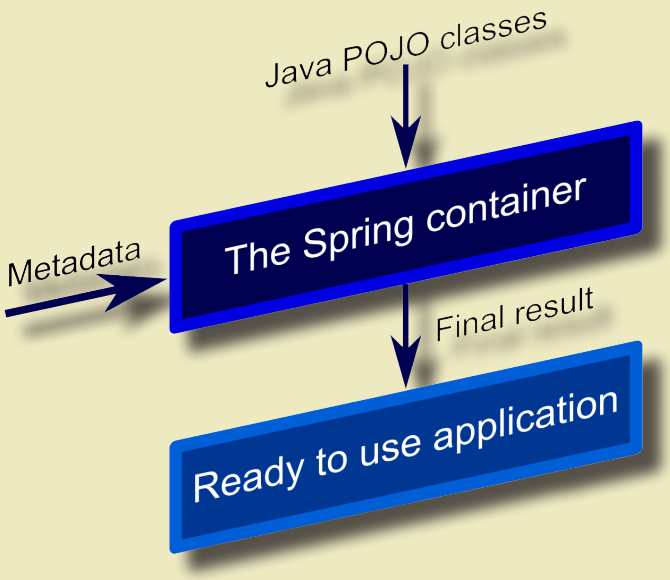
Politechnika Lubelska

PODRĘCZNIKI

Beata Pańczyk

Programowanie aplikacji internetowych JEE w SPRING

Przykłady i zadania



Lublin 2022

Programowanie aplikacji internetowych JEE w SPRING

Przykłady i zadania

Podręczniki – Politechnika Lubelska

POLITECHNIKA
LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI

Beata Pańczyk

Programowanie aplikacji internetowych JEE w SPRING

Przykłady i zadania

POLITECHNIKA
LUBELSKA
WYDAWNICTWO

Lublin 2022

Recenzenci:
dr hab. Andrzej Bochniak, Uniwersytet Przyrodniczy w Lublinie
dr inż. Jakub Smołka, Politechnika Lubelska

,,Konkurs na wydanie podręcznika akademickiego lub skryptu" edycja I

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

ISBN: 978-83-7947-527-8

Wydawca: Wydawnictwo Politechniki Lubelskiej
www.biblioteka.pollub.pl/wydawnictwa
ul. Nadbystrzycka 36C, 20-618 Lublin
tel. (81) 538-46-59

Druk: Soft Vision Mariusz Rajski
www.printone.pl

Nakład: 50 egz.

Spis treści

WSTĘP 8

PRZYGOTOWANIE ŚRODOWISKA PRACY 10

Wymagania wstępne 10
Niezbędne instalacje 10
Konfiguracja i tworzenie projektu aplikacji *web* w *NetBeans* 10

LABORATORIUM 1. ŚRODOWISKO *JEE* I INTERFEJS SERWLETÓW 18

Wprowadzenie 18
Zadanie 1.1. Pierwszy serwlet 19
Zadanie 1.2. Metadane żądania *HTTP* 22
Zadanie 1.3. Metoda *init()* w serwlecie 22
Zadanie 1.4. Dostęp do parametrów żądania 23
Zadanie 1.5. Walidacja danych 24
Zadanie 1.6. Mechanizm sesji i ciasteczko 25

LABORATORIUM 2. *JSP* I *JAVA BEAN* 27

Wprowadzenie 27
Zadanie 2.1. Wprowadzenie do stron *JSP* 28
Zadanie 2.2. Obsługa formularza w *JSP* 29
Zadanie 2.3. Zastosowanie *Java Bean* 30
Zadanie 2.4. Strona do obsługi błędów – *ErrorPage* 32

LABORATORIUM 3. SERWLETY, *JAVA BEAN, JSP* I *MYSQL* 34

Wprowadzenie 34
Zadanie 3.1. Przygotowanie bazy danych na serwerze *MySQL* 35
Zadanie 3.2. Interfejs *JDBC* i praca z bazą danych w serwlecie 35
Zadanie 3.3. Prosty *MVC* – serwlet, model *JavaBean*, widok *JSP* 36
Zadanie 3.4. Identyfikator obiektu w adresie *URL* 40

LABORATORIUM 4. WPROWADZENIE DO *SPRING MVC* 42

Wprowadzenie 42
Zadanie 4.1. Przygotowanie bazy danych *MySQL* 43
Zadanie 4.2. Utworzenie projektu *Spring MVC* 44
4.2.1. Klasa *Pracownik* jako *POJO* 46
4.2.2. Klasa *PracownikDao* do pracy z danymi 46
4.2.3. Klasa kontrolera *PracownikController* 47
4.2.4. Konfiguracja w plikach *web.xml* i *spring-servlet.xml* 49
4.2.5. Widoki *JSP* 52
Zadanie 4.3. Zbudowanie i uruchomienie projektu 55
Zadanie 4.4. Akcja kontrolera *DELETE* 57
Zadanie 4.5. Akcja kontrolera *EDIT* 57

ZADANIE 4.6. OBSŁUGA WYJĄTKÓW W KONTROLERZE ... 58

LABORATORIUM 5. *SPRING BOOT* I *JPA* ... 60

WPROWADZENIE ... 60
ZADANIE 5.1. KONFIGURACJA PROJEKTU W *SPRING BOOT* ... 61
5.1.1. Oddzielenie kontrolera ... 65
5.1.2. Eksport aplikacji do pliku *war* lub *jar* ... 67
ZADANIE 5.2. *SPRING BOOT* I *SPRING DATA JPA* ... 68
5.2.1. Dodatkowe zależności ... 68
5.2.2. Klasa encji ... 69
5.2.3. Konfiguracja bazy danych ... 70
5.2.4. Repozytorium *CRUD* ... 71
5.2.5. Zastosowanie repozytorium ... 72
ZADANIE 5.3. *MYSQL* I *SPRING BOOT* ... 74
ZADANIE 5.4. METODY WYSZUKIWANIA W REPOZYTORIUM ... 75
ZADANIE 5.5. WYSZUKIWANIE W BAZIE *WORLD* ... 76
ZADANIE 5.6. WSTĘP DO *SPRING SECURITY* ... 77

LABORATORIUM 6. *SPRING SECURITY* I *THYMELEAF* ... 80

WPROWADZENIE ... 80
ZADANIE 6.1. UTWORZENIE PROJEKTU W *SPRING INITIALIZR* ... 81
ZADANIE 6.2. DODATKOWE ELEMENTY KONFIGURACJI PROJEKTU ... 82
ZADANIE 6.3. KONFIGURACJA *SPRING SECURITY* ... 83
6.3.1. Klasa *Security* i jej metody *configure()* ... 83
6.3.2. Klasa encji *User* ... 86
6.3.3. Interfejs *UserDao* ... 86
6.3.4. Klasa *UserAuthenticationDetails* ... 87
ZADANIE 6.4. KONTROLER I WIDOKI *THYMELEAF* ... 88
ZADANIE 6.5. USUWANIE I EDYCJA DANYCH UŻYTKOWNIKA ... 95
ZADANIE 6.6. WALIDACJA DANYCH Z FORMULARZA REJESTRACJI ... 96

LABORATORIUM 7. *SPRING BOOT* I *REST API* ... 98

WPROWADZENIE ... 98
ZADANIE 7.1. INICJALIZACJA PROJEKTU *REST API* ... 99
ZADANIE 7.2. PODSTAWOWE ELEMENTY *REST API* ... 99
7.2.1. Adnotacje *@JsonView* w klasie encji ... 99
7.2.2. Repozytorium i klasa z adnotacją *@Service* ... 100
7.2.3. *REST* kontroler ... 100
7.2.4. Utrwalenie danych z żądania ... 102
ZADANIE 7.3. OBSŁUGA METOD *DELETE* I *PUT* ... 104
ZADANIE 7.4. *JAVASCRIPT* Z *FETCH API* ... 105

LABORATORIUM 8. *SPRING BOOT* I *DTO* ... 106

WPROWADZENIE ... 106
ZADANIE 8.1. APLIKACJA BEZ OBIEKTÓW *DTO* ... 107
8.1.1. Klasy encji w relacji 1:1 ... 107

8.1.2. Repozytoria i serwis ... 109
8.1.3. Kontroler ... 110
8.1.4. Plik *import.sql* ... 110
ZADANIE 8.2. APLIKACJA Z *DTO* ... 113
8.2.1. Klasa *DTO* ... 113
8.2.2. Klasa konwertera ... 114
8.2.3. Wykorzystanie *DTO* w serwisie i kontrolerze ... 115
8.2.4. Test działania obiektu *DTO* ... 116
ZADANIE 8.3. *DTO* W REPOZYTORIACH ... 116
ZADANIE 8.4. BIBLIOTEKA *MAPSTRUCT* ... 118

LABORATORIUM 9. *SPRING* I *JSON WEB TOKEN* ... 121

WPROWADZENIE ... 121
ZADANIE 9.1. *SPRING BOOT* I *JWT* ... 122
9.1.1. Konfiguracja *Spring Security* z *JWT* ... 123
9.1.2. Pobranie nazwy i hasła użytkownika ... 125
9.1.3. Kontroler do autentykacji użytkownika ... 126
9.1.4. Klasa *JwtRequest* ... 128
9.1.5. Klasa *JwtResponse* ... 129
9.1.6. Filtr *JWT* w pakiecie *config* ... 129
9.1.7. Klasa *JwtAuthenticationEntryPoint* ... 131
9.1.8. Konfiguracja *Web Security* ... 132
ZADANIE 9.2. TEST DZIAŁANIA W NARZĘDZIU *POSTMAN* ... 134

LABORATORIUM 10. *SPRING, JWT* I *MYSQL* ... 137

ZADANIE 10.1. *JWT* I *MYSQL* ... 137
10.1.1. Klasy *UserDao* i *UserDto* ... 139
10.1.2. Repozytorium *JPA* ... 140
10.1.3. Konfiguracja *Web Security* ... 141
10.1.4. Implementacja klasy *JwtAuthenticationController* ... 143
ZADANIE 10.2. TEST DZIAŁANIA W NARZĘDZIU *POSTMAN* ... 146

PODSUMOWANIE ... 149

BIBLIOGRAFIA ... 150

STRESZCZENIE ... 152

ABSTRACT ... 153

Wstęp

Skrypt jest zbiorem przykładów i zadań proponowanych do realizacji na laboratorium programowania aplikacji internetowych dla studentów kierunku Informatyka. Jego treść została dostosowana do obowiązującego programu studiów, z uwzględnieniem uzyskiwanych efektów kształcenia w ramach przedmiotu „Programowanie aplikacji internetowych". Przedmiot jest realizowany w ramach 30 godzin wykładu i 30 godzin laboratorium.

Zbiór zadań obejmuje zagadnienia związane z programowaniem aplikacji internetowych w środowisku *Java Enterprise Edition* (*JEE)* z wykorzystaniem szkieletu programistycznego *Spring*. Tematyka związana jest z poznaniem zaawansowanych wzorców programowania obiektowego i jest realizowana dla studentów II stopnia kierunku Informatyka. Zakładana jest znajomość podstawowych zagadnień związanych z tworzeniem aplikacji internetowych, które studenci nabyli w trakcie studiów I stopnia na kierunku Informatyka. Do realizacji zadań wymagana jest dobra znajomość języka *HTML* oraz solidnych podstaw programowania w języku *Java* i *Java Script*.

Zadania proponowane w skrypcie wykorzystują rozwiązania i wzorce projektowe, stosowane aktualnie w programowaniu aplikacji internetowych. Platformą programistyczną jest zaawansowane środowisko *Java Enterprise Edition* i *Spring*, jako najbardziej popularny szkielet programistyczny języka *Java*.

Skrypt zawiera przykłady i zadania, które umożliwiają indywidualną organizację pracy studenta. Przygotowane instrukcje, prowadzą czytelnika krok po kroku do rozwiązania zadanego problemu, pozwalając przy tym poznać zaawansowane techniki programowania obiektowego. Każdy rozdział skryptu, poprzedzony krótkim wprowadzeniem do omawaianych zagadnień, zawiera zestawy zadań programistycznych, częściowo rozwiązanych tak, aby łagodnie wprowadzić studenta do zadanego problemu i zasugerować właściwe metody do rozwiązania kolejnych etapów zadania. Realizacja tych zadań pomoże Czytelnikowi w poznaniu, zrozumieniu i utrwaleniu prezentowanego materiału.

Zadania umieszczone w niniejszym opracowaniu zostały przetestowane na laboratorium, prowadzonym przez autorkę ze studentami II stopnia kierunku Informatyka na Wydziale Elektrotechniki i Informatyki Politechniki Lubelskiej. Dodatkowo studenci: Patryk Drozd, Kamil Jaskot i Sebastian Iwanowski brali czynny udział w opracowaniu wybranych zadań.

W skrypcie zaproponowano zadania do realizacji w trakcie dziesięciu spotkań laboratoryjnych. Podział ten traktować należy bardzo elastycznie, do wykonania niektórych tematów studenci mogą potrzebować więcej czasu. W zwyczajowym cyklu 15 spotkań po 2 godziny laboratorium, pozostały czas można przeznaczyć na samodzielną pracę studenta i realizację aplikacji, wykorzystującej i rozszerzającej rozwiązania pokazane w przykładach.

Podziękowania

Bardzo dziękuję studentom Informatyki, którzy brali aktywny udział w przygotowaniu treści zadań, wykorzystanych w niniejszym opracowaniu.

Szczególne podziękowania kieruję do recenzentów, dr hab. Andrzeja Bochniaka, prof. uczelni i dr inż. Jakuba Smołki, za ich cenne i bardzo szczegółowe uwagi, których uwzględnienie pozwoliło wyeliminować niedociągnięcia i poprawić wartość merytoryczną prezentowanego materiału.

Autorka

Przygotowanie środowiska pracy

Do realizacji zadań proponowanych w niniejszym skrypcie potrzebna jest instalacja odpowiedniego oprogramowania i skonfigurowanie środowiska pracy.

Wymagania wstępne

Do wykonania zadań konieczna jest znajomość *HTML* oraz języka *Java* i języka *SQL*. Potrzebny jest także serwer aplikacji *JEE* (np. *Apache Tomcat*).

Prezentowane zrzuty ekranu wykonanych zadań zostały zrealizowane w środowisku *NetBeans* 12.6, ale można również skorzystać z innych środowisk programistycznych umożliwiających pracę w środowisku *JEE*.

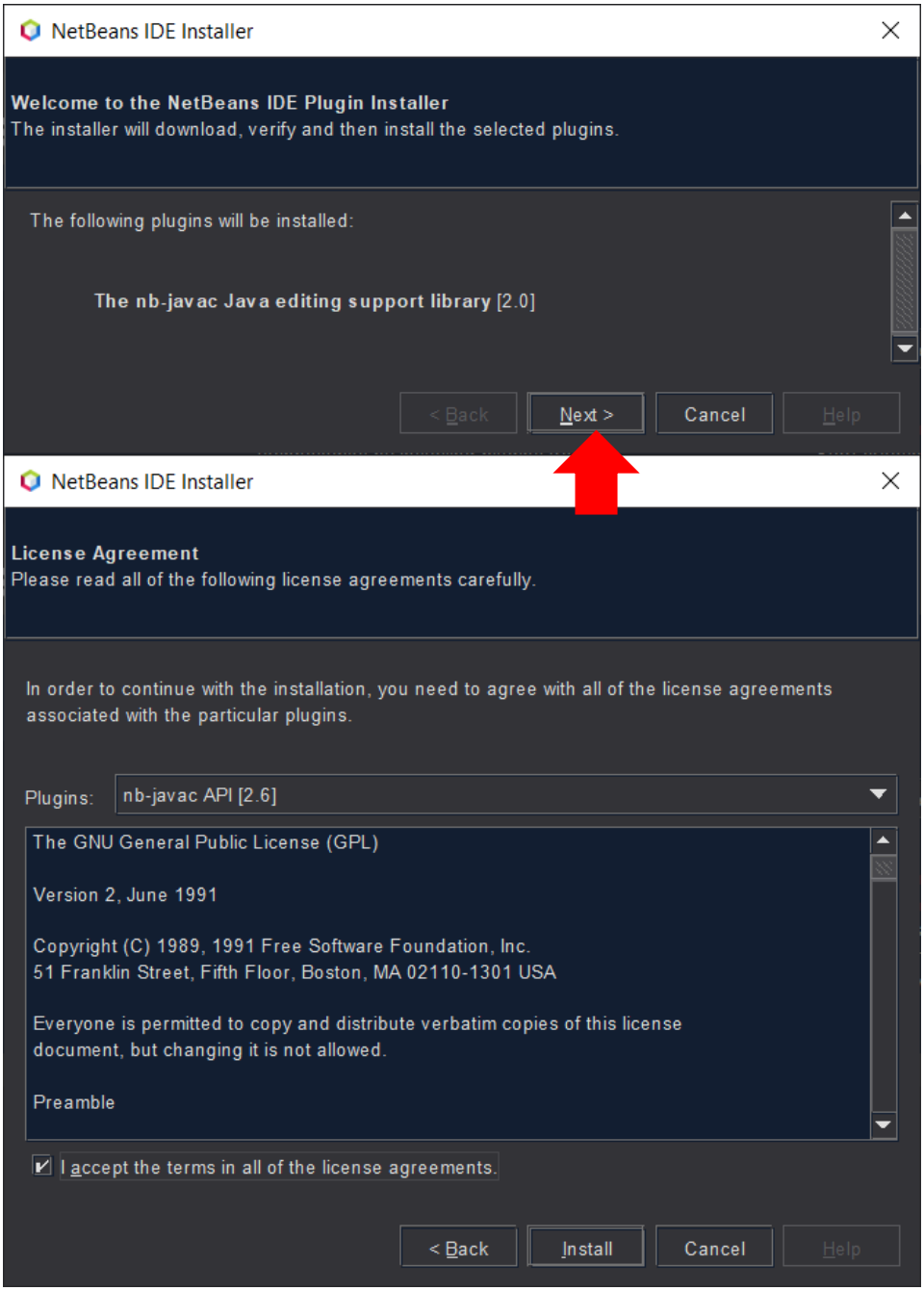
Niezbędne instalacje

Do wykonania zadań laboratoryjnych należy zainstalować:

- ***Java SE Development Kit*** (wersję minimum 8) [9]: *https://www.oracle.com/pl/java/technologies/javase-jdk15-downloads.html*
- środowisko programistyczne, np. ***N***etBeans [20]: *https://netbeans.apache.org/download/index.html lub* ***Intellij****: https://www.jetbrains.com/idea/download/#section=windows*
- serwer ***MySQL*** i serwer aplikacji ***JEE*** (np. ***Apache Tomcat*****).** Oba serwery są dostępne w pakiecie ***XAMPP*** [15]: *https://www.apachefriends.org/pl/download.html*

Konfiguracja i tworzenie projektu aplikacji *web* w *NetBeans*

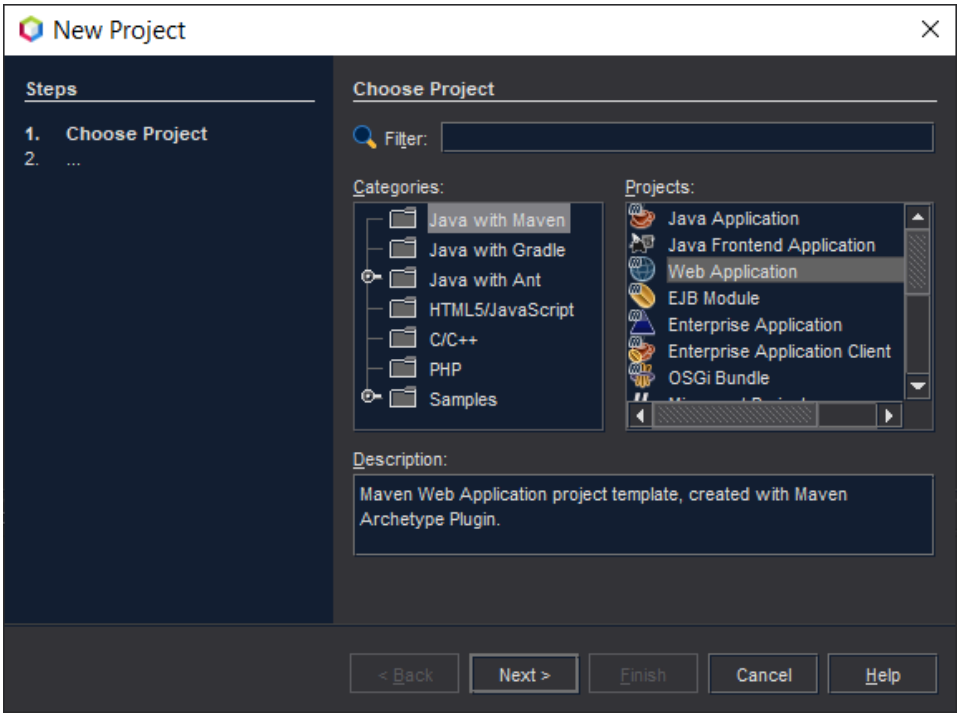
Prezentowane w niniejszym skrypcie zrzuty ekranu zostały zrealizowane w środowisku *NetBeans* 12.6. Po instalacji i pierwszym uruchomieniu programu *NetBeans*, pojawi się zapytanie o doinstalowanie dodatkowej biblioteki *nb-javac*, która będzie potrzebna do tworzenia aplikacji internetowych (Rys. 1).



Rys. 1. Instalacja dodatkowej biblioteki

1. Tworzenie nowego projektu *Java with Maven* → *Web Application*

Na rysunku 2 przedstawiono okno kreatora nowego projektu aplikacji webowej w *NetBeans* 12.6. Przykłady projektów prezentowane w skrypcie są zarządzane za pomocą narzędzia *Maven*. Narzędzia *Gradle* czy *Ant* także mogą być zastosowane w celu automatyzacji procesu budowania kodu, jednak to *Maven* jest uznawany za najbardziej popularne narzędzie wspomagające proces budowania aplikacji w języku *Java*.

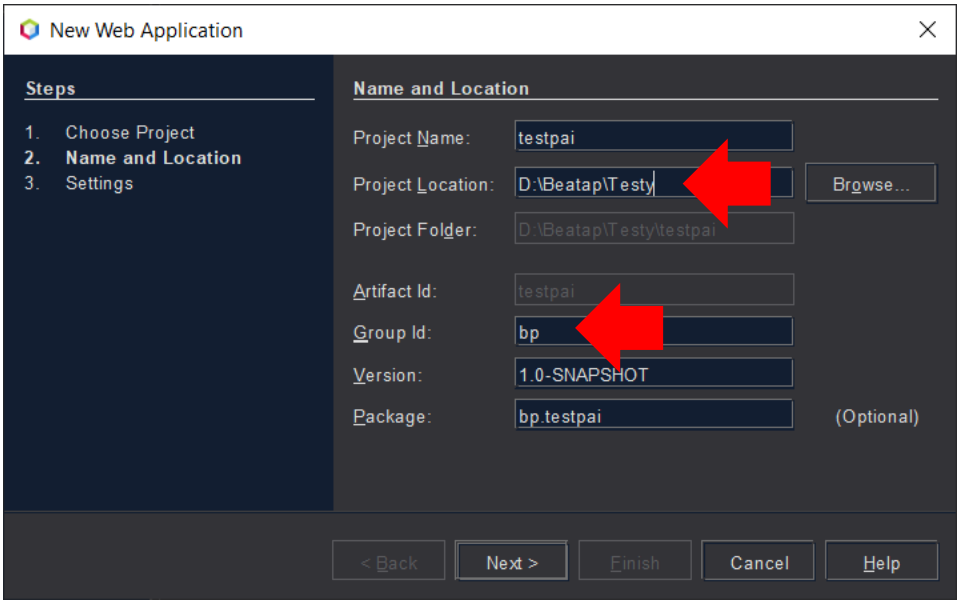


Rys. 2. Tworzenie nowego projektu *Maven* aplikacji internetowej

Podstawowym plikiem konfiguracyjnym projektu *Maven* jest plik ***pom.xml***. Plik ten zawiera całą konfigurację związaną z projektem, opis jak ma być budowany, konfigurację dodatkowych wtyczek, itp.
Plik ***pom.xml*** musi zawierać minimum:

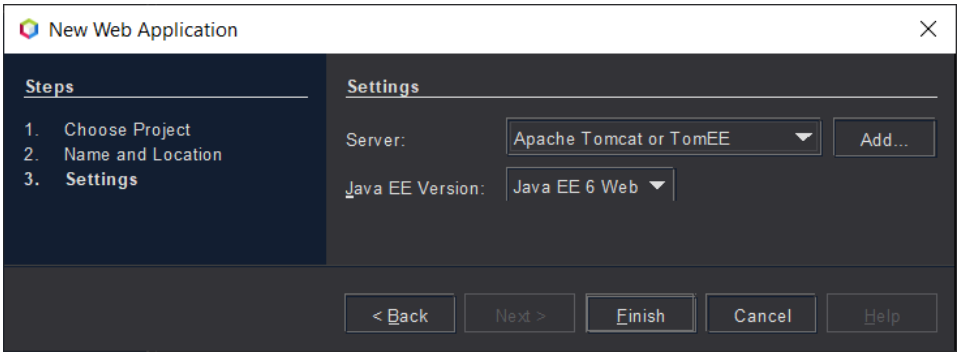
- koordynaty projektu (ang. *coordinates*), czyli unikalną parę:
  - identyfikator grupy (ang. *groupId*), czyli organizacji, która utworzyła projekt,
  - identyfikator głównego artefaktu (ang. *artifactId*) generowanego przez projekt.
- numer wersji (ang. *version*).

Moduły tego samego projektu mają wspólny *groupId*, ale unikalny *artifactId*. Ustawienie właściwości projektu (nazwa, lokalizacja, koordynaty projektu *Maven*) zaprezentowano na rysunku 3.



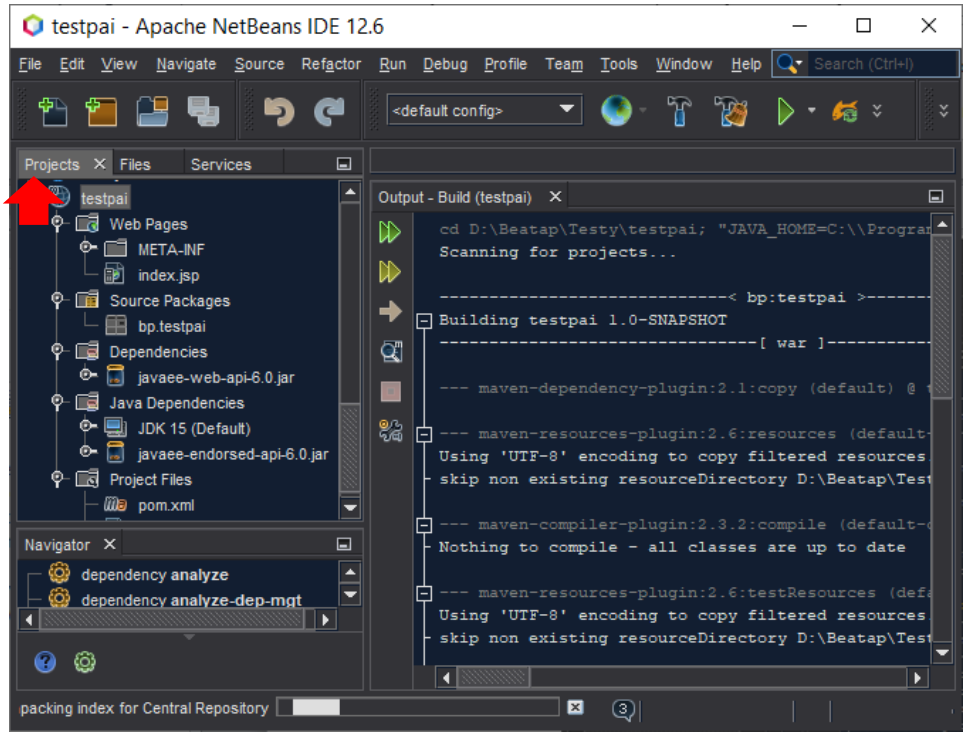
Rys. 3. Ustawienie nazwy, lokalizacji i wskazanie pakietu (group id) tworzonego projektu *Maven*

W kolejnym kroku wskazano serwer aplikacji *JEE* (Rys. 4).



Rys. 4. Wybór serwera

Pomyślne zakończenie tworzenia projektu powinno wyglądać podobnie jak na rysunku 5.



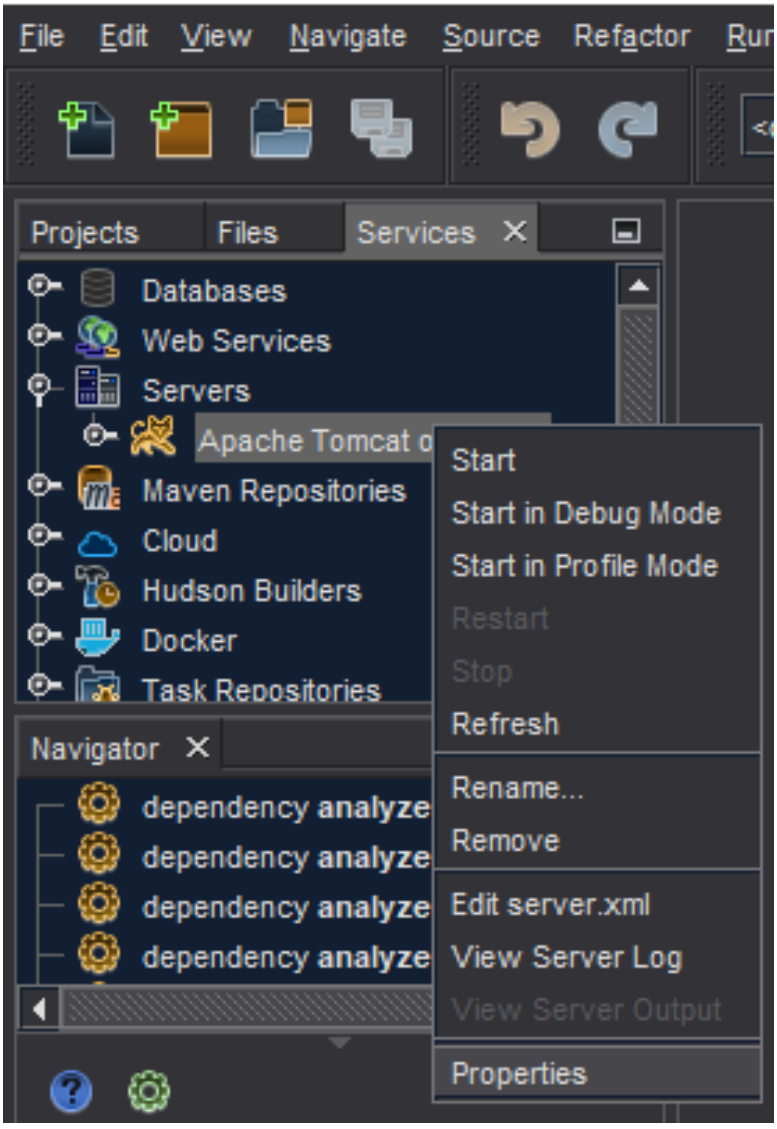
Rys. 5. Przykładowy układ okien i widok struktury plików utworzonego projektu

Na rysunku 5 pokazano strukturę katalogów i plików utworzonego projektu (zakładka *Project)*, okno nawigatora (*Navigator*) oraz okno *Output* z opisem etapów tworzenia projektu *Maven*.

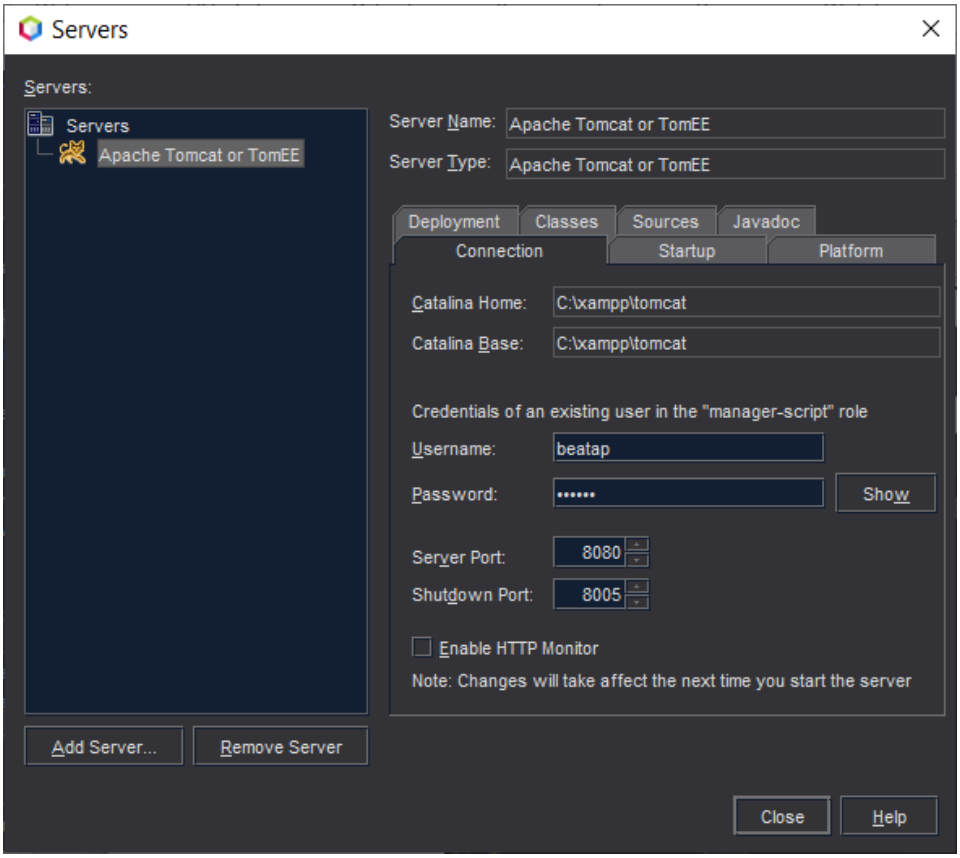
2. Konfiguracja serwera *Tomcat* (z pakietu *XAMPP*)

W celu konfiguracji serwera należy przejść do widoku na zakładce *Services* (Rys. 6a) oraz (Rys. 6b):

- wybrać nazwę i typ serwera z listy rozwijanej (jeśli nie ma na liście odpowiedniego serwera, to należy skorzystać z przycisku *Add Server*, który pozwala dodać do listy wyboru i skonfigurować nowy serwer),
- wskazać lokalizację folderu z programem serwera (*Catalina Home*),
- podać parametry użytkownika (*Username*, *Password*),
- ewentualnie zmienić numery portów (*Server Port*, *Shutdown Port*).



Rys. 6a. Wybór serwera do konfiguracji w celu ustawienia jego właściwości (*Properties*)

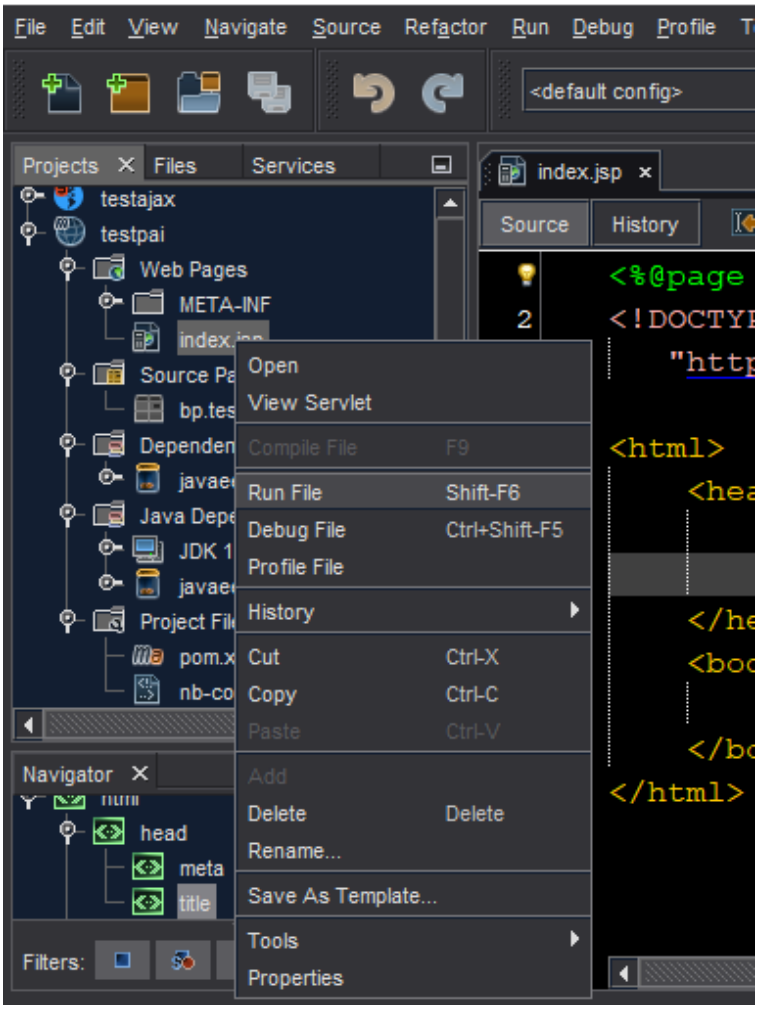


Rys. 6b. Wskazanie lokalizacji i konfiguracja ustawień serwera *Tomcat*

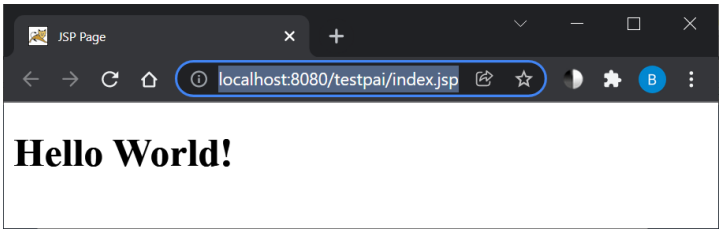
3. Uruchomienie utworzonego projektu

Punktem startowym projektu jest obecnie plik ***index.jsp***. Po wyborze przeglądarki i aktywowaniu menu kontekstowego w okienku z plikiem ***index.jsp***, można skorzystać z opcji ***Run*** w celu uruchomienia projektu i wyświetlenia w przeglądarce strony startowej ***index.jsp*** (Rys. 7).

Uzyskanie efektu jak pokazano na rysunku 8, świadczy o prawidłowym przygotowaniu środowiska do pracy.



