ĆWICZENIE 1.

APLIKACJA WEBOWA DO ZARZĄDZANIA PROJEKTAMI

Aplikacja webowa do zarządzania projektami będzie ewoluować podczas realizacji kolejnych ćwiczeń laboratoryjnych. W zdecydowanej większości zadań z nią związanych będziemy korzystać z frameworku Spring Boot i narzędzia Gradle, m.in. rozdzielimy aplikację na back-end i front-end, stworzymy REST API, użyjemy Thymeleafa i RestTemplate'a lub opcjonalnie innego rozwiązania do stworzenia warstwy prezentacji, a także będziemy testować oprogramowanie za pomocą MockMVC i zabezpieczymy aplikację używając modułu Spring Security itd.

Jednak na początek utworzymy podstawową, prostą aplikację webową uruchamianą na serwerze Tomcat. Struktura oprogramowania powinna być zgodna z wzorcem projektowym MVC (*Model-View-Controller*). Warstwą logiki biznesowej będą serwisy domenowe korzystające z modelu opartego na mapowaniu obiektowo relacyjnym - JPA/Hibernate, serwlety będą pełniły funkcję kontrolerów, a do prezentacji danych użyte zostaną strony JSP. Aplikacja powinna umożliwiać wyświetlanie listy wszystkich projektów przechowywanych w bazie danych, a także ich tworzenie, modyfikowanie i usuwanie. Trzeba również zapewnić możliwość przeglądania zadań wybranego projektu, a także przypisywania do niego nowych pozycji. Stronicowanie listy projektów oraz mechanizm wyszukiwania są zadaniami dodatkowymi, tylko dla chętnych. Przed implementacją aplikacji webowej każda grupa powinna przygotować listę zadań cząstkowych, oszacować czas ich realizacji i rozdzielić je między członków zespołu. Ponadto wygenerowany projekt wzorcowy musi być umieszczony w zdalnym repozytorium (można skorzystać z platformy *Bitbucket* obsługującej system kontroli wersji *Git*). Każdy użytkownik przed rozpoczęciem realizacji danego zadania powinien pobrać/uaktualnić swoją lokalną wersję projektu, a po zaimplementowaniu nowej funkcjonalności przesłać zmiany na serwer.

Lista projektów

Dodaj projekt

Szukaj

Rozmiar strony: 5

Poprzednia strona Następna strona

Lp.	Id	Nazwa	Opis	Utworzony	Data obrony	Edycja
6.	42	Projekt testowy 9	Opis projektu testowego 9	2020-04-25 12:37:24	2020-06-22	Zadania Edycja Usuń
7.	38	Projekt testowy 5	Opis projektu testowego 5	2020-04-25 12:34:12	2020-06-15	Zadania Edycja Usuń
8.	37	Projekt testowy 4	Opis projektu testowego 4	2019-05-30 05:44:49	2020-06-15	Zadania Edycja Usuń
9.	20	Projekt testowy 1	Opis projektu testowego 1	2018-06-05 12:13:42	2020-06-15	Zadania Edycja Usuń
10.	36	Projekt testowy 2	Opis projektu testowego 2	2019-05-30 05:26:00	2020-06-15	Zadania Edycja Usuń

