

Laboratorium 6. *Spring Security* i *Thymeleaf*

Cel zajęć

Realizacja zadań z niniejszego laboratorium umożliwi studentom poznanie zasad konfiguracji modułu *Spring Security* [5] w celu uwierzytelniania użytkowników aplikacji internetowej za pomocą loginu i hasła, z wykorzystaniem technologii widoków *Thymeleaf* [23] oraz bazy danych *MySQL*.

Zakres tematyczny

- Utworzenie projektu aplikacji internetowej *Spring Boot* za pomocą kreatora *Spring Initializr*.
- Konfiguracja *Spring Security* w celu implementacji klasycznego uwierzytelniania użytkownika za pomocą loginu i hasła, przechowywanych w bazie danych *MySQL*.
- Wykorzystanie technologii widoków *Thymeleaf* w celu utworzenia i obsługi własnego formularza rejestracji i logowania.
- Implementacja operacji typu *CRUD* dla danych użytkownika.
- Walidacja danych z formularza rejestracyjnego.

Wprowadzenie

Strukturę projektu *Spring Boot* można bardzo łatwo utworzyć, korzystając z darmowego, dostępnego online (<https://start.spring.io>) narzędzia *Spring Initializr*. Narzędzie to oferuje prosty graficzny interfejs użytkownika, który pozwala wskazać m.in. wersję języka *Java*, określić koordynaty projektu, wybrać z listy odpowiednie zależności. Wygenerowany przez *Initializr* gotowy projekt można po rozpakowaniu, otworzyć w wybranym *IDE* i dalej dowolnie go modyfikować i rozbudowywać.

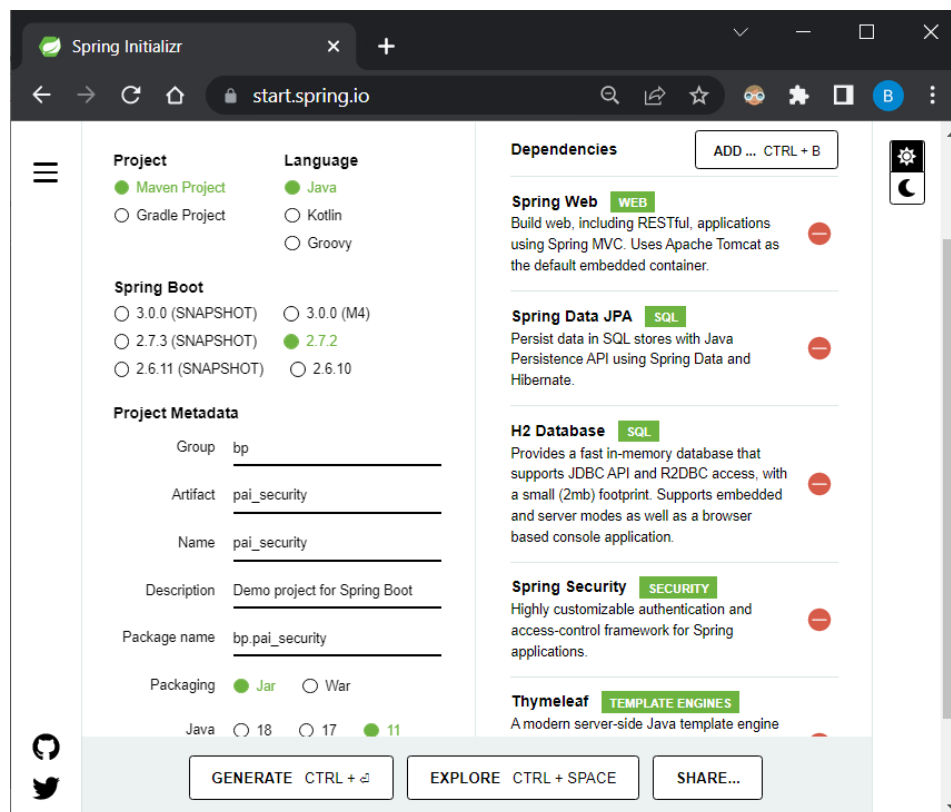
Dodanie do pliku *pom.xml* zależności *Spring Security*, powoduje, że *Spring Boot* automatycznie zabezpiecza wszystkie punkty docelowe *HTTP* (ang. *endpoints*) za pomocą podstawowej autentykacji (ang. *Basic Authentication*). Taki podstawowy mechanizm zastosowano w zadaniu 5.6. Dodatkowe zabezpieczenia można dodać za pomocą definicji własnej klasy, poprzedzonej specjalną adnotacją *@EnableWebSecurity*, która dziedziczy po klasie *WebSecurityConfigurerAdapter*. Metody tej klasy umożliwiają dodatkową specyfikację zabezpieczeń. *Spring* nie narzuca żadnej technologii do tworzenia widoków **po stronie serwera**. Można korzystać ze standardu *JSP* (jak w poprzednich laboratoriach), jednak nowocześniejszym rozwiązaniem, wykorzystywanym w połączeniu z projektami *Spring* jest silnik szablonów

Thymeleaf [23]. *Thymeleaf* umożliwia budowę elementów interfejsu użytkownika **po stronie serwera**, definiuje bogaty zestaw funkcjonalności, dostępnych w kodzie dokumentu *HTML* za pośrednictwem charakterystycznego przedrostka *th*.