Rys. 7. Uruchomienie strony startowej projektu – *index.jsp*



Rys. 8. Widok strony *index.jsp* w przeglądarce

Podobne etapy związane ze wskazaniem i skonfigurowaniem serwera dla aplikacji internetowej należy wykonać w przypadku korzystania z innego środowiska programistycznego, na przykład wspomnianego wcześniej *Intellij*.

Laboratorium 1. Środowisko *JEE* i interfejs serwletów

Cel zajęć

Realizacja zadań z niniejszego laboratorium pozwoli studentom zapoznać się z zasadami tworzenia aplikacji internetowych z wykorzystaniem podstwowych elementów środowiska *Java Enterprise Edition* (*JEE*), klasy *HttpServlet* serwletów (ang. *Java Servlet*) [3, 13, 21] oraz z interfejsami *HttpServletRequest* i *HttpServletResponse*.

Zakres tematyczny

- Przygotowanie prostej aplikacji internetowej w *JEE*, korzystającej z klasy *HttpServlet*.
- Poznanie i zastosowanie metod interfejsów *HttpServletRequest* oraz *HttpServletResponse* do:
    - wyświetlania metadanych żądania *HTTP*,
    - pobrania i walidacji parametrów przekazanych z formularza w żądaniu *HTTP*,
    - zarządzania ciasteczkami i sesją *HTTP*.

Wprowadzenie

Podstawą aplikacji internetowych tworzonych w języku *Java* jest serwlet (ang. *Java Servlet)*. Serwlet jest programem napisanym w języku *Java*, wykonywanym na kontenerze aplikacji *JEE* (np. na serwerze *Apache Tomcat*). Serwlety stanowią warstwę pośrednią pomiędzy żądaniami przesyłanymi przez przeglądarkę (klienta *HTTP*) oraz aplikacjami działającymi po stronie serwera. Serwlety obsługują żądania *HTTP* (ang. *request*) i generują odpowiedź *HTTP* (ang. *response*) przesyłaną do klienta.

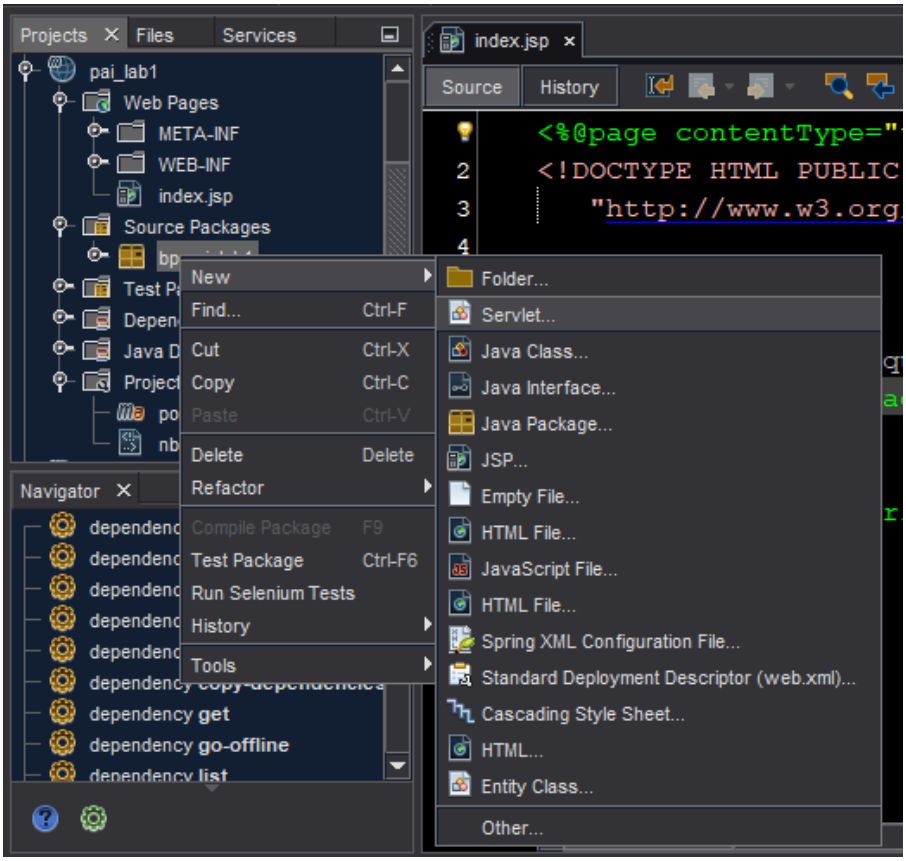
Serwlet umożliwia:

- odczytywanie jawnych informacji przesyłanych przez użytkownika (np. danych z formularza *HTML*),
- odczytywanie niejawnych informacji przesyłanych przez przeglądarkę w żądaniu *HTTP* (np. nagłówków żądania *HTTP*),
- generowanie wyników w różnych formatach (np. *text/html*, *text/json*, *image/jpg* itp.),
- przesyłanie jawnych informacji w różnych formatach do klienta,
- przesyłanie niejawnych informacji w odpowiedzi *HTTP* (np. nagłówków odpowiedzi *HTTP*).

Podstawą działania serwletów są dwa interfejsy: ***HttpServletRequest*** i ***HttpServletResponse***. Całą aplikację internetową można utworzyć korzystając jedynie z możliwości serwletów, jednak przy rozbudowanych aplikacjach nie jest to efektywne. Poznanie interfejsów klasy serwletu pomoże zrozumieć kolejne poznawane w tym skrypcie technologie, które dodają pewien poziom abstrakcji, umożliwiając łatwiejsze i szybsze tworzenie kodu w języku *Java*.

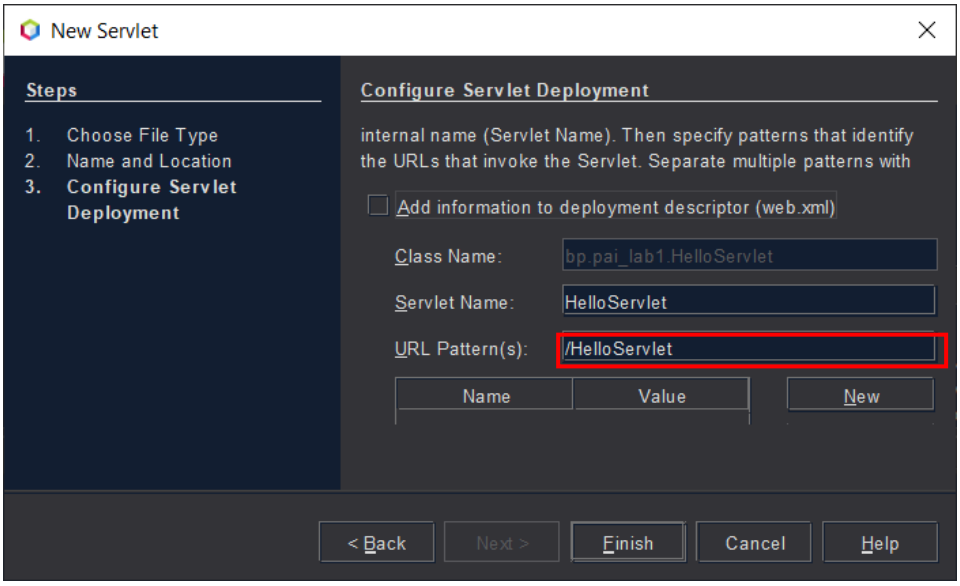
Zadanie 1.1. Pierwszy serwlet

Przygotuj środowisko programistyczne do pracy, jak pokazano na przykładzie *Netbeans* 12.6 w rozdziale wstępnym. Utwórz nowy projekt aplikacji webowej o nazwie ***pai_lab1*** (*File → New Project → Java with Ma ven/Web Application*). Do projektu dodaj nowy plik z kategorii serwletu o nazwie *HelloServlet* (menu kontekstowe projektu lub *File → New File → Servlet* – Rys. 1.1). Tworzony serwlet (klasę dziedziczącą po klasie bazowej serwletu *HttpServlet)* dodaj do pakietu (w przykładzie jest to pakiet *bp.pai_lab1*).



Rys. 1.1. Tworzenie serwletu *HelloServlet* jako plik w projekcie

Przy tworzeniu serwletu zwróć uwagę na konfigurację związaną z jego wdrożeniem (Rys. 1.2) i sprawdź ustawienie dla ***URLPattern***.



Rys. 1.2. Konfiguracja serwletu

Sprawdź, jak wygląda wygenerowany kod serwletu *HelloServlet* oraz usuń zbędne komentarze. Sprawdź kod pomocniczej metody ***processRequest()*** oraz rozwiń fragment kodu na końcu serwletu i dokładnie przejrzyj umieszczone tam metody (***doGet()***, ***doPost()***, ***getServletInfo()***) – Rys. 1.3. **Nie usuwaj ani nie modyfikuj metod *doGet()* i *doPost()***. Zwróć uwagę, że obie metody wywołują pomocniczą metodę ***processRequest()***, wykorzystywanej w kolejnych zadaniach. Parametrami tych metod są dwa ważne obiekty:

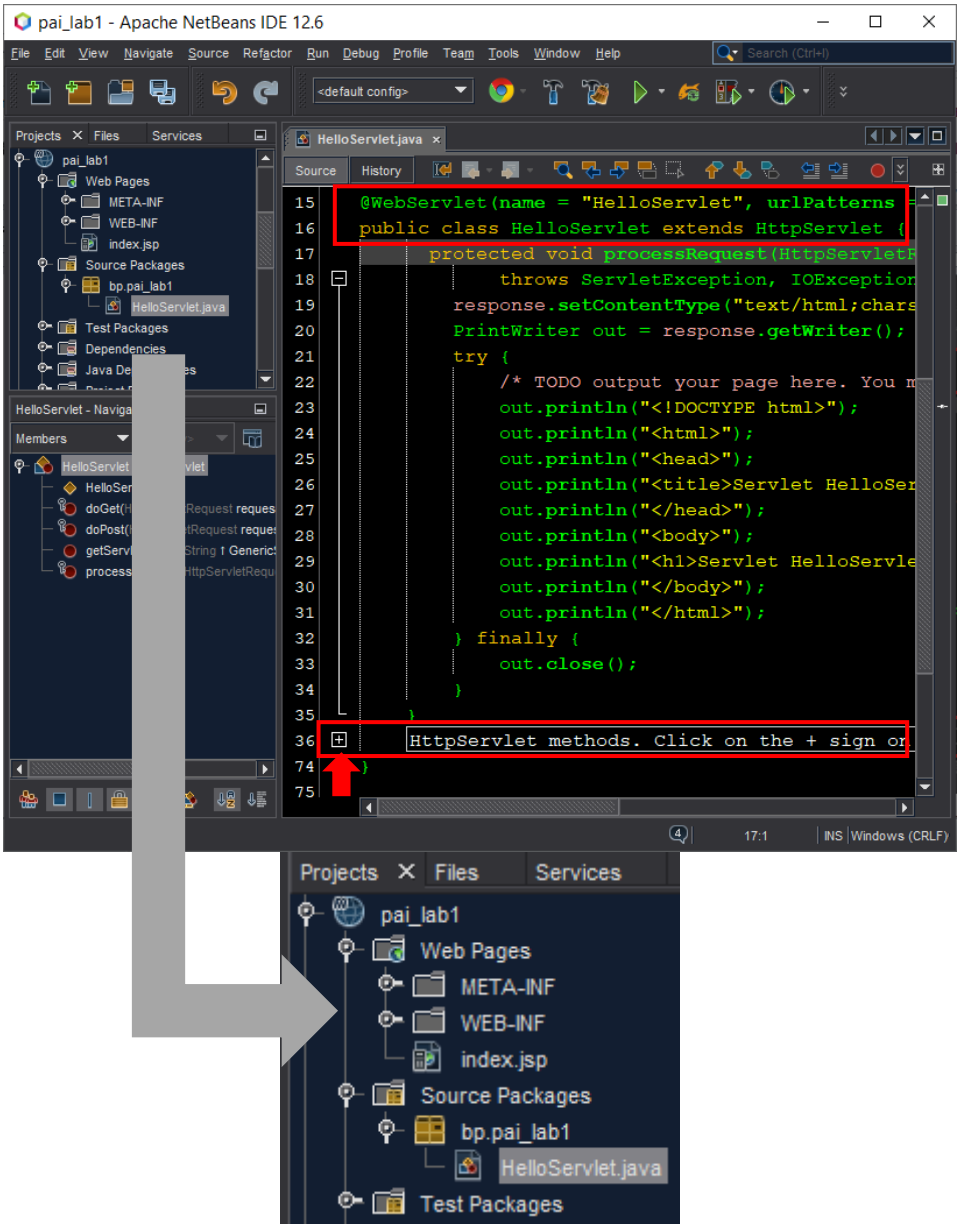
- ***request*** klasy ***HttpServletRequest***, reprezentujący przychodzące żądanie *HTTP*,
- ***response*** klasy ***HttpServletResponse***, reprezentujący obiekt odpowiedzi *HTTP*.

Zwróć uwagę na adnotację poprzedzającą definicje klasy serwletu:

```
@WebServlet(name = "HelloServlet",
            urlPatterns = {"/HelloServlet"})
```

Na pasku narzędziowym *NetBeans* ustaw wybraną przeglądarkę, a następnie uruchom serwlet *Run→Run File*. Sprawdź adres *URL* w pasku adresowym przeglądarki.

W okienku *Output* kontroluj proces budowania, kompilacji i wdrażania projektu w celu uruchomienia na serwerze *Tomcat*, który wskazano w trakcie tworzenia nowego projektu.



Rys. 1.3. Kod serwletu z pomocniczą metodą *processRequest()*

Zadanie 1.2. Metadane żądania *HTTP*

Do metody ***processRequest()*** serwletu ***HelloServlet*** dopisz (w odpowiednim miejscu w bloku ***try***) kod wyświetlający wybrane informacje z obiektu ***request*** (klasy ***HttpServletRequest)***, który w serwlecie reprezentuje obiekt żądania *HTTP* (Przykład 1.1).

Przykład 1.1. Obiekt request i metadane żądania HTTP

```
out.println("<h2>Dane serwera</h2>");
out.println("<p>request.getServerName(): " + request.getServerName() +
       "</p>");
out.println("<p>request.getServerPort(): " + request.getServerPort() +
       "</p>");
out.println("<p>request.getRemoteHost(): " + request.getRemoteHost() +
       "</p>");
out.println("<p>request.getRemoteAddr(): " + request.getRemoteAddr() +
       "</p>");
out.println("<h2>Szczegóły żądania</h2>");
out.println("<p>request.getMethod(): " + request.getMethod() + " </p>");
out.println("<p>request.getQueryString(): " + request.getQueryString() +
"</p>");
```

Uruchom projekt oraz serwlet wpisując w przeglądarce *URL*: ***http://localhost:8080/pai_lab1/HelloServlet***. Przeanalizuj otrzymane wyniki.

Zadanie 1.3. Metoda *init()* w serwlecie

Do klasy serwletu:

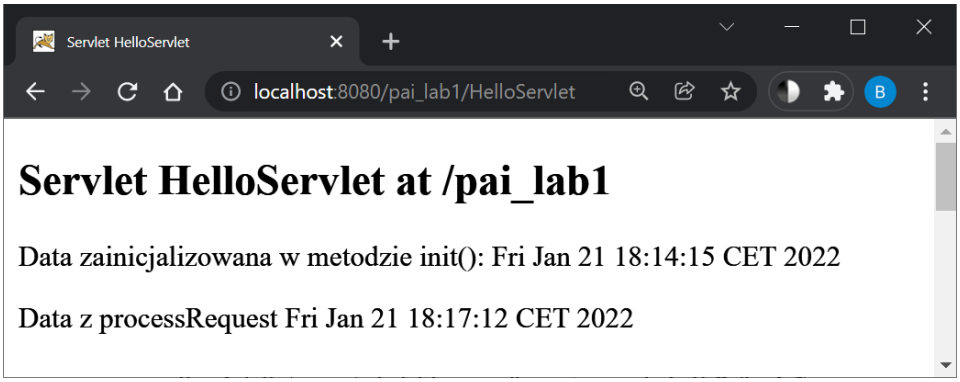
- dodaj deklarację pola ***data1*** typu ***Date*** (zaimportuj bibliotekę ***java.util.Date***),
- dodaj specjalną (nadpisaną) metodę: ***public void init() { }***, w której zainicjalizuj wartość daty: ***data1 = new Date();***

Wyświetl wartość ***data1*** w metodzie ***processRequest()***. Uruchom serwlet i sprawdź, czy data zmienia się po odświeżeniu strony w przeglądarce.

Następnie w **metodzie *processRequest()*** dodaj następujący kod:

```
out.println("<p>data z processRequest  " + new Date() + "</p>");
```

Która data zmienia się po ponownym uruchomieniu i odświeżeniu strony w przeglądarce? (Rys. 1.4).



Rys. 1.4. Data z metody *init* i *processRequest* w serwlecie *HelloServlet*

Korzystając z klas ***java.text.SimpleDateFormat*** i ***java.text.DateFormat*** przekształć wyświetlaną datę do formatu „*yyyy-MM-dd*”:

```
DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
Date d = new Date();
// zastosowanie formatu daty:
// dateFormat.format(d)
```

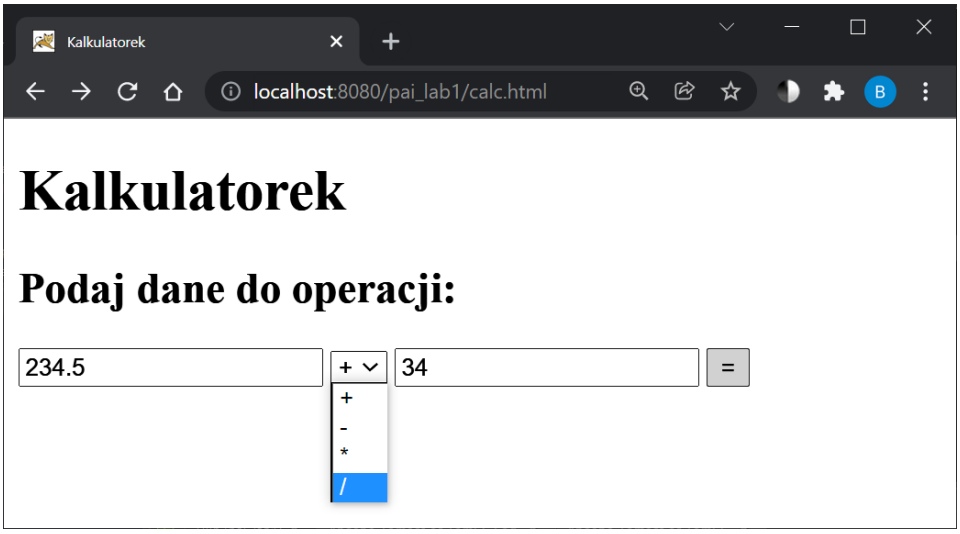
Zadanie 1.4. Dostęp do parametrów żądania

W projekcie utwórz nowy serwlet ***CalcServlet***, a następnie statyczną stronę ***calc.html*** (w głównym folderze ***Web Pages*** projektu, wybierz kategorię tworzonego pliku *HTML*) z formularzem kalkulatorka jak na rysunku 1.5. Formularz powinien być przesyłany za pomocą metody *POST* do serwletu ***CalcServlet***:

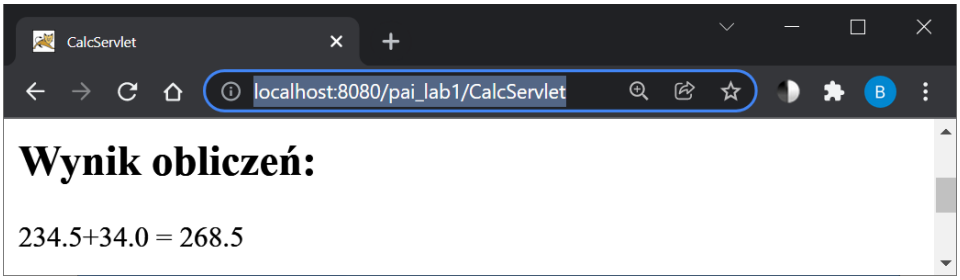
```
<form method="POST" action="CalcServlet" > … </form>
```

Kliknięcie na przycisk ze znakiem równości ma wysłać żądanie z danymi do serwletu ***CalcServlet,*** który powinien pobrać wartości parametrów i wyświetlić wynik obliczeń (Rys. 1.6). Do obliczeń wykorzystaj pomocniczą metodę zdefiniowaną w klasie serwletu, np. ***oblicz(…)***. Metodzie tej należy przekazać w parametrze obiekt ***request.*** Pamiętaj, że wartości parametrów w żądaniu są wysyłane jako typ ***String***. Wartość parametru przesłanego z danymi z formularza można uzyskać za pomocą metody ***getParameter*** obiektu ***request*:**

```
String param=request.getParameter("param");
```



Rys. 1.5. Formularz kalkulatora



Rys. 1.6. Wynik działania kalkulatora

Zadanie 1.5. Walidacja danych

Uzupełnij (jeśli jeszcze tego nie ma) serwlet ***CalcServlet*** tak, aby w przypadku podania niepoprawnego formatu liczby lub w przypadku próby dzielenia przez 0, pojawiał się stosowny komunikat. Walidację zrealizuj po stronie serwletu.

Pamiętaj też, że w wyniku próby pobrania parametru, który nie został przekazany w żądaniu, pojawi się **błąd serwera: *NullPointerException,*** dlatego dla każdego parametru należy sprawdzać przynajmniej warunek:

```
if ( (param == null) || (param.trim().equals("") ))
{
       obsluga_Brakujacej_Wartosci_Parametru(...);
}
```

Zadanie 1.6. Mechanizm sesji i ciasteczko

Zmodyfikuj serwlet ***CalcServlet*** tak, aby wyświetlał historię operacji na kalkulatorze, korzystając z obiektu sesji (w tym celu wygodne jest przechowywanie danych historii w obiekcie ***ArrayList***).
Do pracy z sesją w serwlecie skorzystaj z obiektu ***HttpSession***:

```
HttpSession session=request.getSession(true);
```

i jego metod ***getAttribute*** i ***setAttribute*** (Przykład 1.1).

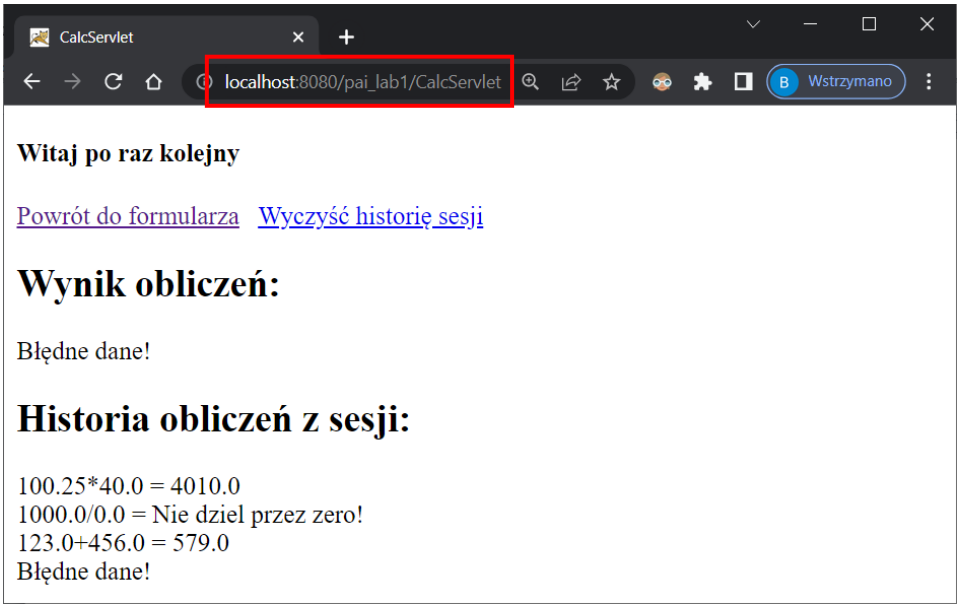
Przykład 1.1. Schemat pracy z obiektem sesji w serwlecie

```
HttpSession sesja = request.getSession(true);
JakasKlasa wartosc = (JakasKlasa) sesja.getAttribute("jakis_id");
if (wartosc==null) {  //nie ma szukanego obiektu w sesji
       wartosc = new JakasKlasa(...);
       sesja.setAttribute("jakis_id",wartosc);
}
zrobCosZ(wartosc);
```

Dodaj także dwa linki: powrotny do formularza i do usunięcia operacji z historii (Rys. 1.7). W metodzie ***processRequest*** wykorzystaj ciasteczko (Przykład 1.2) z powitaniem „Witaj po raz pierwszy” lub „Witaj po raz kolejny” (Rys. 1.7).

Przykład 1.2. Praca z ciasteczkami w serwlecie

```
String nazwaCookie = "UserId";
Cookie [ ] cookies = request.getCookies();
if ( cookies != null )
{      for (int i=0; i<cookies.lenght; i++) {
              Cookie c=cookies[i];
              if (nazwaCookie.equals(c.getName()))
                     jakasOperacjaNaCookie(c.getValue());
       }
}
```



Rys. 1.7. Kalkulator z historią operacji

Laboratorium 2. *JSP* i *Java Bean*

Cel zajęć

Realizacja zadań proponowanych w niniejszym laboratorium pozwoli studentom poznać wyrażenia, skryptlety i wybrane dyrektywy technologii widoków *JavaServer Pages* (*JSP*) oraz podstawowe zasady stosowania i działania komponentów *Java Bean* w aplikacji internetowej.

Zakres tematyczny

- Przygotowanie aplikacji internetowej opartej na możliwościach oferowanych przez technologię stron *Java Server Pages* (*JSP*) bez jawnego programowania serwletów.
- Poznanie i zastosowanie podstawowych dyrektyw *JSP*, wyrażeń *JSP*, oraz skryptletów.
- Poznanie i zastosowanie w aplikacji klasy komponentu *JavaBean* do pracy z danymi pobieranymi z formularza generowanego przez stronę *JSP*.

Wprowadzenie

Technologia *JSP* pozwala na umieszczanie w jednym dokumencie statycznego kodu *HTML* oraz fragmentów generowanych dynamicznie przez kod pisany w języku *Java*. Kod generowany dynamicznie jest umieszczany w dokumencie o strukturze *HTML* za pomocą specjalnych elementów skryptowych (znaczników *JSP*). Elementy skryptowe *JSP* pozwalają na wstawianie kodu *Java* do serwletów, automatycznie generowanych na podstawie stron *JSP*. Strony *JSP* są kompilowane do wynikowej klasy serwletu, dlatego w elementach skryptowych można korzystać z interfejsów i predefiniowanych obiektów dostępnych w klasie serwletu (np. ***request***, ***response***, ***session***).

Stosowane są trzy rodzaje elementów skryptowych *JSP*:

- **wyrażenia:** `<%= wyrażenie javy %>`
- **skryptlety:** `<% instrukcje języka Java %>`
- **deklaracje:** `<%! deklaracja pola lub metody %>`

Na ogólną strukturę serwletu generowanego na podstawie strony *JSP* mają wpływ specjalne dyrektywy umieszczane w kodzie strony *JSP*. Niezbędną dyrektywą (pierwszy wiersz na stronie *JSP*) jest dyrektywa *@page*:

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

Najprościej mówiąc serwlet można interpretować jako kod w języku *Java* zawierający umieszczony w nim kod *HTML*, natomiast strony *JSP* można interpretować jako kod *HTML* zawierający umieszczony wewnątrz kod *Java*.

Bardzo istotnym elelemntem poruszanym w niniejszym laboratorium (Zadanie 2.3) jest pomocniczy komponent danych. W celu zwiększenia poziomu separacji pomiędzy zawartością a prezentacją w zaawansowanych aplikacjach internetowych (opartych najczęściej na wzorcu architektonicznym *MVC* lub jego odmianach), stosowane są specjalne klasy pomocnicze, ułatwiające operacje na danych. W języku *Java* są to ziarna ***JavaBean***. Stosowanie ***JavaBean*** upraszcza używanie obiektów w różnych miejscach aplikacji i ułatwia definiowanie związków pomiędzy parametrami żądania oraz właściwościami obiektu reprezentującego dane, definiowanego jako klasa ***JavaBean***. Zadanie 2.3 ułatwia zrozumienie idei korzystania z takiego ziarna na przykładzie danych przekazanych w parametrach żądania.

Zadanie 2.1. Wprowadzenie do stron *JSP*

Utwórz nowy projekt aplikacji webowej o nazwie ***pai_lab2*** a następnie dodaj do niego nowy plik ***calc.jsp*** (*File→New file→JSP*). Strony *JSP* są plikami tekstowymi o strukturze dokumentu *HTML* z możliwością osadzania w nich kodu *Java* za pomocą odpowiednich znaczników (wyrażenia, skryptlety, deklaracje). W utworzonym pliku ***calc.jsp*** zwróć uwagę na dyrektywę ***@page***, która została umieszczona przed znacznikiem `<!DOCTYPE html>`.

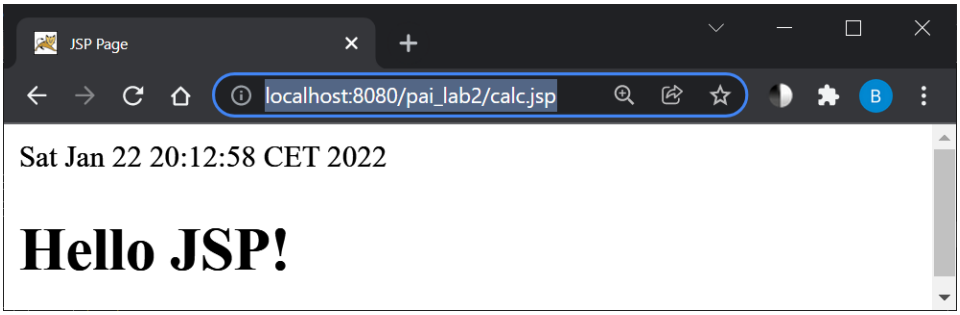
Na górze strony wyświetl bieżącą datę, korzystając z możliwości osadzania **wyrażeń Javy** za pomocą znacznika `<%= new Date() %>` (Rys. 2.1). Za pomocą wyrażenia `<%= … %>` można wyświetlić na stronie wartość wyrażenia (w tym przykładzie datę, ale może to być wynik zwracany przez metodę lub rezultat bardziej złożonych operacji). Na stronach *JSP* dodatkowo można korzystać ze wszystkich klas języka *Java*, po ich wcześniejszym zaimportowaniu. W przykładzie zaimportuj klasę ***Date*** za pomocą dyrektywy:

```
<%@page import="java.util.Date" %>
```

Wykorzystaj następnie klasy ***SimpleDateFormat*** i ***DateFormat*** do przekształcenia wyświetlanej daty do postaci „*yyyy-MM-dd*”, tak jak poprzednio w serwlecie. W tym celu należy zastosować skryptlet (w znacznikach `<% … %>`):

```
<%
DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
Date now = new Date();
String date = dateFormat.format(now);
```

```
// skorzystanie z predefiniowanego obiektu strumienia out,
// znanego z serwletów:
out.println(date);
%>
```



Rys. 2.1 Wyświetlanie daty za pomocą wyrażenia na stronie *JSP*

Zamiast wyświetlać wartość obiektu ***date*** za pomocą instrukcji ***out.println()*** w skryptlecie, można zastosować (jak w pierwszym przykładzie) wyrażenie:

`<%= date %>.`

Zmienna ***date*** powinna być zdefiniowana we wcześniejszym skryptlecie.

W wyrażeniach i skryptletach na stronach *JSP* można stosować predefiniowane obiekty takie jak *out, request, response* czy *session*, co znacznie przyspiesza generowanie wynikowego kodu *HTML*.

Zadanie 2.2. Obsługa formularza w *JSP*

Do strony ***calc.jsp*** dodaj formularz kalkulatora rat (Rys. 2.2) do obliczania raty kredytu na podstawie wzoru:

$$rata = \frac{K \cdot p}{1 - \frac{1}{(1+p)^n}}$$

gdzie:

K – kwota pożyczki,
pr – oprocentowanie roczne,
n – liczba rat,
$p = pr/12$ – oprocentowanie w skali miesiąca.

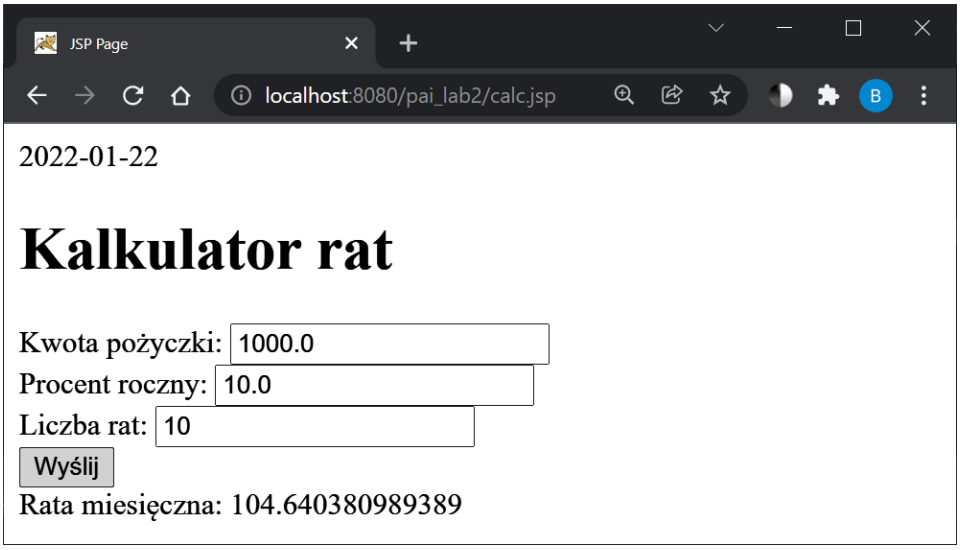
Obliczenia zrealizuj w skryptlecie na stronie ***calc.jsp*** (dane z formularza powinny być wysyłane do tej samej strony). Fragment skryptletu zaprezentowano na Przykładzie 2.1.

Przykład 2.1. Fragment skryptletu obliczającego ratę

```
<% if (request.getParameter("wyslij")!=null){
    String res="";
    try {
      k = Double.parseDouble(request.getParameter("kwota"));
      //Pobierz kolejne wartości parametrów
      //Oblicz ratę lub pokaż komunikat o błędnych danych
    }
    catch (Exception ex) {   }
    out.println(res);
   }
%>
```

Do wyświetlenia obliczonej raty z dwoma miejscami po kropce wykorzystaj klasę ***DecimalFormat*:**

```
DecimalFormat df = new DecimalFormat("#.00");
String rataf=df.format(rata);
```



Rys. 2.2. Formularz kalkulatora rat i przykładowy rezultat obliczeń

Zadanie 2.3. Zastosowanie *Java Bean*

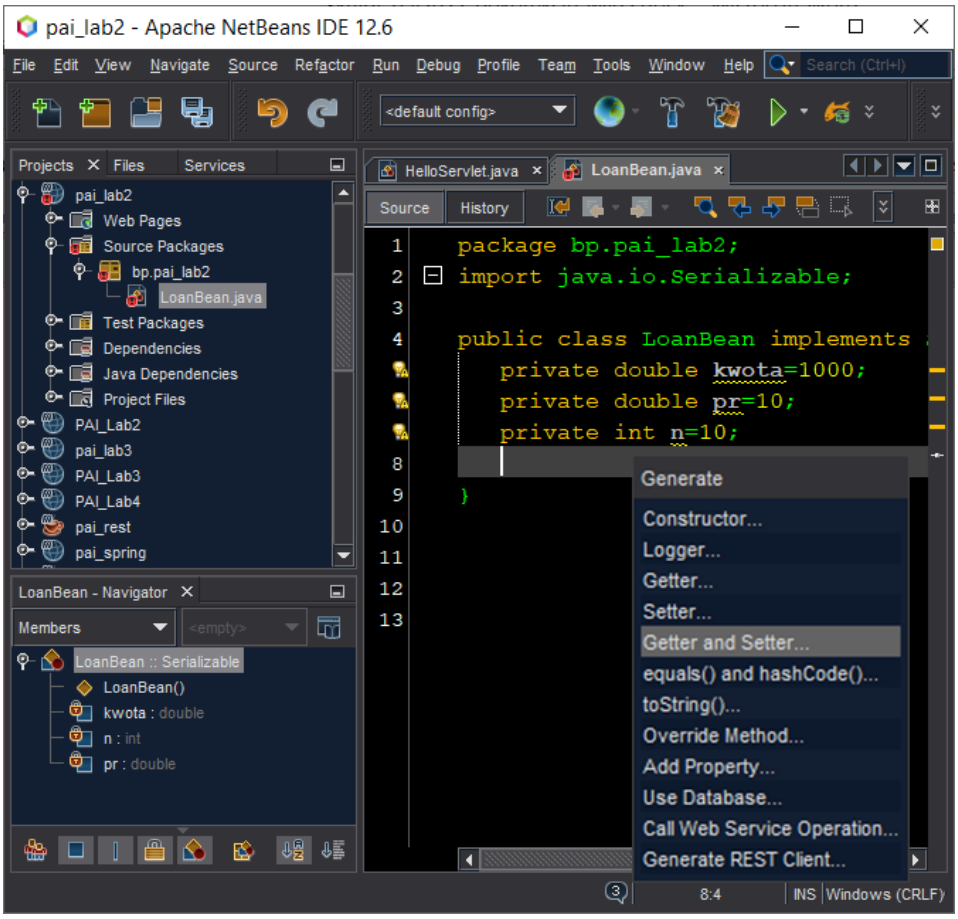
Przebuduj mechanizm obliczania rat kredytów tworząc klasę pomocniczą ***LoanBean*** (Rys. 2.3) oraz wykorzystaj odpowiednie znaczniki *JSP* do pracy z komponentem ***JavaBean***. W tym celu:

- Zdefiniuj w pakiecie projektu klasę ***LoanBean*** (implementującą interfejs ***Serializable***, umożliwiający poprawne serializowanie danych do wykorzystania ich np. w obiekcie sesji), która ma zawierać

wszystkie niezbędne atrybuty (kwota, procent, liczba rat – nazwij pola klasy tak samo jak odpowiadające im pola w formularzu kalkulatora). Następnie, korzystając ze wsparcia środowiska *IDE*, wygeneruj potrzebne metody ***get()*** i ***set()*** – Rys. 2.3.

