

Informatyka, Aplikacje internetowe i mobilne, semestr 5

Programowanie aplikacji webowych

Laboratorium nr 5

Aplikacja WWW w technologiach Jakarta Faces i Facelets z obsługą bazy danych użytkowników w bazie PostgreSQL

Celem aplikacji webowej jest zarządzanie tabelą danych użytkowników opartej na motorze baz danych users.

Wymagania aplikacji

Dostępne w DevEnv:

- Apache Tomcat 10.x -> środowisko jest dostępne w DevEnv
- IDE ECLIPSE Java EE // dostępne
- PostgreSQL 16
- JDK 20 -> na takim java pracuje DevEnv
- JDBC PostgreSQL
- JSP // zaimplementowane w Tomcat

Do wgrania do katalogu lib:

- sterowniki PostgreSQL – plik postgresql-42.6.0.jar
- JSTL 3.0 – plik jakarta.servlet.jsp.jstl-3.0.0.jar
- Jakarta Faces 4.0 – plik jakarta.faces-4.0.4.jar
- Weld 5.1.2.Final – implementacja CDI (Contexts and Dependency Injection) czyli mechanizm zarządzania aplikacją, plik weld-servlet-shaded-5.1.2.Final.jar

Projekt Dynamic Web Project aplikacji WebAppUsers

Utwórz nowy Dynamic Web Project w wersji 5.0 o nazwie **WebAppUsers** z plikiem web.xml.

Zmień ustawienia przestrzeni nazw dołączanych w znaczniku <web-app> pliku web.xml na pakiety jakarta. Jako plik startowy ustaw tylko jeden plik index.xhtml. Dodaj również informacje na temat biblioteki z silnikiem servletów Faces oraz przekieruj/dokonaj mapowania do przetwarzania przez servlet Faces wszystkie pliki z rozszerzeniem .xhtml. Plik powinien mieć postać jak poniżej.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  xmlns="https://jakarta.ee/xml/ns/jakartaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
  https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
  version="5.0">
  <display-name>WebAppUsers</display-name>

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
```



Katedra Systemów Informatycznych
Uniwersytet Morski w Gdyni

```
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>

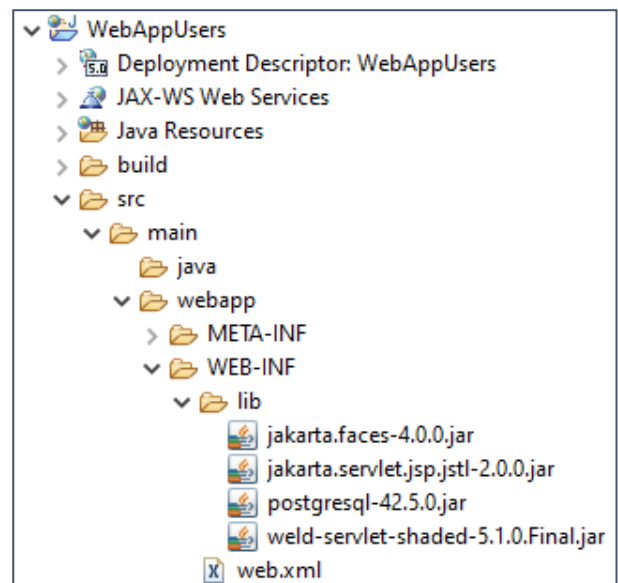
<welcome-file-list>
  <welcome-file>index.xhtml</welcome-file>
</welcome-file-list>
</web-app>
```

Dodanie zależności/niezbędnych bibliotek

Do katalogu lib (webapp\WEB-INF\lib) dodaj pliki z katalogu lib materiałów pobranych z Iliasa:

- jakarta.servlet.jsp.jstl-3.0.0.jar
- jakarta.faces-4.0.4.jar
- postgresql-42.6.0.jar
- weld-servlet-shaded-5.1.2.Final.jar

Projekt powinien mieć teraz postać jak na obrazku.



Baza danych – schemat users z tabelą users

Uruchom klienta bazy danych PostgreSQL i utwórz w swojej bazie danych nowy schemat users:

```
CREATE SCHEMA users AUTHORIZATION s<numer_indeksu>;
```

Utwórz odpowiednią sekwencję oraz tabelę users według poniższej składni SQL:

```
CREATE SEQUENCE users.users_id_seq;
CREATE TABLE IF NOT EXISTS users.users
(
  id integer NOT NULL DEFAULT nextval('users.users_id_seq'::regclass),
  first_name character varying(50) COLLATE pg_catalog."default" NOT NULL,
  last_name character varying(70) COLLATE pg_catalog."default" NOT NULL,
  year_of_study integer,
  email character varying(120) COLLATE pg_catalog."default" NOT NULL,
  personal_id bigint,
  country character varying(120) COLLATE pg_catalog."default" NOT NULL,
  CONSTRAINT users_pkey PRIMARY KEY (id),
  CONSTRAINT users_personal_id_key UNIQUE (personal_id)
)
```

Zwróć uwagę na ustawienie klucza na polu `id` oraz unikalności wartości w polu `personal_id`.

Dopisz do tabeli `users` następujące dane:

Jan Nowak, 2, Jan.nowak@domena.pl, 90041255213, Polska

Użyty numer pesel został wygenerowany za pomocą generatora online:

<https://pesel.cstudios.pl/o-generatorze/generator-on-line>

Uzupełnienie źródła projektu o odpowiednie foldery i pakiety dla aplikacji `WebAppUsers`

Pliki aplikacji podzielimy na katalogi zgodnie z architekturą MVC.

W folderze `src/main/java` projektu utwórz podane poniżej trzy pakiety/foldery. Skorzystaj z opcji `New | New Package`

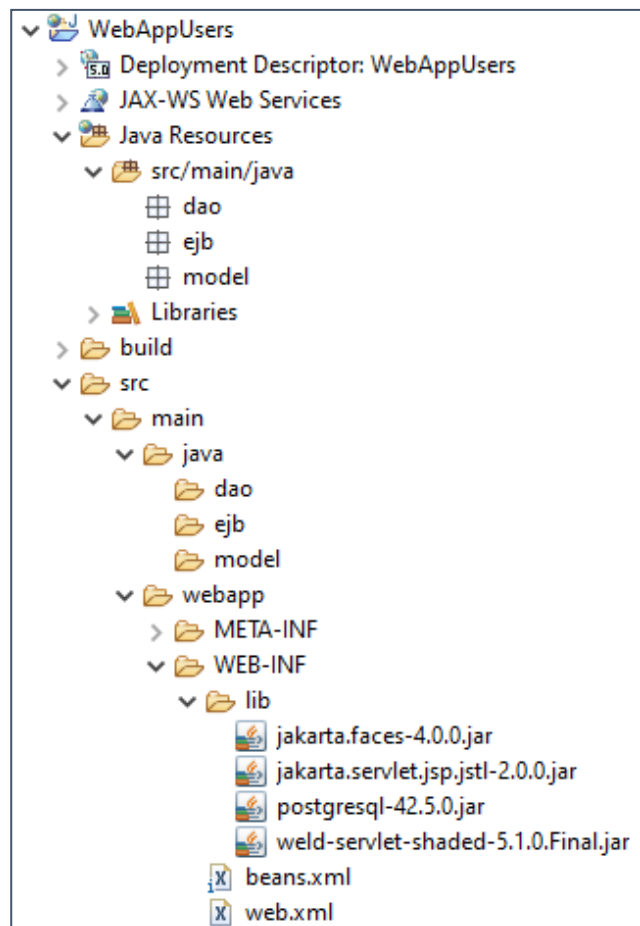
- `dao`
- `ejb`
- `model`

W folderze `webapp` dodaj dwa podfoldery:

- `includes`
- `resources`

W `includes` są najczęściej przechowywane części stron lub kodów takie jak nawigacja. W `resources` pliki zewnętrznych technologii jak style `.css`, grafiki czy skrypty.

Dodatkowo, żeby Weld prawidłowo zarządzał plikami, aplikacja powinna mieć zdefiniowany plik konfiguracyjny dla fasolek o nazwie `beans`. Może być pusty, ale musi zawierać prawidłową strukturę. W folderze `WEB-INF` utwórz plik `beans.xml` i umieść w nim następujący kod:



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee">
  <scan>
  </scan>
</beans>
```

W efekcie, struktura projektu aplikacji powinna mieć postać jak na obrazku.

Model danych dla tabeli user w klasie User

Utwórz model danych dla tabeli user w postaci klasy User wykorzystując JavaBean. W tym celu utwórz nową klasę Java o nazwie User w pakiecie model.

Dodaj odpowiednie składowe klasy oraz pusty konstruktor. W efekcie powinien powstać kod w postaci:

```
package model;

public class User
{
    private int id;
    private String first_name;
    private String last_name;
    private int year_of_study;
    private String email;
    private Long personal_id;
    private String country;

    public User()
    {
    }
}
```

Dodaj metody pobierające i ustawiające dane w klasie User. Ustaw kursor edycyjny pod ostatnią składową klasy User, kliknij prawy klawisz myszy, a następnie z menu wybierz Source | Generate Getters and Setters. Wybierz przycisk Select All. W ten sposób zostaną dodane odpowiednie metody publiczne get... i set... dla poszczególnych składowych klasy User.

Dodaj do klasy User dwa konstruktory:

1. konstruktor dla wszystkich danych
2. konstruktor dla wszystkich danych bez pola id.

Żeby zdefiniować pierwszy konstruktor ustaw kursor za istniejącym już pustym konstruktorem i zrób dwa wiersze odstepu. Następnie kliknij prawy klawisz myszy i wybierz Source | Generate constructor using Fields. Dla zaznaczonych wszystkich pól, użyj przycisku Generate.

Analogicznie dodaj drugi konstruktor, nie zaznaczaj dla niego pola/attributu id (w niektórych przypadkach nie będzie potrzebne ponieważ w bazie danych wartość tego pola nadawana jest automatycznie).

W ten sposób został zbudowany model danych w klasie User.

Komunikacja z bazą danych w klasie UserDao

Przeznaczeniem klasy UserDao będzie komunikacja z bazą danych PostgreSQL.

W pakiecie dao utwórz nową klasę Java o nazwie UserDao z metodą main (do testowania połączeń i zapytań do bazy danych). Zaimportuj model.User i dodaj metody do nawiązywania i kończenia połączenia z Twoją bazą danych PostgreSQL, schematem users.