Zadanie 6.1. Utworzenie projektu w *Spring Initializr*

Do utworzenia projektu *Maven*, skorzystaj z narzędzia *Spring Initializr*, które jest dostępne na stronie <https://start.spring.io> (Rys. 6.1):

1. Wskaż, że nowy projekt ma być generowany jako *Maven*.
2. Podaj nazwę pakietu głównego dla nowej aplikacji (w przykładzie *bp*).
3. Podaj nazwę artefaktu – nazwę aplikacji (w przykładzie *pai_security*). Zwróć też uwagę na wersję Javy (w tym przykładzie wybrano v11).
4. Wybierz zależności (wpisując ich nazwy w przeznaczonym do tego okienku), które zostaną dodane do pliku *pom.xml*. Wskaż: *Spring Web*, *Spring Data JPA*, *H2 Database*, *Spring Security* i *Thymeleaf*.
5. Wygeneruj projekt (przycisk *Generate*).



Rys. 6.1. Okno *Spring Initializr* z metadanymi i dodanymi zależnościami

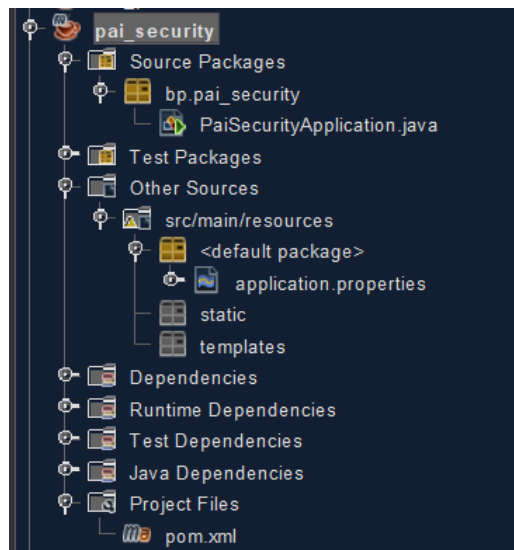
Zadanie 6.2. Dodatkowe elementy konfiguracji projektu

Rozpakuj wygenerowane archiwum **zip** oraz:

1. Otwórz i zbuduj projekt w swoim *IDE*. *IDE* pobierze wszystkie zależności wskazane w **pom.xml** i postara się zbudować projekt. Po zbudowaniu projektu, jego struktura powinna wyglądać jak na rysunku 6.2.
2. Przejrzyj zawartość plików **pom.xml** i **application.properties** oraz utworzonej już klasy startowej (w przykładzie jest to klasa o nazwie **PaiSecurityApplication.java**).

W pliku **pom.xml** warto jeszcze dodać jeden wiersz wskazujący klasę startową projektu. Tak jak na laboratorium 5, do istniejącego już elementu **<properties>** dodaj wiersz:

```
<start-class>bp.pai_security.PaiSecurityApplication</start-class>
```



Rys. 6.2. Struktura plików w projekcie

W celu skonfigurowania połączenia aplikacji z bazą danych *H2*, w pliku **application.properties**, dodaj parametry serwera i połączenia z bazą danych:

```
#wybór portu na którym pracuje nasz serwer HTTP
server.port = 8080
#konfiguracja połączenia z bazą danych
spring.datasource.url = jdbc:h2:mem:paiapp
#pring.datasource.url=jdbc:h2:file:./h2db
spring.datasource.username = admin
```

```
spring.datasource.password = admin
spring.jpa.database-platform = org.hibernate.dialect.H2Dialect
spring.datasource.driverClassName = org.h2.Driver
#wyświetlanie w dzienniku serwera wszystkich poleceń SQL:
spring.jpa.show-sql = true
#aktualizacja struktury bazy danych przy starcie aplikacji:
spring.jpa.hibernate.ddl-auto = update
```

Uruchom projekt i przetestuj jego działanie, wpisując w przeglądarce adres, np. **localhost:8080/login**. Przeglądarka, przed udostępnieniem aplikacji, wyświetla domyślny formularz logowania (jak w zadaniu 5.6), co wynika z domyślnej konfiguracji *Spring Security*.

Zadanie 6.3. Konfiguracja *Spring Security*

W celu przygotowania własnej konfiguracji *Spring Security* należy zdefiniować klasę dziedziczącą po adapterze ***WebSecurityConfigurerAdapter*** i odpowiednio nadpisać jego metody ***configure()***. Kolejny etap to przygotowanie własnego formularza logowania z zastosowaniem szablonu *Thymeleaf*.