- W klasie ***LoanBean*** dodaj metodę obliczającą wartość kredytu ***getRata()***.
- Utwórz nową wersję strony z kalkulatorem (np. ***calcwithbean.jsp***) i wykorzystaj w niej znaczniki *JSP* (podaj tu pełną nazwę kwalifikowaną: ***pakiet.LoanBean***):

```
<jsp:useBean id="loan" class="pakiet.LoanBean" scope="session" />
```



Rys. 2.3. Klasa implementująca interfejs *Serializable* oraz korzystanie z wbudowanych mechanizmów generowania metod *get* i *set* w *NetBeans IDE*

UWAGA! Jeśli pola klasy ***LoanBean*** są tak samo nazwane jak pola formularza to po dodaniu do strony *JSP* znacznika:

```
<jsp:useBean id="loan" class="pakiet.LoanBean" scope="session" />
```

można wykorzystać mechanizm automatycznego wiązania danych (ang. *data binding*):

```
<jsp:setProperty name="loan" property="*" />
```

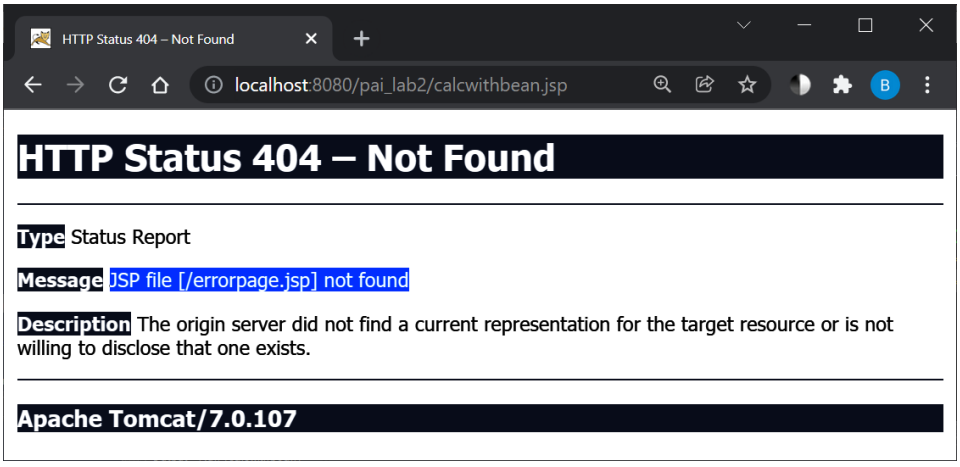
a do właściwości ***LoanBean*** odwoływać się na stronie *JSP* za pomocą np.:

```
Kwota pożyczki: <input name='kwota'
                       value="<%= loan.getKwota() %>">
```

Zadanie 2.4. Strona do obsługi błędów – *ErrorPage*

Przetestuj kalkulator rat, wprowadzając w pola błędny format danych liczbowych (Rys. 2.4) a następnie przygotuj stronę ***errorPage.jsp*** do obsługi błędów (Rys. 2.5). Umieść ją w głownym folderze projektu (tam, gdzie ***index.jsp***). To, że strona jest dedykowana do obsługi błędów wskazuje atrybut ***isErrorPage*** dyrektywy ***@page***:

```
<%@page contentType="text/html" pageEncoding="UTF-8"
        isErrorPage="true" %>
```



Rys. 2.4. Wynik działania aplikacji po wprowadzeniu błędnych danych do pola liczbowego

W momencie wystąpienia niekontrolowanego wyjątku, poszukiwana jest domyślna strona obsługi błędów o nazwie ***errorPage.jsp***. Jeśli zostanie znaleziona – ma możliwość skorzystania z obiektu ***Exception*** i wyświetlenia stosownego komunikatu o zaistniałym problemie (Przykład 2.2).

Przykład 2.2. Schemat strony Error Page

```
<%@page contentType="text/html" pageEncoding="UTF-8" isErrorPage="true"
%>
<!DOCTYPE html>
<html>
    <head>...</head>
    <body>
        <h2>Wprowadzono błędne dane!</h2>
        <p>Pojawił się następujący błąd:
           <%= exception.getMessage() %>. <br />
        </p>
    </body>
</html>
```

Po dodaniu strony ***errorpage.jsp***, wynik jej działania przedstawia rysunek 2.5.

Rys. 2.5. Wynik działania aplikacji po zdefiniowaniu strony do obsługi błędów

Laboratorium 3. Serwlety, *Java Bean*, *JSP* i *MySQL*

Cel zajęć

Realizacja zadań z niniejszego laboratorium pozwoli studentom poznać zasady realizacji wzorca *MVC* na przykładzie implementacji aplikacji współpracującej z bazą danych *MySQL* z wykorzystaniem podstawowego interfejsu *JDBC*.

Zakres tematyczny

- Przygotowanie prostej aplikacji internetowej z wykorzystaniem wzorca projektowego *MVC*, współpracującej z bazą danych *MySQL* [3, 13, 24].
- Wykorzystanie:
  - serwletu (kontroler we wzorcu *MVC*),
  - stron *JSP* (widoki w *MVC*),
  - komponentu *JavaBean* (model w *MVC*).
- Wykonywanie operacji na danych w serwlecie z wykorzystaniem podstawowego interfejsu *JDBC*.

Wprowadzenie

Wzorzec projektowy *MVC* (*ang Model View Controller*) zakłada separację kodu tworzącego i operującego na danych od kodu służącego do przedstawiania tych danych. Najprostszą realizację aplikacji internetowej w języku *Java*, spełniającej założenia *MVC* można utworzyć z wykorzystaniem *Java Bean* jako modeli, serwletu jako kontrolera oraz widoków *JSP*. Podstawowe narzędzia implementacji takiej separacji są standardowo dostępne w *API* serwletów (interfejs ***RequestDispatcher***).

Implementacja wzorca *MVC* z zastosowaniem interfejsu ***RequestDispatcher*** składa się z następujących kroków:

1. Zdefiniowanie komponentów reprezentujących dane jako ***JavaBean***.
2. Zastosowanie serwletów do obsługi żądań.
3. Zapisanie danych w komponentach (w serwlecie).
4. Zapisanie (w serwlecie) komponentów w obiekcie żądania, sesji lub kontekście serwletu (metodą ***setAttribute***).
5. Przekazanie żądania z serwletu do strony *JSP* (metoda ***forward*** obiektu ***requestDispatcher***).
6. Pobranie danych (utworzonych w serwlecie) na stronie *JSP* za pomocą znaczników ***jsp:useBean*** i ***jsp:getProperty***.

W *MVC* serwlety odpowiadają za nadsyłane żądania, w ogóle nie generują wyników. Zadanie to realizują strony widoków *JSP*, które z kolei nie tworzą i nie modyfikują komponentów danych a jedynie odwołują się do nich.

Zadanie 3.1. Przygotowanie bazy danych na serwerze *MySQL*

Korzystając z pliku ***world.sql*** z przykładową bazą danych (do pobrania ze strony: *https://dev.mysql.com/doc/index-other.html*), utwórz na serwerze *MySQL* bazę danych ***world*** i przejrzyj jej strukturę.

Utwórz nowy projekt aplikacji webowej o nazwie ***pai_lab3***, a następnie do pliku ***pom.xml*** dodaj następującą zależność, która dołączy do projektu sterownik do *MySQL*:

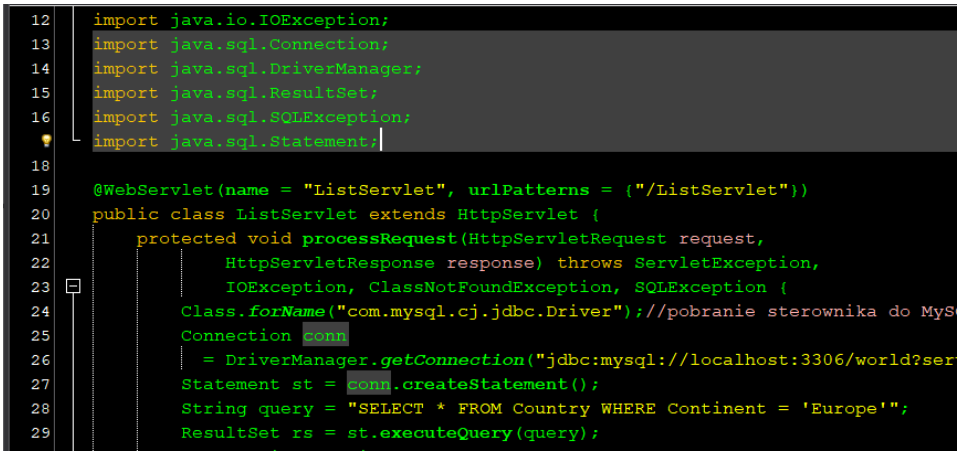
```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
     <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
     <version>8.0.11</version>
</dependency>
```

Zadanie 3.2. Interfejs *JDBC* i praca z bazą danych w serwlecie

W pakiecie projektu utwórz serwlet ***ListServlet***, który będzie łączył się z bazą danych. W serwlecie pobierz i wyświetl na stronie *HTML* informacje np. o krajach Europy:

```
//pobranie sterownika do MySQL:
Class.forName("com.mysql.cj.jdbc.Driver");
//utworzenie obiektu połączenia do bazy danych MySQL:
Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/world?
serverTimezone=UTC", "root", "");
//utworzenie obiektu do wykonywania zapytań do bd:
Statement st = conn.createStatement();
String query="SELECT * FROM Country WHERE Continent = 'Europe'";
//wykonanie zapytania SQL:
ResultSet rs = st.executeQuery(query);
```

Wprowadź odpowiednią nazwę bazy danych, użytkownika oraz hasło. Metoda pracująca z bazą danych powinna wyrzucać (lub obsługiwać) wyjątki ***ClassNotFoundException*** i ***SQLException***. Jeśli korzystasz z pomocniczej metody serwletu ***processRequest()***, jak na rysunku 3.1, to także metody ***doGet()*** i ***doPost()***, które ją wywołują, powinny obsługiwać te wyjątki. Zwróć też uwagę, że wszystkie klasy do pracy z bazą danych (***Connection, Statement, ResultSet***) są importowane z pakietu ***java.sql***.



Rys. 3.1. Utworzenie połączenia do bazy danych w serwlecie *ListServlet*

Korzystając z metod klasy ***ResultSet*** sprawdź możliwości (Rys. 3.2) konwersji danych pobieranych z odpowiednich kolumn tabeli ***Country***. Wyświetl informacje o nazwie, kodzie oraz liczbie ludności w krajach Europy (Rys. 3.3):

```
while (rs.next()) {
   //pobierz i wyświetl dane z odpowiedniej kolumny
   out.print(rs.getString("name"));
   //out.println …
}
```

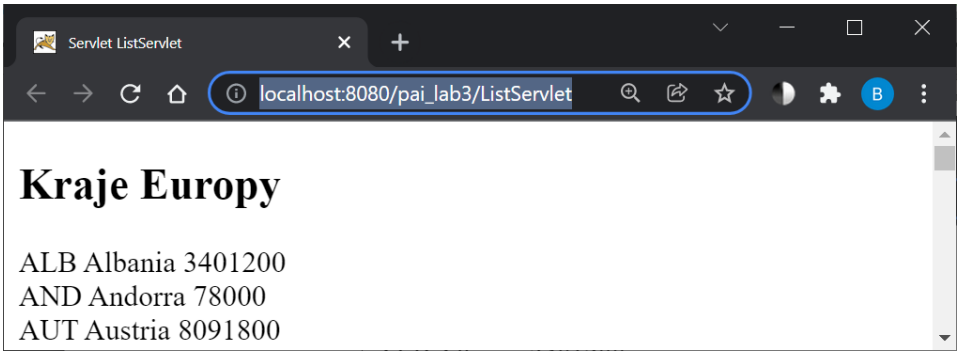
Zadanie 3.3. Prosty *MVC* – serwlet, model *JavaBean*, widok *JSP*

W poprzednim zadaniu dostęp do danych i generowanie widoku realizowano za pomocą pojedynczej klasy serwletu. Teraz zmodyfikuj poprzedni projekt tak, aby rozdzielić funkcje na kilka klas. Serwlet (klasa kontrolera w *MVC*) nadal obsługuje żądanie i pobiera dane z bazy (jak poprzednio), ale przekazuje je w obiekcie sesji za pomocą klasy modelu (***JavaBean***) do strony widoku *JSP*.

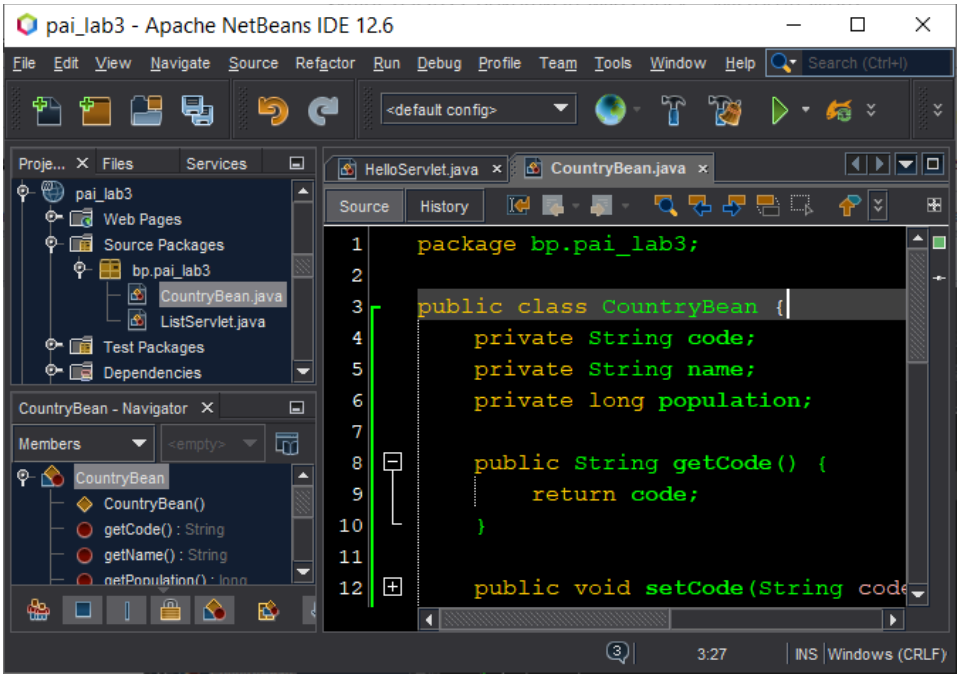
W pakiecie projektu utwórz klasę modelu ***CountryBean*** (spełniającą założenia klasy ***JavaBean***), która pomoże przetwarzać dane pomiędzy serwletem a stronami widoków *JSP*. Klasa powinna implementować interfejs ***Serializable*** oraz zawierać pola prywatne odpowiadające kolumnom tabeli ***Country*** z bazy danych ***world*** (zdefiniuj na razie tylko 3 pola klasy: ***code, name, population***). Korzystając ze wsparcia *IDE*, wygeneruj automatycznie kod metod ***get*** oraz ***set*** w klasie ***CountryBean*** (Rys. 3.4).

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x												
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x												
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x												
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x												
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x												
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x												
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x												
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x												
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x						
getBytes														X	X	x									
getDate											x	x	x				X		x						
getTime											x	x	x					X	x						
getTimestamp											x	x	x				x	x	X						
getAsciiStream											x	x	X	x	x	x									
getUnicodeStream											x	x	X	x	x	x									
getBinaryStream														x	x	X									
getClob																				X					
getBlob																					X				
getArray																						X			
getRef																							X		
getCharacterStream											x	x	X	x	x	x									
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	X	X

Rys. 3.2. Metody do konwersji typów *SQL* na typy języka *Java* (*ResultSet.getXXX*) [12]
http://info.ee.pw.edu.pl/Java/1.4.2/docs/guide/jdbc/getstart/mapping.html



Rys. 3.3. Wynik działania serwletu *ListServlet*



Rys. 3.4. Definicja klasy modelu *CountryBean*

Korzystając z klasy ***CountryBean*** oraz mechanizmu sesji, zmodyfikuj serwlet ***ListServlet*** tak, aby przekazywał dane o krajach (w obiekcie sesji) do strony widoku ***countryList.jsp***. W sesji, pod kluczem "***list***" zapisz listę ***ArrayList*** zawierającą obiekty typu ***CountryBean***. W serwlecie pozostaw **tylko** instrukcje konieczne do pracy z bazą danych i obiektem sesji. Samym wyświetlaniem listy krajów z obiektu sesji zajmie się już strona widoku *JSP*.

W serwlecie **nie twórz** już i nie korzystaj z obiektu ***out***, nie potrzebny jest też cały blok ***try-catch***, który w zadaniu 3.2 wykorzystywano do tworzenia treści odpowiedzi):

```
HttpSession session=request.getSession(true);
CountryBean country;
ArrayList<CountryBean> list=new ArrayList<CountryBean>();
  while (rs.next()) {
     country = new CountryBean();
     //pobierz dane z odpowiedniej kolumny
     //przypisz je do właściwości obiektu CountryBean
     country.setName(rs.getString("name"));
     // …
     list.add(country);
  }
session.setAttribute("list", list);
```

Następnie w serwlecie dodaj przekierowanie do strony *JSP*:

```
response.sendRedirect("countryList.jsp");
```

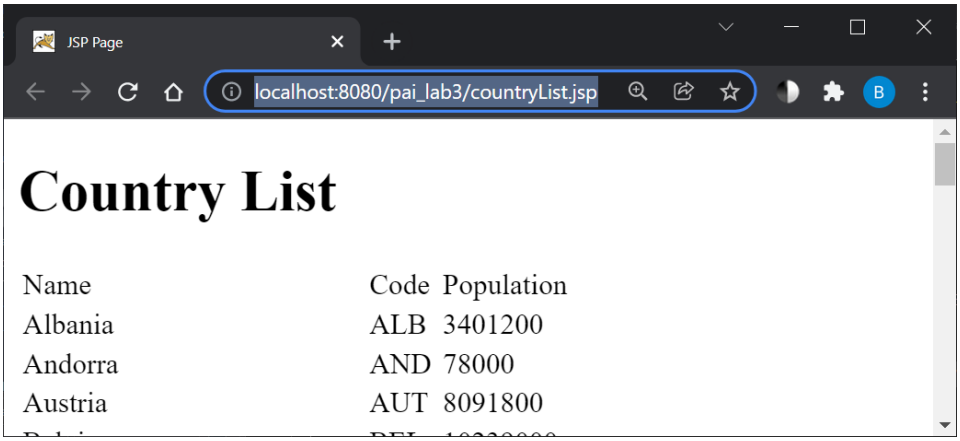
Do projektu dodaj nową stronę widoku ***countryList.jsp*** oraz wyświetl tam dane pobrane z obiektu sesji (Rys. 3.5):

```
<% ArrayList<CountryBean> list =
 (ArrayList<CountryBean>)session.getAttribute("list");
%>
```

Do wyświetlenia danych na stronie *JSP* skorzystaj z pętli:

```
for(CountryBean country:list){}
```

Uruchom ponownie ***ListServlet*** (Rys. 3.5) i zwróć uwagę na adres *URL*, który jest widoczny w przeglądarce po jego uruchomieniu (porównaj z adresem z rysunku 3.3).

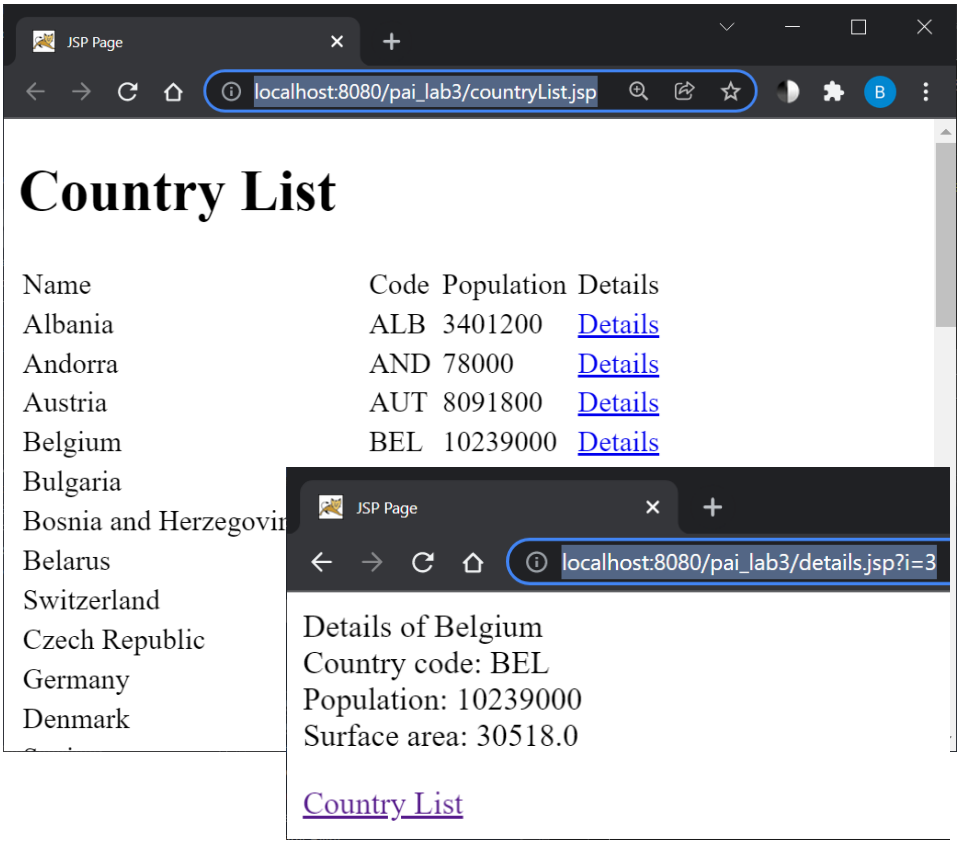


Rys. 3.5. Widok wygenerowany przez stronę *countryList.jsp* po uruchomieniu serletu *ListServlet*

W zmodyfikowanej aplikacji, serwlet zajmuje się logiką operacji na danych a strona *JSP* wyświetla przygotowane przez serwlet dane, korzystając z modelu ***CountryBean***.

Zadanie 3.4. Identyfikator obiektu w adresie *URL*

Zmodyfikuj stronę *JSP* tak, aby obok każdego kraju wyświetlane było hiperłącze prowadzące do szczegółowych informacji o wybranym państwie (Rys. 3.6). W hiperłączu wskaż plik, do którego ma nastąpić przekierowanie, łącznie z przekazaniem ***indeksu*** wybranego państwa z listy (***list.indexOf(country)***) jako parametru do nowej strony ***details.jsp***. Na stronie ***details.jsp*** wyświetl pełniejsze informacje o wybranym kraju (***list.get(indeks)***). Do strony ***details.jsp*** dodaj także link prowadzący z powrotem do listy krajów (Rys. 3.6).



Rys. 3.6. Lista krajów europejskich z tabeli *Country* wyświetlona przez stronę *countryList.jsp*

Laboratorium 4. Wprowadzenie do *Spring MVC*

Cel zajęć

Realizacja zadań proponowanych w niniejszym laboratorium pozwoli studentom poznać podstawowe zasady tworzenia aplikacji internetowych z wykorzystaniem szkieletu programistycznego *Spring MVC* [6, 19, 25], na przykładzie aplikacji typu *CRUD* do zarządzania pracownikami, przechowywanymi w bazie *MySQL*. Zadania opracowano na podstawie [17]:
https://www.javatpoint.com/spring-mvc-crud-example

Zakres tematyczny

- Przygotowanie aplikacji internetowej typu *CRUD* z wykorzystaniem szkieletu programistycznego *Spring MVC*.
- Przygotowanie bazy danych *MySQL* do obsługi pracowników.
- Konfiguracja projektu *Spring MVC*.
- Implementacja kontrolera i komponentów danych z wykorzystaniem modułów *Spring MVC*.
- Przygotowanie stron widoków w *JSP*.
- Wykonywanie operacji na danych (*CREATE*, *READ*, *UPDATE* i *DELETE*) z wykorzystaniem biblioteki *JdbcTemplate*.
- Implementacja obsługi wyjątków w klasie kontrolera.

Wprowadzenie

Spring jest wielowarstwowym, modułowym, najpopularniejszym obecnie szkieletem programistycznym dla języka *Java* [6, 24]. Projekt w *Spring* tworzony jest z perspektywy aplikacji, a nie serwera. *Spring* promuje dobre praktyki programowania obiektowego i jest w pełni modułowy (można wykorzystać dowolną część *Spring*, niezależnie od pozostałych). Podstawowe moduły *Spring* to *Core* i *Beans*. Moduły te zapewniają podstawowe części struktury projektu.

Zasadniczym elementem projektu *Spring* są zwykłe klasy *POJO* (ang. *Plain Old Java Object*), czyli obiekty aplikacji, zarządzane poprzez kontener *Spring Core*, nazywane ziarnami (ang. *Beans*). Ziarno *Spring Bean* jest obiektem, który jest instancjonowany, montowany i zarządzany przez kontener. *Spring* nie nakłada na klasy *POJO* żadnych dodatkowych wymagań dotyczących dziedziczenia lub implementowania interfejsów. Kontener podstwowy *Spring Core Container* czyta metadane konfiguracyjne, które mogą być reprezentowane przez pliki

XML, adnotacje w klasach *Java* lub bezpośrednio w kodzie. Kontener *Spring* korzysta z klas *POJO* i metadanych konfiguracyjnych, do utworzenia gotowej aplikacji. Metadane konfiguracyjne określają obiekty, które zawierają aplikację oraz współzależności między tymi obiektami. Kontener *Spring Core* czyta metadane i na tej podstawie określa, które obiekty mają być instancjonowane, konfigurowane oraz montowane. Następnie kontener wstrzykuje zależności (ang. *dependency injection – DI*) podczas tworzenia ziarna.

Spring Web MVC to moduł *Spring* ukierunkowana na tworzenie aplikacji sieciowych, oparty na architekturze wzorca *MVC*. Dostarcza komponenty do tworzenia elastycznych i luźno połączonych aplikacji internetowych. Model łączy dane aplikacji, które definiowane są za pomocą klas *POJO*. Widok jest odpowiedzialny za renderowanie danych modelu i generowanie wyjściowego *HTML*. Kontroler jest odpowiedzialny za przetwarzanie żądań użytkownika, budowę odpowiedniego modelu i przekazywanie go do renderowania przez widok. Moduł *Web* zapewnia podstawowe funkcje zorientowane na sieć, inicjalizuje kontener *Spring* z wykorzystaniem servletów i kontekstu aplikacji webowej.

Zadanie 4.1. Przygotowanie bazy danych *MySQL*

Na serwerze *MySQL* w bazie danych o nazwie ***test*** utwórz tabelę ***pracownik*** (Rys. 4.1), korzystając z kodu *SQL* z przykładu 4.1.

#	Nazwa	Typ	Metoda porównywania napisów	Atrybuty	Null	Ustawienia domyślne
1	**id** 🔑	int(11)		UNSIGNED	Nie	*Brak*
2	**nazwisko**	varchar(50)	utf8_general_ci		Nie	*Brak*
3	**pensja**	double		UNSIGNED	Nie	0
4	**firma**	varchar(400)	utf8_general_ci		Nie	*Brak*

Rys. 4.1. Tabela *pracownik* w bazie *test*

Przykład 4.1. Kod SQL do utworzenia tabeli pracownik

```
--
-- Struktura tabeli `pracownik`
--
```

```
CREATE TABLE `pracownik` (
  `id` int(11) UNSIGNED NOT NULL AUTO_INCREMENT,
  `nazwisko` varchar(50) NOT NULL,
  `pensja` double UNSIGNED NOT NULL DEFAULT '0',
  `firma` varchar(400) NOT NULL,
   PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Zadanie 4.2. Utworzenie projektu *Spring MVC*

Utwórz nowy projekt aplikacji webowej o nazwie ***pai_spring*** (*File → New Project → Java with Maven/Web Application)*. W wygenerowanym pliku konfiguracyjnym ***pom.xml*** zmień wersję *JEE* 6 na *JEE* 7 (jeśli trzeba):

```
<dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId>
    <version>7.0</version>
</dependency>
```

W elemencie **<build>** dla elementu **<plugin>** zmodyfikuj element **<configuration>**:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.3.2</version>
           <configuration>
               <source>1.7</source>
               <target>1.7</target>
               <compilerArguments>
                   <endorseddirs>${endorsed.dir}</endorseddirs>
               </compilerArguments>
           </configuration>
</plugin>
```

W kolejnym elemencie **<plugin>** (jeśli trzeba) zmień wersję *javaee-endorsed-api* z 6.0 na 7.0:

```
<artifactItem>
    <groupId>javax</groupId>
    <artifactId>javaee-endorsed-api</artifactId>
    <version>7.0</version>
    <type>jar</type>
</artifactItem>
```

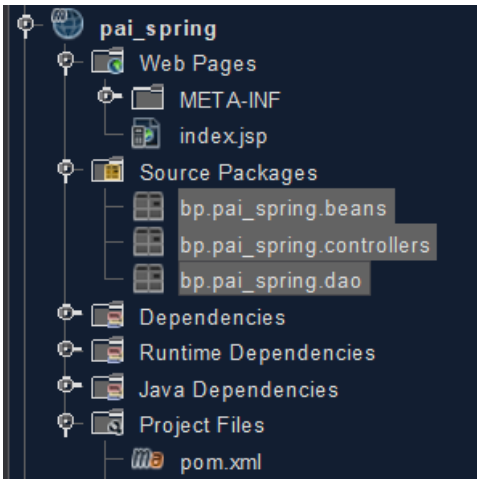
Następnie dodaj do pliku ***pom.xml*** kolejne zależności (bibliotekę *spring-webmvc*, sterownik *mysql-connector-java* i *spring-jdbc* do pracy z *MySQL* oraz bibliotekę znaczników *jstl*) (Przykład 4.2).

Przykład 4.2. Dodatkowe zależności w pliku pom.xml

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-
webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.11</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc
-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
```

Po tych modyfikacjach zbuduj projekt: *Run→Clean and Build Main Project*.
Następnie w pakiecie (tutaj o nazwie ***bp.pai_spring***) utwórz 3 kolejne pakiety (*New → Java package*) o nazwach: ***beans***, ***controllers*** i ***dao*** tak, jak przedstawiono na rysunku 4.2. Przy tworzeniu zwróć uwagę, żeby pakiety te były **bezpośrednimi podpakietami** pakietu projektu (np. ***bp.pai_spring***).

Do utworzenia funkcjonalnej aplikacji webowej typu *CRUD* należy wykonać kilka podstawowych kroków, pokazanych w punktach 4.2.1–4.2.5.



Rys. 4.2. Struktura projektu z dodanymi pakietami

4.2.1. Klasa *Pracownik* jako *POJO*

Do pakietu ***beans*** dodaj definicję klasy o nazwie ***Pracownik***, której obiekt będzie odwzorowany na rekord w tabeli ***pracownik*** (Przykład 4.3). Do klasy dodaj odpowiednie metody publiczne ***get*** i ***set***.

Przykład 4.3. Definicja klasy Pracownik.java

```
package bp.pai_spring.beans;

public class Pracownik {
    private int id;
    private String nazwisko;
    private float pensja;
    private String firma;
    //dodaj metody get i set
}
```

4.2.2. Klasa *PracownikDao* do pracy z danymi

W pakiecie ***dao*** utwórz klasę ***PracownikDao*** (Przykład 4.4), której metody, korzystają z interfejsu ***JdbcTemplate*** i pomogą wykonać zapytania *SQL* do bazy danych ***test*** z tabelą ***pracownik***.

Przykład 4.4. Definicja klasy PracownikDao.java

```
package bp.pai_spring.dao;

import bp.pai_spring.beans.Pracownik;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import java.util.List;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
public class PracownikDao {
    JdbcTemplate template;

    public void setTemplate(JdbcTemplate template) {
        this.template = template; //wstrzyknięcie przez metodę set
    }
    public int save(Pracownik p) {
        String sql = "insert into pracownik (nazwisko,pensja,firma) "
                + "values('" + p.getNazwisko() + "'," + p.get…()
                + ",'" + p.get…() + "')";
        return template.update(sql);
    }
    public List<Pracownik> getAll() {
        return template.query("select * from pracownik",
                  new RowMapper<Pracownik>() {
            @Override
            public Pracownik mapRow(ResultSet rs, int row)
                    throws SQLException{
                Pracownik e = new Pracownik();
                e.setId(rs.getInt(1));
                e.setNazwisko(rs.getString(2));
                // ustaw właściwość pensja
                // ustaw właściwość firma
                return e;
            }
        });
    }
}
```

4.2.3. Klasa kontrolera *PracownikController*

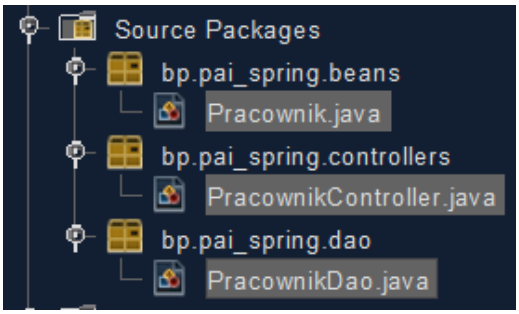
W pakiecie ***controllers*** utwórz klasę o nazwie ***PracownikController*** (Przykład 4.5), która będzie pełniła rolę kontrolera. Metody tej klasy (**akcje kontrolera**) będą obsługiwały odpowiednie żądania *HTTP*, wskazane w parametrach adnotacji ***@RequestMapping***.

Przykład 4.5. Definicja klasy PracownikController.java

```
package bp.pai_spring.controllers;

import bp.pai_spring.beans.Pracownik;
import bp.pai_spring.dao.PracownikDao;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class PracownikController {
    @Autowired
    PracownikDao dao; //wstrzyknięcie dao z pliku XML

/* Wynikiem działania metody jest przekazanie danych w modelu do
* strony widoku addForm.jsp, która wyświetla formularz
* wprowadzania danych, a „command” jest zastrzeżonym atrybutem
* żądania, umożliwiającym wyświetlenie danych obiektu pracownika
* w formularzu.
*/
    @RequestMapping("/addForm")
    public String showform(Model m){
        m.addAttribute("command", new Pracownik());
        return "addForm"; //przekiekrowanie do addForm.jsp
    }

/* Metoda obsługuje zapis pracownika do BD. @ModelAttribute
* umozliwia pobranie danych z żądania do obiektu pracownika.
* Jawnie wskazano RequestMethod.POST (domyślnie jest to GET)
*/
    @RequestMapping(value="/save",method =
      RequestMethod.POST)
    public String save(@ModelAttribute("pr") Pracownik pr){
        dao.save(pr);
        return "redirect:/viewAll";
        //przekierowanie do endpointa o URL: /viewAll
    }

/* Metoda pobiera listę pracowników z BD i umieszcza je w modelu */
    @RequestMapping("/viewAll")
    public String viewAll(Model m){
        List<Pracownik> list=dao.getAll();
        m.addAttribute("list",list);
        return "viewAll"; //przejście do widoku viewAll.jsp
    }
}
```

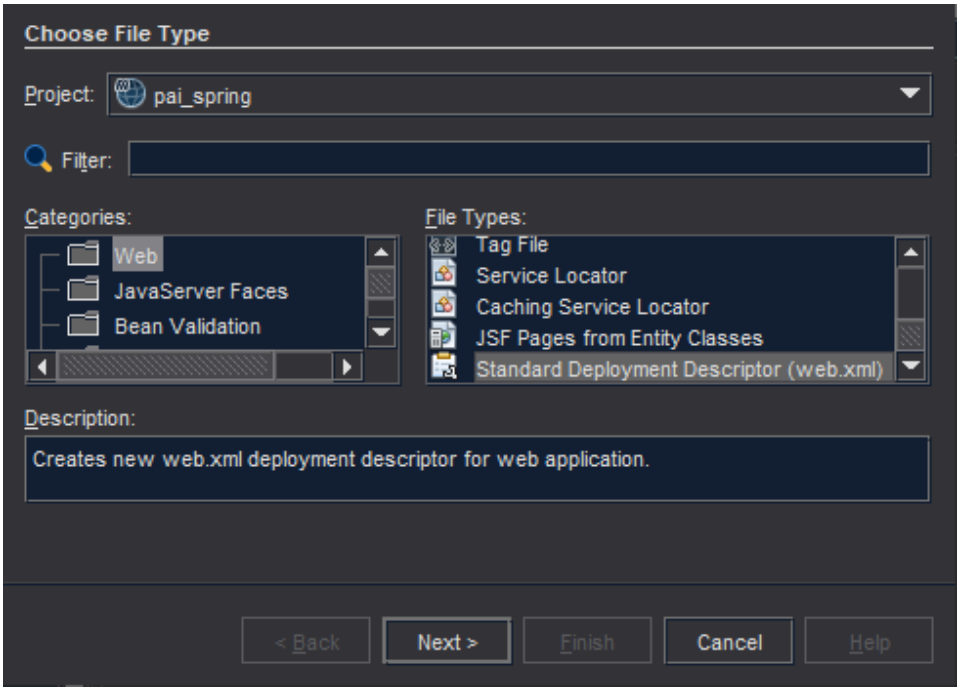
Struktura plików w projekcie po wykonaniu kroków z podrozdziałów 4.2.1–4.2.3 przedstawiona jest na rysunku 4.3.



Rys. 4.3. Pakiety z klasami *Pracownik*, *PracownikDao* i *PracownikController*

4.2.4. Konfiguracja w plikach *web.xml* i *spring-servlet.xml*

W folderze ***WEB-INF*** projektu utwórz (jeśli jeszcze nie istnieje) standardowy plik ***web.xml*** (deskryptor wdrożenia – Rys. 4.4) i dodaj do niego konfigurację punktu wejścia aplikacji – serwletu o nazwie ***spring***, jak pokazuje przykład 4.6 (kolorem zielonym oznaczono dodany fragment kodu).



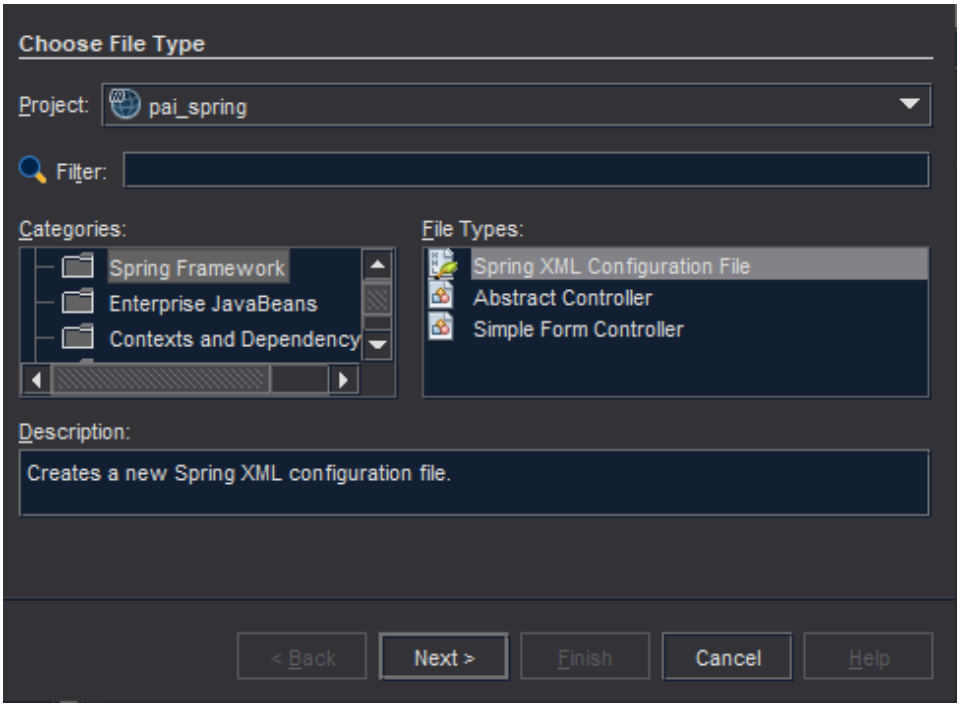
Rys. 4.4. Dodanie deskryptora wdrożenia *web.xml* w *NetBeans IDE*

Przykład 4.6. Uzupełniony plik web.xml

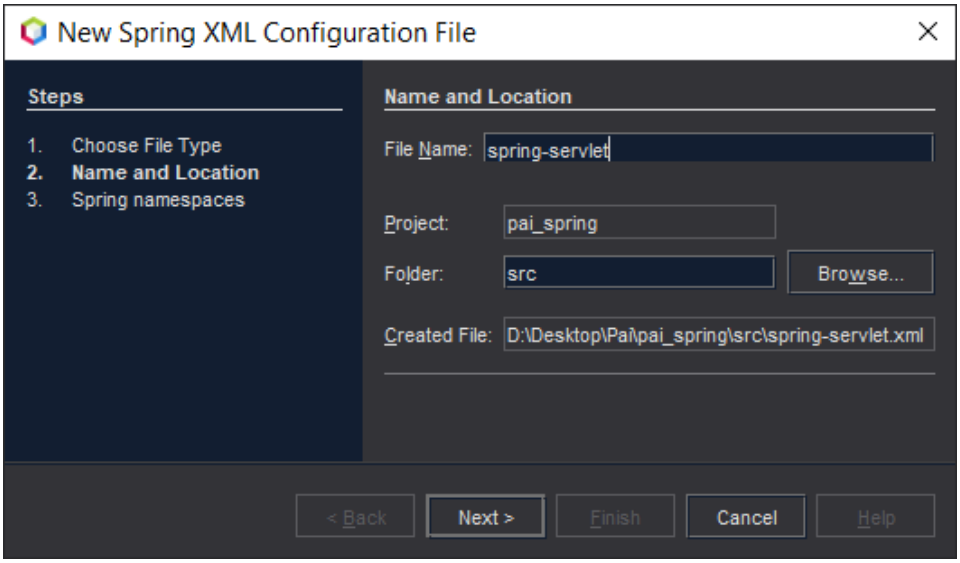
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
             http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>
          org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

W tym samym folderze utwórz także plik konfiguracyjny ***spring-servlet.xml*** serwletu (Rys. 4.5 i 4.6), w którym należy dodać definicję ziaren (`bean`).

Przykład 4.7 przedstawia prawidłowy, gotowy plik konfiguracyjny. Zwróć uwagę na właściwe wskazanie nazw pakietów w projekcie oraz na konfigurację ziarna z połączeniem do bazy danych (Rys. 4.5 i 4.6).



Rys. 4.5. Dodanie pliku konfiguracyjnego *spring-servlet.xml* w *NetBeans*



Rys. 4.6. Ustawienia pliku konfiguracyjnego *spring-servlet.xml*

Przykład 4.7. Konfiguracja ziaren w pliku spring-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
<context:component-scan base-package="bp.pai_spring.controllers">
</context:component-scan>
<bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver
">
<property name="prefix" value="/WEB-INF/jsp/"></property>
<property name="suffix" value=".jsp"></property>
</bean>

<bean id="ds"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"></property>
<property name="url"
value="jdbc:mysql://localhost:3306/test?serverTimezone=UTC"> </property>
<property name="username" value="root"></property>
<property name="password" value=""></property>
</bean>
<bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="ds"></property>
</bean>

<bean id="dao" class="bp.pai_spring.dao.PracownikDao">
<property name="template" ref="jt"></property>
</bean>
</beans>
```

4.2.5. Widoki *JSP*

Jako stronę startową aplikacji wykorzystaj istniejący (lub utwórz jeśli nie istnieje) plik ***index.jsp*** z zawartością jak podano w przykładzie 4.8. Zwróć uwagę, że adresy w hiperłączach są mapowane na odpowiednie metody kontrolera ***PracownikController***. Lokalizacja pliku ***index.jsp*** pokazana jest na rysunku 4.7.