Utworzona klasa z metodą DBSQLConnection oraz parametrami połączenia ze schematem users powinna wyglądać następująco:

```
package dao;

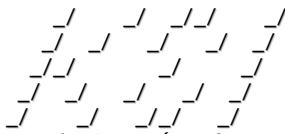
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

import model.User;

public class UserDao {
    // definicja parametrów połączenia z bazą danych PGSQL za pomocą JDBC
    private final String url =
        "jdbc:postgresql://localhost:5432/nazwabazy?currentSchema=\"users\"";
    private final String user = "";
    private final String password = "";
    // koniec - parametrów połączenia z PGSQL
    Connection connection = null;

    /**
     * Connect to the PostgreSQL database
     *
     * @return a Connection object
     */
    //rozpoczęcie połączenia
    public Connection DBSQLConnection() {
        try {
            Class.forName("org.postgresql.Driver");
            connection = DriverManager.getConnection(url, user, password);
            if(connection.isValid(5)) System.out.println("Connection is working");
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return connection;
    }

    //zakończenie połączenia
    private void DBSQLConnectionClose(){
        if(connection==null) return;
    }
}
```



```
try{
    connection.close();
    if(!connection.isValid(5)) System.out.println("Connection closed");
}
catch(SQLException e){
    e.printStackTrace();
}

// konstruktor, który nic nie wykonuje
public UserDAO() {
}

// metoda main do testowania
public static void main(String[] args)
{
    UserDAO dao = new UserDAO();
    dao.DBSQLConnection();
    dao.DBSQLConnectionClose();
}
}
```

Ustaw własne dane połączenia i sprawdź czy działa uruchamiając klasę UserDAO jako Java Application. Sprawdź, czy w konsoli Eclipse pojawił się komunikat o prawidłowym nawiązaniu i zamknięciu połączenia.

Instrukcje SQL. Przygotowanie instrukcji SQL do realizacji operacji CRUD

Do klasy UserDAO zaraz za wierszem

```
// koniec - parametrów połączenia z PGSQL
```

dodaj stałe definiujące schematy zapytań SQL do realizacji 5-ciu podstawowych operacji CRUD w bazie danych dla tabeli users. Kod może wyglądać następująco:

```
//instrukcje SQL do realizacji CRUD (Create, Read, Update, Insert, Delete)
// read / select data - all users
private static final String SELECT ALL USERS = "select * from users";
// read / select data user by id
private static final String SELECT USER BY ID = "select id, first_name
,last_name, year_of_study, email, personal_id, country from users where id =?";
// insert data
private static final String INSERT USERS SQL = "INSERT INTO users" + "
(first_name, last_name, year_of_study, email, personal_id, country) VALUES (?,
?, ?, ?, ?)";
// update user by id
// delete user by id
```

W dalszej części aplikacji analogicznie utwórz i zapisz w zmiennych instrukcje do uaktualniania i usuwania danych użytkownika o podanym id.

Opracowanie metod obsługujących instrukcje SQL

Obsługa błędów

Wart zadbać o sprawne testowanie instrukcji SQL. Jednym ze sposobów jest konstrukcja jednej spójnej metody do generowania jasnych komunikatów z kodami błędów SQL w konsoli Java. W klasie UserDAO dodaj metodę printSQLException, która może mieć postać podaną poniżej.

```
private void printSQLException(SQLException ex) {
    for (Throwable e: ex) {
        if (e instanceof SQLException) {
            e.printStackTrace(System.err);
            System.err.println("SQLState: " + ((SQLException) e).getSQLState());
            System.err.println("Error Code: " + ((SQLException) e).getErrorCode());
            System.err.println("Message: " + e.getMessage());
            Throwable t = ex.getCause();
            while (t != null) {
                System.out.println("Cause: " + t);
                t = t.getCause();
            }
        }
    }
}
```

Metoda printSQLException odbiera obiekt SQLException, który jak wszystkie obiekty wyjątków zawiera wskazanie na listę wyjątków, które go spowodowały - łańcuch przyczyn wyjątków. Oprócz tego standardowego łańcucha, w klasie SQLException istnieje inny łańcuch – łańcuch obiektów SQLException, które można pobrać za pomocą metody getNextException. Korzystając z iteratora zdefiniowanego dla tego łańcucha można przejrzeć listę w poszukiwaniu wyjątków klasy SQLException. Zdefiniowana metoda wyświetla dla każdego z nich w konsoli błędów System.err: stos wywołań – printStackTrace, stan wyjątku oraz inne informacje.

Odczytanie danych wybranego użytkownika

W klasie UserDAO dodaj metodę selectUser, która odczyta dane użytkownika o podanym id. Metoda odda w wyniku obiekt klasy User. Metoda może mieć postać:

```
/*pobranie z DB danych użytkownika o podanym id i zapisanie w obiekcie*/
public User selectUser(int id) {
    User user = null;
    //połączenie
    try(
        Connection connection = DBSQLConnection();
        //utworzenie obiektu reprezentującego prekompilowane zapytanie SQL
```

```
PreparedStatement preparedStatement =
    connection.prepareStatement(SELECT_USER_BY_ID);
{
    //ustawienie parametru
    preparedStatement.setInt(1, id);
    //wyświetlenie zapytania w konsoli
    System.out.println(preparedStatement);
    // Wykonanie zapytania
    ResultSet rs = preparedStatement.executeQuery();
    // Proces obsługi rezultatu.
    while (rs.next()) {
        String t_first_name = rs.getString("first_name");
        String t_last_name = rs.getString("last_name");
        int t_year_of_study = rs.getInt("year_of_study");
        String t_email = rs.getString("email");
        Long t_personal_id = rs.getLong("personal_id");
        String t_country = rs.getString("country");
        user = new User(id, t_first_name, t_last_name, t_year_of_study,
            t_email, t_personal_id, t_country);
    }
}
catch (SQLException e) { printSQLException(e); }
return user;
}
```

Zauważ, że w instrukcji try oprócz nawiązania połączenia tworzony jest obiekt klasy PreparedStatement, który reprezentuje wstępnie skompilowane wyrażenie SQL. Jest on tworzony na podstawie szablonu wyrażenia zdefiniowanego jako stała SELECT_USER_BY_ID. Umożliwia również podstawienie, w miejsce kolejnych znaków ?, konkretnych wartości. W poniższym przykładzie instrukcja preparedStatement.setInt(1, id) zamienia pierwszy znak ? stałej SELECT_USER_BY_ID na wartość zmiennej id. Wartość ta została odebrana jako parametr. Zapytanie SQL skonstruowane po podstawieniu konkretnych wartości wyświetlane jest w konsoli.