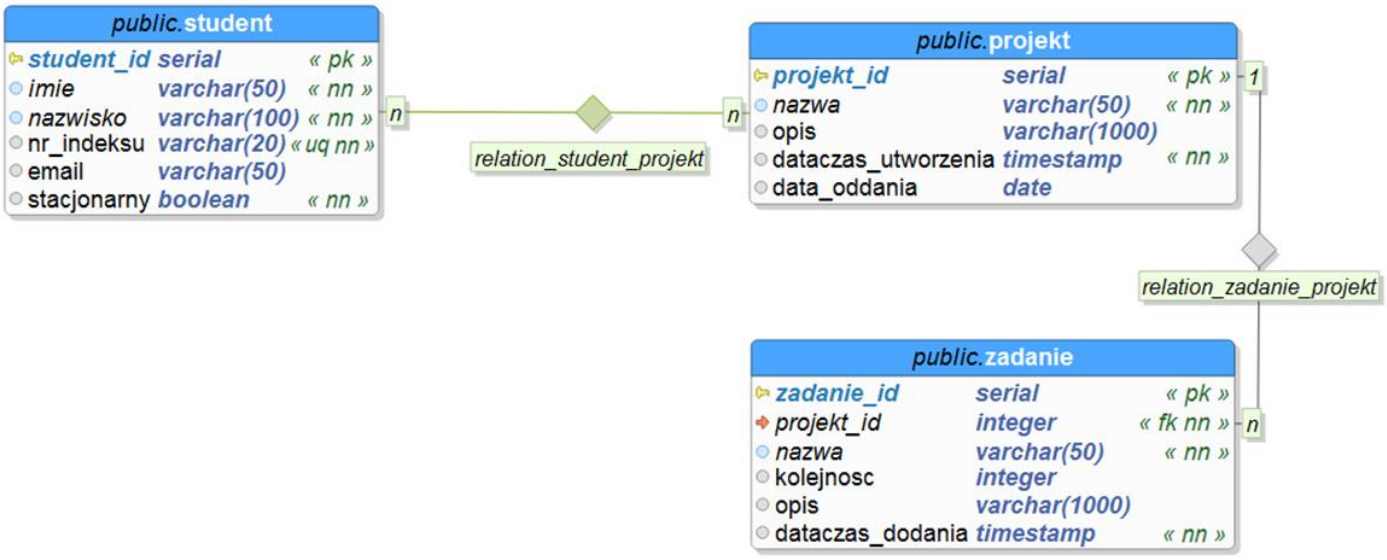
Edycja projektu

Zapisz Anuluj

Id:	42
Nazwa:	Projekt testowy 9
Opis:	Opis projektu testowego 9
Data oddania:	2020-06-22 *(RRRR-MM-DD)*

1. Utworzenie modelu. Do projektu należy dodać pakiet *com.project.model*, w którym trzeba zdefiniować klasę odwzorowującą bazodanową tabelę *projekt*.

Rys. 1. Model bazy danych *projekty*




```
package com.project.model;

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

@Entity
@Table(name="projekt") //potrzebne tylko jeżeli nazwa tabeli w bazie danych ma być inna od nazwy klasy
public class Projekt {

	@Id
	@GeneratedValue
	@Column(name="projekt_id") //tylko jeżeli nazwa kolumny w bazie danych ma być inna od nazwy zmiennej
	private Integer projektId;

	@Column(nullable = false, length = 50)
	private String nazwa;

	/* TODO Uzupełnij kod o zmienne reprezentujące pozostałe pola tabeli projekt (patrz rys. 1),
	.	a następnie wygeneruj dla nich tzw. akcesory (Source -> Generate Getters and Setters),
	.	ponadto dodaj pusty konstruktor.
	*/

	public Projekt(){

	}

	public Integer getProjektId() {
		return projektId;
	}

	public void setProjektId(Integer projektId) {
		this.projektId = projektId;
	}

	public String getNazwa() {
		return nazwa;

	}
```

```
	public void setNazwa(String nazwa) {
		this.nazwa = nazwa;
	}

	/* TODO Getters and Setters
	.
	*/
}
```

Do odwzorowania bazodanowych pól *data_oddania* i *dataczas_utworzenia* można by użyć odpowiednio zmiennych typu *java.sql.Date* i *java.sql.Timestamp*. Jednak wygodniej jest korzystać ze zmiennych typu *java.time.LocalDate* i *java.time.LocalDateTime* np.

```
	private LocalDateTime dataczasUtworzenia;
	private LocalDate dataOddania;
```

Przykłady tworzenia obiektów:

```
	LocalDateTime.now(); //aktualny czas i data
	LocalDate.of(2018, 6, 22);
```

W klasach modelu można też korzystać z adnotacji spoza pakietu *javax.persistence* np. *@CreationTimestamp* i *@UpdateTimestamp*, które pozwalają na automatyczne przypisywanie dat i czasu podczas tworzenia lub modyfikacji rekordu.

```
import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

// ...

@CreationTimestamp
@Column(name = "dataczas_utworzenia", nullable = false, updatable = false)
private LocalDateTime dataCzasUtworzenia;

// ...

@UpdateTimestamp
@Column(name = "dataczas_modyfikacji", nullable = false)
private LocalDateTime dataCzasModyfikacji;
```