Przykład 4.8. Strona index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
```

```
    <head>
        <meta charset="UTF-8">
        <title>Pracownicy</title>
    </head>
    <body>
        <div>
            <h3>Pracownicy</h3>
            <p>
                <a href="addForm">Dodaj pracownika</a><br />
                <a href="viewAll">Pokaż listę pracowników</a>
            </p>
        </div>
    </body>
</html>
```

Do projektu dodaj 2 dodatkowe pliki widoków *JSP* (w lokalizacji jak na rysunku 4.7):

- ***viewAll.jsp* –** tabela z listą pracowników (Przykład 4.9),
- ***addForm.jsp*** – strona z formularzem dodawania nowego pracownika (Przykład 4.10).

Strony *JSP* korzystają z języka wyrażeń *JSTL* oraz z bibliotek znaczników ***form*** i ***core***, które pozwoliły w prosty sposób wyświetlić dane przekazane z kontrolera do widoków w atrybutach modelu, dostępnych przez nazwę klucza. Klucze i wartości atrybutów zostały ustawione w odpowiednich akcjach kontrolera ***PracownikController***: ***showform()*** i ***viewAll()***. W kodzie strony ***viewAll.jsp,*** **przy każdym wierszu** zostały już dodane hiperłącza do wykonania kolejnych akcji: edycji i usunięcia wskazanego przez ***id*** obiektu (**do uzupełnienia w kolejnych zadaniach**).

Przykład 4.9. Strona viewAll.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://www.springframework.org/tags/form"
prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Pracownicy</title>
    </head>
 <body>
   <div>
     <h1>Lista pracowników</h1>
     <table border>
      <tr> <th>Id</th> <th>Nazwisko</th> <th>Pensja</th> <th>Firma</th>
           <th>Edytuj</th> <th>Usuń</th>
```

```
      </tr>
      <c:forEach var="pr" items="${list}">
          <tr>
               <td> ${pr.id} </td>
               <td> ${pr.nazwisko} </td>
               <td> ${pr.pensja} </td>
               <td> ${pr.firma} </td>
               <td><a href="edit/${pr.id}"> Edytuj </a></td>
               <td><a href="…"> Usuń </a></td>
          </tr>
      </c:forEach>
     </table>
     <br/>
     <a href="addForm">Dodaj nowego pracownika</a>
    </div>
 </body>
</html>
```

Przykład 4.10. Strona addForm.jsp

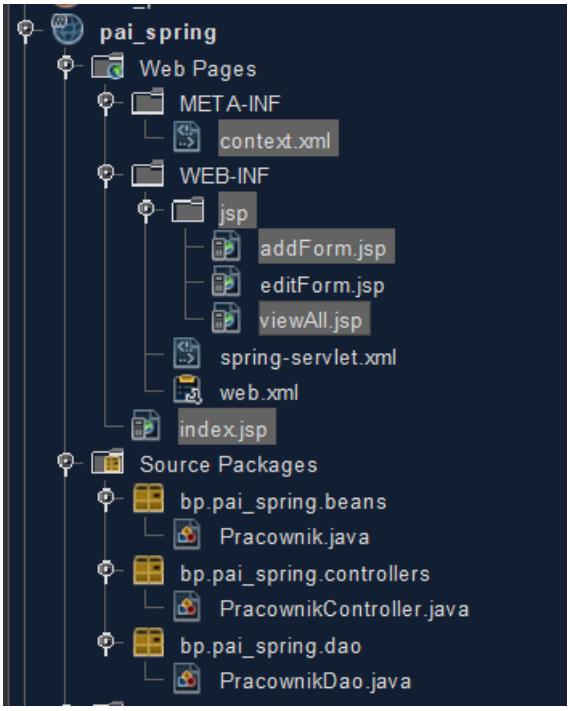
```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://www.springframework.org/tags/form"
    prefix="form"%>
<!DOCTYPE html>
<html>
  <head>
        <meta charset="UTF-8">
        <title>Pracownicy</title>
  </head>
  <body>
    <div>
     <h1>Dodaj dane nowego pracownika</h1>
     <form:form method="post" action="save">
      <table >
       <tr>
          <td>Nazwisko : </td>
          <td> <form:input path="nazwisko"  /> </td>
       </tr>
       <tr>
         <td>Pensja :</td>
         <td> <form:input path="pensja" /> </td>
       </tr>
       <tr>
         <td>Firma :</td>
         <td> <form:input path="firma" /> </td>
      </tr>
       <tr>
        <td> </td>
        <td> <input type="submit" value="Zapisz" /> </td>
       </tr>
      </table>
```

```
    </form:form>
   </div>
  </body>
</html>
```

UWAGA! Jeśli w projekcie nie istnieje folder ***META-INF***, to go utwórz (w tej samej lokalizacji co folder ***WEB-INF***), a w nim umieść plik ***context.xml*** z zawartością:

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/pai_spring"/>
```

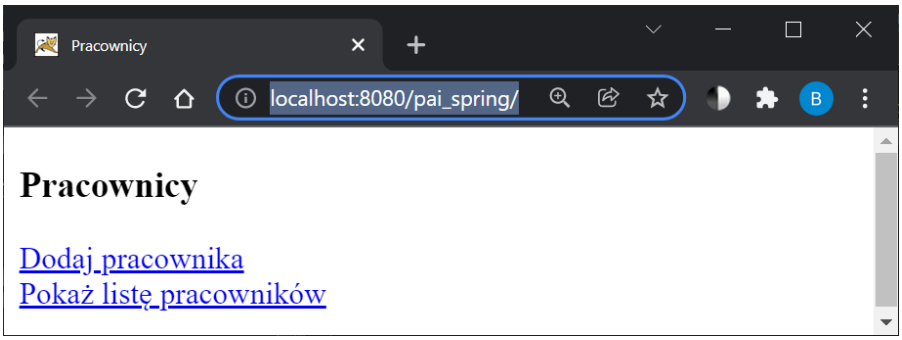
Struktura plików w gotowym projekcie pokazana jest na rysunku 4.7.



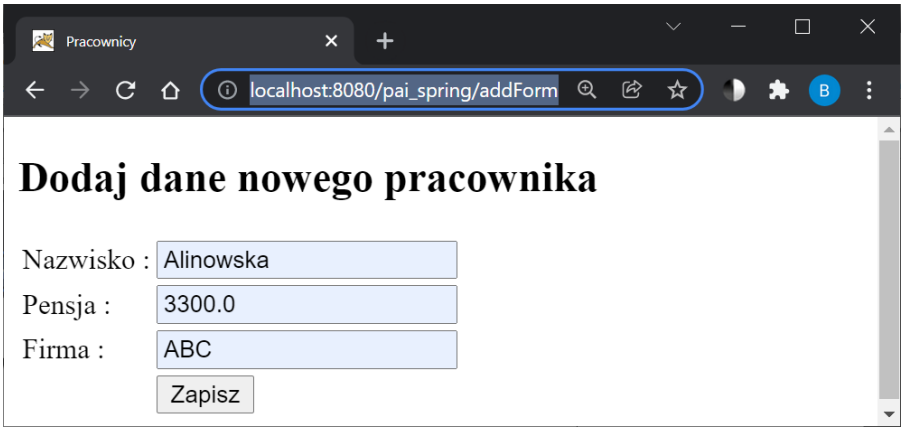
Rys. 4.7. Rozmieszczenie plików gotowego projektu

Zadanie 4.3. Zbudowanie i uruchomienie projektu

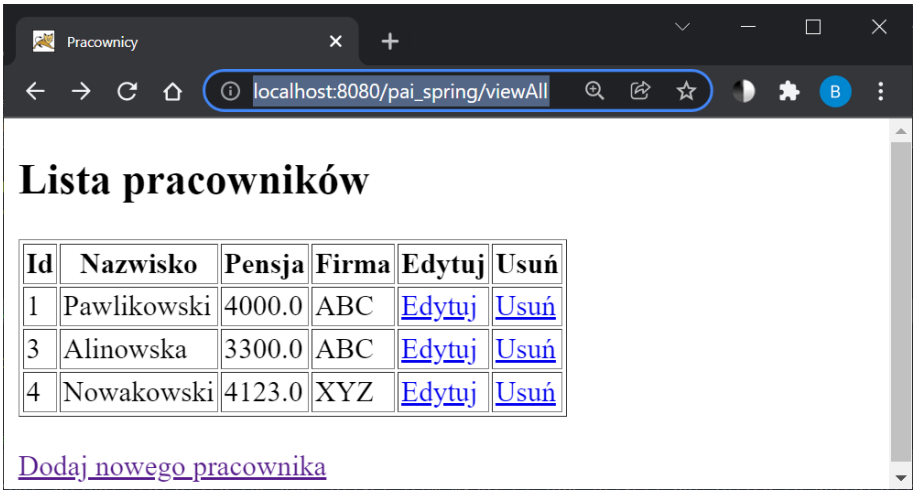
Zbuduj i uruchom projekt (*Run→Build Main Project*). Jeśli wszystko zostało poprawnie zrealizowane, to po uruchomieniu projektu w przeglądarce powinna pokazać się strona ***index.jsp*** (Rys. 4.8). Sprawdź efekt działania hiperłączy (Rys. 4.9) i poprawność pracy z bazą danych. Przeanalizuj kod klasy kontrolera i zwróć szczególną uwagę na jego metody obsługujące poszczególne żądania.



Rys. 4.8. Strona startowa projektu



Rys. 4.9. Formularz dodawania nowego pracownika



Rys. 4.10. Widok listy pracowników

Zadanie 4.4. Akcja kontrolera *DELETE*

Zaimplementuj możliwość usuwania wskazanego pracownika. W widoku listy pracowników istnieje już odpowiedni link do akcji kontrolera (sprawdź jaki).
W celu implementacji usunięcia pracowanika:

- do klasy ***PracownikDao*** dodaj metodę ***delete*** z atrybutem ***int id***,
- do klasy ***PracownikController*** dodaj metodę ***delete***, która obsłuży żądanie ***/delete/{id}.*** Nagłówek tej metody powinien mieć postać: `public String delete(@PathVariable int id) {}`, gdzie ***@PathVariable*** umożliwia pobranie parametru przekazanego w ścieżce z adresem *URL* żądania.

Przetestuj działanie akcji usunięcia pracownika.

Zadanie 4.5. Akcja kontrolera *EDIT*

Zaimplementuj możliwość edycji danych wskazanego pracownika. W widoku listy pracowników istnieje już odpowiedni link do akcji kontrolera (sprawdź jaki).

Wykonaj podobne operacje jak w przypadku dodawania nowego pracownika do bazy:

- Do klasy ***PracownikDao*** dodaj metodę ***update*** (analogiczną do metody ***save***, tylko z zapytaniem ***update***).
- Do klasy ***PracownikDao*** dodaj metodę ***getPracownikById()***:

```
public Pracownik getPracownikById(int id){
  String sql="select * from pracownik where id=?";
  return
   template.queryForObject(sql, new Object[]{id},
      new BeanPropertyRowMapper<>(Pracownik.class));
}
```

- Do klasy ***PracownikController*** dodaj metodę ***edit***, która obsłuży żądanie przesłane metodą *GET* ***/edit/{id}*** i przekaże do widoku ***editForm*** obiekt pracownika o podanym ***id***.
- Do klasy ***PracownikController*** dodaj metodę ***editsave***, która obsłuży przychodzące żądanie ze zmodyfikowanymi danymi pracownika, przesłane z formularza ***editForm*** metodą *POST* (sprawdź, jak działa metoda ***save***).
- Utwórz stronę widoku ***editForm.jsp*** do edycji danych pracownika, analogicznie jak w ***addForm.jsp,*** tylko dodatkowo do formularza dodaj ukryte pole do przekazania ***id***:

```
<form:hidden path="id" />
```

Przetestuj działanie akcji edycji danych pracownika.

Zadanie 4.6. Obsługa wyjątków w kontrolerze

W poprzednich zadaniach nie było jeszcze kontroli poprawności danych przesyłanych w żądaniu z formularza. Sprawdź, jaki będzie efekt po próbie zapisania danych pracownika z pensją wpisaną jako łańcuch, np. "abc". W takim przypadku nastąpi problem związany z pojawieniem się niekontrolowanego wyjątku.

Aby tego uniknąć, przygotuj strony widoków do obsługi błędów (umieść je w folderze ***jsp*** razem z istniejącymi już tam widokami) oraz w kontrolerze dodaj specjalną metodę, poprzedzoną adnotacją ***@ExceptionHandler***.
W zależności od potrzeby, metoda taka może mieć postać prezentowaną w przykładzie 4.11 lub 4.12.

Przykład 4.11. Prosta obsługa wyjątku przez metodę kontrolera

```
//Metoda zwraca nazwę widoku, wykorzystanego do pokazania komunikatu
//o błędzie (sam obiekt Exception nie jest dostępny w widoku)
@ExceptionHandler({Exception.class}) //tu można wymienić wyjątki
public String error() {
        return "errorpage";
}
```

Przykład 4.12. Obsługa błędu z przekazaniem modelu do widoku

```
//Totalna kontrola - ustawienie danych o błędzie w modelu oraz
//zwrócenie nazwy widoku i modelu w obiekcie ModelAndView
@ExceptionHandler(Exception.class)
public ModelAndView handleError(HttpServletRequest req,
                                Exception ex) {
        ModelAndView mav = new ModelAndView();
        mav.addObject("exception", ex);
        mav.addObject("url", req.getRequestURL());
        mav.setViewName("error");
        return mav;
}
```

Przykład 4.13 przedstawia stronę widoku ***error.jsp***, która pobiera dane przekazane w metodzie kontrolera z przykładu 4.12 w obiekcie ***ModelAndView***.

Przykład 4.13. Strona error.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
```

```
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" >
  <title>JSP Page</title>
 </head>
 <body>
  <h1>Error Page</h1>
  <p>Failed URL: ${url}
     Exception:  ${exception.message}
     <c:forEach items="${exception.stackTrace}" var="ste">
              ${ste}
     </c:forEach>
  </p>
 </body>
</html>
```

Laboratorium 5. *Spring Boot* i *JPA*

Cel zajęć

Realizacja zadań z niniejszego laboratorium umożliwi studentom poznanie elementów konfiguracji projektu *Spring* z wykorzystaniem startera *Spring Boot* [4, 18] i metod pracy z bazą danych za pomocą repozytorium *JPA* [7, 16, 22].

Zakres tematyczny

- Konfiguracja projektu *Spring Boot* (plik konfiguracyjny ***pom.xml*** oraz ***application.properties***, adnotacje do konfiguracji stosowane w startowej klasie projektu).
- Implementacja aplikacji współpracującej z danymi z bazy *H2* (lub *MySQL*) za pomocą interfejsu *Spring Data JPA* i *Hibernate*:
    - definicja klasy encji do obsługi zadań, podstawowe adnotacje w klasie encji i konfiguracja dostępu do bazy danych,
    - definicja interfejsu repozytorium dziedziczącego po interfejsie ***CrudRepository***,
    - wykorzystanie gotowych metod repozytorium do obsługi zadań w aplikacji typu *CRUD* (dodawanie nowych zadań, wyświetlanie listy istniejących, usuwanie wskazanego zadania),
    - implementacja metod wyszukiwania danych za pomocą zapytań wbudowanych lub metod z adnotacją ***@Query***.

Wprowadzenie

Projekt *Spring Boot* ułatwia start z projektem *Spring* i eliminuje on potrzebę pracochłonnego tworzenia konfiguracji w plikach *XML*. Gotową aplikację można utworzyć i uruchomić za pomocą jednej klasy startowej. Dzięki o wiele prostszej konfiguracji, *Spring Boot* nadaje się do projektów studenckich i szybkiego prototypowania aplikacji.

Plikiem konfiguracyjnym środowiska *Spring Boot* jest plik ***application.properties***. Większość konfiguracji tworzy się przy pomocy adnotacji *Spring Boot*, pozostała część znajduje się w pliku ***application.properties***. W *Spring Boot* można korzystać z adnotacji auto-konfiguracji, skanowania w poszukiwaniu komponentów oraz z możliwości definiowania dodatkowych konfiguracji dla klasy startowej aplikacji, poprzedzonej specjalną adnotacją ***@SpringBootApplication***.

Praca z danymi zarówno w projektach *Spring*, jaki i *Spring Boot*, może być realizowana na różne sposoby (podstawowy interfejs *JDBC*, biblioteka *JdbcTemplat*e, gotowe interfejsy repozytoriów). Interfejs *JDBC* i *JdbcTemplate* wykorzystano w poprzednich laboratoriach. Nie są to jednak rozwiązania idealne, ponieważ trzeba ręcznie budować zapytania *SQL*, które często się powtarzają, szczególnie w przypadku podstawowych zapytań typu *CRUD*.

Hibernate jest rozwiązaniem tego problemu, ponieważ:

- pozwala automatycznie mapować obiekty języka *Java* na wiersze w bazie danych,
- pozwala odczytywać rekordy z bazy danych i automatycznie tworzyć z nich obiekty w języku *Java*.

Wykorzystując *Hibernate* teoretycznie nie trzeba mieć większego pojęcia o poprawnym konstruowaniu zapytań w języku *SQL*, ponieważ *Hibernate* buduje je sam. *Hibernate* jest najpopularniejszą biblioteką *ORM* do mapowania obiektowo-relacyjnego w Javie. Specyfikacją *JEE* do mapowania obiektowo-relacyjnego jest *JPA* (*Java Persistance API*). Specyfikacja oznacza, że nie jest to żadna biblioteka, a jedynie zbiór definicji i interfejsów, utworzonych przez grono ekspertów, których celem było wprowadzenie do Javy standardu mapowania obiektowo relacyjnego. Główną implementacją standardu *JPA* (*Java Persistence API*) jest obecnie właśnie *Hibernate*. Z kolei *Spring Data JPA* to niewielka biblioteka upraszczająca pracę z *JPA* poprzez automatyczne tworzenie kodu repozytoriów. Centralnym interfejsem dostarczanym przez bibliotekę jest ***CrudRepository*** i rozszerzający go interfejs ***JpaRepository***.

Zadanie 5.1. Konfiguracja projektu w *Spring Boot*

Utwórz projekt *Maven* **zwykłej** aplikacji (*New Project → Java with Maven →Java Application*) o nazwie np. ***pai_springboot***. Do pliku ***pom.xml*** w folderze ***Project Files*** dodaj element **<*parent*>** oraz zależność z informacją o tworzonej aplikacji *Web* z wykorzystaniem startera ***Spring Boot***. Przy dodawaniu zależności nie podawaj teraz wersji, gdyż wersje poszczególnych artefaktów wyspecyfikowane są w projekcie nadrzędnym, zdefiniowanym jako element **<*parent*>.** Gotowy plik ***pom.xml*** przedstawia przykład 5.1 (z *Java* v15, gdzie wersję Javy wskazano kolorem czerwonym).

Przykład 5.1. Plik pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.bp</groupId>
    <artifactId>SpringBoot1</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>15</maven.compiler.source>
      <maven.compiler.target>15</maven.compiler.target>
    </properties>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.5</version>
    </parent>
    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
    </dependencies>
</project>
```

Po uzupełnieniu ***pom.xml***, zbuduj projekt i sprawdź, jakie zależności zostały do niego dodane (folder ***Dependencies***). Następnie w głównym pakiecie projektu (w tym przykładzie ***bp.pai_springboot***) utwórz główną klasę ***Main,*** której zadaniem będzie uruchomienie aplikacji i obsługa żądań *HTTP* (Przykład 5.2).

Przykład 5.2. Klasa Main

```
package bp.pai_springboot;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@EnableAutoConfiguration
public class Main {
   public static void main(String[] args) {
        SpringApplication.run(Main.class, args);
    }

    @RequestMapping("/")
    @ResponseBody
    public String mainPage() {
        return "Hello Spring Boot!";
    }
}
```

Utworzona klasa ***Main*** posiada dwie adnotacje:

- ***@EnableAutoConfiguration*** z ***org.springframework.boot.autoconfigure***, dzięki której aplikacja dokona samokonfiguracji według domyślnych wartości, załaduje potrzebne moduły itp.,
- ***@Controller*** z pakietu ***org.springframework.stereotype*** informuje, że klasa obsługuje żądania *HTTP*.

Ważnym składnikiem tej klasy jest zwykła metoda ***main()***, wywoływana na początku każdego programu. W niej aplikacja uruchamiana jest za pomocą jednej linii:

```
SpringApplication.run(Main.class, args);
```

Drugą metodą klasy ***Main*** jest metoda ***mainPage()*** z dwoma adnotacjami:

- ***@RequestMapping("/")*** wskazuje, że żądanie z przeglądarki o stronę główną będzie obsługiwała właśnie ta metoda,
- ***@ResponseBody*** wskazuje, że metoda zwróci ciało odpowiedzi, przesłane do przeglądarki w formacie tekstu (w tym przypadku jest to łańcuch ***String*** z napisem *Hello Spring Boot*).

Aby uruchomienie aplikacji *Spring Boot* nie stwarzało problemów, w pliku ***pom.xml*** warto wskazać klasę główną. W tym celu w istniejącym już elemencie **<*properties*>** dodaj wpis:

```
<start-class>bp.pai_springboot.Main</start-class>
```

wskazujący na pakiet i klasę startową z metodą ***main()*.**

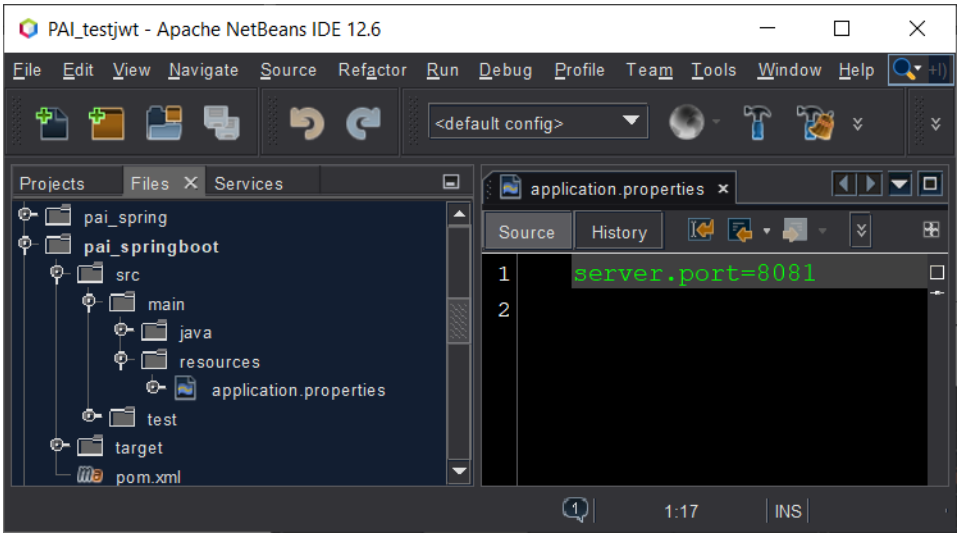
Uruchom projekt, co spowoduje także próbę uruchomienia serwera *TomcatEmbeddedServletContainer on port(s): 8080 (http). Spring Boot* uruchamia serwer *Tomcat* (*Embedded Tomcat*) za nas i wdraża tam aplikację. Dzięki temu nie trzeba już korzystać z serwera zewnętrznego (jak do tej pory), ponieważ konfiguracja startera *Spring Boot* dostarcza wbudowany serwer *Tomcat*, na którym aplikacja jest wdrażana i uruchamiana. Jeśli próba uruchomienia skończy się komunikatem o błędzie (konflikt z portami): *„Identify and stop the process that's listening on port 8080 or configure this application to listen on another port.”*, to należy skonfigurować serwer tak, aby nasłuchiwał na innym porcie niż domyślny *8080*.

SpringBoot korzysta z dwóch podstawowych plików konfiguracyjnych:

- ***pom.xml*** – znany już, podstawowy plik konfiguracyjny dla projektu *Maven*,
- ***application.properties*** – plik konfiguracyjny do wprowadzenia dodatkowych ustawień.

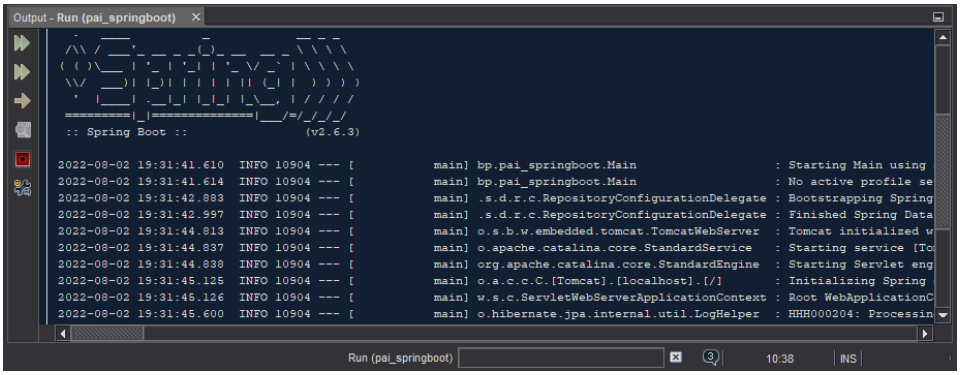
Utwórz plik konfiguracyjny ***application.properties*** (nowy plik z kategorii *Other* i *Empty file*), w którym będą dodawane kolejne elementy związane z ustawieniami aplikacji. Taki plik najlepiej jest umieścić w katalogu ***resources*** (Rys. 5.1 – widok w zakładce ***Files***). Jeśli katalog ***resources*** nie istnieje, to go utwórz. W ***application.properties*** dodaj jedno polecenie:

```
server.port=8081
```



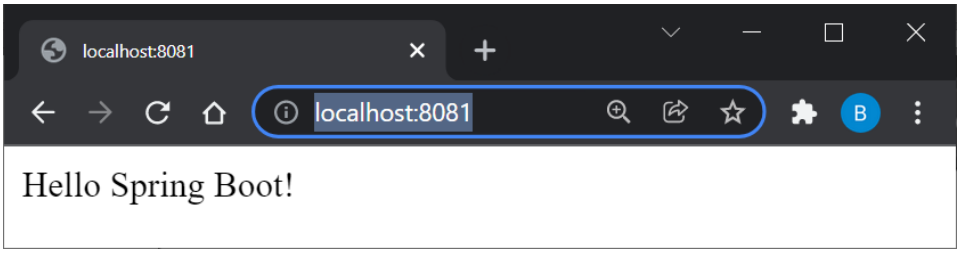
Rys. 5.1. Lokalizacja pliku *application.properties*

Po prawidłowym zbudowaniu i uruchomieniu aplikacji, wynik w oknie *Output* powinien być podobny do tego z rysunku 5.2.



Rys. 5.2. Komunikaty dla prawidłowo uruchomionego projektu *Spring Boot*

Po wpisaniu w oknie przeglądarki adresu *URL*: *http://localhost:8081/* widok strony w przeglądarce powinien być postaci jak na rysunku 5.3.



Rys. 5.3. Efekt działania pierwszej aplikacji *Spring Boot*

5.1.1. Oddzielenie kontrolera

W przypadku bardzo prostej aplikacji takie podejście z klasą ***Main*** i dodatkowymi metodami, obsługującymi żądania *HTTP*, sprawdza się, jednak pomieszane są tutaj różne warstwy aplikacji. Aby kod był czytelniejszy, należy wydzielić metody do obsługi żądań *HTTP* do oddzielnej klasy kontrolera (jak w poprzednim laboratorium).

W pakiecie projektu utwórz pakiet c***ontrollers*** na kontrolery i zdefiniuj tam klasę ***PageController,*** która powinna zawierać część kodu z klasy ***Main*** i dodatkową metodę do obsługi adresu, np.: ***/hello*** (Przykład 5.3).

Przykład 5.3. Klasa PageController w pakiecie controllers

```
package bp.pai_springboot.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class PageController {

    @RequestMapping("/")
    @ResponseBody
    public String mainPage() {
        return "Hello Spring Boot from mainPage() method!";
    }
    @RequestMapping("/hello")
    @ResponseBody
    public String pageTwo() {
        return "Hello Spring Boot from pageTwo() method!";
    }
}
```

W pliku z klasą ***Main*** usuń fragment kodu przeniesiony do kontrolera oraz adnotację ***@Controller*** i zbędne deklaracje importu, a z kolei dodaj adnotację ***@ComponentScan***, która każe aplikacji przeszukiwać inne klasy w poszukiwaniu adnotacji, takich jak np. ***@Controller***. Jeśli klasy są w innych pakietach, należy je wskazać w atrybucie adnotacji, przykładowo:

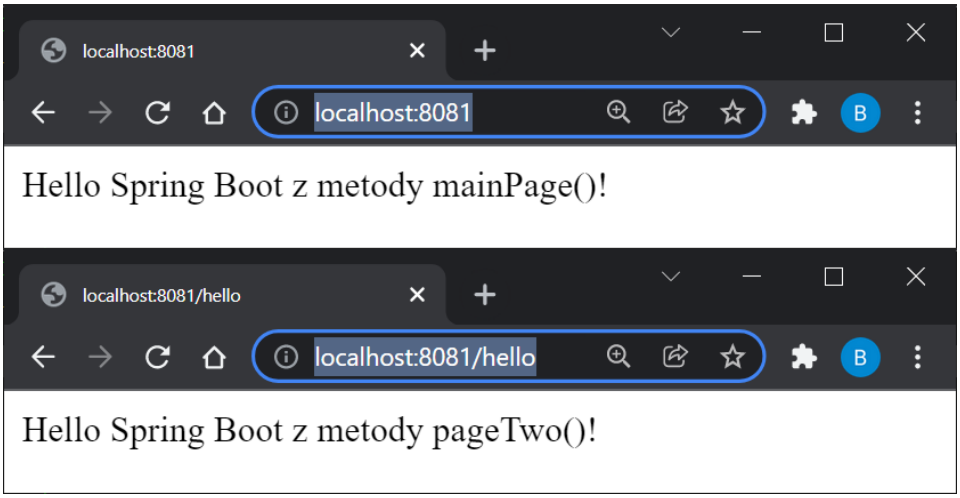
```
@ComponentScan({"bp.pai_springboot.controllers",
                "bp.pai_springboot.innypakiet"})
```

Klasę ***Main*** po zmianach przedstawia przykład 5.4.

Przykład 5.4. Klasa Main po modyfikacji

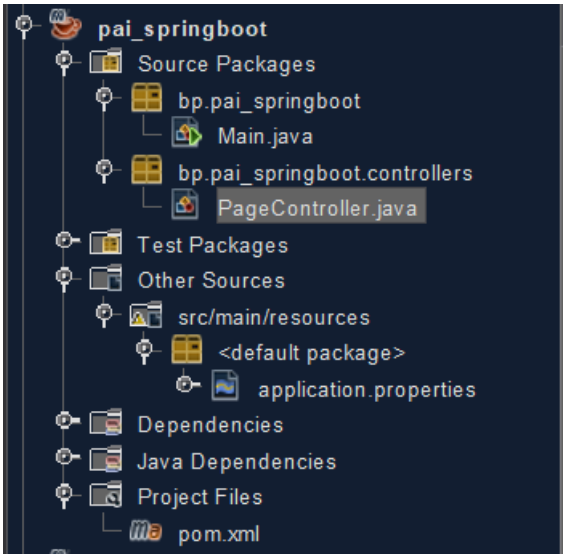
```
package bp.pai_springboot;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;

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

Po zatrzymaniu serwera *Tomcat* (*Run* → *Stop Build/Run*) i ponownym uruchomieniu aplikacji, w przeglądarce można podejrzeć wynik działania dwóch akcji, które zostały zdefiniowane w kontrolerze (Rys. 5.4).



Rys. 5.4. Wynik działania dwóch metod kontrolera

Strukturę plików projektu pokazano na rysunku 5.5.



Rys. 5.5. Końcowa struktura projektu

5.1.2. Eksport aplikacji do pliku *war* lub *jar*

Aplikacja może być wyeksportowana do pliku ***war*** tak, aby można było ją uruchomić na zewnętrznym serwerze *Tomcat*. Jednak ciekawą opcją jest utworzenie tak zwanego ***fatJar***, czyli pliku ***jar*** z wszystkimi zależnościami i kontenerem *Tomcat*, który można uruchomić przy pomocy polecenia:

```
java -jar plik.jar
```

W tym celu dodaj (po sekcji zależności `<dependencies>`) do pliku ***pom.xml*** wpis:

```
<build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
</build>
```

Uruchomienie projektu spowoduje teraz utworzenie odpowiedniego pliku ***jar*** w katalogu ***target***. Więcej na ten temat można znaleźć na stronie: *http://docs.spring.io/spring-boot/docs/current/reference/html/build-tool-plugins-maven-plugin.html*.

Zadanie 5.2. *Spring Boot* i *Spring Data JPA*

Spring Data to jeden z kluczowych dodatków do *Spring*, który umożliwia integrację aplikacji z bazą danych. Domyślna implementacja bazuje na *Hibernate*.