6.3.1. Klasa *Security* i jej metody *configure()*

W istniejącym pakiecie *pai_security* utwórz nowy pakiet o nazwie ***configuration*** (dla klas konfiguracyjnych *Spring Security*) a w nim klasę ***Security*** (Rys. 6.3).



Rys. 6.3. Pakiet *configuration* z klasą *Security*

Klasa ***Security*** jako klasa konfiguracyjna, wymaga dodatkowej adnotacji ***@Configuration***. Aby wyłączyć domyślną konfigurację zabezpieczeń aplikacji internetowej, dodana zostanie adnotacja ***@EnableWebSecurity*** (nie wyłącza to konfiguracji menedżera uwierzytelniania). Aby skonfigurować własne zabezpieczenia, zwykle używa się właściwości i metod gotowej klasy ***WebSecurityConfigurerAdapter***. Własna obsługa logowania w oparciu o login i hasło przechowywane w bazie danych, wymaga przygotowania specjalnej klasy konfiguracyjnej. Klasa taka powinna dziedziczyć po klasie ***WebSecurityConfigurerAdapter*** (Przykład 6.1). Do tworzonej klasy

konfiguracyjnej należy dodać deklarację („wstrzyknąć”) obiektu klasy **UserAuthenticationDetails** (ta klasa zostanie zdefiniowana nieco później w tym samym pakiecie) – Przykład 6.4, Rys. 6.4.

Przykład 6.1. Klasa Security

```
package bp.pai_security.configuration;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.authentication.builders.A
uthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecu
rityConfigurerAdapter;
import
org.springframework.security.config.annotation.web.configuration.EnableW
ebSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
```

@Configuration

@EnableWebSecurity

```
public class Security extends WebSecurityConfigurerAdapter {
```

@Autowired

```
private UserAuthenticationDetails userAuthenticationDetails;
```

@Override

```
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
    auth.userDetailsService(userAuthenticationDetails);
    auth.authenticationProvider(authenticationProvider());
}
```

@Bean

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

@Bean

```
public DaoAuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider authenticationProvider =
        new DaoAuthenticationProvider();
    authenticationProvider.setUserDetailsService(
        userAuthenticationDetails);
    authenticationProvider.setPasswordEncoder(passwordEncoder());
}
```

```

        return authenticationProvider;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic().disable()
            .csrf().disable()
            .authorizeRequests()
            .antMatchers("/register").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login")
            .usernameParameter("login")
            .passwordParameter("passwd")
            .defaultSuccessUrl("/profile", true)
            .permitAll()
            .and()
            .logout()
            .logoutUrl("/logout")
            .logoutSuccessUrl("/login")
            .invalidateHttpSession(true);
    }
}

```

W klasie *Security* należy zwrócić szczególną uwagę na dwie metody *configure*:

- a) Pierwsza z nich ustawia klasę odpowiadającą za uwierzytelnianie użytkowników aplikacji na podstawie danych znajdujących się w bazie danych.
- b) Druga metoda:
 - wyłącza podstawowe, proste uwierzytelnienie użytkowników (*httpBasic*),
 - ustawia autoryzację użytkowników i pozwala określić, w jaki sposób adresy *URL* będą interpretowane przez środowisko (w aplikacji dostęp do strony rejestracji */register* będzie dostępny dla wszystkich użytkowników – *permitAll*),
 - ustawia podstawowe informacje dotyczące formularza logowania i jego parametrów,
 - ustawia podstawową stronę domową, na którą przekierowany jest użytkownik po pomyślnym logowaniu (*/profile*),
 - ustawia adres */logout*, pod którym użytkownicy mogą wylogować się z systemu i wyczyścić sesję *HTTP*.

Metody oznaczone adnotacją *@Bean* pozwalają na utworzenie instancji klasy, która może zostać później „wstrzyknięta” i wykorzystywana w różnych miejscach aplikacji.

6.3.2. Klasa encji *User*

Do skonfigurowania własnych zabezpieczeń potrzebna będzie klasa encji *User*.

Utwórz pakiet *entity* i dodaj do niego klasę *User*, która będzie reprezentować użytkownika zapisanego w bazie danych (Rys. 6.4). Zdefiniuj pola klasy *User* (Przykład 6.2), a metody *get* i *set* wygeneruj automatycznie w IDE.

Przykład 6.2. Klasa *User*

```
package bp.pai_security.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "Users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer userid;
    private String name;
    private String surname;
    private String login;
    private String password;

    public User() {
    }

    public User(String name, String surname, String login,
                String password) {
        this.name = name;
        this.surname = surname;
        this.login = login;
        this.password = password;
    }
    //metody get i set
}
```

6.3.3. Interfejs *UserDao*

Kolejnym krokiem jest implementacja interfejsu dostępu do danych.

Utwórz pakiet *dao* (Rys. 6.4) z interfejsem *UserDao* (Przykład 6.3), w którym zdefiniuj tylko jedną metodę *findByLogin()*.