2. Utwórz klasę odwzorowującą tabelę *zadanie*.

```
package com.project.model;

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

@Entity
@Table(name="zadanie")
public class Zadanie {
	@Id
	@GeneratedValue
	@Column(name="zadanie_id")
	private Integer zadanieId;

	/* TODO Uzupełnij kod o zmienne reprezentujące pozostałe pola tabeli projekt (patrz rys. 1),
	.	a następnie wygeneruj dla nich tzw. akcesory (Source -> Generate Getters and Setters),
	.	ponadto dodaj pusty konstruktor.
	*/
}
```

3. Realizacja dwukierunkowej relacji jeden do wielu.

W klasie *Zadanie* dodaj zmienną *projekt* oraz adnotację `@ManyToOne`. Użyj też adnotacji `@JoinColumn`. Wygeneruj akcesory dla nowo utworzonej zmiennej.

```
@ManyToOne
@JoinColumn(name = "projekt_id")
private Projekt projekt;
```

W klasie *Projekt* dodaj listę *zadania* z adnotacją `@OneToMany` i parametrem `mappedBy`, którego wartość wskazuje zmienną po drugiej stronie relacji tj. *projekt* z klasy *Zadanie*. Pamiętaj o wygenerowaniu akcesorów.

```
@OneToMany(mappedBy = "projekt")
private List<Zadanie> zadania;
```

4. Utwórz klasę odwzorowującą tabelę *student* z dodatkowymi konstruktorami oraz zaimplementuj relację wiele do wielu między tabelami *projekt* i *student*.

```
@Entity
@Table(name = "student")
public class Student {

	/* TODO Uzupełnij kod o zmienne reprezentujące pozostałe pola tabeli projekt (patrz rys. 1),
	.	a następnie wygeneruj dla nich tzw. akcesory (Source -> Generate Getters and Setters),
	.	ponadto dodaj pusty konstruktor.
	*/
	public Student() {
	}

	public Student(String imie, String nazwisko, String nrIndeksu, Boolean stacjonarny) {
		this.imie = imie;
		this.nazwisko = nazwisko;
		this.nrIndeksu = nrIndeksu;
		this.stacjonarny = stacjonarny;
	}

	public Student(String imie, String nazwisko, String nrIndeksu, String email,
	                                                                Boolean stacjonarny) {
		this.imie = imie;
		this.nazwisko = nazwisko;
		this.nrIndeksu = nrIndeksu;
		this.email = email;
		this.stacjonarny = stacjonarny;
	}
}
```

Do istniejącej klasy *Projekt* dodaj zmienną *studenci* z adnotacją `@ManyToMany`

```
@ManyToMany
@JoinTable(name = "projekt_student",
   joinColumns = {@JoinColumn(name="projekt_id")},
   inverseJoinColumns = {@JoinColumn(name="student_id")})
private Set<Student> studenci;
```

Do nowo utworzonej klasy *Student* dodaj zmienną *projekty* z adnotacją `@ManyToMany` (`mappedBy` wskazuje na zmienną w klasie *Projekt*).

```
@ManyToMany(mappedBy = "studenci")
private Set<Projekt> projekty;
```

W każdej z tych klas wygeneruj akcesory dla utworzonych zmiennych.

5. Utwórz serwlet np. *ProjektEdycja*. We wnętrzu jego metody `doGet` wpisz wywołanie metody `doPost(request, response);`. Natomiast w metodzie `doPost` dodaj poniższy fragment, który zapisze przykładowy projekt w bazie danych.

```
if(request.getParameter("btn_zapisz") != null) {
	EntityManager entityManager = HibernateUtil.getInstance().createEntityManager();
	Projekt projekt = new Projekt();
	projekt.setNazwa("Projekt testowy");
	/* TODO Uzupełnij kod ustawiając wartości pozostałych pól tabeli projekty
	.
	*/
	entityManager.getTransaction().begin();
	entityManager.persist(projekt);
	entityManager.getTransaction().commit();
	entityManager.close(); // zalecane umieszczenie metody close() w bloku finally
}
```

Następnie utwórz stronę JSP (np. *projekt_edycja.jsp*) z formularzem zawierającym przycisk *Zapisz*.

```
<form action="ProjektEdycja" method="POST">
	<input name="btn_zapisz" value="Zapisz" type="submit">
</form>
```

Zmodyfikuj stronę JSP i serwlet tak, aby po każdym wciśnięciu przycisku *Zapisz* na stronie wyświetlany był identyfikator zapisanego projektu. W tym celu do strony JSP można dodać fragment:

```
ID zapisanego projektu: ${projekt.projektId}
```

(`${…}` jest wyrażeniem EL, patrz p. 21).