Zwróć uwagę na konstrukcję try. Bezpośrednio za słowem try, w nawiasie okrągłym, zdefiniowane zostały zasoby, które będą używane w tym bloku. Taka konstrukcja umożliwia automatyczne zamknięcie zasobu jeśli tylko implementuje on interfejs AutoClosable. Tak właśnie jest w tym przypadku. Dzięki temu nie trzeba deklarować bloku finalny, ani wywoływać metody DBSQLConnectionClose().

Przetestuj działanie metody selectUser w metodzie main. Przykładowy test dla użytkownika o identyfikatorze 1 może mieć postać:

```
User u1= dao.selectUser(1);
System.out.println(u1.getFirst_name());
System.out.println(u1.getLast_name());
System.out.println(u1.getPersonal_id());
```

Na podobnej zasadzie działają pozostałe metody CRUD, przykłady kodów takich metod do pobierania listy użytkowników oraz wstawiania nowego użytkownika podano poniżej.

Wstawianie nowych danych do tabeli users

Metoda `insertUser` jest odpowiedzialna za dopisanie nowego rekordu do tabeli `users` w bazie danych. Ta metoda nie zwraca żadnego wyniku.

```
/* dodanie rekordu z danymi nowego użytkownika */
public void insertUser(User user) throws SQLException
{
    System.out.println(INSERT_USERS_SQL);
    try (Connection connection = DBSQLConnection();
        PreparedStatement preparedStatement =
            connection.prepareStatement(INSERT_USERS_SQL))
    {
        preparedStatement.setString(1, user.getFirst_name());
        preparedStatement.setString(2, user.getLast_name());
        preparedStatement.setInt(3, user.getYear_of_study());
        preparedStatement.setString(4, user.getEmail());
        preparedStatement.setLong(5, (Long)user.getPersonal_id());
        preparedStatement.setString(6, user.getCountry());
        System.out.println(preparedStatement);
        preparedStatement.executeUpdate();
    }
    catch (SQLException e) { printSQLException(e); }
}
```

Warto zwrócić uwagę, że do wykonywania zapytań, które nie zwracają wartości, czyli takich jak `INSERT`, `UPDATE` czy `DELETE` przeznaczona jest metoda `executeUpdate`.

Skonstruuj test sprawdzający działanie metody `insertUser`. Pamiętaj o konieczności obsłużenia wyjątku. Do wygenerowania prawidłowego numeru pesel wykorzystaj URL podany na początku przewodnika. Zwróć uwagę, że liczbę należy wpisać jako `long`.

Odczytanie listy użytkowników

Metoda `selectAllUsers` odczytuje z bazy danych dane wszystkich użytkowników i zapisuje w postaci listy `ArrayList`.

```
/* lista użytkowników */
public ArrayList<User> selectAllUsers() {
    ArrayList<User> users = new ArrayList<>();
    //połączenia
    try (Connection connection = DBSQLConnection();
        PreparedStatement preparedStatement =
            connection.prepareStatement(SELECT_ALL_USERS);)
    {
        System.out.println(preparedStatement);
        ResultSet rs = preparedStatement.executeQuery();
        // odebranie wyników z obiektu ResultSet.
        while (rs.next()) {
            int t_id = rs.getInt("id");
            String t_first_name = rs.getString("first_name");
        }
    }
}
```

```
String t_last_name = rs.getString("last_name");
int t_year_of_study = rs.getInt("year_of_study");
String t_email = rs.getString("email");
Long t_personal_id= rs.getLong("personal_id");
String t_country = rs.getString("country");
users.add(new User(t_id, t_first_name, t_last_name, t_year_of_study,
    t_email, t_personal_id, t_country));
}
}
catch (SQLException e) { printSQLException(e);}
return users;
}
```

Skonstruuj test sprawdzający działanie metody selectAllUsers w metodzie main.

Aktualizacja i usuwanie danych w tabeli users

Metody do aktualizacji i usuwania danych użytkowników skonstruuj samodzielnie w dalszej części aplikacji.

Zarządzanie aplikacją za pomocą fasolki CDI (CDI bean)

Utworzenie fasolki UserManager

W pakiecie ejb utwórz klasę Java o nazwie UserManager implementującą interfejs Serializable.

Zgodnie z podpowiedzią dodaj default serial version ID.

Przed definicją klasy dodaj adnotację @Model. Adnotacja @Model może zostać użyta zamiast dwóch innych: @Named i @RequestScoped. Adnotacje, które są zamiennikami dla grupy innych nazywane są stereotypami, można je również tworzyć samodzielnie.