Przykład 6.3. Interfejs UserDao

```
package bp.pai_security.dao;

import bp.pai_security.entity.User;
import org.springframework.data.repository.CrudRepository;

public interface UserDao extends CrudRepository<User, Integer> {
    public User findByLogin(String login);
}
```

6.3.4. Klasa UserAuthenticationDetails

Po utworzeniu warstwy dostępu do danych, w pakiecie *configuration* utwórz (wspomnianą wcześniej) klasę *UserAuthenticationDetails* (Rys. 6.4). Klasa ta powinna implementować interfejs *UserDetailsService* z jedną metodą *loadUserByUsername()* (Przykład 6.4).

Przykład 6.4. Klasa UserAuthenticationDetails

```
package bp.pai_security.configuration;

import bp.pai_security.dao.UserDao;
import bp.pai_security.entity.User;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Component;

@Component
public class UserAuthenticationDetails implements UserDetailsService {

    @Autowired
    private UserDao dao;

    @Override
    public UserDetails loadUserByUsername(String login)
        throws UsernameNotFoundException {
        User user = dao.findByLogin(login);
        if (user != null) {
            List<GrantedAuthority> grupa = new ArrayList<>();
            grupa.add(new SimpleGrantedAuthority("normalUser"));
            return new
```



```
        org.springframework.security.core.userdetails.User(
            user.getLogin(), user.getPassword(),
            true, true, true, true, grupa);
    } else {
        throw
            new UsernameNotFoundException("Zły login lub hasło.");
    }
}
}
```

Działanie klasy *UserAuthenticationDetails* umożliwia skorzystanie z logowania do środowiska *Spring Security*. Klasa posiada tylko jedną metodę *loadUserByUsername*, której parametrem jest login użytkownika. Metoda wywoływana jest w trakcie logowania użytkownika, po wpisaniu i zaakceptowaniu przez niego danych w formularzu logowania. Zadaniem w tym przypadku jest pobranie z bazy danych użytkownika o wskazanym loginie. W przypadku, gdy nie ma takiego użytkownika, wyrzucany jest wyjątek ze stosowną informacją. W ten sposób na podstawie połączenia do bazy danych realizowane jest logowanie do systemu przy pomocy klas zawartych w pakiecie **.configuration*.

Po przygotowaniu klas i interfejsów z punktów 6.3.1–6.3.4, konfiguracja *Spring Security* jest już gotowa.

Potrzebny jest jeszcze kontroler i widoki do obsługi logowania.

Zadanie 6.4. Kontroler i widoki *Thymeleaf*

Utwórz pakiet *controllers* i klasę kontrolera *UserController* (Przykład 6.5, Rys. 6.4). Jest to ostatnia klasa potrzebna do uruchomienia aplikacji. Foldery pakietów z klasami przedstawia rysunek 6.4.

Przykład 6.5. Klasa *UserController*

```
package bp.pai_security.controllers;

import bp.pai_security.dao.UserDao;
import bp.pai_security.entity.User;
import java.security.Principal;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
```

```

@Controller
public class UserController {
    @Autowired
    private PasswordEncoder passwordEncoder;
    @Autowired
    private UserDao dao;
    @GetMapping("/login")
    public String loginPage() {
        //zwrócenie nazwy widoku logowania - login.html
        return "login";
    }
    @GetMapping("/register")
    public String registerPage(Model m) {
        //dodanie do modelu nowego użytkownika
        m.addAttribute("user", new User());
        //zwrócenie nazwy widoku rejestracji - register.html
        return "register";
    }
    @PostMapping("/register")
    public String registerPagePOST(@ModelAttribute User user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        dao.save(user);
        //przekierowanie do adresu url: /login
        return "redirect:/login";
    }
    @GetMapping("/profile")
    public String profilePage(Model m, Principal principal) {
        //dodanie do modelu aktualnie zalogowanego użytkownika:
        m.addAttribute("user", dao.findByLogin(principal.getName()));
        //zwrócenie nazwy widoku profilu użytkownika - profile.html
        return "profile";
    }
    // @GetMapping("/users")
    //definicja metody, która zwróci do widoku users.html listę
    //użytkowników z bd
}