Natomiast w serwlecie, po fragmencie zapisującym projekt, należy zaimplementować przekierowanie przekazujące stronie JSP obiekt reprezentujący zapisany rekord np.:

```
if(request.getParameter("btn_zapisz") != null) {
	...
	request.setAttribute("projekt", projekt);
}
ServletContext context = getServletContext();
RequestDispatcher dispatcher = context.getRequestDispatcher("/projekt_edycja.jsp");
dispatcher.forward(request, response);
```

6. Sprawdź poniższe wyszukiwanie projektu za pomocą identyfikatora przekazywanego w drugim parametrze metody *find*.

```
int projektId = 1;
Projekt projekt = null;
try {
   projekt = entityManager.find(Projekt.class, projektId);
} finally {
   entityManager.close();
}
if (projekt != null) {
   System.out.println("Projekt ID: " + projektId + ", nazwa: "  + projekt.getNazwa());
}
```

7. Przetestuj usuwanie projektu z bazy danych.

```
Projekt projekt = entityManager.find(Projekt.class, projektId);
entityManager.getTransaction().begin();
entityManager.remove(projekt);
entityManager.getTransaction().commit();
```

8. Wyszukiwanie za pomocą zapytań w JPQL.
Uwaga! W zapytaniu używamy zawsze nazw klas i ich zmiennych a nie rzeczywistych nazw pól i tabel z bazy danych!

```
TypedQuery<Projekt> query = entityManager
    .createQuery("SELECT p FROM Projekt p WHERE p.projektId = 1", Projekt.class);
Projekt projekt = query.getSingleResult();
if (projekt!= null) {
       System.out.println("Projekt ID=1: " + projekt.getNazwa());
}
```

9. Pobieranie listy wyników za pomocą JPQL.

```
TypedQuery<Projekt> query = entityManager
    .createQuery("SELECT p FROM Projekt p WHERE p.projektId > 1", Projekt.class);
List<Projekt> projekty = query.getResultList();
if (projekty != null) {
   for (Projekt p : projekty) {
     System.out.println("Projekt ID = " + p.getProjektId() + ": " + p.getNazwa());
   }
 }
```

10. Przykład pobierania wyników za pomocą sparametryzowanego zapytania w JPQL.

```
TypedQuery<Projekt> query = entityManager
    .createQuery("SELECT p FROM Projekt p WHERE p.projektId > :id", Projekt.class);
query.setParameter("id", 1);
List<Projekt> projekty = query.getResultList();
if (projekty != null) {
   for (Projekt p : projekty) {
      System.out.println("Projekt ID = " + p.getProjektId() + ": " + p.getNazwa());
   }
}
```

11. Przykład pobierania wyników za pomocą sparametryzowanego zapytania w JPQL (z użyciem listy).

```
TypedQuery<Projekt> query = entityManager
   .createQuery("SELECT p FROM Projekt p WHERE p.projektId IN :ids", Projekt.class);
List<Integer> ids = new ArrayList<Integer>();
ids.add(1);
ids.add(2);
query.setParameter("ids", ids);
```

```
List<Projekt> projekty = query.getResultList();
if (projekty != null) {
   for (Projekt p : projekty) {
       System.out.println("Projekt ID = " + p.getProjektId() + ": " + p.getNazwa());
   }
}
```

12. Utrwalanie nowych obiektów (wykonywane w bloku transakcyjnym).

```
entityManager.persist(projekt);
```

13. Modyfikacja obiektów (wykonywana w bloku transakcyjnym).

```
projektKopia = entityManager.merge(projekt); // zwracany obiekt projektKopia jest encją
                                             // zarządzaną, a projekt encją odłączoną
```