Zarządzając aplikacją będziemy korzystać z obiektu reprezentującego wybranego użytkownika oraz listy użytkowników. Dodaj do klasy UserManager pole user i users oraz egzemplarz klasy userDAO, za pośrednictwem którego odbywać się będzie komunikacja z bazą danych.

```
private User user;
private ArrayList<User> users;
private UserDAO userDAO=new UserDAO();
```

Dodaj również metody get... i set... dla pól user i users.

Ponieważ obiekty klasy UserManager będą inicjalizowane z przeglądarki trzeba dodać metodę init(). Żeby metoda init została wykonana przez kontener zanim klasa zacznie odpowiadać na zapytania klienta web musi mieć adnotację @PostConstruct. Zadaniem init jest wygenerowanie obiektu klasy User oraz pobranie z bazy listy użytkowników, która będzie wyświetlana na stronie. Na początku klasy, za polem userDAO dodaj następujący kod:

```
@PostConstruct
private void init()
{
    user=new User();
    setUsers(userDAO.selectAllUsers());
}
```

Analogicznie można używać adnotacji `@PreDestroy` dla metody, która powinna zostać wykonana przez kontener przed usunięciem obiektu.

Strony WWW Jakarta Faces

Układ/szablon strony z dołączonym stylem .css

Ponieważ aplikacje powinny być spójne pod względem wyglądu wygodnie jest zdefiniować wspólny szablon. Różne jego części można definiować w osobnych plikach. W tym przykładzie zdefiniujemy osobny plik do przechowywania menu.

W katalogu webapp utwórz plik o nazwie `template.xhtml`. Skorzystaj z generatora plików HTML, podaj nazwę z rozszerzeniem `.xhtml` i wybierz szablon New Faces Template. Trzeba będzie dokonać kilku zmian. Przede wszystkim zmień deklarację typu dokumentu z XHTML na HTML 5 oraz dodaj do przestrzeni nazw bibliotekę `jsf/html`. Kod na początku pliku powinien mieć postać:

```
<!DOCTYPE html>
<html xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
```

Znaczniki `<head>` zamień na `<h:head>`, a wewnątrz dodaj meta dane na temat kodowania oraz podłączenie pliku ze stylem. Zmień również nazwę wstawianego elementu `ui` z `title` na `htmlTitle`. Głowa dokumentu HTML powinna teraz wyglądać następująco:

```
<h:head>
    <meta charset="UTF-8"/>
    <h:outputStylesheet name="default.css" />
    <title><ui:insert name="htmlTitle">Default title</ui:insert></title>
</h:head>
```

Plik `default.css` został dołączony do materiałów, natomiast w aplikacji powinien się znaleźć w katalogu `resources`, który utworzyliśmy wcześniej. W tym miejscu będzie go szukał servlet Faces. Skopiuj plik ze stylem `default.css` do katalogu `webapp/resources`.

W dalszej części dokumentu `template.xhtml` zamień znacznik `<body>` na `<h:body>`, natomiast znacznik `<div id="header">` na `<nav>`. Zmień również nazwę (name) wstawianego elementu `ui:insert` z `header` na `nav`, usuń komentarz dla znacznika `<ui:include ..>` i podaj w `src` jako wstawiany plik `"/includes/mainNav.xhtml"`. Znaczniki z prefiksem `ui:` są znacznikami biblioteki Facelets, która odpowiada za układy stron. Część nawigacyjna powinna mieć teraz postać:

```
<nav>
  <ui:insert name="nav">
    <p>Header area. See comments below this line in the source.</p>
    <!-- include your header file or uncomment the include below and create
         header.xhtml in this directory -->
    <ui:include src="/includes/mainNav.xhtml"/>
  </ui:insert>
</nav>
```

Za częścią nav dodaj jeszcze nagłówek dla strony w postaci:

```
<h2><ui:insert name="pageHeader"/></h2>
```

Części content i footer pozostaw bez zmian.

Strona główna w pliku index.xhtml

Strona główna, tak jak zdefiniowaliśmy w pliku web.xml będzie w pliku index.xhtml. Plik ten jednak nie będzie zawierał bezpośrednio znaczników HTML tylko informacje o kompozycji strony w notacji używanej przez Facelets. Kompozycję definiuje znacznik `<ui:composition>`.

Wygeneruj plik index.xhtml korzystając z generatora plików HTML, wybierz szablon New Facelt Composition Page. Podobnie jak w poprzednio zamień definicje typu dokumentu na HTML 5. Żeby nie zmieniać tych elementów przy tworzeniu kolejnych plików, można zmienić szablony Eclipse, opcja Window | Preferences | Web | HTML Files | Editor | Templates.

```
<!DOCTYPE html>
<html xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
```

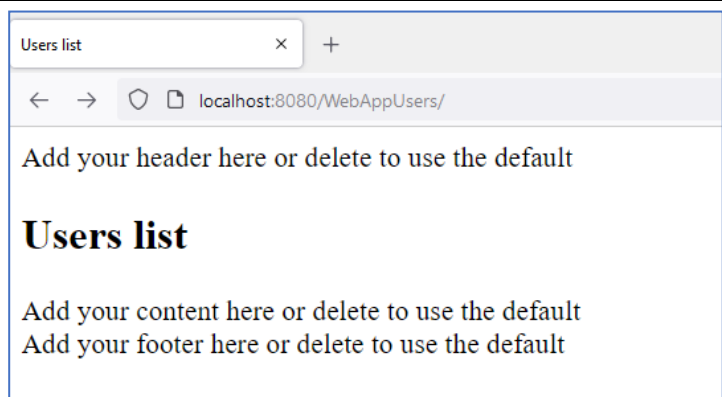
Ustaw parametr template w znaczniku `ui:composition` na "template.xhtml".