```

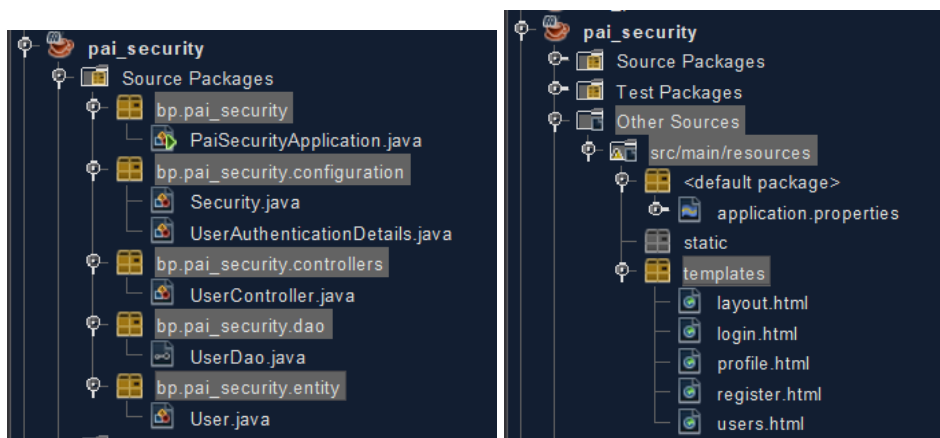
Do klasy kontrolera „wstrzyknięto” dwa ziarna: *PasswordEncoder* i *UserDao*. Każda z metod kontrolera obsługuje inny adres *URL* określony przy pomocy adnotacji *@GetMapping* lub *@PostMapping*. Po utworzeniu klasy kontrolera brakuje już tylko plików odpowiedzialnych za widoki.

Pliki widoków przygotowane będą z zastosowaniem szablonów *Thymeleaf*. W celu zminimalizowania kodu i ujednolicenia wyglądu aplikacji wykorzystany zostanie plik z szablonem strony *layout.html* (Przykład 6.6), ze wspólnymi elementami każdej ze stron widoku (nagłówek, nawigacja i stopka).

Pliki widoków powinny być umieszczone (domyślna lokalizacja) w folderze *src/main/resources/templates* (Rys. 6.4).

Do folderu *templates* dodaj pięć plików z widokami (Przykłady 6.6–6.9):

- *layout.html* (szablon),
- *login.html*, *profile.html*, *register.html* (pliki obsługujące rejestrację i logowanie),
- *users.html* (do przygotowania samodzielnie).



Rys. 6.4. Struktura pakietów, klas i plików widoków w aplikacji *PaiSecurityApplication*

Przykład 6.6. Plik szablonu strony – *layuot.html*

```
<!DOCTYPE html>
<html lang="pl" xmlns:th="http://www.thymeleaf.org">
  <head th:fragment="head">
    <meta charset="utf-8" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1"/>
    <title>Spring login application - PL</title>
    <style>
      .footer {
        position: fixed; bottom: 0px; left: 1px;
        height: 50px; width: 100%;
      }
      .nav > li {
        display: inline;
        background: limegreen;
      }
      a { text-decoration: none; }
      th,td {border: solid 1px black;padding: 5px;}
    </style>
  </head>
```

```

<body>
  <div th:fragment="navigationPanel" >
    <ul class="nav">
      <li sec:authorize="!isAuthenticated()">
        <a th:href="@{/register}">Zarejestruj się</a></li>
      <li sec:authorize="isAuthenticated()">
        <a th:href="@{/profile}">Wyświetl swoje konto</a>
      </li>
      <li sec:authorize="isAuthenticated()">
        <a th:href="@{/users}">Wyświetl użytkowników</a></li>
      <li sec:authorize="!isAuthenticated()">
        <a th:href="@{/login}">Zaloguj się</a></li>
      <li sec:authorize="isAuthenticated()">
        <a th:href="@{/logout}">Wyloguj się</a></li>
    </ul>
  </div>
  <div th:fragment="footer">
    <footer class="footer"><hr/>
    <p>Politechnika Lubelska, wprowadzenie do Spring Boot</p>
  </div>
</body>
</html>

```

W pliku *layout.html* warto zwrócić szczególną uwagę na fragmenty:

- **th:fragment="head"** – definiuje fragment strony z definicją nagłówka dokumentu *HTML*,
- **th:fragment="navigationPanel"** – definiuje element nawigacji,
- **sec:authorize** – pozwala określić widoczność danego elementu *HTML* w zależności od tego, czy użytkownik jest zalogowany (jeśli nie, to element nie jest wyświetlany),
- **th:href** – pozwala wskazać ścieżkę za pomocą bezwzględnego adresu *URL* poprzez *@{url}* lub względnego *URL* w kontekście naszej aplikacji: *@{/login}*,
- **th:fragment="footer"** – definiuje fragment kodu ze stopką strony.

Plik *login.html* (Przykład 6.7) i każdy z kolejnych plików korzysta z szablonu strony *layout.html*. W celu dołączenia wybranych fragmentów zdefiniowanych w szablonie *layout.html* do innego pliku należy skorzystać atrybut *th:include* i określić nazwę pliku oraz nazwę sekcji, oddzielonych podwójnym dwukropkiem, np. `<head th:include="layout :: head">`

Przykład 6.7. Plik *login.html*

```

<!DOCTYPE html>
<html lang="pl" xmlns:th="http://www.thymeleaf.org">

```

```

<head th:include="layout :: head">
</head>
<body>
  <div th:include="layout :: navigationPanel"></div>
  <h1>Logowanie do systemu:</h1>
  <form th:action="@{/login}" method="POST">
    <input type="text" name="login" placeholder="Login"/>
    <input type="password" name="passwd" placeholder="Hasło"/>
    <button type="submit">Zaloguj się</button>
  </form>
  <div th:include="layout :: footer"></div>
</body>
</html>

```

Plik *profile.html* przedstawia przykład 6.8.

