Katedra Systemów Informacyjnych
Uniwersytet Morski w Gdyni

Informatyka, Aplikacje internetowe i mobilne, semestr 5

Programowanie aplikacji webowych

Laboratorium nr 4

Podstawowa aplikacja WWW z obsługą bazy danych na przykładzie przechowywania i zarządzania informacjami o książkach

Aplikacja Web Ksiazki

Napisz aplikację Web o nazwie `WebAppKsiazki` umożliwiającą użytkownikowi podstawową obsługę informacji na temat książek dla księgarni lub biblioteki.

Potrzebne oprogramowanie:

- środowisko **Open JDK 20**,

- serwer WWW, czyli tutaj kontener servletów **Apache Tomcat 10.1**,

- baza danych **postgreSQL 14** z programem do obsługi bazy danych, np. **pgAdmin4**, **DBeaver**.

Oprogramowanie DevEnv

Można skorzystać z bazy danych PostgreSQL i programu (klienta) przygotowanego w środowisku DevEnv korzystając z poniższych informacji:

Baza danych **PostgreSQL** 14 (instalacja portable) jest zainstalowana w katalogu C:\DevEnv\xampp\pgsql Usługę bazy danych można uruchomić poprzez wykonanie pliku wsadowego C:\DevEnv\xampp\pgsql_start.bat. Zatrzymanie działania bazy danych plik wsadowy C:\DevEnv\xampp\pgsql_stop.bat. Dostęp z prawami roli administratora – login: postgres, hasło: puste.

Potrzebne pliki .jar:

- postgresql-42.6.0.jar – biblioteka ze sterownikami do obsługi bazy danych PostgreSQL,

- jakarta.servlet.jsp.jstl-3.0.0.jar – standardowa biblioteka do obsługi tagów JSTL w plikach JSP,

- jakarta.servlet.jsp.jstl-api-3.0.0.jar - zawiera implementację klas API.

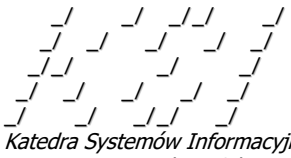
Pliki nie są dołączane do serwera Tomcat.

Instrukcje

Baza danych

1. W swojej bazie PostgreSQL utwórz schemat o nazwie `ksiazki`. Następnie uruchom kody SQL, które utworzą i wypełnia przykładowymi danymi trzy tabele: `ksiazka`, `kategoria` i `wydawnictwo` opisane poniżej Skorzystaj z pgAdmin4 lub DBeaver i kodów SQL zapisanych w dołączonych plikach .sql.

> Zwróć uwagę, że typem pola z identyfikatorem we wszystkich tabelach, czyli `idk`, `idk` i `idw` powinien być `serial`, pozwoli to na automatyczne indeksowanie identyfikatorów,
>
> Kod SQL do utworzenia tabel można znaleźć w plikach: `sql_ksiazka.sql`, `sql_kategoria.sql` i `sql_wydawnictwo.sql`, natomiast do importu danych, odpowiednio w plikach: `sql_ksiazka_dane.sql`, `sql_kategoria_dane.sql` i `sql_wydawnictwo_dane.sql`.



Katedra Systemów Informacyjnych
Uniwersytet Morski w Gdyni

Tabele powinny mieć postać:

`wydawnictwo`

idw [PK] integer	nazwa character varying (50)	adres character varying
1	Helion	Gliwice, Polska
2	PWN	Warszawa, Polska
3	O'REILLY	Boston, USA

`ksiazka`

idk [PK] integer	tytul character varying (60)	idwyd integer	idkat integer
1	Java. Podstawy	1	4
2	Projektownie serwisów WW...	1	1
3	Zrozumieć JavaScript	1	3
4	Head first Java	3	4
5	HTML5. Komponenty	2	2
6	Wydajny JavaScript	2	3

`kategoria`

idk [PK] integer	opis character varying (50)
1	WWW
2	HTML
3	JavaScript
4	Java

W tabeli `ksiazka` ustaw klucze obce o nazwach: `ksiazka_kategoria_fkey` oraz `ksiazka_wydawnictwo_fkey` łączące kolumny `idwyd` i `idkat` z kolumnami kluczy głównych odpowiednich tabel `kategoria` i `wydawnictwo`. Zabroń dokonywania zmian i usuwania danych.

Aplikacja

Model – logika aplikacji

2. Utwórz projekt aplikacji typu `Dynamic Web Project` o nazwie `WebAppKsiazki`. Zmień domyślne kodowanie znaków projektu na `utf-8`. Wgraj pliki `.jar` z bibliotekami do katalogu `webapp\WEB-INF\lib` projektu.

3. W folderze `src/main/java` projektu utwórz klasę Java o nazwie `modelKsiazka` do przechowywania obiektów reprezentujących rekordy tabeli `ksiazka`. Słowo model w nazwie

oznacza, że plik należy do części aplikacji reprezentującej jej struktury tzw. logikę aplikacji zgodnie ze wzorcem projektowym MVC czyli Model-View-Controller.