W tym zadaniu zostanie utworzona aplikacja typu *CRUD*, współpracująca z bazą danych *H2* za pomocą *Spring Data JPA*. Potrzebne będą:

- dodatkowe zależności,
- klasa encji, czyli klasa *POJO* z odpowiednimi adnotacjami, która zostanie wykorzystana do utworzenia tabeli w bazie danych,
- konfiguracja bazy danych,
- repozytorium, czyli interfejs z definicjami operacji, które można wykonać na klasie encji (dodawanie, pobieranie, usuwanie, modyfikowanie, wyszukiwanie).

5.2.1. Dodatkowe zależności

Spring Data JPA łączy się z wieloma różnymi bazami danych (typu *embedded* jak *H2*, relacyjnymi jak *MySQL*, *Oracle* itp.). Najszybszym jednak sposobem na sprawdzenie działania kodu w akcji jest użycie bazy ***H2*** [1] w trybie *memory*. Jest ona uruchamiana wraz z aplikacją i przetrzymywana w całości w pamięci (domyślnie), więc po zrestartowaniu aplikacji – baza zostanie usunięta. Baza ta może być również zapisywana do pliku. Aby dodać bazę *H2* do projektu, wystarczy dodać następującą zależność do pliku ***pom.xml***:

```
<dependency>
          <groupId>com.h2database</groupId>
          <artifactId>h2</artifactId>
</dependency>
```

Dodatkowo *Spring* musi wiedzieć, że aplikacja używa interfejsu *JPA*. W tym celu należy dodać do pliku ***pom.xml*** kolejną zależność:

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

UWAGA! Zawsze pamiętaj o przebudowaniu projektu po dodaniu dodatkowych zależności do pliku ***pom.xml*** lub dokonaniu innych modyfikacji.

5.2.2. Klasa encji

Następny krok to utworzenie klasy encji, która będzie odwzorowana na rekord w tabeli bazy danych. Każde pole obiektu klasy encji będzie odwzorowane jako oddzielna kolumna w tabeli w bazie. W klasie encji można także opisać relacje pomiędzy obiektami, które zostaną odwzorowane w bazie danych za pomocą kluczy obcych. Instancja obiektu jest odwzorowywana w bazie danych jako pojedynczy wiersz (rekord) w odpowiedniej tabeli bazy danych.

Utworzenie klas encji spowoduje, że *Spring* sam (za pośrednictwem *Hibernate*) przy uruchomieniu aplikacji, połączy się ze wskazaną w konfiguracji bazą danych (domyślnie *H2* w trybie memory), sprawdzi strukturę bazy i dokona potrzebnych modyfikacji (utworzy nieistniejące tabele, doda pola do tabel itp.).

W głównym folderze projektu utwórz kolejny pakiet ***entities*** (lokalizacja jak na rysunku 5.8), a w nim klasę encji ***Zadanie***. Klasa posiada wszystkie pola prywatne: ***nazwa***, ***opis***, ***budżet*** i ***czy zostało wykonane*** (Przykład 5.5).

UWAGA! Importy klas wskazanych jako adnotacje ***@Entity***, ***@Column***, ***@Id***, ***@GeneratedValue*** powinny pochodzić z pakietu ***javax.persistence***.

Przykład 5.5. Klasa encji Zadanie

```
package bp.pai_springboot.entities;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;

@Entity
public class Zadanie{
   @GeneratedValue
   @Id
   private Long id;

   @Column
   private String nazwa;

   @Column
   @Lob
   private String opis;
```

```
    @Column
    private Double koszt;

    @Column
    private Boolean wykonane=false;

    public Zadanie() {
         this.koszt = 2000.0;
         this.nazwa="Zadanie";
         this.opis="Zadanie do wykonania";
    }

    //nadpisana metoda toString
    @Override
    public String toString() {
         return "Encja Zadanie{ id=" + id + ", " + nazwa + ", " +
         opis + ", koszt=" + koszt + ", wykonane=" + wykonane +
          "}";
    }
    //dodaj metody get i set
}
```

Do klasy ***Zadanie*** dodaj jeszcze metody ***get*** i ***set***.

W klasie encji ***Zadanie***:

- ***@Entity*** – adnotacja informuje, że obiekt klasy będzie odwzorowany na rekord tabeli w bazie danych.
- ***@Id*** – definiuje pole, które będzie odwzorowane na klucz główny (unikatowy identyfikator rekordu) w tabeli, w przykładzie występuje z adnotacją ***@GeneratedValue,*** co oznacza, że wartość ta powinna zostać wygenerowana automatycznie.
- ***@Column*** – informuje, że pole jest kolumną tabeli. Adnotację tę można pominąć, jeśli nazwa pola klasy i kolumny w bazie danych są takie same. Jeśli są różne, to dodaje się atrybut ***name*** jako parametr adnotacji.
- ***@Lob*** – informuje, że w tym polu będą przechowywane duże obiekty (***L****arge* ***OB****ject*). Na przykład, pole typu *String* z samą adnotacją ***@Column*** w *SQL* zostanie utworzone domyślnie jako *VARCHAR (255)*. Jeżeli dołączona zostanie adnotacja ***@Lob*** pole to będzie typu ***TEXT***.

5.2.3. Konfiguracja bazy danych

Po uruchomieniu aplikacji struktura bazy danych zostanie automatycznie utworzona. Istniejący już plik konfiguracyjny ***application.properties*** wykorzystaj do konfiguracji pracy z bazą danych, dodając do niego wpisy:

```
spring.jpa.properties.hibernate.hbm2ddl.auto=update
#spring.datasource.url=jdbc:h2:file:./bazaDanych
spring.jpa.show-sql = true
```

Pierwszy wiersz informuje o strategii generowanego schematu bazy danych (schemat bazy danych zostanie utworzony i później aktualizowany automatycznie). Drugi wiersz (z komentarzem #) dotyczy ewentualnej lokalizacji bazy danych *H2* w pliku (domyślnie baza jest w trybie ***memory***). *H2* umożliwia tworzenie lokalnych baz danych, a struktura bazy danych może znajdować się w systemie plików. Uruchamia się ona razem z aplikacją. Ostatnie polecenie pozwala zobaczyć na konsoli zapytania *SQL* (budowane przez *Hibernate*), wykorzystane do utworzenia bazy danych, dodania rekordów, pobrania ich z bazy, itp.

Na rysunku 5.7 zaznaczono przykładowe polecenia *SQL* wykorzystane w przykładzie.

5.2.4. Repozytorium *CRUD*

Model definiuje strukturę danych, a repozytoria definiują jakie operacje można wykonać na danych. Podstawowe operacje *CRUD* (ang. *Create, Read, Update, Delete*) udostępnia sam interfejs *JPA*, inne operacje, jak wyszukiwanie po polach, trzeba dodać samodzielnie. Najprostsze repozytoria tworzone są jako **interfejsy.** Cały kod, który potrzebny jest do wykonania akcji, jest generowany przez *Spring*.

W kolejnym pakiecie ***repositories*** (lokalizacja jak na Rys. 5.8) utwórz interfejs repozytorium (nowy plik z kategorii ***interface***) o nazwie ***ZadanieRepository***, który udostepni standardowe operacje (np. *CRUD*) wykonywane na klasie encji ***Zadanie*** (Przykład 5.6). Interfejs ***ZadanieRepository*** dziedziczy po gotowym repozytorium ***CrudRepository***. Dodatkowe operacje na danych udostępnia interfejs ***JpaRepository***, który rozszerza możliwości ***CrudRepository*** (sprawdź jakie).

Przykład 5.6. Interfejs repozytorium dla klasy encji Zadanie

```
package bp.pai_springboot.repositories;

import bp.pai_springboot.entities.Zadanie;
import org.springframework.data.repository.CrudRepository;

public interface ZadanieRepository
                    extends CrudRepository<Zadanie, Long>{
    }
```

UWAGA! Definiując interfejs repozytorium, poza typem encji należy wskazać typ pola, które jest mapowane na klucz podstawowy w bazie danych. W przykładzie wskazano: **`<Zadanie, Long>`**.

5.2.5. Zastosowanie repozytorium

Ostatnim krokiem jest „**wstrzyknięcie" obiektu repozytorium** do klasy kontrolera i wykorzystanie jego gotowych metod, na przykład:

- ***save(Zadanie t)*** – zapisuje obiekt ***Zadanie t*** do bazy danych,
- ***Zadanie findOne(Long id)*** – znajduje i zwraca obiekt ***Zadanie*** o podanym ***id***,
- ***Iterable findAll()*** – pobiera wszystkie rekordy z tabeli bazy danych i zwraca je w postaci kolekcji ***Iterable***,
- ***long count()*** – zwraca liczbę rekordów tabeli ***zadanie***,
- ***void delete(Long id)*** **–** usuwa z obiekt o zadanym ***id***,
- ***void delete(Zadanie t)*** – usuwa wskazany obiekt ***Zadanie*** z tabeli w bazie danych,
- ***deleteAll()*** – usuwa wszystkie rekordy z tabeli.

Aby poinformować *Spring*, gdzie znajduje się repozytorium, przed klasą ***Main*** dodaj adnotację ***@EnableJpaRepositories*** z parametrem ***basePackagesClasses*** (Przykład 5.7). Pamiętaj też o dodaniu kolejnego importu dla klasy ***EnableJpaRepositories***.

Przykład 5.7. Zastosowanie repozytorium – klasa Main

```
@EnableAutoConfiguration
@ComponentScan({"bp.pai_springboot.controllers"})
@EnableJpaRepositories({"bp.pai_springboot.repositories"})

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

UWAGA! Jeśli pakiety z kontrolerami i interfejsem repozytorium są umieszczone jako bezpośrednie podpakiety głównego pakietu projektu, to nie trzeba dodawać parametrów do odpowiednich adnotacji i można uprościć je do postaci:

```
@ComponentScan
@EnableJpaRepositories
```

Utworzony wcześniej kontroler ***PageController*** wykorzystaj teraz do pracy z danymi z bazy. Do klasy kontrolera dodaj deklarację obiektu repozytorium, poprzedzoną odpowiednią adnotacją ***@Autowired***:

```
@Autowired
public ZadanieRepository rep;
```

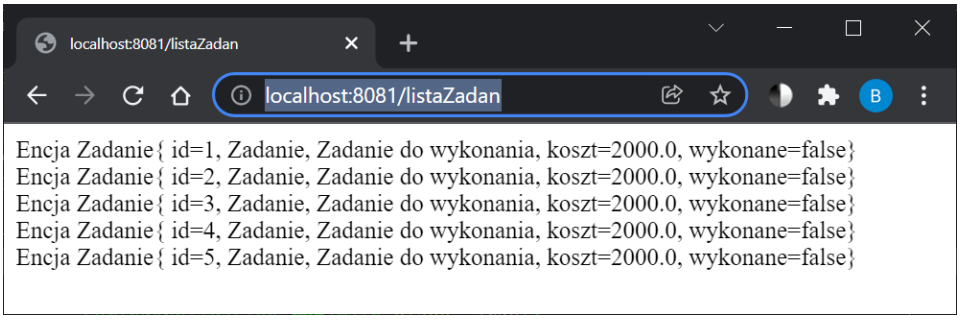
Adnotacja ***@Autowired*** umożliwia wykorzystanie mechanizmu programowania obiektowego, znanego jako **„wstrzykiwanie zależności”** (ang. *dependency injection – DI)*.

Następnie dodaj nową metodę ***listaZadan()***, która zdefiniuje i zapisze obiekt klasy ***Zadanie*** do bazy danych oraz zwróci w odpowiedzi do klienta listę aktualnych rekordów pobranych z bazy danych (Przykład 5.8).

Przykład 5.8. Dodatkowa metoda w kontrolerze PageController

```
@RequestMapping("/listaZadan")
@ResponseBody
public String listaZadan() {
  StringBuilder odp = new StringBuilder();
  Zadanie zadanie = new Zadanie();
  //korzystając z obiektu repozytorium zapisujemy zadanie do bazy
  rep.save(zadanie);
  //korzystając z repozytorium pobieramy wszystkie zadania z bazy
  for(Zadanie i: rep.findAll()) {
            odp.append(i).append("<br>");
  }
  return odp.toString();
}
```

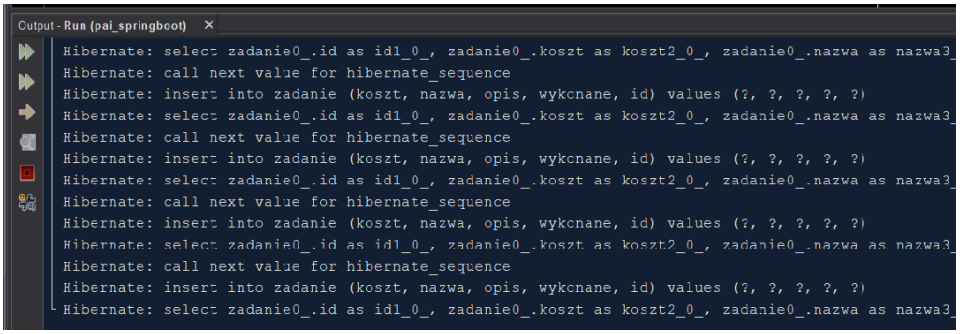
Po uruchomieniu aplikacji i przejściu na stronę ***http://localhost:8081/listaZadan*** powinno się wyświetlić dodane zadanie. Po kilkukrotnym odświeżeniu strony wynik będzie postaci jak na rysunku 5.6.



Rys. 5.6. Wynik pracy z bazą *H2*

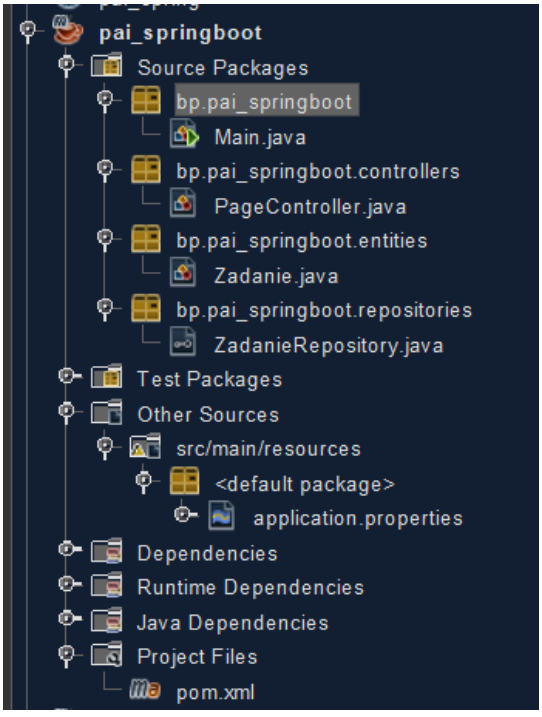
W okienku *Output* w *NetBeans* można zaobserwować zapytania do bazy generowane przez *Hibernate* (Rys. 5.7).

Struktura projektu w *NetBeans* pokazana jest na rysunku 5.8.



```
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3_
Hibernate: call next value for hibernate_sequence
Hibernate: insert into zadanie (koszt, nazwa, opis, wykonane, id) values (?, ?, ?, ?, ?)
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3_
Hibernate: call next value for hibernate_sequence
Hibernate: insert into zadanie (koszt, nazwa, opis, wykonane, id) values (?, ?, ?, ?, ?)
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3_
Hibernate: call next value for hibernate_sequence
Hibernate: insert into zadanie (koszt, nazwa, opis, wykonane, id) values (?, ?, ?, ?, ?)
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3_
Hibernate: call next value for hibernate_sequence
Hibernate: insert into zadanie (koszt, nazwa, opis, wykonane, id) values (?, ?, ?, ?, ?)
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3_
```

Rys. 5.7. Polecenia *Hibernate* widoczne w okienku *Output*



Rys. 5.8. Struktura projektu po wykonaniu zadania 5.2

Zadanie 5.3. *MySQL* i *Spring Boot*

W projekcie z zadania 5.2 wykorzystano *Spring Data JPA*, *Hibernate* oraz bazę danych *H2*. Aby zmienić konfigurację do pracy z serwere *MySQL*:

1. Dodaj do pliku ***pom.xml*** zależność (i zakomentuj zależność dla *H2*):

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

2. Do pliku ***application.properties*** dodaj konfigurację dla *MySQL* (oraz zakomentuj dla *H2*):

```
#Jeśli jest problem z tworzeniem tabeli w mysql – dodaj:
#spring.jpa.hibernate.ddl-auto=update
spring.datasource.url =
jdbc:mysql://localhost:3306/test?serverTimezone=UTC&useUnicode
=yes&characterEncoding=UTF-8
spring.datasource.username = root
spring.datasource.password =
# Strategia nazewnictwa dla Hibernate (Naming strategy)
spring.jpa.hibernate.naming-strategy =
org.hibernate.cfg.ImprovedNamingStrategy
# Określenie dialektu SQL pozwala Hibernate generować
# odpowiednią składnię SQL dla wskazanej bazy danych:
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQL5Dialect
```

Reszta plików projektu pozostaje bez zmian. Tabele i kolumny będą tworzone zgodnie z definicjami pól w encjach (klasach z adnotacją ***@Entity***) tak samo jak dla bazy *H2*. Po zrestartowaniu aplikacji i jej uruchomieniu z konfiguracją dla *MySQL*, dane o zadaniach będą utrwalone w bazie ***test*** w tabeli ***zadanie***.
UWAGA! Pamiętaj o uruchomieniu serwera *MySQL* **przed** uruchomieniem aplikacji z zadaniami.

Zadanie 5.4. Metody wyszukiwania w repozytorium

Rozszerz działanie aplikacji o kolejne metody:

- do generowania dodatkowych zadań testowych,
- do usuwania wskazanego zadania,
- do wyszukiwania zadań według zadanego kryterium.

Dzięki dziedziczeniu po interfejsie ***CrudRepository*** (lub ***JpaRepository***) można korzystać z gotowych metod do wykonywania podstawowych operacji typu *CRUD*. Jednak aby wyszukiwać dane, należy do repozytorium dodać dodatkowe metody do filtrowania danych według zadanych kryteriów.

a) Korzystając z istniejącej już metody kontrolera ***listaZadan()*** (lub nowej pomocniczej metody) wygeneruj więcej zadań, np. za pomocą instrukcji (sprawdź, czy nowe rekordy pojawią się w bazie danych ***test*** w tabeli ***zadanie***):

```
Zadanie z;
double k=1000;
boolean wyk=false;
for (int i=1;i<=10;i++){
```

```
        z = new Zadanie();
        z.setNazwa("zadanie "+i);
        z.setOpis("Opis czynnosci do wykonania w zadaniu "+i);
        z.setKoszt(k);
        z.setWykonane(wyk);
        wyk=!wyk;
        k+=200.50;
        rep.save(z);
    }
```

b) Do kontrolera dodaj nową metodę ***delete***, do usuwania rekordu w odpowiedzi na zapytanie postaci ***http://localhost:8081/delete/5***. Przetestuj działanie tej metody.
c) Do repozytorium ***ZadanieRepository*** dodaj kolejne metody do wyszukiwania rekordów w tabeli ***zadanie***:
   - ***findByWykonane(boolean)*** – zwraca rekordy z wykonanymi lub niewykonanymi już zadaniami;
   - ***findByKosztLessThan(double)*** – zwraca rekordy zadań, których koszt jest mniejszy niż zadany;
   - ***findByKosztBetween(double, double)*** – zwraca rekordy zadań, których koszt należy do wskazanego przedziału wartości.

Do wyszukiwania wykorzystaj *Spring Data JPA* i tzw. **zapytania wbudowane**. Zapytania wbudowane umożliwiają tworzenie zapytań w oparciu o mechanizmy składania zapytań z nazwy akcji (np**. *findBy***) oraz części własnej w postaci nazw pól encji. Więcej szczegółów i przykłady zapytań wbudowanych i metod z adnotacją ***@Query*** można znaleźć na stronie:
https://www.javappa.com/kurs-spring/spring-data-jpa-zapytania-wbudowane

Przetestuj działanie tych metod, dodając do kontrolera odpowiednie akcje.

UWAGA! W celu przekazania wielu parametrów do akcji kontrolera, należy w ***@RequestMapping*** podać je w **{}** w adresie *URL*, np.:

```
@RequestMapping("/koszt/{min}/{max}")
```

Zadanie 5.5. Wyszukiwanie w bazie *world*

W oparciu o *Spring Boot* utwórz nową aplikację, która będzie korzystała z bazy ***world*** i tabeli ***country***. Zadaniem aplikacji ma być obsługa wyszukiwania:

- krajów z danego kontynentu,
- krajów o liczbie ludności z zadanego przedziału,
- krajów danego kontynentu o powierzchni z zadanego przedziału.

Tak jak w zadaniu poprzednim, akcje kontrolera powinny zwracać typ ***String*** i być poprzedzone adnotacją ***@ResponseBody***. Na potrzeby tego zadania w definicji klasy encji ***Country*** wystarczy dodać **tylko** te pola, które są potrzebne w metodach wyszukiwania. Pola powinny być deklarowane odpowiednio jako typ ***String, Double*** (lub ***BigDecimal***) i nie trzeba stosować adnotacji ***@Column*** poprzedzających nazwy pól.

UWAGA 1. W pliku ***application.properties*** należy zmienić domyślną strategię nazewnictwa dla *Hibernate* na (należy wpisać w jednym wierszu!):

```
spring.jpa.hibernate.naming.physical-
strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyS
tandardImpl
```

Pozwoli to właściwie zmapować nazwę pól składających się z wielu wyrazów (np. ***surfaceArea*** w klasie encji ***Country*)** na nazwę kolumny ***SurfaceArea*** w tabeli. Brak tego ustawienia (ustawienie domyślne) powoduje tworzenie nowej kolumny o nazwie ***surface_area,*** co stwarza problemy z działaniem metod wyszukiwania opartych na zapytaniach wbudowanych.

UWAGA 2. Jeśli nie uda się skorzystać z zapytania wbudowanego, zawsze można zastosować własne metody z adnotacją ***@Query*.** Przykłady takich zapytań można znaleźć na wskazanej już wcześniej stronie:
https://www.javappa.com/kurs-spring/spring-data-jpa-zapytania-wbudowane

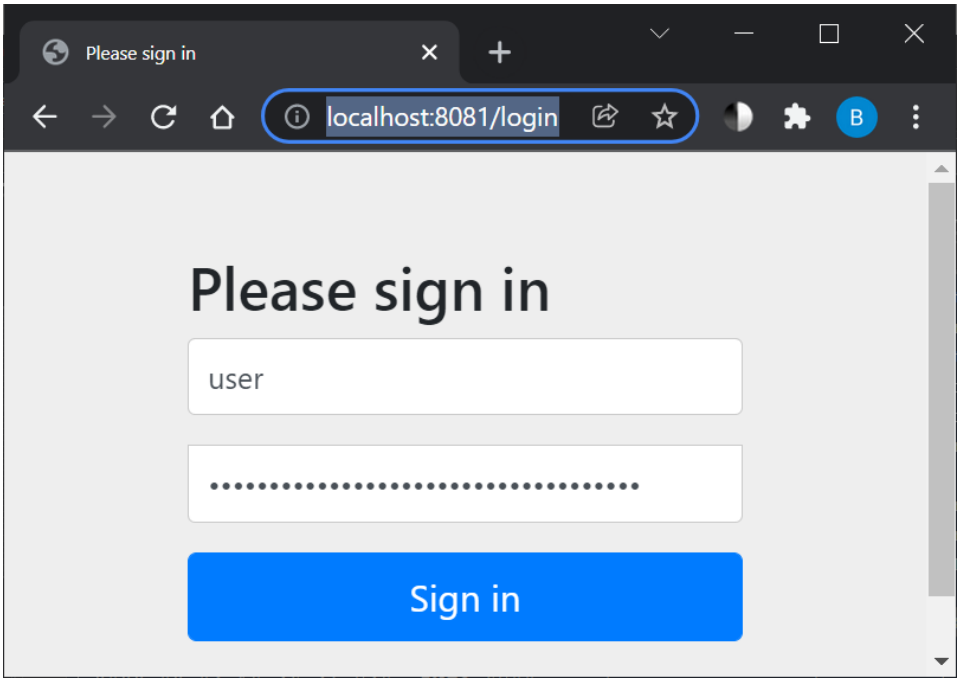
Zadanie 5.6. Wstęp do *Spring Security*

Spring udostępnia gotowy mechanizm autentykacji i autoryzacji użytkowników, z którego można skorzystać, dodając do ***pom.xml*** zależność do *Spring Security* (Przykład 5.9).

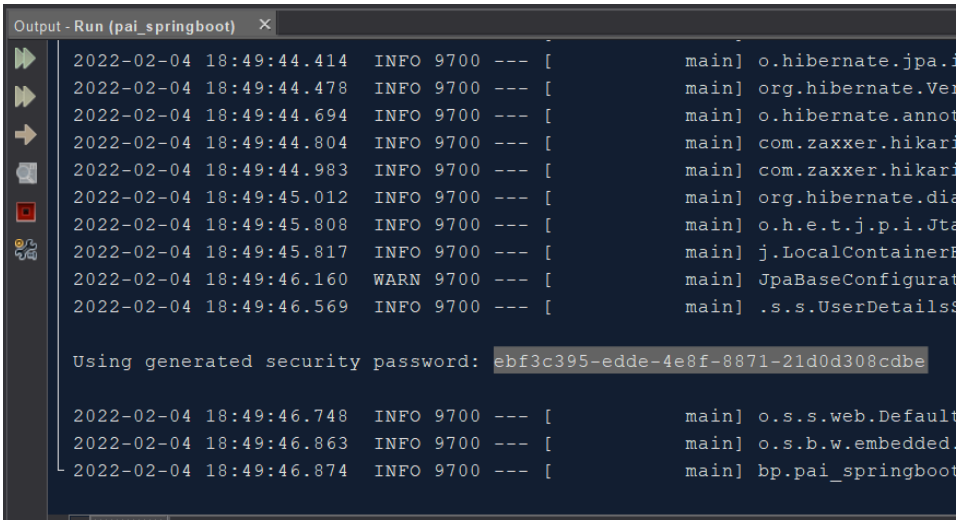
Przykład 5.9. Zależność dla Spring Security

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

Po przebudowaniu projektu z dodaną zależnością dla *Sprig Security* oraz po wejściu na adres strony z listą zadań, nastąpi przekierowanie do strony logowania (Rys. 5.9). Domyślna nazwa użytkownika to ***user***, a hasło jest generowane losowo przez serwer i wypisywane na konsolę (Rys. 5.10).



Rys. 5.9. Domyslny formularz logowania dla *Spring Security*



Rys. 5.10. Losowo wygenerowane hasło dla użytkownika *user*

Wpisanie poprawnych danych pozwoli ponownie zobaczyć strony. W przypadku podania nieprawidłowych danych system poprosi o nie kolejny raz. Anulowanie spowoduje wyświetlenie domyślnej strony błędu.

Dane logowania można zmienić, dodając do pliku konfiguracyjnego ***application.properties*** kolejne wiersze z ustawieniami, np.:

```
spring.security.user.name=beata
spring.security.user.password=beata
spring.security.user.roles=admin
```

Bardziej zaawansowana konfiguracja uwierzytelniania użytkownika będzie tematem kolejnych laboratoriów.

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.

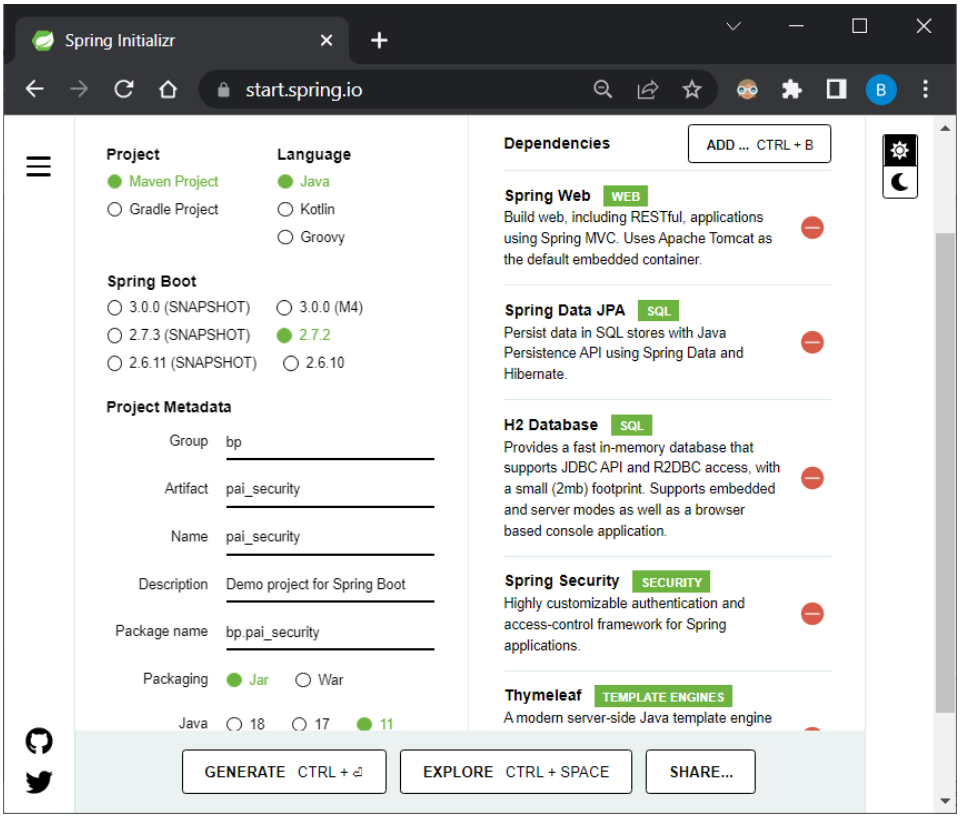
Dodanie do pliku *pom.xml* zależności *Spring Security*, powoduje, że *Spring Boot* automatycznie zabezpiecza wszystkie punkty docelowe *HTTP* (ang. *endpoints*) za pomocą podstawowej autentykacji (ang. *Basic Authentication*). Taki podstawowy mechanizm zastosowano w zadaniu 5.6. Dodatkowe zabezpieczenia 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

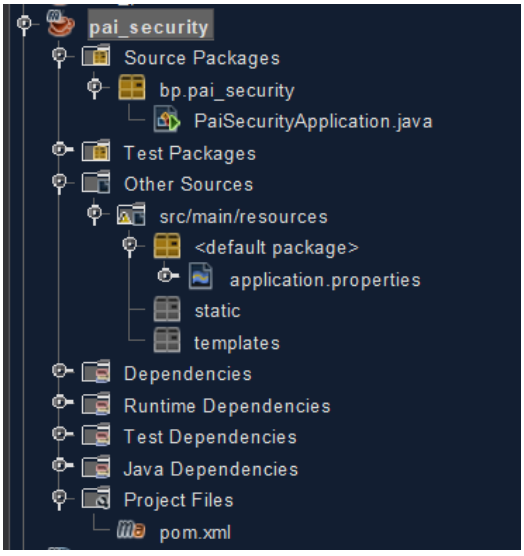
Zadanie 6.2. Dodatkowe elementy konfiguracji projektu

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
```

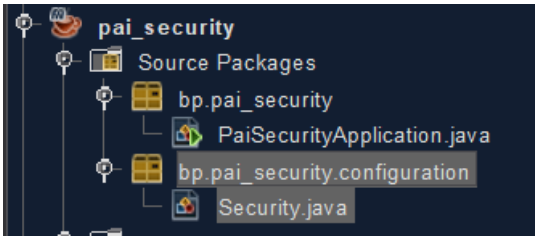
Uruchom projekt i przetestuj jego działanie, wpisując w przeglądarce adres, np. ***localhost:8080/login***. Przeglądarka, przed udostępnieniem aplikacji, wyświetla domyślny formularz logowania (jak w zadaniu 5.6), co wynika z domyślnej konfiguracji *Spring Security*.

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

}
```

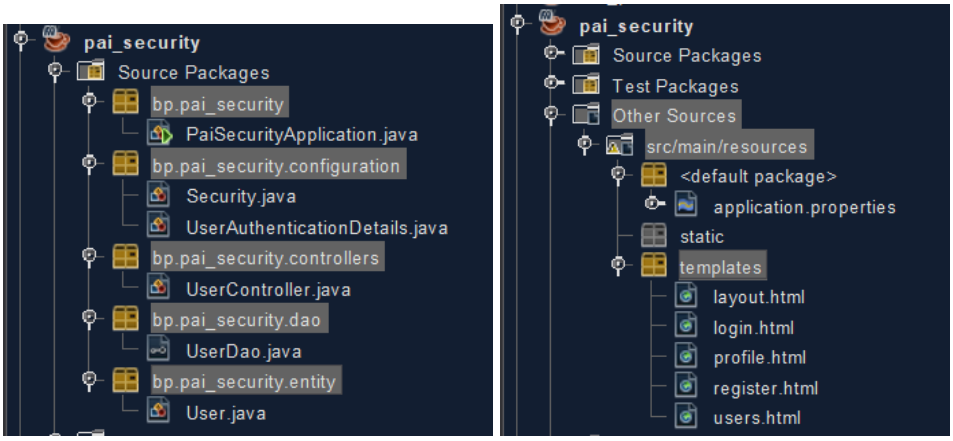
Do klasy kontrolera „wstrzyknięto” dwa ziarna: ***PasswordEncoder*** i ***UserDao***. Każda z metod kontrolera obsługuje inny adres *URL* określony przy pomocy adnotacji ***@GetMapping*** lub ***@PostMapping***. Po utworzeniu klasy kontrolera brakuje już tylko plików odpowiedzialnych za widoki.

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>
            </footer>
        </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"/>
```

```
    <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).

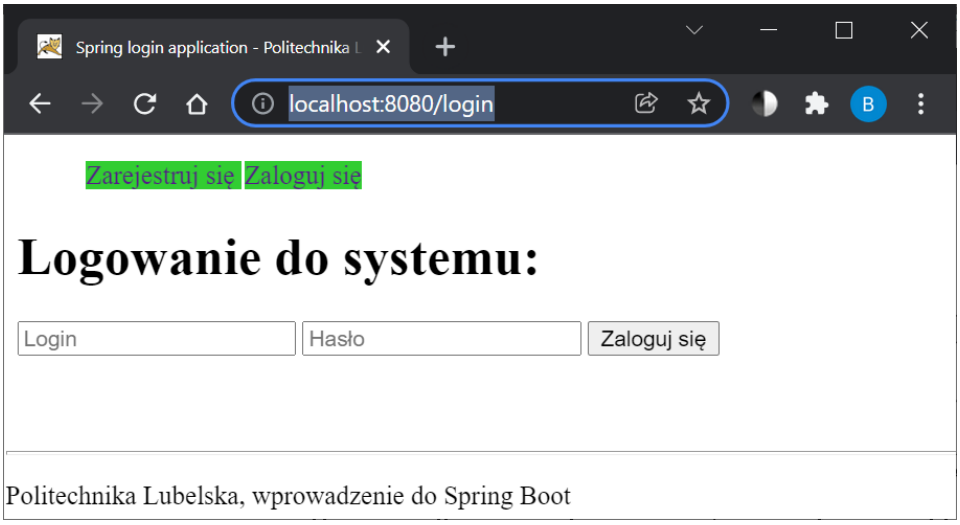
Aby można było przetestować działanie aplikacji, na początek dodaj dwóch użytkowników do bazy danych przy pomocy specjalnej metody ***init()*** w **klasie głównej**. Metoda ***init()*** z adnotacją ***@PostConstruct***, wywoływana jest tuż po uruchomieniu aplikacji. Do klasy startowej projektu (***PaiSecurityApplication***)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

Przykład 6.10. Głowna 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")));
    }
}
```

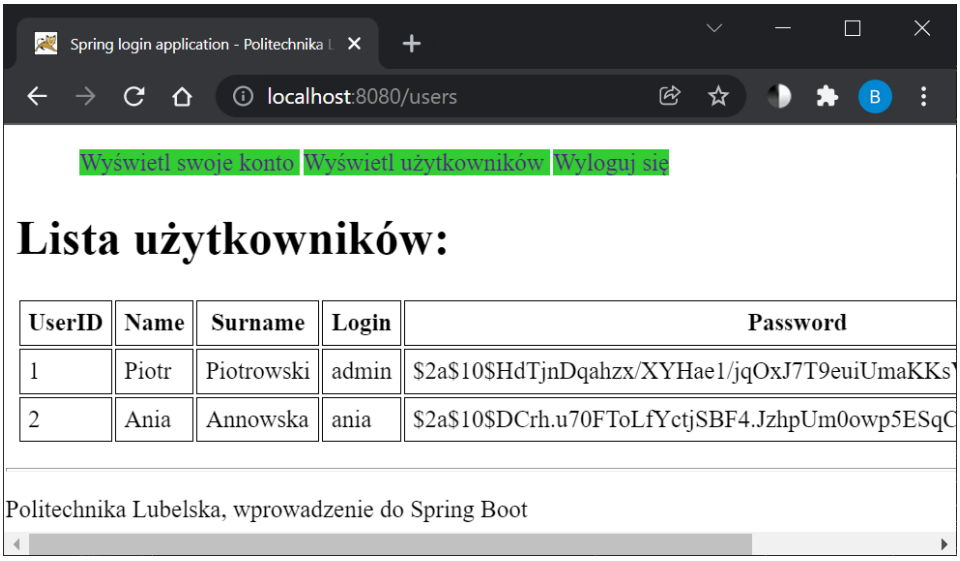
W metodzie ***init()*** utworzono pierwszych użytkowników za pomocą obiektu ***dao*** i ***passwordEncoder*** (wykorzystany do haszowania hasła). Pierwszy użytkownik posiada dane logowania: **login: admin, hasło: admin.** Dla drugiego użytkownika ustawiono **login: ania, hasło: ania.**

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*

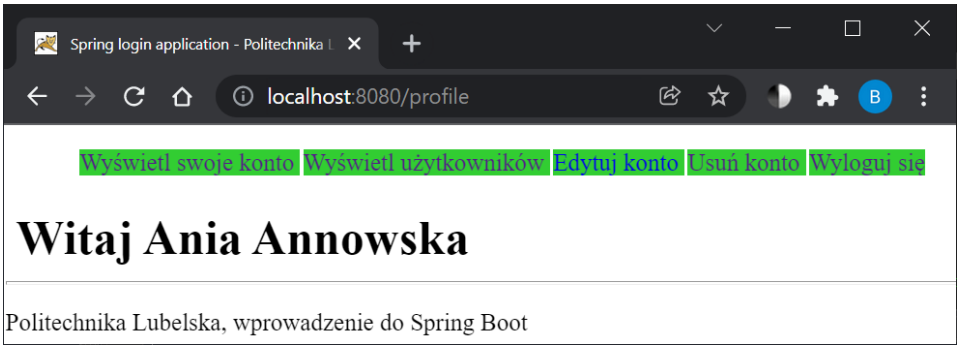
Po prawidłowym zalogowaniu (i uzupełnieniu pliku widoku ***users.html***) wyświetl listę wszystkich użytkowników (Rys. 6.6).