14. Odświeżenie stanu zarządzanej encji danymi z bazy.

```
entityManager.refresh(projekt);
```

15. Stronicowanie.

```
TypedQuery<Projekt> query = entityManager.createQuery("SELECT p FROM Projekt p ORDER BY
                                                       p.dataCzasUtworzenia DESC", Projekt.class);
query.setFirstResult(offset);
query.setMaxResults(limit);
List<Projekt> projekty = query.getResultList();
```

16. Dokonywanie masowych zmian w bazie danych.

```
int alteredRows = em.createQuery("UPDATE Student s SET s.stacjonarny = true").executeUpdate();
```

17. Przetestuj poniższy fragment kodu zapisujący powiązane tabele *projekt* i *zadania*.

```
Projekt projekt = new Projekt();
projekt.setNazwa("Projekt testowy");
/* TODO Uzupełnij kod ustawiając wartości pozostałych pól tabeli projekty
.
*/

Zadanie zadanie1=new Zadanie();
zadanie1.setNazwa("Zadanie 1");
/* TODO Uzupełnij kod ustawiając wartości pozostałych pól tabeli zadanie
.
*/

Zadanie zadanie2=new Zadanie();
zadanie2.setNazwa("Zadanie 2");
/* TODO Uzupełnij kod ustawiając wartości pozostałych pól tabeli zadanie
.
*/

//przypisujemy do zadań projekt
zadanie1.setProjekt(projekt);
zadanie2.setProjekt(projekt);

entityManager.getTransaction().begin();
//utrwalanie zawsze dla wszystkich obiektów - projektu i jego zadań
entityManager.persist(projekt);
entityManager.persist(zadanie1);
entityManager.persist(zadanie2);
entityManager.getTransaction().commit();
```

```
entityManager.refresh(projekt); //odświeżenie stanu zarządzanej encji
                                //na podstawie informacji z bazy danych
//sprawdzamy czy w bazie danych do projektu zostały przypisane zadania
List<Zadanie> zadania = projekt.getZadania();

System.out.printf("Projekt - Id: %d, Nazwa: %s%n", projekt.getProjektId(), projekt.getNazwa());
for (Zadanie zad : zadania) {
   System.out.printf(" Zadanie - Id: %d, Nazwa: %s%n", zad.getZadanieId(), zad.getNazwa());
}
```

18. Przykład drukowania danych z podziałem na strony. Liczbę elementów na stronie określa wartość zmiennej *pageSize*.

```
EntityManager entityManager = HibernateUtil.getInstance().createEntityManager();

boolean next = false;
int page = 0;
Integer offset = 0;
Integer pageSize = 10;
   do {
      TypedQuery<Projekt> query = entityManager.createQuery("SELECT p FROM Projekt p
                                          ORDER BY p.dataczasUtworzenia", Projekt.class);
      query.setFirstResult(offset);
      query.setMaxResults(pageSize);
      List<Projekt> projekty = query.getResultList();
      if(projekty != null && !projekty.isEmpty()) {
          page += 1;
          System.out.println("Strona " + page);
          for (Projekt projekt : projekty) {
              System.out.println(" - projekt ID: " + projekt.getProjektId()
                                                   + ", nazwa: " + projekt.getNazwa());
          }
          next = true;
          offset = offset + pageSize;
       } else {
          next = false;
       }
    } while (next);

    entityManager.close();
```

19. Przykładowy sposób realizacji natywnych zapytań w języku SQL

```
List<Object[]> results = entityManager.createNativeQuery("SELECT projekt_id, nazwa FROM
                                                          projekt").getResultList();
for (Object[] objects : results) {
      Integer projektId = (Integer) objects[0];
      String nazwa = (String) objects[1];
      System.out.println("ID: " + projektId + ", NAZWA: " + nazwa);
}
```

Można również przypisywać wyniki do encji np.

```
List<Projekt> projekty = entityManager.createNativeQuery("SELECT * FROM projekt ORDER BY
                                                  nazwa", Projekt.class).getResultList();
```

Uwaga! Należy unikać nieuzasadnionego korzystania z natywnych zapytań SQL.