Ponieważ dane z/do baz są zwykle przesyłane przez sieć, klasa `modelKsiazka` powinna implementować interfejs `Serializable` (informację o tym można dodać podczas tworzenia klasy). Interfejs `Serializable` wymaga zdefiniowania zmiennej `serialVersionUID`, która jest ustawiana na losową wartość i pomaga rozpoznać czy operacje serializacji i deserializacji są wykonywane dla tej samej klasy. Można to zrobić automatycznie po wygenerowaniu kodu.

```
private static final long serialVersionUID = 1L;
```

W klasie `modelKsiazka` utwórz zmienne/właściwości odpowiadające polom w tabeli `ksiazka` bazy danych oraz metody do ich odczytywania i zapisywania.

W przypadku tabeli `ksiazka`, której odpowiednikiem jest klasa `modelKsiazka` zmiennymi będą:

```
private int idk;
private String tytul;
private modelWydawnictwo wyd;
private modelKategoria kat;
```

W tabeli `ksiazka` w bazie danych są dwa pola z identyfikatorami do innych tabel `idwyd` i `idkat`. W modelu będą one reprezentowane przez odpowiednie klasy `modelWydawnictwo` i `modelKategoria`, które zostaną utworzone w następnym punkcie.

Zgodnie z zasadami dobrego programowania właściwości są ustawione na dostęp prywaty, a więc ich obsługa wymaga zdefiniowania metod `get…` i `set…` do pobierania i zapisywania wartości. Przykłady takich metod dla pierwszego pola podano poniżej.

```
public int getIdk() {
	return this.idk;
}

public void setIdk(int idk) {
	this.idk = idk;

}
```

Warto zwrócić uwagę, że zapis `this.idk` wskazuje właściwość klasy, podczas gdy `idk` to parametr metody podany w nawiasach. Używając takiego zapisu można wykorzystać te same nazwy zmiennych.

Zdefiniuj analogiczne metody `get…` i `set…` dla zmiennej `tytul` klasy `modelKsiazka`, pamiętaj, że jest ona typu `String`.

Zdefiniuj metody `get…` i `set…` dla zmiennej obiektu reprezentującego wydawnictwo, maja one postać:

```
public modelWydawnictwo getWyd() {
	return wyd;
}

public void setWyd(modelWydawnictwo wyd) {
	this.wyd=wyd;

}
```

Analogicznie zdefiniuj metody dla zmiennej obiektu reprezentującego kategorię.

W środowisku Eclipse można wygenerować metody get… i set… automatycznie ustawiając kursor w odpowiednim miejscu w kodzie klasy i wybierając z menu podręcznego Source | Generate Getters and Setters.

4. W folderze src/main/java projektu utwórz klasę Java o nazwie modelKategoria i modelWydawnictwo do przechowywania obiektów reprezentujących rekordy tabel kategoria i wydawnictwo. Utwórz odpowiednie właściwości zgodnie z opisem tabel i wygeneruj metody, analogiczne jak w klasie modelKsiazka. Po ich utworzeniu powinny zniknąć informacje o błędach w klasie modelKsiazka.

Obiekt sterownika bazy danych aplikacji - DAO

5. W folderze src/main/java utwórz klasę Java o nazwie daoKsiazki.java, która będzie pełniła rolę sterownika do bazy danych naszej aplikacji (DAO - Data Access Object). W kreatorze dołącz metodę main, która przyda się do testowania.

W klasie daoKsiazki zadeklaruj zmienne dbcon i dbstat do obsługi bazy..

```
private Connection dbcon=null;
```

```
private Statement dbstat=null;
```

Deklaracje wymagają dołączenia do pliku odpowiednich klas Connection i Statement z pakietu java.sql:

Do wygenerowania obiektu połączenia potrzebne będzie również dołączenie klasy sterownika do języka SQL java.sql.DriverManager.

6. W klasie daoKsiazki utwórz metodę otwierającą połączenie z bazą danych, np.:

```
private void otworzCon()
{
      String login = "<login>"; //postgres dla lokalnego serwera
      String haslo = "<haslo>"; //puste dla lokalnego serwera
      String url="jdbc:postgresql://localhost:5432/postgres?currentSchema
      =\"ksiazki\"";
      try {
            Class.forName("org.postgresql.Driver");
            dbcon = DriverManager.getConnection(url, login, haslo);
            dbstat=dbcon.createStatement();
            System.out.println("Połączenie otwarte");
      }
      catch (ClassNotFoundException ex)
      {System.err.println("ClassNotFoundException z init:"
         +ex.getMessage());}
      catch (SQLException ex)
      {System.err.println("SQLException z init: " + ex.getMessage());}
}
```

Wartości potrzebne do nawiązania połączenia można zapisać w zmiennych, tutaj są to: `login`, `haslo` i `url`. Na razie wykorzystujemy `login` i `haslo` do bazy na lokalnym serwerze PostgreSQL. Standardowo ustawiany jest użytkownik `postgres`. W bazach danych działających w Sieci ze względów bezpieczeństwa definiowani są różni użytkownicy dla różnych baz, albo nawet dla części tej samej bazy, natomiast hasła zapisywane są w postaci szyfrowanej w plikach konfiguracyjnych.

