywane w obiekcie Authentication, który jest również przechowywany w kontekście sesji użytkownika. Obiekt Authentication zawiera obiekt UserDetails.

Witaj na stronie głównej!

Jesteś zalogowany!

Wyloguj się

Wszystkie zakładki

Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder

Application

Filter

JSESSIONID

06854071C6181F7F5DDDE30BD334836F

localhost

/

Session

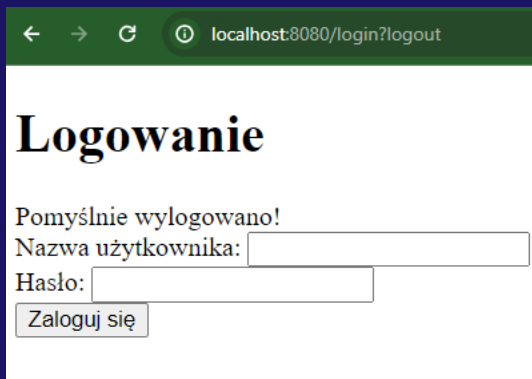
42

✓

Medium

Logowanie

Wylogowanie



← → ↻ ⓘ localhost:8080/login?logout

Logowanie

Pomyślnie wylogowano!

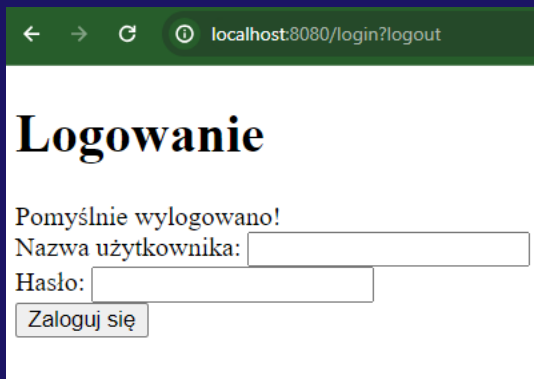
Nazwa użytkownika:

Hasło:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Strona główna</title>
</head>
<body>
<h1>Witaj na stronie głównej!</h1>
<p>Jesteś zalogowany!</p>
<form th:action="@{/logout}" method="post">
  <button type="submit">Wyloguj się</button>
</form>
</body>
</html>
```

Logowanie

Wylogowanie



← → ↻ ⓘ localhost:8080/login?logout

Logowanie

Pomyślnie wylogowano!

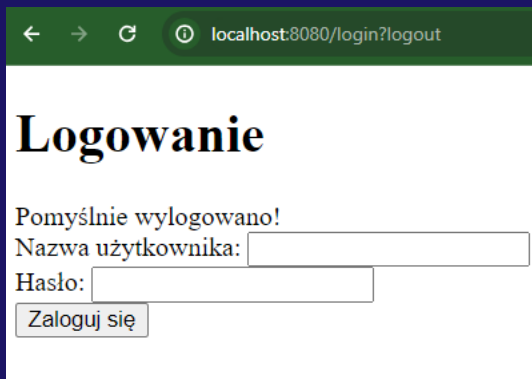
Nazwa użytkownika:

Hasło:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Strona główna</title>
</head>
<body>
<h1>Witaj na stronie głównej!</h1>
<p>Jesteś zalogowany!</p>
<form th:action="@{/logout}" method="post">
  <button type="submit">Wyloguj się</button>
</form>
</body>
</html>
```

Logowanie

Wylogowanie



← → ↻ ⓘ localhost:8080/login?logout

Logowanie

Pomyślnie wylogowano!

Nazwa użytkownika:

Hasło:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Strona główna</title>
</head>
<body>
<h1>Witaj na stronie głównej!</h1>
<p>Jesteś zalogowany!</p>
<form th:action="@{/logout}" method="post">
  <button type="submit">Wyloguj się</button>
</form>
</body>
</html>
```

Logowanie

Dokładniejsze wyjaśnienie procesu uwierzytelniania w Spring Security:

- 1.Żądanie logowania: np. poprzez przesłanie danych (nazwa użytkownika i hasło).
- 2.Filtracja żądania: Żądanie logowania jest przekazywane przez filtr Spring Security, który jest odpowiedzialny za obsługę żądań związanych z bezpieczeństwem.
- 3.AuthenticationManager: Filtr Spring Security używa AuthenticationManager, aby uwierzytelnić użytkownika na podstawie dostarczonych danych.
AuthenticationManager może zawierać wiele różnych AuthenticationProviderów, z których każdy obsługuje inny sposób uwierzytelniania (na przykład uwierzytelnianie na podstawie nazwy użytkownika i hasła, uwierzytelnianie tokenem)
- 4.AuthenticationProvider: W trakcie uwierzytelniania, AuthenticationManager przekazuje żądanie uwierzytelnienia do każdego AuthenticationProvidera w kolejności, aż któryś z nich będzie w stanie pomyślnie uwierzytelnić użytkownika. Każdy AuthenticationProvider jest odpowiedzialny za sprawdzenie poprawności dostarczonych danych uwierzytelniających.
- 5.AuthenticationProvider sprawdza dane: otrzymuje obiekt Authentication (z credentialami) i np. sprawdza poprzez porównanie hasła zapisanego w bazie danych z hasłem podanym przez użytkownika.
Jeśli dane uwierzytelniające są poprawne, AuthenticationProvider tworzy nowy obiekt Authentication, który reprezentuje uwierzytelnionego użytkownika (principal - UserDetails). Ten obiekt zawiera informacje o użytkowniku (np. nazwa użytkownika, lista ról, itp.) oraz jego statusie uwierzytelnienia.
- 6.SecurityContext: Poprawnie uwierzytelniony obiekt Authentication jest umieszczany w SecurityContext, który jest przechowywany w sesji użytkownika. Dzięki temu informacje o uwierzytelnionym użytkowniku są dostępne w całej sesji.
- 7.Logowanie udane: Użytkownik zostaje przekierowany na stronę docelową lub otrzymuje potwierdzenie sukcesu logowania.

Logowanie