20. Parametr *fetch*

Wartość parametru `fetch` równa `FetchType.LAZY` (jest to wartość domyślna, nie wymaga jawnego deklarowania) oznacza, że powiązane dane są pobierane z bazy dopiero w momencie ich pierwszego użycia. Nawet jeżeli pobierzemy dane np. za pomocą poniższego kodu

```
int projektId = 1;
```

```
Projekt projekt = entityManager.find(Projekt.class, projektId);
```

to realizująca np. jednokierunkową relację jeden do wielu lista *zadania* klasy *Projekt* będzie w rzeczywistości pusta. Dopiero pierwsze wywołanie metody `projekt.getZadania()` pobierającej listę encji powoduje chwilę przed zwróceniem listy wykonanie bazodanowego zapytania ustawiającego jej encje. Jest to bardzo użyteczny mechanizm zwiększający wydajność, jeżeli jednak zachodzi potrzeba wymuszenia w naszym programie natychmiastowego pobierania powiązanych danych to musimy użyć `FetchType.EAGER`, np.

```
@OneToMany(fetch = FetchType.EAGER)
 private List<Zadanie> zadania;
```

21. Wyrażenia EL (*Expression Language*) na stronach JSP

Każde wyrażenie języka EL zaczyna się od znaku $, zaraz zanim jest para nawiasów klamrowych z treścią np. wyrażenie wyświetlające host ma postać `${header.host}`.

Zamiast używać `<%= request.getAttribute("info")%>` można skorzystać z wyrażenia `${requestScope.info}` (lub bez określania zasięgu `${info}`)

Natomiast do pobrania parametru przekazywanego za pomocą metody GET lub POST można użyć wyrażenia `${param.x_projekt_id}`, co odpowiada wcześniej poznanemu `<%=request.getParameter("x_projekt_id")%>`

W wyrażeniach możemy określać zasięg - strony, żądania, sesji, kontekstu aplikacji, korzystając odpowiednio z `pageScope`, `requestScope`, `sessionScope` i `applicationScope`. Jeżeli nie podamy konkretnego zasięgu to zostaną przeszukane wszystkie (od strony do kontekstu aplikacji), aby odnaleźć dany atrybut.

Wyróżniamy kilka obiektów, które można wykorzystać w EL m.in.:

- `param` – mapa parametrów żądania (lub `paramValues` zwracająca tablicę wartości),
- `header` – mapa nagłówków żądania (lub `headerValues` zwracająca tablicę wartości)
- `cookie` – mapa ciasteczek
- `initParam` – mapa parametrów kontekstu.

UWAGA! Język wyrażeń EL pozwala na dostęp do właściwości przekazywanego ziarna!

Po utworzeniu w serwlecie obiektu klasy `Projekt` (będącego w istocie javowym ziarnem) i przekazaniu go, np. za pomocą poniższego kodu, do strony JSP (zakładamy, że strona jest umieszczona w podkatalogu *pages*)

```
int projektId = 1;
Projekt projekt = entityManager.find(Projekt.class, projektId);
request.setAttribute("projekt", projekt);
ServletContext context = getServletContext();
RequestDispatcher dispatcher = context.getRequestDispatcher("/pages/projekt_edit.jsp");
dispatcher.forward(request, response);
```

mamy na tej stronie dostęp do wszystkich właściwości przekazanego ziarna. Możemy np. pobrać nazwę projektu za pomocą wyrażenia `${requestScope.projekt.nazwa}`, co odpowiada wywołaniu metody `projekt.getNazwa();`.

W serwlecie można też utworzyć listę obiektów i przekazać ją stronie JSP np.

```
TypedQuery<Projekt> query = em.createQuery("SELECT p FROM Projekt p", Projekt.class);
List<Projekt> projekty = query.getResultList();
request.setAttribute("projekty", projekty);
ServletContext context = getServletContext();
RequestDispatcher dispatcher = context.getRequestDispatcher("/pages/projekt_list.jsp");
dispatcher.forward(request, response);
```

Natomiast na stronie JSP do drukowania można użyć pętli *For-Each* np.

```
<c:forEach var="projekt" items="${requestScope.projekty}" varStatus="info">
      <c:out value="${projekt.nazwa}" /><br>
```

```
</c:forEach>
```

22. JSTL (*JSP Standard Tag Library*) - standardowa biblioteka znaczników ułatwiająca pisanie stron JSP

W podkatalogu */WebContent/WEB-INF/lib/* naszego projektu znajduje się kilka takich bibliotek tj.