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 umieszczny 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:…"*

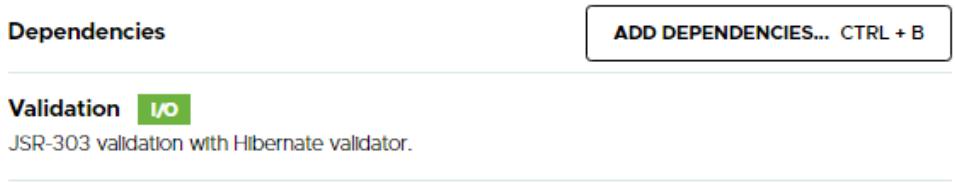
Aby skorzystać z adnotacji do walidacji danych w klasie encji, należy dodać do pliku ***pom.xml*** zależność:

```
<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).

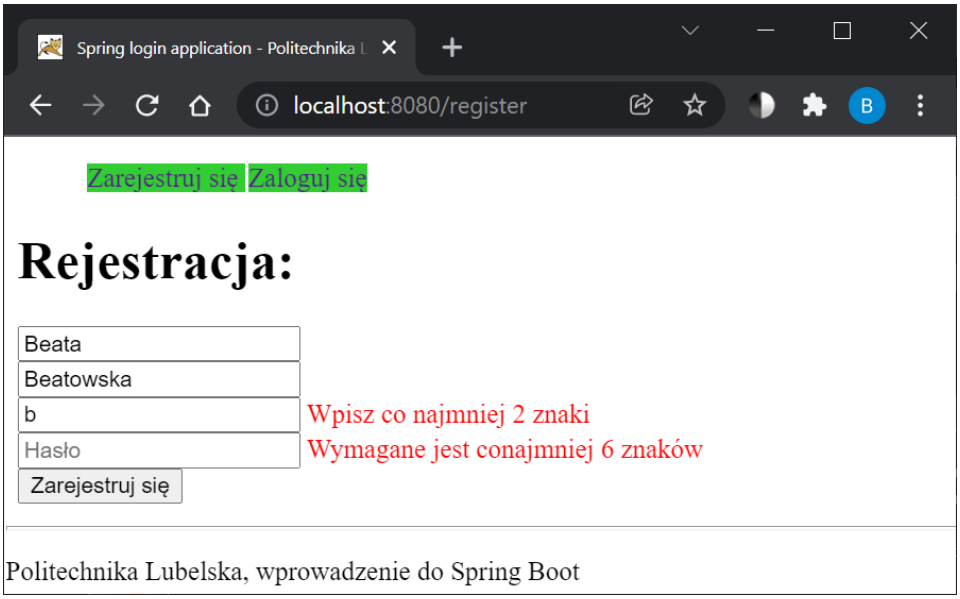
Przykład 6.11. Przykładowa akcja kontrolera z walidacją

```
@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";
    }
```



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

Przykładowy efekt walidacji danych rejestracyjnych przedstawia rysunek 6.9.



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.

Laboratorium 7. *Spring Boot* i *REST API*

Cel zajęć

Realizacja zadań z niniejszego laboratorium pozwoli studentom poznać zasady tworzenia warstwy serwerowej aplikacji typu *REST* w *Spring* [2], współpracującej z aplikacją klienta, zaimplementowaną w języku *JavaScript* z wykorzystaniem interfejsu *Fetch API* [8].

Zakres tematyczny

- Utworzenie *REST API* z funkcjonalnością *CRUD* w *Spring Boot* za pomocą narzędzia *Initializr*.
- Konfiguracja *REST API* do pracy z danymi w formacie *JSON*.
- Przetestowanie *REST API* za pomocą narzędzia *Postman*.
- Utworzenie aplikacji klienckiej, współpracującej z *REST API* za pomocąinterfejsu *Fetch API* w *JavaScript*.

Wprowadzenie

Tworzone do tej pory przykładowe aplikacje były aplikacjami typu *MPA* (ang. *Multi Page Application*), dla których widoki (*JSP*, *Thymeleaf*) generowane były **po stronie serwera** i wysyłane do przeglądarki internetowej w czasie rzeczywistym. Obsługę żądań w tego typu aplikacjach realizuje się w trybie synchronicznym.

Innym rozwiązaniem są aplikacje typu *SPA* (ang. *Single Page Application*), korzystające tylko z *REST API*, udostępnianego przez serwer. Taka aplikacja składa się zwykle tylko z jednego, głównego dokumentu *HTML*, który poprzezu *JavaScript* komunikuje się z serwerem WWW w architekturze *REST*, za pośrednictwem protokołu *HTTP*, wysyłając żądania w trybie asynchronicznym (w tle). Do implementacji *SPA* po stronie klienta można skorzystać z czystego języka *JavaScript* i jego interfejsu *Fetch API* (do przesyłania żądań w trybie asynczronicznym) lub z dedykowanych szkieletów programistycznych (np. *Angular*, *React*, *Vue*). Do implementacji *REST API* po stronie serwera w języku *Java* wykorzystuje się najczęściej *Spring REST API*. *REST API* jest elastycznym i szeroko rozpoznawanym interfejsem dla systemów stron trzecich, również dla innych klientów (np. aplikacji mobilnych).

Do realizacji zadań z niniejszego laboratorium potrzebny będzie klient do przetestowania utworzonego *REST API*. W przykładach wykorzystano narzędzie *Postman*, które można pobrać ze strony: *https://www.postman.com/downloads/*.

Zadanie 7.1. Inicjalizacja projektu *REST API*

Do utworzenia projektu zastosuj *Spring Initializr* (*https://start.spring.io/*). Dodaj zależności:

- *Rest Repositories*,
- *Spring Data JPA*,
- *MySQL Driver*,
- *Lombok* (opcjonalna, wygodna biblioteka wspierająca m.in. automatyczne generowanie konstruktorów i metod ***get*** i ***set***).

Otwórz projekt w swoim *IDE*. Do pliku ***application.properties*** dodaj konfigurację do połączenia z bazą danych za pomocą sterownika dla *MySQL* (pamiętaj aby przed uruchomieniem projektu uruchomić też serwer *MySQL*). Sprawdź, czy podana w przykładzie 7.1 konfiguracja jest odpowiednia dla bazy danych (w przykładzie wykorzystana jest baza danych ***test***). Zbuduj projekt z dodanymi zależnościami.

Przykład 7.1. Zawartość pliku application.properties

```
server.port=8080
#Konfiguracja BD i MySQL:
#struktura BD ma być zaktualizowana przy starcie aplikacji:
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = true
spring.datasource.url =
jdbc:mysql://localhost:3306/test?serverTimezone=UTC&useUnicode=yes&chara
cterEncoding=UTF-8
spring.datasource.username = root
spring.datasource.password =
#Ustawienie strategii nazewnictwa (Naming strategy) dla Hibernate
spring.jpa.hibernate.naming-strategy =
org.hibernate.cfg.ImprovedNamingStrategy
```

Zadanie 7.2. Podstawowe elementy *REST API*

7.2.1. Adnotacje *@JsonView* w klasie encji

W projekcie utwórz pakiet ***entities*** z klasą encji ***Student***, która zawiera 4 pola prywatne: ***id***, ***name***, ***surname*** i ***average.*** Pamiętaj o dodaniu znanych już adnotacji ***@Entity, @Id*** i ***@GeneratedValue*** (z odpowiednimi parametrami) oraz wszystkie pola poza ***id***, poprzedź adnotacją ***@JsonView***. Adnotacja ***@JsonView*** pozwala mapować łańcuch w formacie *JSON* z danymi studenta (przesyłanymi z formularza w parametrach żądania) na obiekt klasy ***Student*** po stronie aplikacji. Metody ***get*** i ***set*** dodaj za pomocą adnotacji ***@Getter*** i ***@Setter*** poprzedzających klasę encji, korzystając z **biblioteki *Lombok***.

7.2.2. Repozytorium i klasa z adnotacją *@Service*

W pakiecie ***entities*** utwórz interfejs ***StudentRepository*** rozszerzający ***CrudRepository***. Następnie przygotuj klasę (usługę, serwis) wspomagającą pracę z danymi. W tym celu utwórz nowy pakiet ***services***, a w nim klasę ***StudentService*** z adnotacją ***@Service***. Adnotacja ***@Service*** jest dedykowana klasom, które dostarczają usługi. Do klasy ***StudentService*** „wstrzyknij" prywatne pole ***studentRepository*** typu ***StudentRepository***, poprzedzając go adnotacją ***@Autowired***. Do klasy dodaj także metodę publiczną ***getStudentList()*** do pobrania z repozytorium listy obiektów ***Student*** (Przykład 7.2).

Przykład 7.2. Klasa StudentService

```
package bp.pai_rest.services;

import bp.pai_rest.entities.Student;
import bp.pai_rest.entities.StudentRepository;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class StudentService {
  @Autowired
  private StudentRepository studentRepository;

  public List<Student> getStudentList() {
       return (List<Student>) studentRepository.findAll();
  }
}
```

7.2.3. *REST* kontroler

Metody obsługujące żądania *HTTP* w przypadku tworzonego *REST API*, najczęściej otrzymują w żądaniu (*Request Body*) i wysyłają w odpowiedzi do klienta (*Response Body*) dane w formacie *JSON*. Klasa kontrolera poprzedzona adnotacją ***@RestController***, domyślnie obsługuje format danych *JSON*.

Dla przypomnienia, tablica z danymi o studentach w formacie *JSON* jest reprezentowana przez łańcuch postaci, np.:

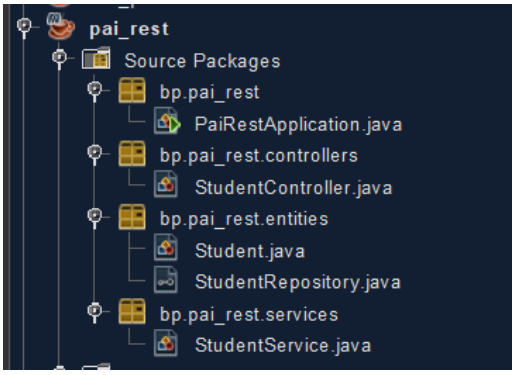
```
[
 { "id":1, "name":"Olek", "surname":"Olewski", "average":4.5},
 { "id":2, "name":"Ola", "surname":"Olewska", "average":4.45}
]
```

Ostatnim elementem aplikacji jest zatem utworzenie klasy kontrolera, z metodami do obsługi żądań. Utwórz pakiet ***controllers*** z klasą ***StudentController*** z adnotacją ***@RestController*** (Przykład 7.3).

Przykład 7.3. Klasa StudentController

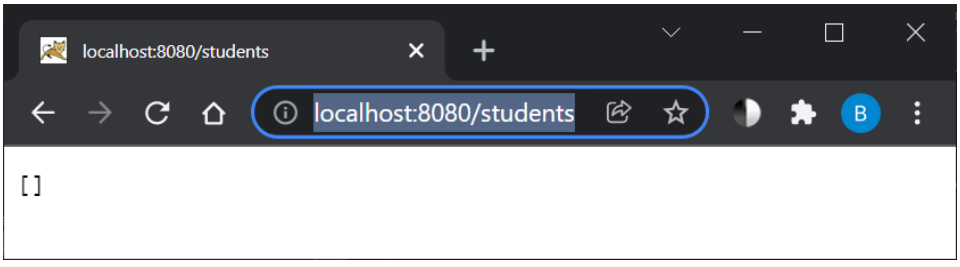
```
@RestController
public class StudentController {
    @Autowired
    private StudentService studentService;

    @GetMapping("/students")
    public List<Student> getAll() {
        return studentService.getStudentList();
    }
}
```

Na rysunku 7.1 pokazano strukturę plików projektu.

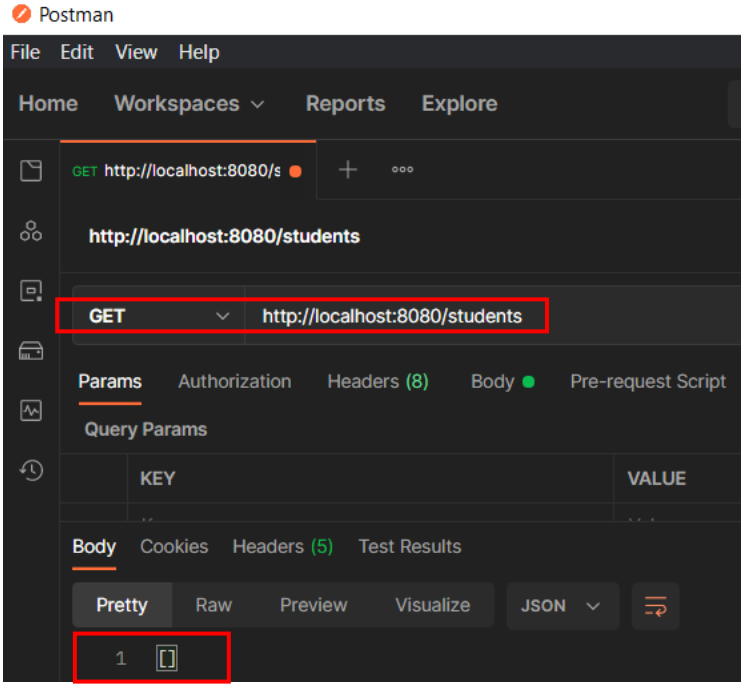


Rys. 7.1. Struktura projektu *pai_rest*

Uruchom aplikację a następnie w przeglądarce (Rys. 7.2) oraz w programie *Postman* (Rys. 7.3) wyślij żądanie postaci: ***http://localhost:8080/students***. W odpowiedzi przesłana zostanie pusta tablica, ponieważ nie utworzono jeszcze rekordów w bazie danych.



Rys. 7.2. Odpowiedź z *REST API* – wynik w przeglądarce



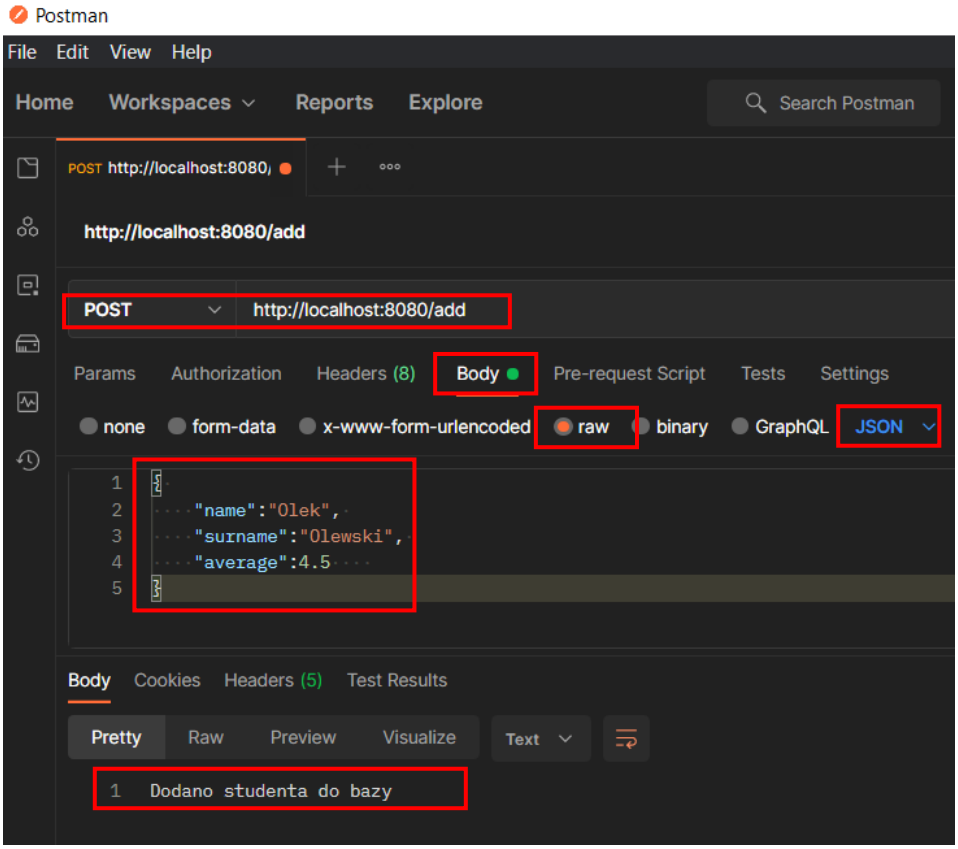
Rys. 7.3. Odpowiedź z *REST API* – fragment okna z wynikiem w narzędziu *Postman*

7.2.4. Utrwalenie danych z żądania

1. Korzystając z biblioteki *Lombok*, do klasy ***Student*** dodaj konstruktor bezparametrowy i konstruktor z wszystkimi parametrami za pomocą adnotacji ***@NoArgsConstructor*** i ***@AllArgsConstructor*** poprzedzających definicję klasy ***Student***.
2. W klasie ***StudentService*** utwórz nową metodę ***addStudent*** z parametrem klasy ***Student***, której zadaniem ma być dodanie obiektu do bazy danych.
3. W kontrolerze utwórz metodę, która obsłuży żądanie *POST*, związane z dodaniem nowego studenta do bazy. Metoda ta powinna posiadać adnotację ***@PostMapping***. Jako parametr metoda przyjmuje obiekt typu ***Student*** poprzedzony adnotacją ***@RequestBody***, dzięki czemu przesłane w żądaniu dane w formacie *JSON* zostaną zmapowane na obiekt ***Student*** (następuje automatyczne pobranie danych *JSON* z ciała żądania i przypisanie ich do odpowiednich pól obiektu ***Student***, o ile tylko nazwy kluczy z *JSON* pokrywają się z nazwami pól w klasie ***Student***).

Zbuduj i uruchom ponownie aplikację, a następnie korzystając z narzędzia *Postman* wyślij żądanie *HTTP* typu *POST* pod adres: ***http://localhost:8080/add***, ale tym razem w ciele żądania dodaj dane w formacie *JSON* postaci (Rys. 7.4):

```
{ "name":"Olek", "surname":"Olewski", "average":4.5}
```



Rys. 7.4. Żądanie i odpowiedź – metoda *POST* z dołączonymi danymi w formacie *JSON*

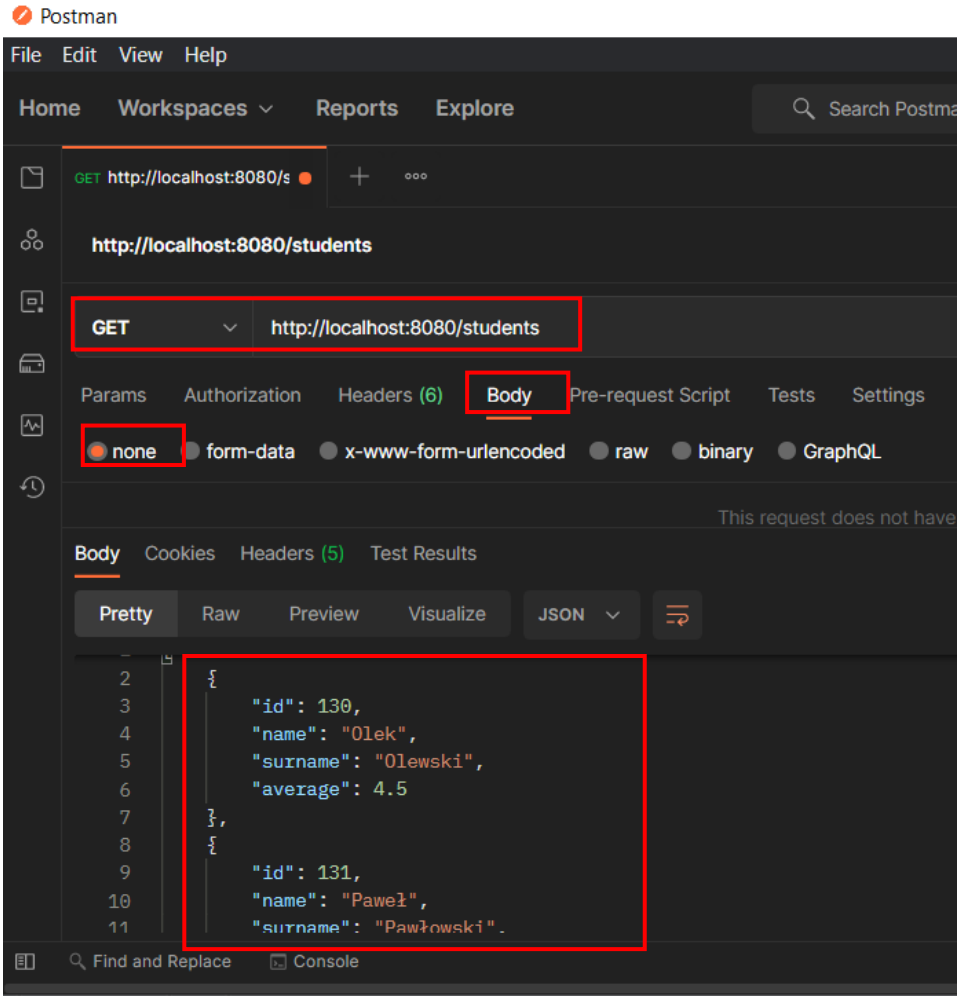
Wyślij kolejne zapytanie z danymi studenta (z polskimi literami):

```
{ "name":"Paweł", "surname":"Pawłowski", "average":4.99}
```

Czy dane studenta z polskimi literami w imieniu lub nazwisku zostały prawidłowo wprowadzone do bazy danych? Jeśli nie i pojawił się problem *com.mysql.cj.jdbc.exceptions.MysqlDataTruncation: Data truncation: Incorrect string value: '\xC5\x82' ...*, to zmień ręcznie w *MySQL* kodowanie w kolumnach ***name*** i ***surname*** na *UTF8-general-ci*.

Powtórz żądanie dodania studenta dla *Pawła Pawłowskiego*. Tym razem rekord powinien być poprawnie dodany do bazy.

Następnie ponownie wyślij zapytanie typu *GET* w celu pobrania wszystkich studentów z bazy (Rys. 7.5). Sprawdź rekordy w bazie danych w tabeli ***student***.



Rys. 7.5. Wynik po wykonaniu dodawania 2 studentów

Zadanie 7.3. Obsługa metod *DELETE* i *PUT*

Uzupełnij funkcjonalność aplikacji o dodatkowe możliwości usuwania i edycji studenta po wskazanym ***id*** (przy czym operacje te można realizować tylko, gdy istnieje student o takim ***id)***. Po dodaniu kolejnej funkcjonalności – testuj jej

działanie za pomocą narzędzia *Postman*, wysyłając odpowiednio żądanie za pomocą metod *DELETE* i *PUT*.

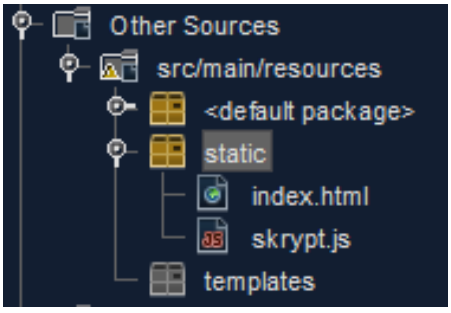
1. Utwórz metodę ***deleteStudent*** w klasie ***StudentService*** i w kontrolerze metodę do obsługi akcji usunięcia studenta z adnotacją ***@DeleteMapping***.
2. Utwórz metodę do modyfikacji (w klasie ***StudentService)*** wszystkich danych studenta poza ***id***. W kontrolerze dodaj metodę z adnotacją ***@PutMapping***, która pozwoli na aktualizację danych studenta.

UWAGA! Usuwanie lub edycję danych można zrealizować po sprawdzeniu, czy obiekt o zadanym ***id*** istnieje w bazie danych. Do sprawdzenia można wykorzystać metodę repozytorium: ***studentRepository.existsById(id)***, która zwraca wartość logiczną.

Zadanie 7.4. *JavaScript* z *Fetch API*

Korzystając z interfejsu *Fetch API* w *JavaScript*, do projektu dodaj plik ***index.html*** oraz odpowiednie funkcje *JavaScript*, które pozwolą na przesyłanie żądań do *REST API* z poziomu strony ***index.html.***

Plik ***index.html*** oraz skrypt z funkcjami *JavaScript* (np. ***skrypty.js***) umieść w folderze projektu: ***resources/static*** (Rys. 7.6).



Rys. 7.6. Zasoby aplikacji w *HTML* i *JavaScript*

Laboratorium 8. *Spring Boot* i *DTO*

Cel zajęć

Realizacja zadań z laboratorium pozwoli studentom poznać zasady stosowania obiektów transferu danych *DTO* [2] i wykorzystać je w *Spring REST API*.

Zakres tematyczny

- Przygotowanie *REST API* za pomocą narzędzia *Spring Initializr*, bez zastosowania obiektów *DTO* (jak w poprzednim laboratorium).
- Definicja klas encji z wykorzystaniem relacji *JPA*.
- Modyfikacja aplikacji i zastosowanie obiektów transferu danych *DTO* (ang. *Data Transfer Object*).
- Zastosowanie w aplikacji biblioteki *MapStruct*, w celu mapowania obiektów encji w obiekty *DTO*.

Wprowadzenie

W aplikacjach w języku *Java* z współpracujących *Hibernate* czy *Java Persistence API*, aby użyć danych korzysta się z różnych obiektów (*POJO*, J*avaBean*, klas encji, *DAO*, *DTO*). Warto usystematyzować, co charakteryzuje te obiekty:

- Klasa ***JavaBean*** musi przestrzegać pewnych konwencji dotyczących nazewnictwa, budowy i zachowania metod:
  - powinna implementować interfejs *Serializable*,
  - posiada konstruktor bezargumentowy,
  - umożliwiać dostęp do właściwości pól prywatnych za pomocą metod *get* i *set*.
- Klasa ***POJO*** (ang. *Plain Old Java Object*) jest podobna do *JavaBean*, przechowuje atrybuty i pozwala je modyfikować (za pomocą metod ***get*** i ***set***), ale nie musi implementować żadnego interfejsu i posiadać konstruktora bezargumentowego. Według definicji klasa *POJO* nie powinna odnosić się do żadnego szkieletu programistycznego.
- Klasa ***encji*** jest modyfikowalna (zawiera metody ***set*** do ustawiania wartości pól reprezentujących kolumny tabeli), ale użycie adnotacji ***@Entity***, ***@Table***, ***@Column*** z *JPA* łamie zasady definicji *POJO*, uzależniając się od *Spring JPA*. Jednak w dokumentacji *Hibernate* znajduje się odniesienie do klas encji, jako do obiektów *POJO* z adnotacjami *javax.persistence.Entity*. Jeżeli logika encji jest prosta, opiera się na danych z encji, nie zawiera skomplikowanej logiki to można traktować encję jako *POJO*.

- Klasa ***DAO*** (ang. *Data Access Object*) zapewnia dostęp do źródła danych (plik, baza danych). Najczęściej jest to interfejs, ponieważ łatwo można wtedy zmienić źródła danych.
- Klasa ***DTO*** (ang. *Data Transfer Object*) definiuje obiekt transferu danych i w programowaniu obiektowym oznacza obiekt, który przechowuje tylko pola publiczne, bez metod. Często używany jest do komunikacji z bazami danych w bibliotekach *ORM*. Podstawowym zadaniem *DTO* jest transfer danych pomiędzy systemami, aplikacjami lub też pomiędzy warstwami w danej aplikacji. *DTO* to bardziej rozbudowany kontener na dane, uzupełniony o dodatkowe możliwości ułatwiające dostęp, ochronę oraz przesyłanie danych.

Zadanie 8.1. Aplikacja bez obiektów *DTO*

Do inicjalizacji projektu o nazwie np. ***pai_dto*** zastosuj *Spring Initializr*, dodając następujące zależności:
- *Spring Web,*
- *Spring Data JPA,*
- *H2 Database,*
- *Lombok.*

Po wygenerowaniu projektu – rozpakuj go, otwórz w swoim IDE i zbuduj. Plik ***application.properties*** może pozostać pusty.

8.1.1. Klasy encji w relacji 1:1

W projekcie utwórz pakiet *domain*, a w nim dwie klasy ***Student*** i ***Address***. Definicja klasy ***Student*** pokazana jest na rysunku 8.1, a klasy ***Address*** na rysunku 8.2 (dodaj odpowiednie importy klas).

Obie klasy są oznaczone adnotacją ***@Entity*** oraz adnotacjami z biblioteki *Lombok*. Dodatkowo klasy te są ze sobą **w relacji 1:1**. Każdy student ma swój własny adres, niezależnie czy istnieje już taki w bazie. Relację 1:1 opisuje adnotacja ***@OneToOne(cascade = CascadeType.ALL)*** z *JPA*. Parametr ***cascade = CascadeType.ALL*** określa, że wszystkie operacje wykonywane na klasie ***Student*** (i tabeli ***student*** w bazie danych) będą automatycznie powtarzane (kaskadowo) na powiązanej z nią klasie ***Address*** (i tabeli ***address*** w bazie danych).

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY )
    private Long id;
    private String name;
    private String surname;
    private Integer age;

    @JoinColumn(name="address_id")
    @OneToOne(cascade = CascadeType.ALL)
    private Address address;
}
```

Rys. 8.1. Klasa *Student*

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY )
    private Long id;

    private String street;
    private String city;
    private String state;
    private String zip;

    @OneToOne(mappedBy = "address")
    private Student student;

}
```

Rys. 8.2. Klasa *Address*

8.1.2. Repozytoria i serwis

W głównej paczce aplikacji utwórz pakiet ***repositories***, a w nim dwa interfejsy: ***StudentRepository*** (Rys. 8.3) oraz ***AddressRepository***. Oba rozszerzają interfejs ***JpaRepository*** i są poprzedzone adnotacją ***@Repository***.

```
package bp.pai_dto.repositories;

import bp.pai_dto.domain.Student;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface StudentRepository extends JpaRepository<Student, Long>{

}
```

Rys. 8.3. Interfejs *StudentRepository*

Repozytorium ***AddressRepository*** jest analogiczne.

Następnie utwórz kolejny pakiet ***services***, a w nim:

- interfejs ***StudentService*** z metodą ***getAllStudents()***, zwracającą listę studentów,
- klasę ***StudentServiceImpl***, implementującą ***StudentService.*** Klasę ***tudentServiceImpl*** (Rys. 8.4) należy oznaczyć adnotacją ***@Service***, aby *Spring* zarejestrował ją w kontekście. Dodatkowo do tej klasy należy „wstrzyknąć" utworzone wcześniej repozytorium ***StudentRepository***, przy pomocy konstruktora (można skorzystać z biblioteki *Lombok* i adnotacji ***@RequiredArgsConstructor***). Obiekt repozytorium i jego metoda ***findAll()*** zostanie wykorzystana dalej w metodzie zwracającej listę studentów.

```
@RequiredArgsConstructor
@Service
public class StudentServiceImpl implements StudentService{
    private final StudentRepository studentRepository;
    @Override
    public List<Student> getAllStudents() {
        return studentRepository.findAll();
    }
}
```

Rys. 8.4. Klasa *StudentServiceImpl* implementująca interfejs *StudentService*

8.1.3. Kontroler

Mając serwis, utwórz klasę kontrolera, który będzie z niego korzystał. Klasę ***StudentController*** umieść w pakiecie ***controllers*** i oznacz adnotacją ***@RestController*** oraz adnotacjami z biblioteki *Lombok*, w celu utworzenia konstruktora, który „wstrzyknie" utworzony wcześniej serwis. Do klasy kontrolera dodaj metodę zwracającą wszystkich studentów. Wykorzystaj adnotację ***@RequestMapping***, dzięki której wszystkie akcje kontrolera będą miały ścieżkę domyślnie zaczynającą się od ***/students*** (Rys. 8.5).

```
@RequiredArgsConstructor
@RequestMapping(value="/students")
@RestController
public class StudentController {
    private final StudentServiceImpl studentService;

    @GetMapping
    public List<Student> getAllStudents(){
        return studentService.getAllStudents();
    }
}
```

Rys. 8.5. Klasa kontrolera *StudentController*

8.1.4. Plik *import.sql*

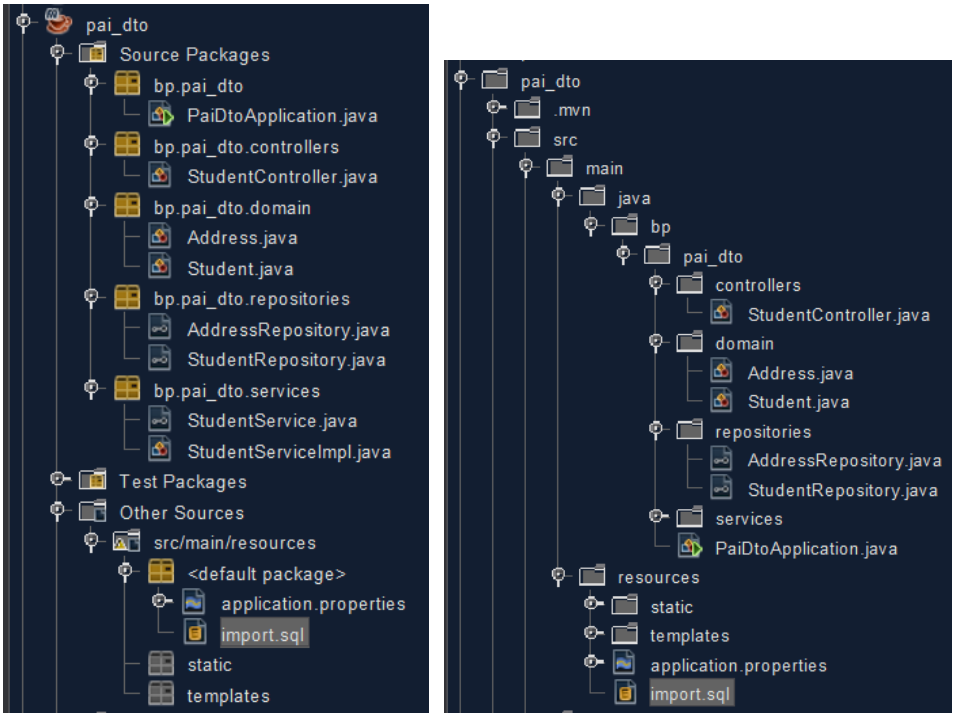
Przed przetestowaniem, czy wszystko jest prawidłowo zaimplementowane, do folderu ***resources*** dodaj plik ***import.sql*** (Przykład 8.1), który wypełni danymi bazę *H2*. **Właściwa nazwa i lokalizacja pliku jest istotna, aby *Spring* załadował go podczas startu.**

Struktura plików projektu oraz lokalizacja pliku ***import.sql*** przedstawiona jest na rysunku 8.6.

Przykład 8.1. Plik import.sql

```
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'Al.
Szucha Jana Chrystiana 1038', 'Warszawa', 'Polska', '00-582');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Bracka 84', 'Tarnowskie Góry', 'Polska', '42-605');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Ratajczaka Franciszka 90', 'Szczecin', 'Polska', '61-816');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Projektowa 142', 'Katowice', 'Polska', '91-493');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Tymiankowa 57', 'Lublin', 'Polska', '20-221');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Zgierska 27', 'Łódź', 'Polska', '91-468');
```

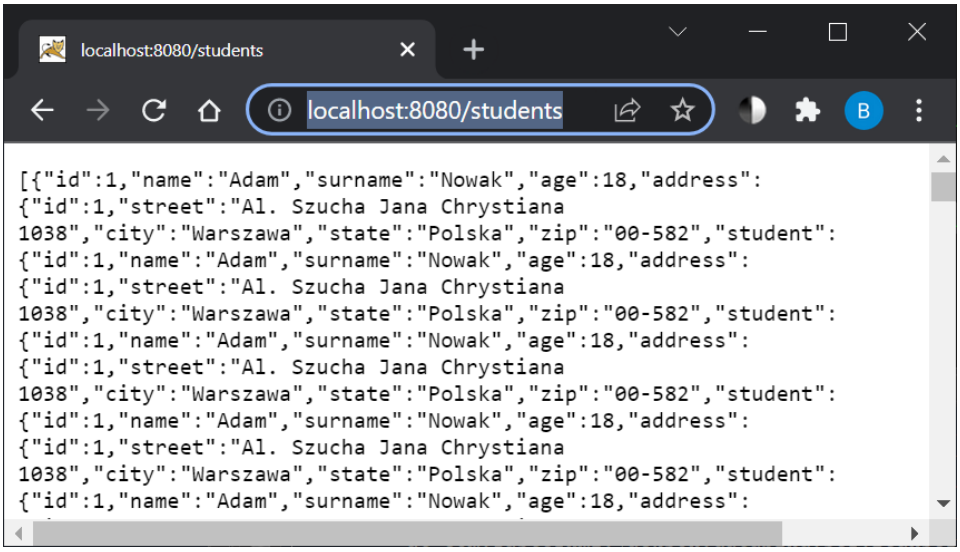
```
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Adam', 'Nowak', 18, 1);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Bartosz', 'Pawlak', 20, 2);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Kinga', 'Kowalczyk', 21, 3);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Wiktor', 'Kowalski', 20, 4);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Piotr', 'Szczepański', 22, 5);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Filip', 'Chmielewski', 19, 6);
```



Rys. 8.6. Widok projektu w zakadce *Project* i *Files* w *Netbeans*

Zbuduj projekt i uruchom aplikację. Przy pomocy narzędzia *Postman* lub przeglądarki pobierz listę studentów, podając adres: ***localhost:8080/students*** (Rys. 8.7).

Rezultat powinien być taki, jak na rysunku 8.7, a dodatkowo w konsoli aplikacji pojawi się komunikat o błędzie (Rys. 8.8).



Na pierwszy rzut oka wszystko wyglądało dobrze, a jednak pojawił się problem, ponieważ doszło do zapętlenia się programu. Obiekt klasy ***Student*** posiada referencję do obiektu klasy ***Address***, który z kolei posiada referencję do obiektu ***Student***. Z tego powodu obie te instancje w kółko siebie wywołują, aż do całkowitego zapełnienia stosu.

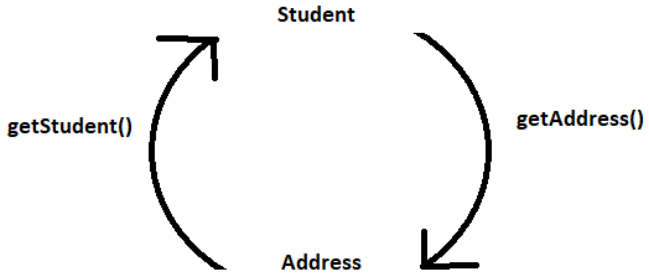
Graficznie można to zinterpretować rysunkiem 8.9.