W części `<ui:composition>` dodaj na początku dwie definicje:

```
<ui:define name="htmlTitle">
  Users list
</ui:define>
<ui:define name="pageHeader">
  Users list
</ui:define>
```

Aplikację w tej postaci można już uruchomić na serwerze.

Strona główna będzie miała postać jak na obrazku. Jak widać ustawiony jest title widoczny na zakładce oraz nagłówek h2 w części body. Oba z napisem Users list.



Na stronie, w części z zawartością, powinna się jeszcze pojawić tabela z danymi użytkowników. Zamień napis w części z zawartością czyli znaczniku `<ui:define name="content">` na kod podany poniżej. Jest to konstrukcja tabeli z pięcioma kolumnami odpowiadającymi informacjom przechowywanym w tabeli `users` w bazie danych. Nazwisko i imię zostało umieszczone w jednej kolumnie.

```
<h:dataTable var="user"
    summary="List of all users"
    value="#{userManager.users}"
    rules="all"
    cellpadding="5">
  <h:column>
    <f:facet name="header">
      <h:outputText value="User" />
    </f:facet>
    <h:outputText value="#{user.first_name} #{user.last_name}" />
  </h:column>
  <h:column>
    <f:facet name="header">
      <h:outputText value="Year of study" />
    </f:facet>
    <h:outputText value="#{user.year_of_study}" />
  </h:column>
  <h:column>
    <f:facet name="header">
      <h:outputText value="Email" />
    </f:facet>
    <h:outputText value="#{user.email}" />
  </h:column>
  <h:column>
    <f:facet name="header">
      <h:outputText value="Personal ID" />
    </f:facet>
    <h:outputText value="#{user.personal_id}" />
  </h:column>
  <h:column>
    <f:facet name="header">
      <h:outputText value="Country" />
    </f:facet>
    <h:outputText value="#{user.country}" />
  </h:column>
</h:dataTable>
```

Jak widać tabela została powiązana bezpośrednio z listą `users` w fasolce `userManager`. Listę wskazuje parametr `value` znacznika `<h:dataTable>`. Parametr `var` tego znacznika definiuje zmienną, za pomocą której można się odwoływać do elementów listy. Ponieważ są to obiekty klasy `User` można korzystać z wszystkich pól dla których zdefiniowano metody `get...` i `set...`

Uruchom i sprawdź działanie aplikacji.

Nawigacja w postaci menu w pliku mainNav.xhtml

W katalogu webapp/includes Utwórz nowy plik mainNav.xhtml z szablonu HTML 5. Usuń zawartość head, która nie jest tu potrzebna. Znaczniki head i body zamień na odpowiedniki Faces (JSF), a do znacznika html dołącz przestrzenie nazw h i ui.

W części h:body dodaj znacznik ui:composition z menu, który może mieć postać:

```
<ui:composition>
  <p>
    <h:link outcome="index" value="Users list" />
    <h:link outcome="add" value="Add new user" />
    <h:link outcome="edit" value="Edit users" />
  </p>
</ui:composition>
```

W parametrze outcome umieszczane są nazwy plików .xhtml lub .html zawierających wskazywane strony. Parametr value definiuje nazwę linku.

Żeby menu było widoczne zamiast napisu w nagłówku trzeba jeszcze usunąć definicję dla nazwy nav z pliku index.html. Zamiast tej informacji pojawić powinno się nav z szablonu template.xhtml.

Uruchom ponownie aplikację i sprawdź jak wygląda strona główna. Zauważ, że linki, dla których strony nie zostały utworzone nie zostały zapisane jako znaczniki zamiast <a>.

Strona index.xhtml z nawigacją powinna mieć teraz postać, jak na obrazku poniżej.

Header area. See comments below this line in the source.

Users list Add new user Edit users

Users list

User	Year of study	Email	Personal ID	Country
Jan Nowak	3	jan.nowak@domena.pl	90041255213	Polska
Dorota Malicka	2	dora@poczta.pl	95113051263	Polska
Piotr Piotrowski	1	pp@onet.com	3221039914	Norwegia
Paweł Pawłowski	1	pp@wp.pl	97031229469	Szwecja
Maria Olszewska	2	maria@wp.pl	97062592831	Polska

Add your footer here or delete to use the default

Strona do dodawania nowych użytkowników w pliku add.xhtml

Dodaj do projektu stronę add.xhtml. Analogicznie jak w przypadku index.xhtml będzie to strona typu New Facelt Composition Page. Zamień definicję typu dokumentu na HTML 5 i podaj nazwę pliku z szablonem dla kompozycji.

Dodaj definicję części `htmlTitle` z napisem Add new user. Zamień definicję części `header` na `pageHeader` i umieść tam taki sam napis: Add new user.