- *javax.servlet.jsp.jstl-api-1.2.1.jar*,
- *javax.servlet.jsp.jstl-1.2.1.jar*,
- *java-time-jsptags-1.1.4.jar*,
- *javax.el-api-3.0.0.jar*.

Aby z nich korzystać na stronie JSP trzeba dodać odpowiednie dyrektywy *taglib* np.:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@ taglib prefix="javatime" uri="http://sargue.net/jsptags/time"%>
```

Dzięki załączonym bibliotekom można m.in.:

- drukować zmienne z automatycznym pomijaniem wartości **null**

```
<c:out value="${projekt.nazwa}" />,
```

- iterować przekazywane kolekcje obiektów (np. `List<Projekt> projekty`)

```
<c:forEach var="projekt" items="${requestScope.projekty}">
	<c:out value="${projekt.projektId}" /><br>
	<c:out value="${projekt.nazwa}" /><br>
	<c:out value="${projekt.opis}" /><br>
</c:forEach>,
```

- formatować wartości z datą i czasem np.

Dla zmiennych klasy *LocalDate* lub *LocalDateTime*

```
<javatime:format value="${projekt.dataczasUtworzenia}" var="fmtDataczasUtworzenia"
 pattern="yyyy-MM-dd hh:mm:ss" />
```

Dla starszych rozwiązań korzystających np. z klasy *Calendar*

```
<fmt:formatDate value="${projekt.dataczasUtworzenia.time}" var="fmtDataczasUtworzenia" type="both"
                                                                pattern="yyyy-MM-dd HH:mm:ss" />
<c:out value="${fmtDataczasUtworzenia}" /><br>,
```

- czy też konstruować linki z automatyczną obsługą mechanizmu sesji np.

```
<c:url value="/pages/zadania.jsp" var="linkZadania">
	<c:param name="x_projekt_id" value="${projekt.projektId}" />
</c:url>
<a href='<c:out value="${linkZadania}" />'>Zadania projektu</a>.
```

- lub używać instrukcji warunkowych np.

```
<c:if test="${projektDostepny}">
	<button id="pobierzBtn" title="Kliknij, aby pobrać projekt studenta.">Pobierz</button>
</c:if>

<c:choose>
	<c:when test="${empty listaStudentow}">
		<span style="text-align: left;">Projekt nie został wybrany.</span>
	</c:when>
	<c:otherwise>
		<span style="text-align: left;">Projekt będzie realizowany.</span>
	</c:otherwise>
</c:choose>
```

23. Przykład strony JSP drukującej listę projektów

```
<%@page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@ taglib prefix="javatime" uri="http://sargue.net/jsptags/time"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Lista projektów</title>
</head>
<body>
	<h2>Lista projektów</h2>
	<table border="1" cellpadding="3">
		<tr>
			<th>Lp.</th>
			<th>Id</th>
			<th>Nazwa</th>
			<th>Opis</th>
			<th>Utworzony</th>
			<th>Data obrony</th>
			<th>Edycja</th>
		</tr>
		<c:forEach var="projekt" items="${requestScope.projekty}" varStatus="info">
			<tr>
				<td>${info.count}.</td>
				<td><c:out value="${projekt.projektId}" /></td>
				<td><c:out value="${projekt.nazwa}" /></td>
				<td><c:out value="${projekt.opis}" /></td>

				 <javatime:format value="${projekt.dataczasUtworzenia}"
				 		var="fmtDataczasUtworzenia" pattern="yyyy-MM-dd hh:mm:ss" />

				<td><c:out value="${fmtDataczasUtworzenia}" /></td>

				<javatime:format value="${projekt.dataOddania}" var="fmtDataOddania"
				 		pattern="yyyy-MM-dd" />
				<td><c:out value="${fmtDataOddania}" /></td>

				<c:url value="/pages/zadania.jsp" var="linkZadaniaProjektu">
					<c:param name="x_projekt_id" value="${projekt.projektId}" />
				</c:url>
				<td><a href='<c:out value="${linkZadaniaProjektu}" />'>Zadania</a></td>
			</tr>
		</c:forEach>
	</table>
</body>
</html>
```