```
[{"id":1,"name":"Adam","surname":"Nowak","age":18,"address":
{"id":1,"street":"Al. Szucha Jana Chrystiana
1038","city":"Warszawa","state":"Polska","zip":"00-582","student":
{"id":1,"name":"Adam","surname":"Nowak","age":18,"address":
{"id":1,"street":"Al. Szucha Jana Chrystiana
1038","city":"Warszawa","state":"Polska","zip":"00-582","student":
{"id":1,"name":"Adam","surname":"Nowak","age":18,"address":
{"id":1,"street":"Al. Szucha Jana Chrystiana
1038","city":"Warszawa","state":"Polska","zip":"00-582","student":
{"id":1,"name":"Adam","surname":"Nowak","age":18,"address":
{"id":1,"street":"Al. Szucha Jana Chrystiana
1038","city":"Warszawa","state":"Polska","zip":"00-582","student":
{"id":1,"name":"Adam","surname":"Nowak","age":18,"address":
```

Rys. 8.7. Widok w przeglądarce listy studentów z bazy *H2*

```
java.lang.StackOverflowError: null
	at com.fasterxml.jackson.databind.ser.std.BeanSerializerBase.serializeFields(Be
	at com.fasterxml.jackson.databind.ser.BeanSerializer.serialize(BeanSerializer.j
	at com.fasterxml.jackson.databind.ser.BeanPropertyWriter.serializeAsField(BeanP
	at com.fasterxml.jackson.databind.ser.std.BeanSerializerBase.serializeFields(Be
	at com.fasterxml.jackson.databind.ser.BeanSerializer.serialize(BeanSerializer.j
	at com.fasterxml.jackson.databind.ser.BeanPropertyWriter.serializeAsField(BeanP
	at com.fasterxml.jackson.databind.ser.std.BeanSerializerBase.serializeFields(Be
```

Rys. 8.8. Widok komunikatu o błędzie na konsoli po uruchomieniu aplikacji



Rys. 8.9. Problem z referencjami w obiektach *Student* i *Address*

Zadanie 8.2. Aplikacja z *DTO*

Problem z zadania 8.1, jak i wiele podobnych, można rozwiązać na kilka sposobów. W tym zadaniu będą zastosowane najbardziej elastyczne i najczęściej używane obiekty *DTO* (ang. *Data Transfer Object)*. Jak sama nazwa wskazuje, obiekty *DTO* są wykorzystywane przy transferze danych pomiędzy klientem a serwerem. Niezależnie od metody transferu (*GET*, *POST* czy *PATCH*), zasada działania jest taka sama. Tworzone są klasy zawierające tylko te pola, które mają być pokazane przez *API*. Do takiej klasy *DTO* potrzebny jest konwerter, przypisujący polom wartości z odpowiednich encji. Utworzona w zadaniu 8.1 aplikacja zostanie teraz zmodyfikowana, przez wprowadzenie obiektów *DTO* i odpowiednich do nich konwerterów.

8.2.1. Klasa *DTO*

Aby uzyskać listę studentów wraz z ich adresami (tak jak w poprzednim przypadku), w nowym pakiecie o nazwie ***dtos*** utwórz klasę ***StudentDto***. Klasa ta powinna zawierać pola zarówno z klasy ***Student***, jak i ***Address*** (Rys. 8.10). Klasa poprzedzona jest znanymi adnotacjami z biblioteki *Lombok*. Na szczególną uwagę zasługuje adnotacja ***@Builder***, pochodząca z tej samej biblioteki. W celu implementacji wzorca projektowego *builder* wystarczy adnotować klasę jako ***@Builder***, a kompilator wygeneruje w klasie głównej statyczną, zagnieżdżoną klasę ***builder*** oraz niezbędny konstruktor. Wzorzec projektowy *builder* to prawdopodobnie najczęściej stosowany wzorzec projektowy w *Javie*. Umożliwia on wieloetapowe budowanie obiektu, które wykonywane jest przez specjalnie przygotowany do tego zadania obiekt budujący (ang. *builder*).

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class StudentDto {
    private String name;
    private String surname;
    private Integer age;
    private String street;
    private String city;
    private String zip;
    private String state;
}
```

Rys. 8.10. Klasa *StudentDto*

8.2.2. Klasa konwertera

Dla klasy *DTO* potrzebny jest mechanizm, który zamieni obiekty klas ***Student*** i ***Address*** na obiekt ***StudentDto***. W pakiecie o nazwie ***converters***, utwórz klasę ***StudentConverter***, dziedziczącą po istniejącej klasie ***Converter*** dostępnej w odpowiednim pakiecie *Spring* (Rys 8.11).

```
import bp.pai_dto.domain.Student;
import bp.pai_dto.dtos.StudentDto;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

@Component
public class StudentConverter implements Converter<Student, StudentDto>{

    @Override
    public StudentDto convert(Student source) {
        return null;
    }
}
```

Rys. 8.11. Klasa *StudentConverter*

Klasa ***StudentConverter****:*

- oznaczona jest adnotacją ***@Component***, aby była dostępna w kontekście aplikacji *Spring* i można było ją „wstrzykiwać" do innych klas komponentów.
- implementuje interfejs ***Converter<S, T>***, gdzie ***S*** (ang. *source*) jest klasą źródłową (z której dane są pobierane), a ***T*** (ang. *target*) jest klasą docelową *DTO* (udostępnianą z naszego *API*).

- nadpisuje metodę ***convert()*** interfejsu ***Converter***, w której tworzona jest logika konwersji (Rys. 8.12). Wybrano tu odpowiednie wartości pól z instancji klasy ***Student*** i przypisano je obiektowi klasy ***StudentDto*** za pomocą metody ***builder()***, dzięki której kod jest znacznie czytelniejszy.

```
@Override
public StudentDto convert(Student source) {
    return StudentDto.builder()
            .name(source.getName())
            .surname(source.getSurname())
            .age(source.getAge())
            .street(source.getAddress().getStreet())
            .city(source.getAddress().getCity())
            .zip(source.getAddress().getZip())
            .state(source.getAddress().getState())
            .build();
}
```

Rys. 8.12. Implementacja metody *convert()* w klasie *SudentConverter*

8.2.3. Wykorzystanie *DTO* w serwisie i kontrolerze

W projekcie wykorzystaj utworzoną klasę i konwerter. W tym celu:

1. W klasach ***StudentService***, ***StudentServiceImpl*** i w kontrolerze ***StudentController*** zmień zwracany typ ze ***Student*** na ***StudentDto***.
2. Do klasy ***StudentServiceImpl*** „wstrzyknij" komponent konwertera ***StudentConverter*** oraz w metodzie ***getAllStudents()*** zastosuj strumień ***stream()*** i metodę ***map()*** do zmapowania obiektów ***Student*** na ***StudentDto***.

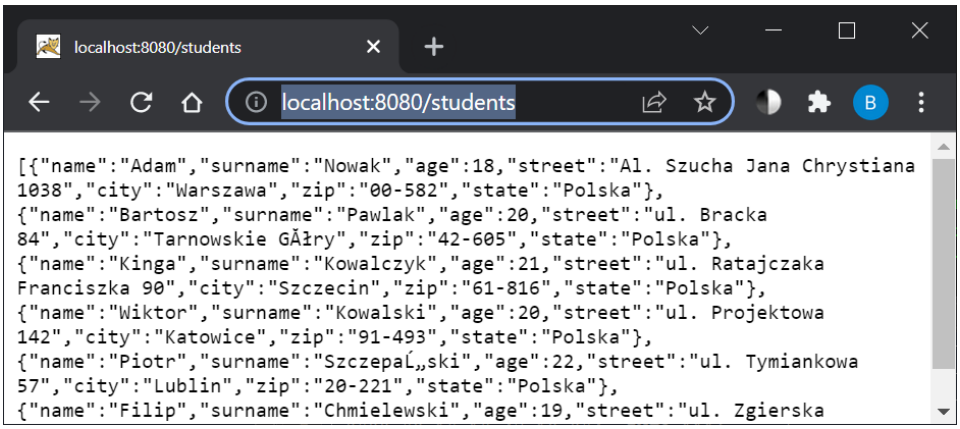
Zmodyfikowany serwis powinien wyglądać jak na rysunku 8.13.

```
@RequiredArgsConstructor
@Service
public class StudentServiceImpl implements StudentService{
    private final StudentRepository studentRepository;
    private final StudentConverter studentConverter;

    @Override
    public List<StudentDto> getAllStudents() {
        return (List<StudentDto>) studentRepository.findAll().stream()
                .map(studentConverter::convert)
                .collect(Collectors.toList());

    }
}
```

Rys. 8.13. Klasa *StudentServiceImpl* po modyfikacjach

8.2.4. Test działania obiektu *DTO*

Ponownie pobierz listę studentów w przeglądarce (Rys. 8.14) lub w *Postman*.



Rys. 8.14. Efekt działania aplikacji z obiektami *DTO* w przeglądarce

Tym razem uzyskano poprawne dane, bez wyrzucenia błędów na konsoli serwera.

Zadanie 8.3. *DTO* w repozytoriach

Przykład z poprzedniego zadania demonstruje jedynie podstawowe możliwości *DTO*. *Spring* zapewnia dużo większe wsparcie dla tego rodzaju mechanizmów, np. związane z ograniczeniem liczby pobieranych bajtów.

Dodaj dodatkowe pole ***attachment*** typu ***byte[]*** do klasy ***Student*** z dodatkową adnotacją ***@Lob***. Z założenia, pole to będzie przechowywało jakiś załącznik (np. dyplom, świadectwo) w postaci pliku ***pdf*** lub ***png***. Pola tego typu zajmują wiele bajtów, ale dla ujednolicenia, każdy taki załącznikw przykładzie niech zajmuje 1MB. Przy 5 studentach pobieranie tego pola z bazy nie stanowi jeszcze problemu. 5MB jest do przyjęcia dla małych aplikacji, ale przy 100 studentach jest to już 100MB, co może być problematyczne. Jeśli pól tego typu w tabeli byłoby kilka, mogą wystąpić poważne problemy z szybkością działania aplikacji. Korzystając z domyślnych metod repozytorium dostarczanych przez *Spring Data*, pobieranych będzie dodatkowe 100MB za każdym razem, niezależnie od tego, że dodatkowe załączniki nie są potrzebne (a i tak będą pobierane z bazy).

Spring pozwala ograniczyć liczbę danych i kontrolować, jakie dane należy pobrać z bazy i można to zrealizować przy użyciu utworzonej już klasy *DTO*.

W interfejsie dziedziczącym po repozytorium *Spring Data JPA* można dodawać własne metody poprzedzone adnotacją ***@Query***, które pozwalają zbudować własne zapytania do bazy za pomocą języka *JPQL* (ang. *Java Persistance Query Language)*. *JPQL* przypomina składnię *SQL*, ale pracuje z obiektami klas w aplikacji *Java*. Przykłady zastosowania zapytań wbudowanych lub metod z adnotacją ***@Query*** realizowane były w zadaniach z laboratorium 5. Na przykład metoda:

```
@Query("select s from Student s")
List<Student> findAllStudents()
```

pozwala pobrać wszystkich studentów z bazy, dzięki zastosowaniu składni języka *JPQL*.

Obiekt *DTO* (ograniczający liczbę pobieranych danych z bazy) w zapytaniu *JPQL* można wykorzystać w klasie ***StudentRepository*** w sposób pokazany na rysunku 8.15.

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long>{
    @Query("select new bp.pai_dto.dtos.StudentDto(s.name, s.surname, s.age, "+
    "s.address.street,s.address.city, s.address.state, s.address.zip) from Student s")
    List<StudentDto> findAllNoAttachment();
}
```

Rys. 8.15. *@Query* w klasie *StudentRepository*

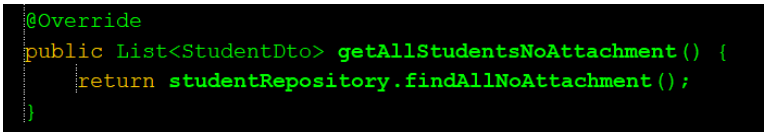
Sztuczka w zapytaniu polega na użyciu konstruktora ***StudentDto***, generowanego dzięki adnotacji ***@AllArgsConstructor*** przez bibliotekę *Lombok* (Przykład 8.2).

Przykład 8.2. Konstruktor z parametrami generowany przez Lombok

```
public StudentDto(final String name, final String surname,
                  final Integer age, final String street,
                  final String city, final String state,
                  final String zip){
       this.name=name;
       this.surname=surname;
       this.age=age;
       //pozostałe przypisania ...
}
```

Wewnątrz zapytania ***@Query*** podano **całą ścieżkę do pakietu z klasą *DTO***. Do pól odwołano się poprzez ***alias.pole***, a ich typy muszą pasować do typów parametrów konstruktora. Dla jednego *DTO* można tworzyć wiele konstruktorów i odpowiednio ich używać dla każdej metody ***@Query***. Ważne jest, że w tym przypadku **nie stosuje się już konwertera**, ponieważ zwracane są gotowe obiekty klasy ***StudentDto*** (metoda ***getAllStudentsNoAttachment()*** w klasie ***StudentService*** i ***StudentServiceImpl*** – Rys. 8.16), więc **typ zwracany**

w repozytorium musi być zmieniony na klasę *DTO*. Wywołując tak zdefiniowaną metodę w kontrolerze, uzyskuje się takie same dane, lecz tutaj nie są pobierane wszystkie dane z bazy, a tylko potrzebne wartości. Dzięki takiemu podejściu można znacznie zoptymalizować i przyspieszyć działanie aplikacji.



Rys. 8.16. Metoda *getAllStudentsNoAttachment()* w klasie *StudentServiceImpl*

Zadanie 8.4. Biblioteka *MapStruct*

Biblioteka ***MapStruct*** jest darmową biblioteką znacznie usprawniającą mapowanie encji na klasy *DTO*. Niestety nie da się jej wskazać podczas tworzenia projektu w *Spring Initializr* i trzeba ręcznie dodać zależność do pliku ***pom.xml:***

```
<dependency>
	<groupId>org.mapstruct</groupId>
	<artifactId>mapstruct</artifactId>
	<version>1.4.2.Final</version>
</dependency>
```

Dodatkowo w sekcji `<plugins>` w elemencie `<build>` należy dodać kolejny `<plugin>` z przykładu 8.3.

Przykład 8.3. Dodatkowy plugin do pliku pom.xml

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <annotationProcessorPaths>
            <path>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>1.4.2.Final</version>
            </path>
            <path>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>1.18.20</version>
            </path>
            <path>
                <groupId>org.projectlombok</groupId>
```

```
                <artifactId>lombok-mapstruct-binding</artifactId>
                <version>0.2.0</version>
            </path>
        </annotationProcessorPaths>
        <compilerArgs>
            <compilerArg>-Amapstruct.defaultComponentModel=spring
        </compilerArg>
        </compilerArgs>
    </configuration>
</plugin>
```

Po przebudowaniu projektu aplikacja jest gotowa do korzystania z biblioteki ***MapStruct***.

Aby przetestować działanie biblioteki, w istniejącym pakiecie ***converters*** utwórz interfejs o nazwie ***StudentMapper*** **z** metodą ***mapStudentToDtoStudent()***, jak na rysunku 8.17.

```
package bp.pai_dto.converters;

import bp.pai_dto.domain.Student;
import bp.pai_dto.dtos.StudentDto;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;

@Mapper
public interface StudentMapper {
    @Mapping(target="name", source="student.name")
    @Mapping(target="surname", source="student.surname")
    @Mapping(target="age", source="student.age")
    @Mapping(target="street", source="student.address.street")
    @Mapping(target="city", source="student.address.city")
    @Mapping(target="state", source="student.address.state")
    @Mapping(target="zip", source="student.address.zip")
    StudentDto mapStudentToStudentDto(Student student);
}
```

Rys. 8.17. Klasa *StudentMapper* w pakiecie *converters*

Adnotacja:

- ***@Mapper*** oznacza interfejs jako źródło, na podstawie którego ***MapStruct*** ma utworzyć implementację dla nowego konwertera.
- ***@Mapping*** pozwala wskazać ***target*** (nazwę pola w klasie docelowej) oraz ***source*** (nazwę pola w klasie źródłowej).

Sama nazwa metody nie ma znaczenia, **musi się jedynie zgadzać zwracany typ** (***StudentDto***) oraz **typ parametru metody** (***Student***). Znaczenie ma natomiast **nazwa argumentu** w parametrze ***source*** (***source = "nazwaArgumentu.pole"***).

Jedną z ważniejszych zalet tej biblioteki jest to, że jeżeli pola nie są zagnieżdżone oraz nazwa i typ pola z *DTO* zgadza się z nazwą i typem pola encji, to nie trzeba definiować dodatkowego mapowania. ***MapStruct*** sam zorientuje się, co ma wziąć i do czego przypisać.

Interfejs z rysunku 8.17 można zatem uprościć do postaci jak na rysunku 8.18.

```
@Mapper
public interface StudentMapper {
    @Mapping(target="street", source="student.address.street")
    @Mapping(target="city", source="student.address.city")
    @Mapping(target="state", source="student.address.state")
    @Mapping(target="zip", source="student.address.zip")
    StudentDto mapStudentToStudentDto(Student student);
}
```

Rys. 8.18. Interfejs *StudentMapper* po uproszczeniu

Po tej modyfikacji można „wstrzyknąć" zdefiniowany maper do serwisu studenta (***StudentServiceImpl***) i wykorzystać go zamiast starego konwertera (Rys. 8.19).

```
final private StudentMapper studentMapper;
@Override
public List<StudentDto> getAllStudents() {
return studentRepository.findAll().stream()
            .map(studentMapper::mapStudentToStudentDto)
            .collect(Collectors.toList());
}
```

Rys. 8.19. Metoda *getAllStudents* z maperem

Testując działanie aplikacji po raz kolejny, wynik powinien być taki sam jak w przypadku poprzedniego konwertera, ale tym razem zautomatyzowano proces przekształcania encji w obiekt *DTO* (oszczędność czasu).
Biblioteka *MapStruct* ma jeszcze wiele innych możliwości, z których korzysta się w bardziej złożonych aplikacjach.

Więcej na temat tej biblioteki można znaleźć na stronach:

- dokumentacja *MapStruct*: *MapStruct 1.5.0.Beta2 Reference Guide*
- krótki poradnik z podstawami biblioteki:
  https://www.baeldung.com/mapstruct

Laboratorium opracowane zostało we współpracy ze studentami Informatyki: Kamilem Jaskotem i Sebastianem Iwanowskim.

Laboratorium 9. *Spring* i *JSON Web Token*

Cel zajęć

Wykonanie zadań z laboratorium pozwoli studentom poznać zaawansowane elementy konfiguracji *Spring Security*, konieczne do implementacji w aplikacjach uwierzytelniających klienta za pomocą tokena *JSON Web Token* (*JWT*) z wykorzystaniem ustalonych danych użytkownika.

Zakres tematyczny

- Przygotowanie *REST API* z autentykacją użytkownika o ustalonym loginie i haśle za pomocą tokena *JWT*.
- Konfiguracja *Spring Security* z autentykacją tokenem *JWT* (definicja klas konfiguracyjnych, generowanie i walidacja *JWT*).
- Testowanie *REST API* z autentykacją tokenem *JWT* w narzędziu *Postman*.

Proces konfiguracji *Spring Security*, generowania i walidacji tokena *JWT*, zrealizowano według przykładu prezentowanego na stronie [10]: *https://www.techgeeknext.com/spring/spring-boot-security-token-authentication-jwt*.
Definicje klas prezentowanych w przykładach niniejszego laboratorium, niezbędne do prawidłowej konfiguracji *Spring Security* do pracy z tokenem *JWT*, pochodzą w całości z pozycji [10]. W celu lepszego zrozumienia działania metod prezentowanych klas, komentarze w kodach, zostały przetłumaczone na język polski.

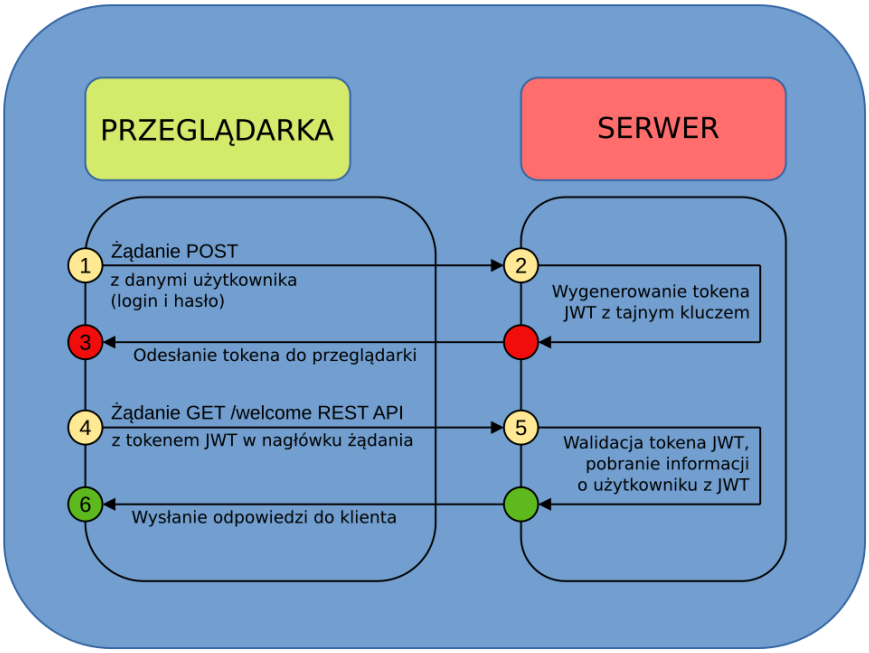
Wprowadzenie

Autentykacja użytkownika za pomocą *JWT* przebiega w następujących krokach:

- Użytkownik uwierzytelnia się wysyłając swoje dane (login i hasło).
- Po udanej autentykacji, serwer generuje token *JWT*, w którym są zaszyte dane użytkownika i informacja o jego uprawnieniach (ang. *credentials*) do korzystania z zasobów serwera oraz data ważności tokena.
- Serwer podpisuje (i jeśli trzeba koduje token *JWT*) oraz wysyła token do klienta w odpowiedzi na pierwsze żądanie.
- W oparciu o datę ważności ustanowioną po stronie serwera – klient odpowiednio długo przechowuje token *JWT* i przesyła go w nagłówku każdego kolejnego żądania.
- Klient identyfikuje użytkownika w oparciu o ten token, wobec czego nie trzeba za każdym kolejnym żądaniem przesyłać loginu i hasła w celu

uwierzytelnienia (robi się to tylko za pierwszym razem, a w odpowiedzi serwer wysyła token, który klient wykorzystuje do autentykacji kolejnych żądań).

Spring Boot REST Authentication z wykorzystaniem tokena *JWT* przedstawia rysunek 9.1.



Rys. 9.1. Uwierzytelnianie za pomocą tokena *JWT* [10]

Zadanie 9.1. *Spring Boot* i *JWT*

Wygeneruj nowy projekt o nazwie np. ***PAI_testjwt*** za pomocą narzędzia *Spring Initializr* z dołączoną zależnością *Web*. W pakiecie głównym projektu dodaj pakiet ***controller***, a w nim utwórz klasę ***TestController*** (Przykład 9.1).

Przykład 9.1. Klasa TestController

```
package bp.PAI_testjwt.controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class TestController {

    @RequestMapping({ "/hello" })
    public String welcomePage() {
```

```
        return "Welcome!";
    }
}
```

Uruchom i przetestuj aplikację z adresu ***http://localhost:8080/hello*** (metodą *GET*) a następnie do pliku ***pom.xml*** dołącz zależności dla *Spring Security* i *JWT*:

```
<dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
       <groupId>io.jsonwebtoken</groupId>
       <artifactId>jjwt</artifactId>
       <version>0.9.1</version>
</dependency>
```

Po ponownym zbudowaniu projektu z dodanymi zależnościami i uruchomieniu, aby zobaczyć efekt działania dla adresu ***/hello*** należy się zalogować jako użytkownik ***user*** za pomocą wygenerowanego przez serwer hasła. Jest to domyślna konfiguracja *Spring Security*, którą teraz należy zmodyfikować tak, aby autentykacja odbywała się za pomocą przekazanego tokena *JWT*.

9.1.1. Konfiguracja *Spring Security* z *JWT*

W celu konfiguracji *Spring Security* oraz wygenerowania i sprawdzenia poprawności tokena *JWT* należy obsłużyć dwie akcje:

- **Generowanie *JWT***: punkt końcowy (ang. *endpoint*) ***/authenticate*** zostanie wykorzystany w celu przekazania metodą *POST* nazwy i hasła użytkownika (***username***, ***password***) i wygenerowania tokena *JWT*.
- **Walidacja *JWT***: zostanie zrealizowana przez utworzoną już akcję kontrolera (*GET*) dla punktu końcowego ***/hello*** z przesłaniem otrzymanego w odpowiedzi z serwera tokena *JWT*.

Utwórz pakiet ***config*** a w nim zdefiniuj klasę ***JwtTokenUtil*** (Przykład 9.2) z metodami niezbędnymi do generowania i walidacji tokena *JWT*. Strukturę plików gotowego projektu pokazano na rysunku 9.2.

Przykład 9.2. Klasa JwtTokenUtil [10]

```
package bp.PAI_testjwt.config;

import java.io.Serializable;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;
```

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

@Component
public class JwtTokenUtil implements Serializable {
    private static final long serialVersionUID = -2550185165626007488L;
    public static final long JWT_TOKEN_VALIDITY = 5 * 60 * 60;

    @Value("")
    private String secret;

    public String getUsernameFromToken(String token) {
        return getClaimFromToken(token, Claims::getSubject);
    }

    public Date getIssuedAtDateFromToken(String token) {
        return getClaimFromToken(token, Claims::getIssuedAt);
    }

    public Date getExpirationDateFromToken(String token) {
        return getClaimFromToken(token, Claims::getExpiration);
    }

    public <T> T getClaimFromToken(String token,
      Function<Claims, T> claimsResolver) {
        final Claims claims = getAllClaimsFromToken(token);
        return claimsResolver.apply(claims);
    }

    private Claims getAllClaimsFromToken(String token) {
        return Jwts.parser()
               .setSigningKey(secret).parseClaimsJws(token)
               .getBody();
    }

    private Boolean isTokenExpired(String token) {
        final Date expiration = getExpirationDateFromToken(token);
        return expiration.before(new Date());
    }

    private Boolean ignoreTokenExpiration(String token) {
        // podaj tokeny, dla których wygaśnięcie jest ignorowane
        return false;
    }

    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
```

```
        return doGenerateToken(claims, userDetails.getUsername());
    }

    private String doGenerateToken(Map<String, Object> claims,
     String subject) {
       return Jwts.builder().setClaims(claims)
          .setSubject(subject)
          .setIssuedAt( new Date(System.currentTimeMillis()))
          .setExpiration(new Date(
              System.currentTimeMillis()+JWT_TOKEN_VALIDITY*1000))
          .signWith(SignatureAlgorithm.HS512, secret).compact();
    }

    public Boolean canTokenBeRefreshed(String token) {
        return (!isTokenExpired(token) || ignoreTokenExpiration(token));
    }

    public Boolean validateToken(String token, UserDetails userDetails){
        final String username = getUsernameFromToken(token);
        return (username.equals(userDetails.getUsername()) &&
                !isTokenExpired(token));
    }
}
```

9.1.2. Pobranie nazwy i hasła użytkownika

W pakiecie ***org.springframework.security.core.userdetails*** *Spring Security* dostarcza interfejs ***UserDetailsService***. Interfejs ten będzie wykorzystany, aby sprawdzić ***username***, ***password*** oraz uprawnienia danego użytkownika (ang. *GrantedAuthorities*).

Interfejs posiada tylko jedną metodę ***loadUserByUsername***. Menadżer autentykacji (***Authentication Manager***) wywołuje tę metodę, aby pobrać dane żytkownika z bazy danych w celu porównania ich z dostarczonymi danymi. W tym zadaniu, do przetestowania konfiguracji *JWT*, wykorzystany zostanie użytkownik z ustalonym, zahaszowym już hasłem (***BCrypt password***). Na kolejnym laboratorium 10 aplikacja zostanie rozbudowana tak, aby korzystała z danych użytkowników zapisanych w bazie *MySQL*.

Utwórz kolejny pakiet ***service*** z klasą ***JwtUserDetailsService*** (Przykład 9.3), która nadpisze metodę interfejsu ***UserDetailsService***.

Przykład 9.3. Klasa JwtUserDetailsService [10]

```
package bp.PAI_testjwt.service;

import java.util.ArrayList;
import org.springframework.security.core.userdetails.User;
```

```
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.Service;
@Service
public class JwtUserDetailsService implements UserDetailsService {

  @Override
  public UserDetails loadUserByUsername(String username) throws
    UsernameNotFoundException {
      if ("pai".equals(username)) {
        return new User("pai",
         "$2a$10$slYQmyNdGzTn7ZLBXBChFOC9f6kFjAqPhccnP6DxlWXx2lPk1C3G6",
         new ArrayList<>());
      } else {
        throw new UsernameNotFoundException(
           "User not found with username: " + username);
      }
  }
}
```

W zadaniu wykorzystano ustalone dane użytkownika i przykładowe, zahaszowane już (bezpieczną funkcją haszującą *BCrypt* wykorzystaną przez serwer) hasło dla łańcucha "password":

```
username="pai"
password="$2a$10$slYQmyNdGzTn7ZLBXBChFOC9f6kFjAqPhccnP6DxlWXx2lP
k1C3G6"
```

9.1.3. Kontroler do autentykacji użytkownika

Do autentykacji użytkownika należy zdefiniować ***JwtAuthenticationController*** z akcją obsługującą żądanie (z danymi ***username*** i ***passsword***), przekazane metodą *POST* do *API*. Za pomocą menadżera autentykacji (ang. *Authentication Manager*) zostanie przeprowadzona autentykacja użytkownika i jego uprawnień (ang. *credentials*). Jeśli weryfikacja zakończy się sukcesem, ***JWTTokenUtil*** wygeneruje token *JWT*, który następnie zostanie przekazany do klienta.

W pakiecie ***controller*** zdefiniuj klasę kontrolera ***JwtAuthenticationController*** (Przykład 9.4).

Przykład 9.4. Klasa JwtAuthenticationController [10]

```
package bp.PAI_testjwt.controller;
import bp.PAI_testjwt.config.JwtTokenUtil;
import bp.PAI_testjwt.model.JwtRequest;
import bp.PAI_testjwt.model.JwtResponse;
import java.util.Objects;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.DisabledException;
import
org.springframework.security.authentication.UsernamePasswordAuthenticati
onToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@CrossOrigin
public class JwtAuthenticationController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtTokenUtil jwtTokenUtil;

    @Autowired
    private UserDetailsService jwtInMemoryUserDetailsService;

    @RequestMapping(value = "/authenticate",
                    method = RequestMethod.POST)
    public ResponseEntity<?> generateAuthenticationToken(@RequestBody
      JwtRequest authenticationRequest) throws Exception {
        authenticate(authenticationRequest.getUsername(),
                     authenticationRequest.getPassword());
        final UserDetails userDetails = jwtInMemoryUserDetailsService
           .loadUserByUsername(authenticationRequest.getUsername());
        final String token = jwtTokenUtil.generateToken(userDetails);
        return ResponseEntity.ok(new JwtResponse(token));
    }

    private void authenticate(String username, String password)
          throws Exception {
        Objects.requireNonNull(username);
        Objects.requireNonNull(password);
        try {
            authenticationManager.authenticate(new
             UsernamePasswordAuthenticationToken(username, password));
        } catch (DisabledException e) {
```

```
            throw new Exception("USER_DISABLED", e);
        } catch (BadCredentialsException e) {
            throw new Exception("INVALID_CREDENTIALS", e);
        }
    }
}
```

Klasa tego kontrolera korzysta z obiektów ***JwtRequest*** i ***JwtResponse***, które zdefiniowane zostaną w kolejnych dwóch punktach. Obie klasy dodaj do kolejnego pakietu ***model*** (Rys. 9.2).

9.1.4. Klasa *JwtRequest*

Obiekt klasy ***JwtRequest*** (Przykład 9.5) jest wykorzystywany do pobrania danych użytkownika (***username*** i ***password***) z żądania wysłanego przez klienta.

Przykład 9.5. Klasa JwtRequest [10]

```
package bp.PAI_testjwt.model;
import java.io.Serializable;
public class JwtRequest implements Serializable {
    private static final long serialVersionUID = 5926468583005150707L;
    private String username;
    private String password;

    // domyślny konstruktor dla parsowania JSON
    public JwtRequest() {
    }

    public JwtRequest(String username, String password) {
        this.setUsername(username);
        this.setPassword(password);
    }

    public String getUsername() {
        return this.username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return this.password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

9.1.5. Klasa *JwtResponse*

Obiekt klasy ***JwtResponse*** (Przykład 9.6) jest wykorzystywany w celu utworzenia odpowiedzi z tokenem *JWT*, zwracanej do klienta.

Przykład 9.6. Klasa JwtResponse [10]

```
package bp.PAI_testjwt.model;
import java.io.Serializable;
public class JwtResponse implements Serializable {

    private static final long serialVersionUID = -8091879091924046844L;
    private final String jwttoken;

    public JwtResponse(String jwttoken) {
        this.jwttoken = jwttoken;
    }

    public String getToken() {
        return this.jwttoken;
    }
}
```

Pozostaje jeszcze do zdefiniowania najważniejsza klasa filtra ***JWT*** oraz konfiguracja ***Web Security***.

9.1.6. Filtr *JWT* w pakiecie *config*

Klasa ***JwtRequestFilter*** (Przykład 9.7) obsługuje wszystkie przychodzące żądania, waliduje tokeny *JWT* (przesyłane z żądaniem) i umieszcza w kontekście aplikacji informację, że zalogowany użytkownik jest uwierzytelniony.

Przykład 9.7. Klasa JwtRequestFilter [10]

```
package bp.PAI_testjwt.config;

import bp.PAI_testjwt.service.JwtUserDetailsService;
import java.io.IOException;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.authentication.UsernamePasswordAuthenticati
onToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
```

```
import
org.springframework.security.web.authentication.WebAuthenticationDetails
Source;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import io.jsonwebtoken.ExpiredJwtException;

@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUserDetailsService jwtUserDetailsService;

    @Autowired
    private JwtTokenUtil jwtTokenUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                    HttpServletResponse response, FilterChain chain)
            throws ServletException, IOException {
        final String requestTokenHeader =
                     request.getHeader("Authorization");
        String username = null;
        String jwtToken = null;
        // Token JWT ma postać "Bearer token". Należy usunąć słowo
        // Bearer i pobrać tylko właściwy token
        if (requestTokenHeader!=null &&
           requestTokenHeader.startsWith("Bearer ")) {
            jwtToken = requestTokenHeader.substring(7);
            try {
                username = jwtTokenUtil.getUsernameFromToken(jwtToken);
            } catch (IllegalArgumentException e) {
                System.out.println("Unable to get JWT Token");
            } catch (ExpiredJwtException e) {
                System.out.println("JWT Token has expired");
            }
        } else {
            logger.warn("JWT Token does not begin with Bearer String");
        }
        // Po uzyskaniu tokena należy go zweryfikować
        if (username != null &&
       SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails =
               this.jwtUserDetailsService.loadUserByUsername(username);
            // Jeśli token jest poprawny, będzie konfigurowane
            // uwierzytelnianie w Spring Security
            if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {
                UsernamePasswordAuthenticationToken
                 usernamePasswordAuthenticationToken = new
                   UsernamePasswordAuthenticationToken(
```

```
                    userDetails, null,
                    userDetails.getAuthorities());
                usernamePasswordAuthenticationToken
                        .setDetails(new
                WebAuthenticationDetailsSource().buildDetails(request));
            // Po powyższych ustawieniach, bieżący użytkownik jest
            // traktowany jako uwierzytelniony, Konfiguracja
            // Spring Security - zakończona powodzeniem
            SecurityContextHolder.getContext()
              .setAuthentication(usernamePasswordAuthenticationToken);
           }
        }
        chain.doFilter(request, response);
    }
}
```

9.1.7. Klasa *JwtAuthenticationEntryPoint*

Do pakietu ***config*** dodaj klasę ***JwtAuthenticationEntryPoint*** (Przykład 9.8), która wysyła kod błędu 401 w odpowiedzi na brak autentykacji użytkownika.

Przykład 9.8. Klasa JwtAuthenticationEntryPoint [10]

```
package bp.PAI_testjwt.config;
import java.io.IOException;
import java.io.Serializable;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

@Component
public class JwtAuthenticationEntryPoint implements
AuthenticationEntryPoint, Serializable {

    private static final long serialVersionUID = -7858869558953243875L;

    @Override
    public void commence(HttpServletRequest request,
                         HttpServletResponse response,
                         AuthenticationException authException)
              throws IOException {
          response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
                             "Unauthorized");
    }
}
```

9.1.8. Konfiguracja *Web Security*

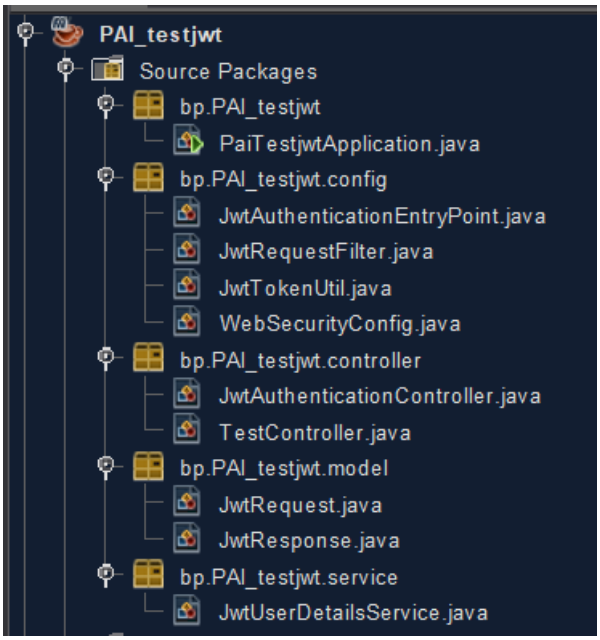
Kolejna klasa ***WebSecurityConfig*** w pakiecie ***config*** (Przykład 9.9) rozszerza ***WebSecurityConfigurerAdapter***, dzięki czemu można dowolnie skonfigurować ***WebSecurity i HTTPSecurity***.