Zamień napis w części `content` na formularz podany poniżej.

```
<h:form id="addUserForm">
  <h:panelGrid id="userGrid" columns="3">
    <h:outputLabel for="userFirstNameInput" value="First name: " />
    <h:inputText id="userFirstNameInput"
      value="#{userManager.user.first_name}" />
    <h:message for="userFirstNameInput" />

    <h:outputLabel for="userLastNameInput" value="Last name: " />
    <h:inputText id="userLastNameInput"
      value="#{userManager.user.last_name}" />
    <h:message for="userLastNameInput" />

    <h:outputLabel for="userYearOfStudyInput"
      value="Year of study: " />
    <h:inputText id="userYearOfStudyInput"
      value="#{userManager.user.year_of_study}" />
    <h:message for="userYearOfStudyInput" />

    <h:outputLabel for="userEmailInput"
      value="Email: " />
    <h:inputText id="userEmailInput" value="#{userManager.user.email}"/>
    <h:message for="userEmailInput" />

    <h:outputLabel for="userPersonal_idInput" value="Personal ID: " />
    <h:inputText id="userPersonalIdInput"
value="#{userManager.user.personal_id}"/>
    <h:message styleClass="error-message" for="userPersonalIdInput" />

    <h:outputLabel for="userCountryInput"
      value="Country: " />
    <h:inputText id="userCountryInput"
      value="#{userManager.user.country}" />
    <h:message for="userCountryInput" />

    <f:facet name="footer">
      <tr>
        <td colspan="2">
          <h:commandButton id="createUserButton"
action="#{userManager.insertUser(userManager.user)}"
          value="Create User" />
          <h:commandButton id="cancelButton"
            immediate="true"
            action="index"
            value="Cancel" />
        </td>
        <td>

```

```

        <h:message for="createUserButton" />
    </td>
</tr>
</f:facet>
</h:panelGrid>
</h:form>

```

Sprawdź jak wygląda strona.

Do ułożenia pól w formularzu wykorzystano znacznik `<h:panelGrid>`. Który również konstruuje tabelę HTML `<table>`. Tabela ma 3 kolumny, w pierwszej znajdują się opisy pól, w drugiej pola, natomiast trzecia, obecnie pusta, została przeznaczona na komunikaty o błędach. Wykorzystano znaczniki JSF `<h:outputLabel>`, `<h:inputText>` i `<h:message>`. Zawartość pól `inputText` została powiązana z obiektem `user` w fasolce `userManager` przez parametr `value`.

Żeby formularz zaczął działać wystarczy zdefiniować metodę obsługi przycisku dodawania `Create User`. Metoda ta została tutaj nazwana `insertUser`, ma jeden parametr w postaci obiektu `user`.

W klasie `userManager` zdefiniuj metodę `insertUser(User u)`. Metoda ta musi wywołać metodę dodającą do bazy dane nowego użytkownika. W naszym przypadku będzie to metoda o tej samej nazwie z klasy `UserDAO`. Metoda może również zwracać nazwę pliku `.xhtml`, do którego ma nastąpić przekierowanie po jej wykonaniu. Najprostsza wersja takiej metody może mieć postać:

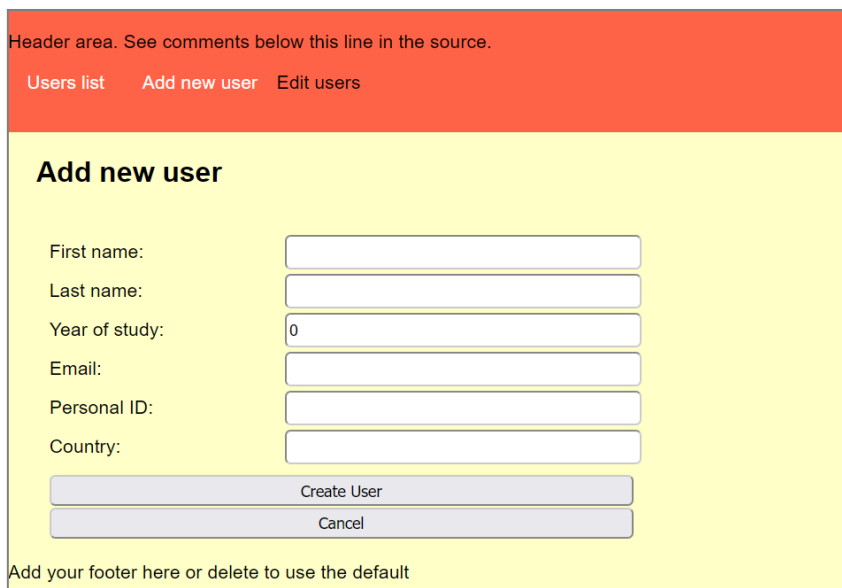
```

public String insertUser(User u) throws SQLException {
    userDao.insertUser(u);
    return ("index");
}

```

Sprawdź działanie aplikacji podając dane nowego użytkownika. Ponieważ aplikacja nie sprawdza jeszcze podawanych danych, żeby dopisać nowego użytkownika do bazy danych trzeba pamiętać tylko o podaniu unikatowego numeru `Pesel` (`pesonal_id`). Pozostałe pola mogą być nawet puste.

Strona powinna mieć postać jak na obrazku.



Header area. See comments below this line in the source.

Users list Add new user Edit users

Add new user

First name:

Last name:

Year of study:

Email:

Personal ID:

Country:

Add your footer here or delete to use the default

Sprawdzanie danych w formularzu na stronie add.xhtml – mechanizm Faces

Podstawowy mechanizm kontroli błędów na stronach WWW został wbudowany w znaczniki Faces bibliotek html i core.