Przykład 6.8. Plik *profile.html*

```

<!DOCTYPE html>
<html lang="pl" xmlns:th="http://www.thymeleaf.org">
  <head th:include="layout :: head"> </head>
  <body>
    <div th:include="layout :: navigationPanel"></div>
    <h1 th:text="'Witaj ' + ${user.name} + ' ' + ${user.surname}"></h1>
    <div th:include="layout :: footer"></div>
  </body>
</html>

```

Plik widoku *register.html* (Przykład 6.9) odpowiada za rejestrację nowych użytkowników. W tym przypadku z wartości pobranych z pól formularza, dołączonych do ciała żądania (*Request Body*), w metodzie kontrolera obsługującej to żądanie, tworzony jest obiekt klasy *User*. W tym celu zastosowano atrybuty *th:object* (jako atrybut dla elementu *form*) i *th:field* (atrybut pola *input*). Atrybut *th:object* odpowiada za nazwę obiektu, który później odbierany jest przez kontroler aplikacji z wykorzystaniem adnotacji *@ModelAttribute(value = "user")* *User user* i mapowany przy pomocy metod *set* na obiekt klasy *User*.

Przykład 6.9. Plik *register.html*

```

<!DOCTYPE html>
<html lang="pl" xmlns:th="http://www.thymeleaf.org">
  <head th:include="layout :: head"> </head>
  <body>
    <div th:include="layout :: navigationPanel"></div>
    <h1>Rejestracja:</h1>
    <form th:action="@{/register}" method="POST" th:object="${user}">
      <input type="text" th:field="*{name}" placeholder="Imię"/>
      <input type="text" th:field="*{surname}" placeholder="Nazwisko"/>
    </form>
  </body>
</html>

```

```

        <input type="text" th:field="*{login}" placeholder="Login"/>
        <input type="password" th:field="*{password}" placeholder="Hasło"/>
        <button type="submit">Zarejestruj się</button>
    </form>
    <div th:include="layout :: footer"></div>
</body>
</html>

```

Widok *users.html* opracuj samodzielnie (na początku dodaj tylko szablon, a resztę kodu dodaj dopiero po przetestowaniu logowania i rejestracji).

Aby można było przetestować działanie aplikacji, na początek dodaj dwóch użytkowników do bazy danych przy pomocy specjalnej metody *init()* w **klasie głównej**. Metoda *init()* z adnotacją *@PostConstruct*, wywoływana jest tuż po uruchomieniu aplikacji. Do klasy startowej projektu (*PaiSecurityApplication*) dodaj kod z przykładu 6.10.

Przykład 6.10. Główna klasa aplikacji (z metodą main)

```

package bp.pai_security;
import bp.pai_security.dao.UserDao;
import bp.pai_security.entity.User;
import javax.annotation.PostConstruct;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.security.crypto.password.PasswordEncoder;

@SpringBootApplication
public class PaiSecurityApplication {
    @Autowired
    private UserDao dao;
    @Autowired
    private PasswordEncoder passwordEncoder;

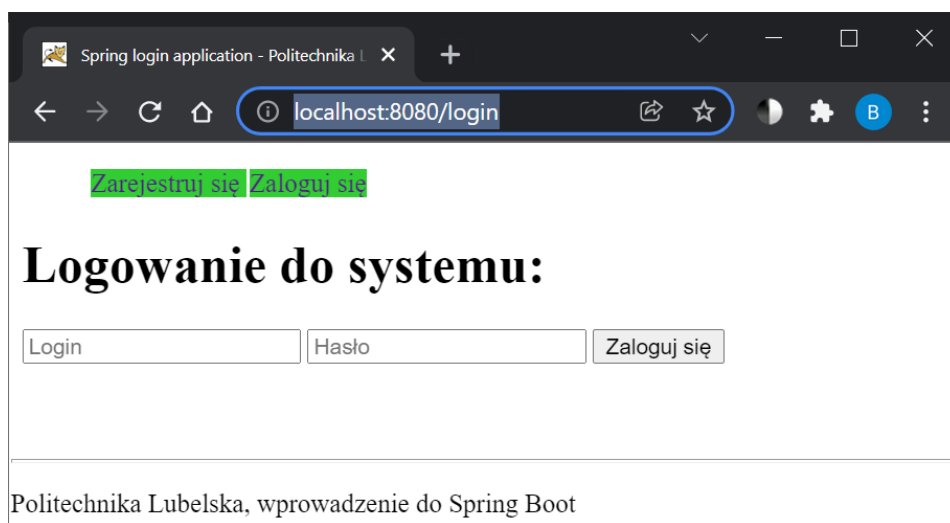
    public static void main(String[] args) {
        SpringApplication.run(PaiSecurityApplication.class, args);
    }

    @PostConstruct
    public void init() {
        dao.save(new User("Piotr", "Piotrowski", "admin",
            passwordEncoder.encode("admin")));
        dao.save(new User("Ania", "Annowska", "ania",
            passwordEncoder.encode("ania")));
    }
}