Przykład 9.9. Klasa WebSecurityConfig [10]

```
package bp.PAI_testjwt.config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.builders.A
uthenticationManagerBuilder;
import
org.springframework.security.config.annotation.method.configuration.Enab
leGlobalMethodSecurity;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity
;
import
org.springframework.security.config.annotation.web.configuration.EnableW
ebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecu
rityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenti
cationFilter;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

    @Autowired
    private UserDetailsService jwtUserDetailsService;

    @Autowired
    private JwtRequestFilter jwtRequestFilter;
```

```
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth)
                throws Exception {
        // Konfiguracja menadżera AuthenticationManager tak, aby
        // wiedział skąd załadować użytkownika w celu dopasowania
        // danych uwierzytelniających
        // Zastosowano haszowanie hasła za pomocą BCryptPasswordEncoder
        auth.userDetailsService(jwtUserDetailsService).
                  passwordEncoder(passwordEncoder());
    }

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

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

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception
{
        // W tym przykładzie nie potrzebne jest zabezpieczenie CSRF
        httpSecurity.csrf().disable()
            // te żądania nie wymagają uwierzytelniania
            .authorizeRequests().antMatchers("/authenticate")
            .permitAll().
            // pozostałe żądania wymagają uwierzytelniania
            anyRequest().authenticated().and()
            // zastosowana sesja bezstanowa - sesja nie przechowuje
            // stanu użytkownika.
            .exceptionHandling()
            .authenticationEntryPoint(jwtAuthenticationEntryPoint).and()
            .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
            // Dodanie filtra do walidacji tokena przy każdym żądaniu
            httpSecurity.addFilterBefore(jwtRequestFilter,
                  UsernamePasswordAuthenticationFilter.class);
    }
}
```

Struktura całego projektu przedstawiona jest na rysunku 9.2.



Rys. 9.2. Struktura gotowego projektu

Zadanie 9.2. Test działania w narzędziu *Postman*

Do pliku ***application.properties*** dodaj wiersze:

```
spring.main.allow-circular-references=true
jwt.secret=Programowanie_ai2021
```

Korzystając z narzędzia *Postman*, wyślij żądanie typu POST na adres ***localhost:8080/authenticate*** z przekazaniem loginu i hasła (Rys. 9.3):

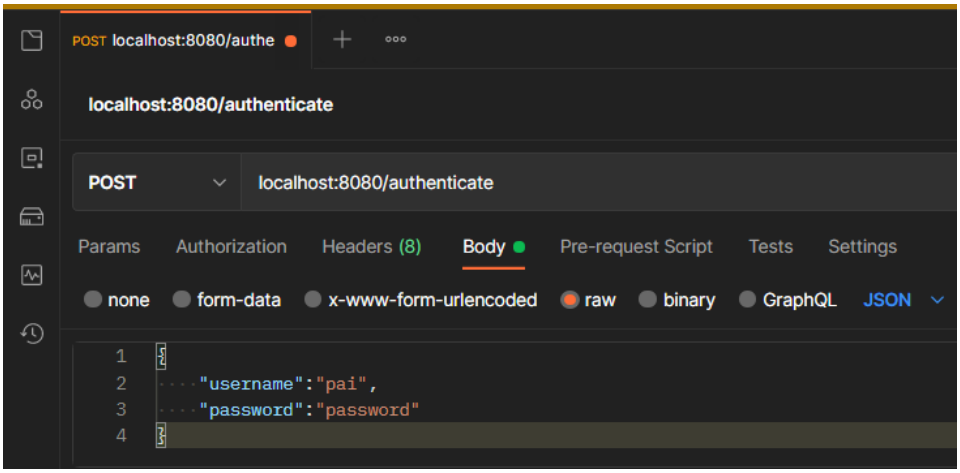
```
{   "username":"pai", "password":"password" }
```

Po sprawdzeniu danych użytkownika, jeżeli są prawidłowe, w odpowiedzi serwer odsyła token (Rys. 9.4).

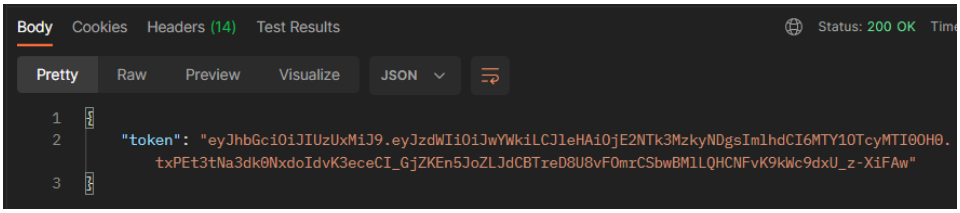
UWAGA! Jeśli korzystasz z wersji języka *Java >8*, może się pojawić wyjątek: *java.lang.NoClassDefFoundError: javax/xml/bind/JAXBException*

Rozwiązaniem tego problemu jest dodanie do pliku *pom.xml* zależności:

```
<dependency>
      <groupId>javax.xml.bind</groupId>
      <artifactId>jaxb-api</artifactId>
      <version>2.3.0</version>
</dependency>
```



Rys. 9.3. ***Postman*** **– żądanie z danymi użytkownika**

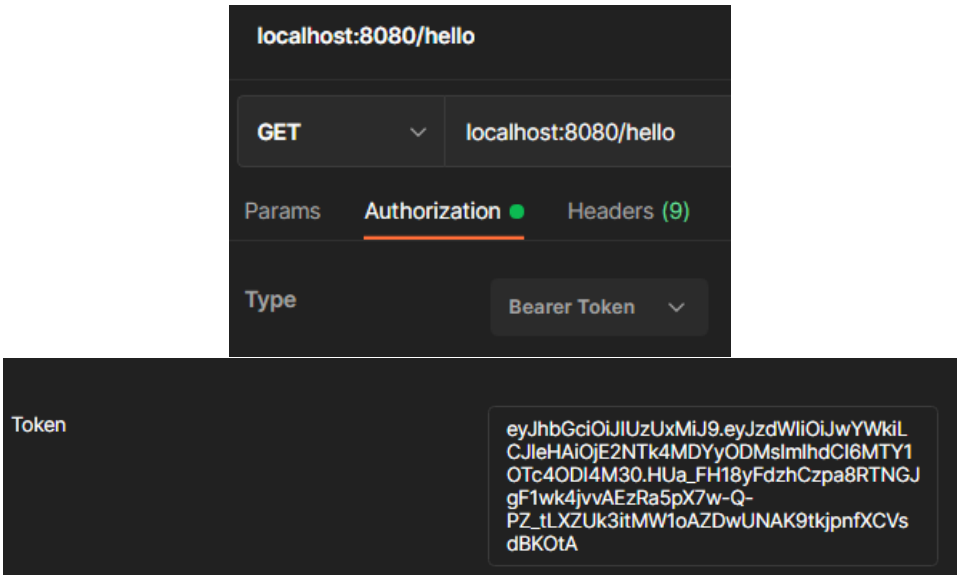


Rys. 9.4. ***Postman*** **– odpowiedź serwera z tokenem** ***JWT***

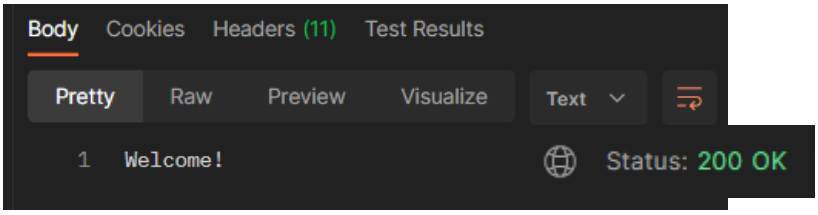
Skopiuj otrzymany token tak, aby można go było wykorzystać do uwierzytelnienia użytkownika w przypadku żądań wymagających autentykacji tokenem.

Przetestuj działanie autentykacji, wysyłając na adres ***localhost:8080/hello***, żądanie wymagające podania tokena do autentykacji. Token należy przekazać w nagłówku ***Authorization*** żądania (Rys. 9.5). Wybierz opcję ***Authorisation*** i wskaż typ tokena jako **Bearer token** oraz w odpowiednim polu tekstowym wklej token skopiowany (bez znaków cudzysłowia) z poprzedniego punktu i wyślij żądanie.

W odpowiedzi, po poprawnej weryfikacji tokena serwer prześle odpowiedź (***Status 200 OK***) postaci jak na rysunku 9.6.

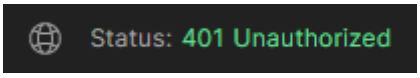


Rys. 9.5. *Postman* – żądanie z tokenem do autentykacji



Rys. 9.6. *Postman* – odpowiedź serwera po poprawnej weryfikacji tokena

Próba dostępu do strony bez przesłania właściwego tokena zakończy się niepowodzeniem – serwer zwróci odpowiedź ***Status 401 Unauthorize*** (Rys. 9.7).



Rys. 9.7. *Postman* – odpowiedź serwera po niepoprawnej weryfikacji tokena

Taką samą odpowiedź jak na rysunku 9.7 serwer zwróci w przypadku próby przesłania w żądaniu *POST*, niepoprawnych danych użytkownika (sprawdź).

W niniejszym laboratorium test poprawności przechodzi tylko 1 użytkownik o podanych wcześniej parametrach, które zakodowano na sztywno w metodzie ***loadUserByUsername*** klasy ***JwtU serDetailsService***. W kolejnym laboratorium zostanie wykorzystana baza danych *MySQL* z danymi do uwierzytelnienia użytkownika.

Laboratorium 10. *Spring*, *JWT* i *MySQL*

Cel zajęć

Realizacja zadań z laboratorium umożliwi studentom przygotowanie konfiguracji *Spring Security* z autentykacją za pomocą tokena *JWT*, podobnie jak w poprzednim laboratorium, ale z wykorzystaniem danych użytkowników przechowywanych w bazie danych *MySQL*.

Zakres tematyczny

- Utworzenie aplikacji z wykorzystaniem klas zdefiniowanych na laboratorium 9 tak, aby aplikacja korzystała z danych użytkowników przechowywanych w bazie *MySQL*.
- Konfiguracja *Spring Security*, definicja klas do pracy z danymi użytkownika z bazy danych, *i*mplementacja klasy kontrolera do obsługi autentykacji użytkowników tokenem *JWT*.
- Testowanie autentykacji w narzędziu *Postman*.

Proces konfiguracji *Spring Security*, generowania i walidacji tokena *JWT*, zrealizowano według przykładu prezentowanego na stronie [11]:
https://www.techgeeknext.com/spring/spring-boot-security-token-authentication-jwt-mysql

Definicje klas prezentowanych w przykładach niniejszego laboratorium, niezbędne do prawidłowej konfiguracji *Spring Security* do pracy z tokenem *JWT* i bazą użytkowników w *MySQL* pochodzą w całości ze strony [11]. W celu lepszego zrozumienia metod prezentowanych klas, komentarze w kodach, zostały przetłumaczone na język polski.

Zadanie 10.1. *JWT* i *MySQL*

Wygeneruj nowy projekt o nazwie np. ***PAI_jwt*** za pomocą narzędzia *Spring Initializr* z dołączonymi zależnościami *Web*, *Security*, *JPA* i *MySQL* (Rys. 10.1).

Rys. 10.1. Tworzenie projektu w *Spring Initializr*

Do pliku ***pom.xml*** dodaj zależność dla *JWT*:

```
<dependency>
	<groupId>io.jsonwebtoken</groupId>
	<artifactId>jjwt</artifactId>
	<version>0.9.1</version>
</dependency>
```

Uruchom serwer *MySQL* i do pliku ***application.properties*** dodaj konfigurację, jak pokazuje Przykład 10.1.

Przykład 10.1. Plik application.properties

```
## secret key wykorzystywany przez algorytm haszujący,
# dodawany przez JWT do kombinacji z nagłówkiem (header)
# i ładunkiem (payload) z danymi
jwt.secret=Programowanie_ai2021
```

```
## Spring DATASOURCE (konfiguracja i właściwości źródła danych
spring.datasource.url =
jdbc:mysql://localhost:3306/notesdb?createDatabaseIfNotExist=true&allowP
ublicKeyRetrieval=true&useSSL=false
spring.datasource.username = root
spring.datasource.password =
spring.datasource.platform=mysql
spring.datasource.initialization-mode=always

## Właściwości dla Hibernate
# dialekt Hibernate
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQL8Dialect

# Ustawienia dla Hibernate dla operacji ddl
# (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = create-drop
```

Zbuduj projekt, a następnie do głównego pakietu dodaj podpakiety: ***config***, ***controller***, ***model***, ***repository*** i ***service***.

10.1.1. Klasy *UserDao* i *UserDto*

W pakiecie ***model*** utwórz klasę encji ***UserDao*** (Przykład 10.2) oraz klasę ***UserDto*** (Przykład 10.3). Klasa ***UserDto*** pobiera wartości użytkownika oraz przekazuje je do warstwy *DAO* w celu utrwalenia w bazie danych.

Przykład 10.2. Klasa UserDao

```
package bp.PAI_jwt.model;
import javax.persistence.*;
import net.minidev.json.annotate.JsonIgnore;

@Entity
@Table(name = "user")
public class UserDao {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Column
    private String username;
    @Column
    @JsonIgnore
    private String password;

    public String getUsername() {
        return username;
    }
```

```
    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Przykład 10.3. Klasa UserDto w pakiecie bp.PAI_jwt.model;

```
/* Klasa modelu UserDto odpowiada za pobranie wartości od użytkownika
   i przekazanie ich do warstwy DAO w celu wstawienia do bazy danych.
*/

public class UserDto {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

10.1.2. Repozytorium *JPA*

W pakiecie ***repository*** utwórz interfejs ***UserRepository*** rozszerzający repozytorium ***CrudRepository*** (Przykład 10.4). Repozytorium pozwala na dostęp do informacji o użytkownikach przechowywanych w bazie danych.

Przykład 10.4. Repozytorium UserRepository

```
package bp.PAI_jwt.repository;
import bp.PAI_jwt.model.UserDao;
import org.springframework.data.repository.CrudRepository;
```

```
public interface UserRepository extends CrudRepository<UserDao, Integer>
{
    UserDao findByUsername(String username);
}
```

10.1.3. Konfiguracja *Web Security*

Do pakietu ***config*** dodaj klasę ***WebSecurityConfig*** (Przykład 10.5), w której należy wskazać punkty końcowe ***/autheticate*** oraz ***/register***, niewymagające autentykacji.

W klasie zostały „wstrzyknięte" obiekty klas ***JwtAuthenticationEntryPoint***, ***JwtUserDetailsService*** i ***JwtRequestFilter***, które zostaną zdefiniowane w dalszych punktach.

Przykład 10.5. Klasa WebSecurityConfig [11]

```
package bp.PAI_jwt.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.builders.A
uthenticationManagerBuilder;
import
org.springframework.security.config.annotation.method.configuration.Enab
leGlobalMethodSecurity;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity
;
import
org.springframework.security.config.annotation.web.configuration.EnableW
ebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecu
rityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenti
cationFilter;

@Configuration
@EnableWebSecurity
```

```
@EnableGlobalMethodSecurity(prePostEnabled = true)

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired
private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

@Autowired
private UserDetailsService jwtUserDetailsService;

@Autowired
private JwtRequestFilter jwtRequestFilter;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
          throws Exception {
  // Konfiguracja menadżera AuthenticationManager, aby wiedział skąd
  // pobrać dane użytkownika do sprawdzenia
  // Haszowania hasła BCryptPasswordEncoder
  auth.userDetailsService(jwtUserDetailsService).
  passwordEncoder(passwordEncoder());
}

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

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

@Override
protected void configure(HttpSecurity httpSecurity) throws Exception {
  // W przykładzie nie ma potrzeby stosowania zabezpieczenia przed CSRF
  httpSecurity.csrf().disable()
   // poniższe żądanie nie wymaga uwierzytelniania
   .authorizeRequests().antMatchers("/authenticate", "/register")
   .permitAll()
   // wszystkie pozostałe żądania wymagają uwierzytelniania
   .anyRequest().authenticated().and()
   // sesja jest bezstanowa, nie przechowuje stanu użytkownika
   .exceptionHandling()
   .authenticationEntryPoint(jwtAuthenticationEntryPoint)
   .and().sessionManagement()
   .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

  // Dodanie filtra w celu walidacji tokena dla każdego żądania
```

```
  httpSecurity.addFilterBefore(jwtRequestFilter,
                UsernamePasswordAuthenticationFilter.class);
 }
}
```

Do odpowiednich pakietów projektu dodaj klasy ***JwtRequestFilter, JwtTokenUtil, JwtAuthenticationEntryPoint*** (zdefiniowane w zadaniach z laboratorium 9) (Rys. 10.2).

10.1.4. Implementacja klasy *JwtAuthenticationController*

Do pakietu ***model*** dodaj definicje 2 klas zdefiniowanych w poprzednim laboratorium: ***JwtRequest*** i ***JwtResponse*** (Rys. 10.2). Utwórz klasę kontrolera ***JwtAuthenticationController*** z punktami końcowymi do autentykacji i rejestracji użytkownika (Przykład 10.6).

Przykład 10.6. Klasa JwtAuthenticationController [11]

```
package bp.PAI_jwt.config;

import bp.PAI_jwt.config.JwtTokenUtil;
import bp.PAI_jwt.model.JwtRequest;
import bp.PAI_jwt.model.JwtResponse;
import bp.PAI_jwt.model.UserDto;
import bp.PAI_jwt.service.JwtUserDetailsService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.DisabledException;
import
org.springframework.security.authentication.UsernamePasswordAuthenticati
onToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.web.bind.annotation.*;

@RestController
@CrossOrigin
public class JwtAuthenticationController {
    @Autowired
    private AuthenticationManager authenticationManager;
    @Autowired
    private JwtTokenUtil jwtTokenUtil;
    @Autowired
    private JwtUserDetailsService userDetailsService;
```

```
    @RequestMapping(value = "/authenticate",
                    method = RequestMethod.POST)
    public ResponseEntity<?> createAuthenticationToken(@RequestBody
                    JwtRequest authenticationRequest) throws Exception {
        authenticate(authenticationRequest.getUsername(),
                     authenticationRequest.getPassword());
        final UserDetails userDetails =
           userDetailsService.loadUserByUsername(authenticationRequest
          .getUsername());
        final String token = jwtTokenUtil.generateToken(userDetails);
        return ResponseEntity.ok(new JwtResponse(token));
    }

    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public ResponseEntity<?> saveUser(@RequestBody UserDto user)
           throws Exception {
        return ResponseEntity.ok(userDetailsService.save(user));
    }

    private void authenticate(String username, String password)
            throws Exception {
        try {
          authenticationManager.authenticate(new
             UsernamePasswordAuthenticationToken(username, password));
        } catch (DisabledException e) {
            throw new Exception("USER_DISABLED", e);
        } catch (BadCredentialsException e) {
            throw new Exception("INVALID_CREDENTIALS", e);
        }
    }
}
```

Zwróć uwagę na adnotację ***@CrossOrigin*** poprzedzającą klasę kontrolera.

CORS (ang. *Cross-Origin Resource Sharing*) to mechanizm bezpieczeństwa, który wykorzystuje dodatkowe nagłówki *HTTP*, aby poinformować przeglądarkę, czy ma udostępnić dane zwrócone klientowi. Serwer decyduje, czy klient jest klientem zaufanym i na tej podstawie ustawia odpowiednie nagłówki, dzięki którym przeglądarka wie, czy ma udostępnić dane klientowi. W ten sposób przeglądarka zabezpiecza użytkownika przed m.in. atakami typu *CrossSite Request Forgery* (atak *CSRF* – polega na wysyłaniu w imieniu klienta żądań *HTTP* do złośliwych serwisów, wykorzystując dane użytkownika np. sesje, ciasteczka, stan zalogowania itp.).

Obsługa *CORS* w *Spring* jest możliwa za pomocą adnotacji ***@CrossOrigin*** poprzedzającej metodę lub klasę kontrolera. Ta adnotacja zezwala przeglądarce udostępniać dane pochodzące ze wszystkich źródeł (ang. *origin*). Czas życia

odpowiedzi (ang. *response*) jest utrzymywany w cache przeglądarki przez 30 minut (domyślna wartość parametru *maxAge*).

W celu przetestowania aplikacji, do pakietu ***controller*** dodaj klasę ***TestController*** jak w poprzednim laboratorium (Rys. 10.2). Klasę ***TestController*** także poprzedź adnotacją ***@CrossOrigin***.

Struktura plików gotowego projektu przedstawiona jest na rysunku 10.2.

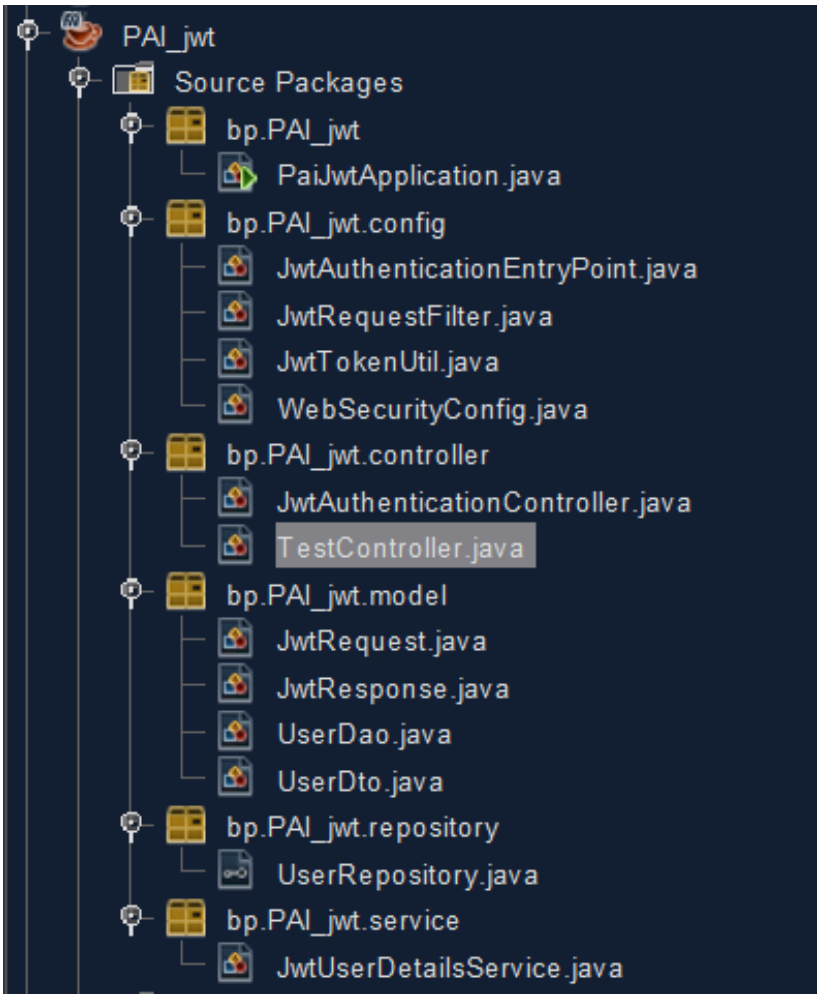
UWAGA! Jeśli korzystasz z wersji *Java* większej niż 8, budowa projektu zakończy się niepowodzeniem. W tym wypadku do pliku ***pom.xml*** dodaj zależność:

```
<dependency>
     <groupId>javax.xml.bind</groupId>
     <artifactId>jaxb-api</artifactId>
</dependency>
```

Jeśli nadal jest problem, do pliku ***application.properties*** dodaj wiersz:

```
spring.main.allow-circular-references=true
```

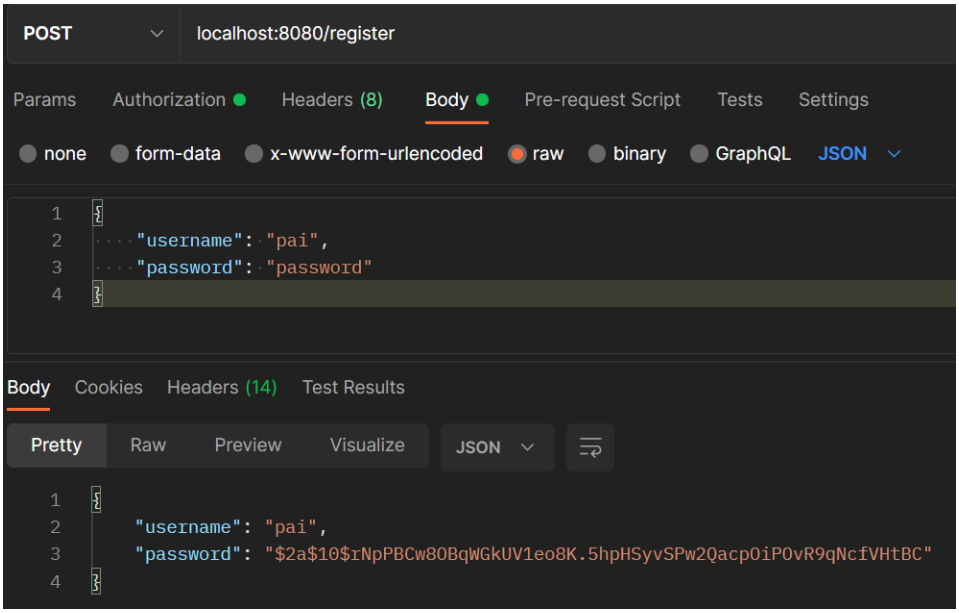
Problem może wynikać z faktu, że od wersji 2.6 w *Spring Boot* domyślnie zabronione są referencje cykliczne (ang. *Circular References Prohibited by Default in Spring Boot version 2.6*).



Rys. 10.2. Struktura projektu z *JWT* i *MySQL*

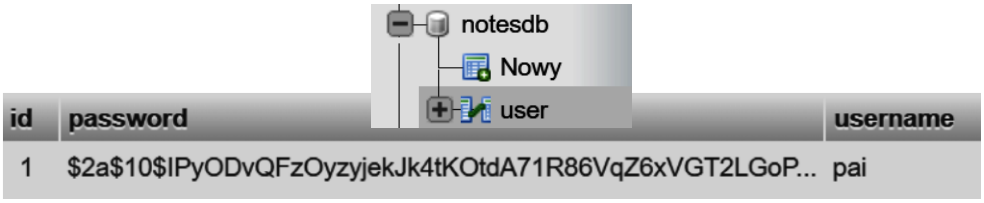
Zadanie 10.2. Test działania w narzędziu *Postman*

Przetestuj działanie aplikacji, wysyłając na początek żądanie metodą *POST* z danymi użytkownika pod adres ***/register*** (Rys. 10.3). W odpowiedzi zostanie zwrócony obiekt ***user*** w formacie *JSON* z zahaszowanym hasłem (Rys. 10.3).



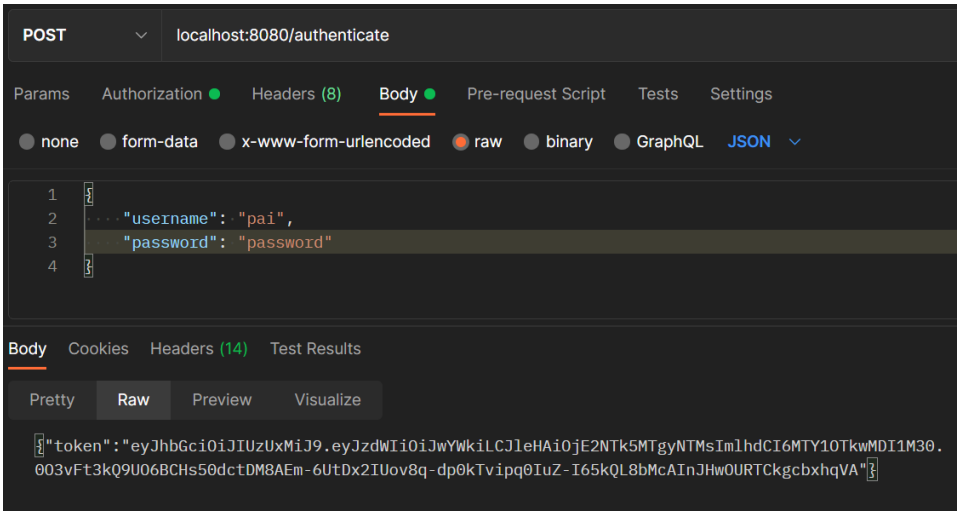
Rys. 10.3. Rejestracja danych użytkownika w bazie *MySQL*

Po prawidłowym wykonaniu akcji, powinna się utworzyć baza danych ***notesdb*** z tabelą ***user***, a w niej pierwszy rekord z danymi użytkownika (Rys. 10.4).



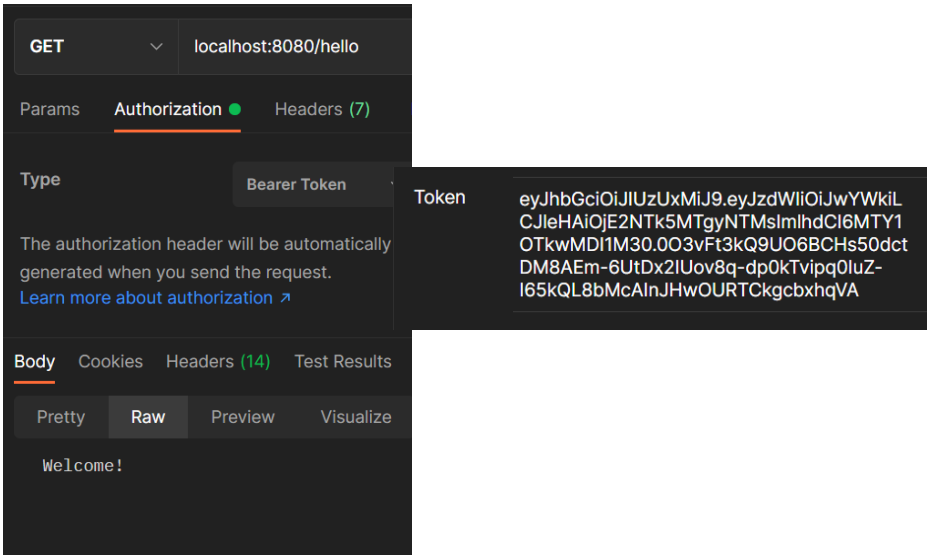
Rys. 10.4. Rekord w tabeli *user* w bazie *notesdb*

Metodą *POST* wyślij żądanie uwierzytelniające użytkownika (na podstawie danych zapisanych w bazie) pod adres ***/authenticate*** (Rys. 10.5). W odpowiedzi, po prawidłowym uwierzytelnieniu użytkownika, serwer powinien zwrócić token *JWT* (Rys. 10.5), analogicznie jak w laboratorium 9.



Rys. 10.5. Żądanie z przekazaniem danych użytkownika i odpowiedź z tokenem *JWT*

Po uzyskaniu tokenu, należy go dodać do nagłówka ***Authorization*** żądania, aby uzyskać dostęp do zawartości strony ***/hello*** (Rys. 10.6).



Rys. 10.6. Żądanie z przekazaniem tokena w nagłówku *Authorization*

Podsumowanie

W skrypcie przedstawio zestaw zadań do realizacji na laboratoriach z programowania aplikacji internetowych w języku *Java* w środowisku *Java Enterprise Edition* z wykorzystaniem szkieletu programistycznego *Spring MVC*.

Pierwsze trzy laboratoria wprowadzały studenta w podstawowe zagadnienia aplikacji webowych w języku *Java* i pozwoliły poznać kolejno:

1. Standard serwletów jako podstawę wszystkich aplikacji internetowych w języku *Java*.
2. Technologię widoków opartą na standardzie *JSP* oraz zasadę działania pomocniczych klas *JavaBean* do operacji na danych.
3. Podstawy działania wzorca projektowego *MVC* na przykładzie własnej implementacji prostej aplikacji typu *CRUD*.

Kolejne laboratoria ukierunkowano na stopniowe poznawanie przez studentów tajników budowania aplikacji internetowych z wykorzystaniem szkieletu programistycznego *Spring*.

Laboratoria 4–10 pozwoliły studentom poznać kolejno:

4. Podstawowe założenia *Spring MVC*.
5. Konfigurację *Spring Boot* i interfejsy repozytorium *JPA*.
6. Zabezpieczenia *Spring Security* i technologię widoków *Thymeleaf*.
7. Strukturę projektu *Spring Rest API*.
8. Zastosowania obiektów *DAO* i *DTO*.
9. Elementy konfiguracji *Spring Security* do uwierzytelniania użytkownika za pomocą tokena *JWT*.
10. Konfigurację *Spring Security* z tokenem *JWT* i bazą danych.

Po wykonaniu proponowanych w skrypcie zadań, Czytelnik powinien umieć wykorzystać w praktyce wiedzę związaną z wytwarzaniem aplikacji internetowych w różnych technologiach związanych z programowaniem na platformie *JEE* w *Spring* z zastosowaniem poznanych metod, technik, wzorców i narzędzi.

Bibliografia

1. Baza danych H2: *http://www.h2database.com/html/main.html* [20.03.2022].
2. Budowa serwisów REST w Spring: *https://spring.io/guides/tutorials/rest/* [20.03.2022].
3. Dokumentacja platformy Java Enterprise Edition: *https://www.oracle.com/technetwork/java/javaee/documentation/ee8-release-notes-3894362.html* [20.03.2022].
4. Dokumentacja Spring Boot: *https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle* [20.03.2022].
5. Dokumentacja Spring Security: *https://docs.spring.io/spring-security/reference/* [20.03.2022].
6. Dokumentacja Spring: *https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/* [20.03.2022].
7. Interakcja z bazą danych za pomocą Spring Data JPA: *http://blog.mloza.pl/spring-boot-interakcja-z-baza-danych-czyli-spring-data-jpa/* [20.03.2022].
8. Interfejs Fetch API: *https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch* [20.03.2022].
9. Java SE Development Kit: *https://www.oracle.com/pl/java/technologies/javase-jdk15-downloads.html* [20.03.2022].
10. Konfiguracja autentykacji użytkownika za pomocą JWT: *https://www.techgeeknext.com/spring/spring-boot-security-token-authentication-jwt* [20.03.2022].
11. Konfiguracja autentykacji użytkownika za pomocą JWT z MySQL: *https://www.techgeeknext.com/spring/spring-boot-security-token-authentication-jwt-mysql* [20.03.2022].
12. Konwersja typów SQL na Java: *http://info.ee.pw.edu.pl/Java/1.4.2/docs/guide/jdbc/getstart/mapping.html* [20.03.2022].
13. Marthy Hall, Larry Brown, Yaakov Chaikin: Core Servlets and JavaServer Pagess, tom II, wydanie II, Helion 2009.
14. Najpopularniejsze pytania z rozmów (Java, Spring): *http://codebykris.com/najpopularniejsze-pytania-z-rozmow-kwalifikacyjnych-ze-spring-framework/* [20.03.2022].

15. Pakiet XAMPP: *https://www.apachefriends.org/pl/download.html* [20.03.2022].
16. Praca z repozytoriami w Spring: *https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories* [20.03.2022].
17. Przykład Spring MVC: *https://www.javatpoint.com/spring-mvc-crud-example* [20.03.2022].
18. Spring Boot – szybkie tworzenie aplikacji web w Javie: *http://blog.mloza.pl/spring-boot-szybkie-tworzenie-aplikacji-web-w-javie/* [20.03.2022].
19. *Spring tutorial: http://www.tutorialspoint.com/spring* [20.03.2022].
20. Środowisko programistyczne Netbeans: *https://netbeans.apache.org/download/index.html* [20.03.2022].
21. Technologia serwletów: *http://www.oracle.com/technetwork/java/javaee/servlet/index.html* [20.03.2022].
22. Tutorial Spring Data JPA: *https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-introduction/* [20.03.2022].
23. Tutorial Thymeleaf + Spring: *https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html* [20.03.2022].
24. Vishal Layka, Java. Projektowanie aplikacji WWW, Helion 2015.
25. Wzorzec Inversion of Control: *http://martinfowler.com/bliki/InversionOfControl.html* [20.03.2022].

Programowanie aplikacji internetowych JEE w Spring
Przykłady i zadania

Streszczenie

Skrypt jest zbiorem przykładów i zadań proponowanych do realizacji na laboratorium programowania aplikacji internetowych dla studentów kierunku Informatyka. Dotyczy zagadnień związanych z programowaniem aplikacji internetowych w środowisku Java Enterprise Edition (JEE) z wykorzystaniem popularnego szkieletu programistycznego Spring. Zakres tematyczny związany jest z poznaniem zaawansowanych wzorców programowania obiektowego. Zakładana jest znajomość podstawowych zagadnień związanych z tworzeniem aplikacji internetowych. Zadania proponowane w skrypcie wykorzystują rozwiązania i wzorce projektowe, stosowane aktualnie w programowaniu aplikacji internetowych. Platformą programistyczną jest zaawansowane środowisko Java Enterprise Edition i Spring, jako najbardziej popularny szkielet programistyczny języka Java. Skrypt oferuje zadania które, umożliwiają indywidualną organizację pracy studenta. Przygotowane instrukcje, prowadzą czytelnika krok po kroku do rozwiązania zadanego problemu, pozwalając przy tym poznać zaawansowane techniki programowania obiektowego.

Słowa kluczowe: aplikacje internetowe, Java Enterprise Edition, Spring

JEE web application programming in Spring
Examples and tasks

Abstract

The script is a set of examples and tasks proposed for implementation in the web application programming laboratory for students of Computer Science. It deals with issues related to programming web applications in the Java Enterprise Edition (JEE) environment with the use of the popular Spring programming framework. The The scope of topics is related to the knowledge of advanced object-oriented programming patterns. Knowledge of the basic issues related to the development of web applications is assumed. The problems proposed in the script use solutions and design patterns currently used in programming web applications. The development platform is based on the advanced Java Enterprise Edition and Spring as the most popular Java programming framework. The script offers exercises that enable the individual organization of the student's work. Prepared instructions lead the reader step by step to solving the given problem, allowing to learn about advanced techniques of object-oriented programming.

Keywords: web applications, Java Enterprise Edition, Spring