Do znaczników `inputText` z imieniem i nazwiskiem dodaj atrybut `required="true"` i sprawdź działanie formularza. Próba wysłania formularza zakończy się wyświetleniem komunikatu np. `addUserForm:userFirstNameInput: Validation Error: Value is required.`

Komunikat można nieco dostosować do własnych potrzeb definiując dodatkowo atrybut `label` z opisem pola, np. `label="First name"`. Po dodaniu atrybutu w komunikacie zostanie wykorzystana etykieta pola, będzie miał więc postać: `First name: Validation Error: Value is required.`

Dodatkowe możliwości sprawdzania poprawności wpisywanych danych dają specjalne znaczniki z biblioteki core. Zdefiniuj zakres liczb podawanych jako rok studiów od 1 do 5. W tym celu użyj znacznika `<f:validateRange>` z parametrami `minimum` i `maximum`. Ponieważ musi się on znaleźć w znaczniku `inputText`, znacznik ten trzeba zmienić na postać dwuczęściową. Po zmianach i dodaniu etykiety kod powinien wyglądać następująco:

```
<h:inputText id="userYearOfStudyInput"
             value="#{userManager.user.year_of_study}"
             label="Year of study">
  <f:validateLongRange minimum="1" maximum="5"/>
</h:inputText>
```

Sprawdź działanie formularza.

Dodaj kolejne reguły i etykiety.

Dla pola Email dodaj wymóg wprowadzenia co najmniej 6 znaków oraz wyrażenie regularne:

```
<f:validateLength minimum="6" />
<f:validateRegex pattern="^[^-\\w\\.]+@([^-\\w]+\\.)+[a-z]+$"/>
```

Dla pola Personal ID dodaj wymóg wprowadzenia wartości oraz etykiety.

Dla pól można również definiować tzw. konwertery dostosowujące postać wprowadzanych danych do potrzeb aplikacji. Dla pola Personal Id zdefiniuj konwerter przekształcający wprowadzoną liczbę na postać złożoną 11 cyfr. Można to zrobić dodając podany poniżej kod, również do pola `inputText`.

```
<f:convertNumber converterId="jakarta.faces.Long" pattern="00000000000"/>
```

Więcej informacji na temat zastosowania walidatorów, konwerterów i słuchaczy zdarzeń, a także zaawansowanej formy walidacji za pomocą fasolek można znaleźć na przykład na stronach:

<https://eclipse-ee4j.github.io/jakartaee-tutorial/#using-converters-listeners-and-validators>
<https://eclipse-ee4j.github.io/jakartaee-tutorial/#introduction-to-jakarta-bean-validation>

Modyfikacja danych zapisanych użytkowników

Aby zmodyfikować dane użytkownika trzeba znaleźć jego dane w bazie danych i skonstruować formularz z tymi danymi. Utworzymy w tym celu dwie dodatkowe strony: `edit.xhtml` oraz `update.xhtml`. Na stronie Edit wyświetlone zostaną na przykład imiona i nazwiska użytkowników oraz odnośniki do strony z formularzem umożliwiającym modyfikację. Do strony `update.xhtml` trzeba będzie przekazać `id` wybranego użytkownika.

Utwórz nową stronę `edit.xhtml` typu New Facelt Composition Page. Uaktualnij typ dokumentu i nazwę szablonu. W części content umieść formularz z listą linków w postaci nazwisk i imion użytkowników. Wybranie linku powinno wywołać metodę `selectUser` z parametrem w postaci `id` wybranego użytkownika. Tym razem do utworzenia listy wykorzystaj konstrukcję `<ui:repeat>` powielając akapity z danymi. Jako źródło danych wykorzystaj listę `users` z fasolki `userManager`. Formularz może mieć postać jak poniżej.

```
<h:form id="editUserForm">
  <ui:repeat value="#{userManager.users}" var="user">
    <p>
      <h:commandLink id="selectUser"
        action="#{userManager.selectUser(user.id)}"
        value="#{user.last_name} #{user.first_name}"
        immediate="true">
      </h:commandLink>
    </p>
  </ui:repeat>
</h:form>
```

W fasolce `userManager` umieść metodę `selectUser(int userId)` pobierającą dane wybranego użytkownika do zmiennej `user` oraz przekierowującą na stronę `update`.

Analogicznie utwórz stronę `update.xhtml` z formularzem jak na stronie `add.xhtml` z dodatkowym polem na `id` oraz przyciskiem wywołującym metodę `updateUser(User u)` z fasolki `userManager`.

Analogicznie utwórz stronę/strony do usuwania wybranego użytkownika.

Zdania dodatkowe

Dopisz odpowiednie elementy walidacji: wszystkie pola są wymagane, imię, nazwisko i kraj mają wymaganą minimalną liczbę znaków, lepsze wyrażenie regularne na adres email.

Dodaj mechanizm pełnej weryfikacji numeru Pesel.

Zmodyfikuj aplikację tak, aby kraje można byłoby wybierać z listy za pomocą pola kombi zasilanego z oddzielnej nowej tabeli `countries` (dołożonej do bazy w PostgreSQL).

Dodaj mechanizm generujący własne komunikaty o błędach.

Inne według własnych pomysłów z zastosowaniem Faces i Facelets.

Materiały pomocnicze:

Jakarta EE 9, Faces, Facelts

<https://eclipse-ee4j.github.io/jakartaee-tutorial/>

Sprawdzenie numeru pesel – algorytm poprawności:

<http://www.algorytm.org/numery-identyfikacyjne/pesel.html>

Sprawdzanie poprawności adresu e-mail:

https://4programmers.net/Forum/Java/152012-Sprawdzanie_poprawno%C5%9Bci_adresu_email

<https://www.nafrontendzie.pl/walidacja-e-mail-za-pomoca-regexp>

Sprawozdanie

Źródła aplikacji (spakowany katalog src) wyślij w sprawozdaniu w systemie Sprawer.

Plik .war umieść na swoim serwerze Tomcat na foce.