```

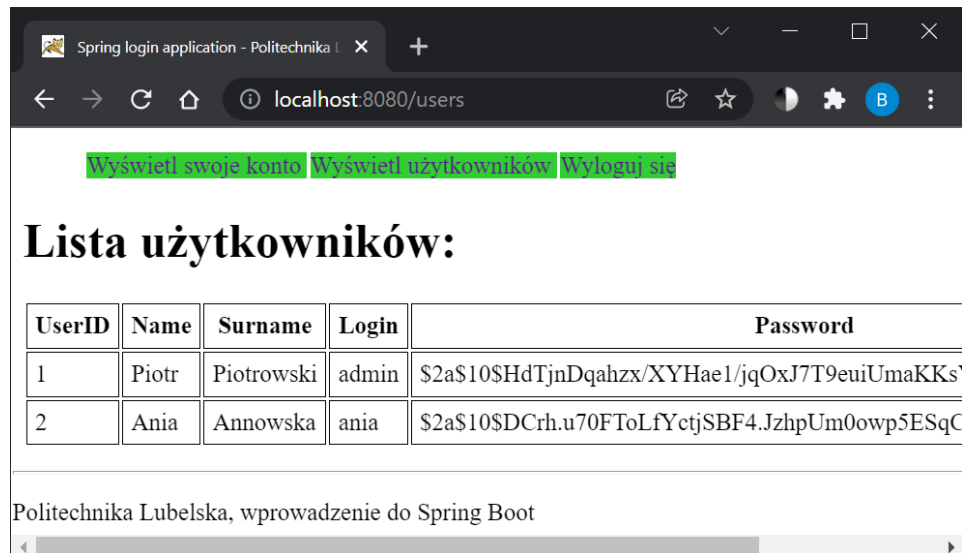
W metodzie *init()* utworzono pierwszych użytkowników za pomocą obiektu *dao* i *passwordEncoder* (wykorzystany do haszowania hasła). Pierwszy użytkownik posiada dane logowania: **login: admin, hasło: admin**. Dla drugiego użytkownika ustawiono **login: ania, hasło: ania**.

Uruchom aplikację – przeglądarka wyświetli widok z pliku *login.html* jak na rysunku 6.5. Zaloguj się na dane utworzonego w metodzie *init()* użytkownika lub dokonaj rejestracji i logowania nowego użytkownika.



Rys. 6.5. Formularz logowania – *login.html*

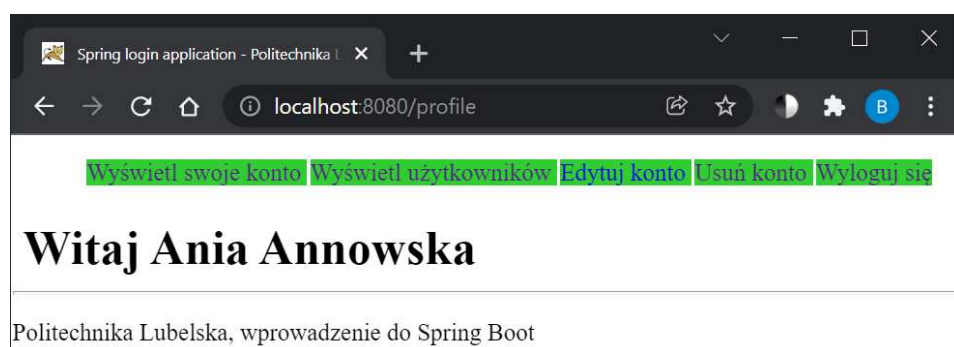
Po prawidłowym zalogowaniu (i uzupełnieniu pliku widoku *users.html*) wyświetli listę wszystkich użytkowników (Rys. 6.6).



Rys. 6.6. Widok listy wszystkich użytkowników

Zadanie 6.5. Usuwanie i edycja danych użytkownika

Do kontrolera *UserController* dodaj metody do obsługi usuwania i edycji danych aktualnie zalogowanego użytkownika (skorzystaj z obiektu klasy *Principal*). Zmodyfikuj w tym celu także plik *layout.html* (dodaj przyciski do usuwania i edycji w elemencie nawigacji – Rys. 6.7). Gdy użytkownik będzie chciał usunąć swoje konto, należy go również wylogować (przekierować na akcję */logout*). W przypadku edycji danych pamiętaj o konieczności haszowania hasła.



Rys. 6.7. Dodatkowe przyciski w nawigacji widoczne dla zalogowanego użytkownika

Zadanie 6.6. Walidacja danych z formularza rejestracji

Uzupełnij pola w klasie encji *User* wybranymi adnotacjami do walidacji (np. *@Size*, *@Pattern*, *@NotNull*) oraz odpowiednio zmodyfikuj plik *register.html* i metodę w kontrolerze obsługującą rejestrację nowego użytkownika tak, aby sprawdzana była poprawność wprowadzonych danych. Przykład 6.11 pokazuje właściwe wykorzystanie adnotacji *@Valid* oraz obiektu *BindingResult* w parametrach metody kontrolera.

UWAGA! Parametr klasy *BindingResult* MUSI być umieszczony bezpośrednio po parametrze z adnotacją *@Valid*. Nieprzestrzeganie tej reguły powoduje wyrzucenie wyjątku: *“java.lang.IllegalStateException: An Errors/BindingResult argument is expected to be declared immediately after the model attribute, the @RequestBody or the @RequestPart arguments to which they apply: ...”*

Aby skorzystać z adnotacji do walidacji danych w klasie encji, należy dodać do pliku *pom.xml* zależność:

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

Zależność tę można też dodać w *Spring Initializr* na etapie tworzenia projektu *Spring Boot* (Rys. 6.8).

Przykład 6.11. Przykładowa akcja kontrolera z walidacją

```
@PostMapping("/register")
public String registerPagePOST(@Valid User user,
                               BindingResult binding) {
    if (binding.hasErrors()) {
        return "register"; //powrót do rejestracji
    }
    //dalsze akcje wykonywane dla prawidłowych danych
    return "login";
}
```

Dependencies

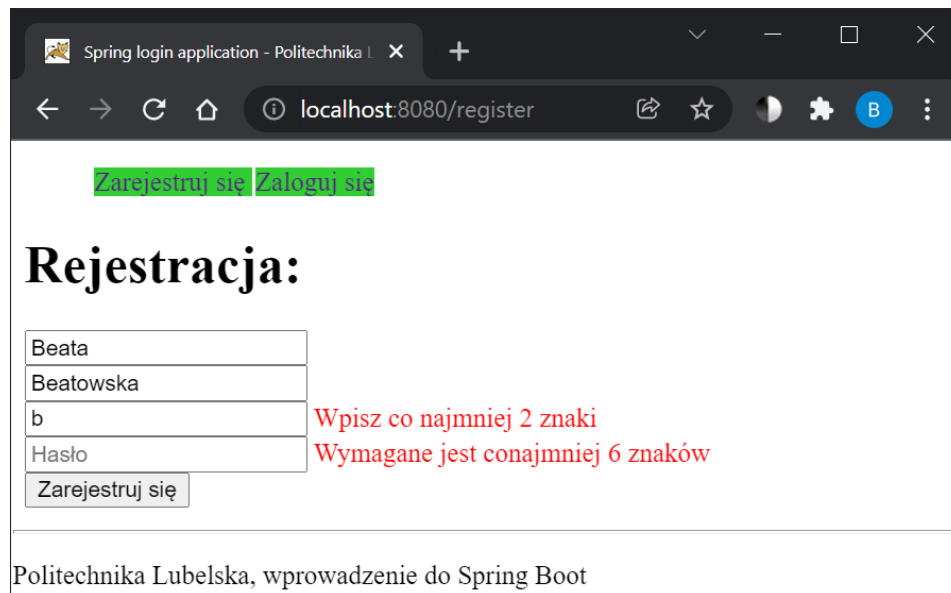
ADD DEPENDENCIES... CTRL + B

Validation 

JSR-303 validation with Hibernate validator.

Rys. 6.8. Dodatkowa zależność do walidacji w *Initializr*

Przykładowy efekt walidacji danych rejestracyjnych przedstawia rysunek 6.9.



The screenshot shows a web browser window with the title "Spring login application - Politechnika L". The address bar displays "localhost:8080/register". The page has a dark theme. At the top, there are two buttons: "Zarejestruj się" and "Zaloguj się". Below them is a large heading "Rejestracja:". The registration form consists of four input fields: "Imię" (containing "Beata"), "Nazwisko" (containing "Beatowska"), "Login" (containing "b"), and "Hasło" (containing "b"). To the right of the "Login" and "Hasło" fields, there are red error messages: "Wpisz co najmniej 2 znaki" and "Wymagane jest co najmniej 6 znaków". Below the form is a "Zarejestruj się" button. At the bottom of the page, the text "Politechnika Lubelska, wprowadzenie do Spring Boot" is displayed.

Rys. 6.9. Wynik działania walidacji

W momencie rejestracji nowego użytkownika, należy też w metodzie kontrolera dodać warunek sprawdzający, czy w bazie nie istnieje już użytkownik o zadanym loginie.

Analogiczne modyfikacje powinny być wprowadzone również w przypadku edycji danych użytkownika.

Laboratorium opracowane zostało we współpracy ze studentem Informatyki Patrykiem Drozdem.