Budowa `url` jest ściśle określona. Na początku wpisujemy jdbc. Interfejs JDBC (Java Database Connectivity) umożliwia pracę w Javie z większością istniejących baz danych. Po dwukropku podajemy z jakim systemem baz danych chcemy pracować, a po kolejnym: adres, port, nazwą bazy danych i ewentualny schemat. Łączymy się z lokalną bazą postgreSQL za pomocą adresu localhost, na standardowym porcie 5432. Użytkownikiem i nazwą bazy jest postgres. Nasz schemat to `ksiazki`. Przy połączeniu z serwerem szkolnym foka.umg.edu.pl (połączenie zrealizowane za pomocą tunelu) trzeba te dane zmienić, jako użytkownika i jednocześnie nazwę bazy danych trzeba podać swoje konto s<*numer indeksu*>, np.

```
"jdbc:postgresql://localhost:5432/s99999?currentSchema=\"ksiazki\""
```

Pozostałe instrukcje trzeba zapisać w bloku `try`, ponieważ wymagają obsługi wyjątków. Dołączyć trzeba brakującą klasę `SQLException`, również z pakietu `java.sql`. W kodzie powyżej zapisano obsługę polegającą na wypisaniu odpowiednich komunikatów w konsoli.

7. W klasie `daoKsiazki` utwórz metodę zamykającą połączenie z bazą danych:

```
private void zamknijCon() {
      if(dbcon==null) return;
      try {
            dbcon.close();
            System.out.println("Połączenie zamknięte");
      }
      catch(SQLException ex)
      {System.out.println("Problem z zamknięciem bazy");}
}
```

Tutaj wymagane jest tylko zamknięcie istniejącego połączenia, ale trzeba to zrobić również w bloku `try` żeby obsłużyć wyjątek `SQLException`.

Przetestuj działanie połączenia z lokalnym pgsql

W metodzie main w klasie daoKsiazki dopisz kod testujący połączenie, np.:

```
daoKsiazki testpgsql = new daoKsiazki();
testpgsql.otworzCon();
testpgsql.zamknijCon();
```

Następnie uruchom plik daoKsiazki.java jako Java Application. Sprawdź w konsoli Eclipse (`Console`), czy udało się nawiązać prawidłowe połączenie z postgresql. Wstaw komentarz (znaki //) dla w/w trzech wierszy w metodzie `main` w przypadku jeśli wszystko przebiegło prawidłowo. Jeśli nie to sprawdź, czy działa serwer pgsql.

8. W klasie `daoKsiazki` utwórz metodę o nazwie `listaKategorii` odczytującą z bazy dane wszystkich kategorii książek. Gotowa lista zostanie później przekazana przez servlet do strony .jsp, w celu utworzenia menu kategorii.

Ponieważ zwykle nie wiemy ile książek zapisano w bazie wykorzystamy do zapisu dynamiczną tablicę `ArrayList`, dla której nie trzeba z góry definiować długości. Metoda będzie zwracała taką tablicę jako wynik. W metodzie `listaKategorii` zdefiniuj tablicę do zapisania wyniku. Kod może mieć postać:

```
public ArrayList<modelKategoria> listaKategorii() {
      ArrayList<modelKategoria> lk= new ArrayList<modelKategoria>();
}
```

Konstrukcja z nawiasem trójkątnym <> umożliwia zdefiniowanie typu obiektów, które będą przechowywane na liście `lk`.

Kolejny krok to zdefiniowanie pytania SQL, zapiszemy je w zmiennej `pyt` klasy `String`:

```
String pyt="SELECT idk, opis FROM ksiazki.kategoria";
```

Następne instrukcje wymagają już połączenia z bazą, a więc muszą być zapisane w bloku `try`.

```
try
{
      otworzCon();
      ResultSet wyniki=dbstat.executeQuery(pyt);
      while(wyniki.next())
      {
            modelKategoria k=new modelKategoria();
            k.setIdk(wyniki.getInt("idk"));
            k.setOpis(wyniki.getString("opis"));
            lk.add(k);
      }
}
catch (Exception e) {System.out.println(e);}
finally {zamknijCon();}
```

W bloku `try` otwieramy połączenie z bazą, wysyłamy pytanie SQL metodą `executeQuery`, a wynik zapisujemy w obiekcie klasy `ResultSet`, tutaj reprezentuje go zmienna `wyniki`. Obiekt klasy `ResultSet` przechowuje dane otrzymane z bazy w odpowiedzi na zadane pytanie SQL w postaci struktury złożonej z wierszy/rekordów. Klasę `ResultSet` trzeba dołączyć z pakietu `java.sql`.

Przeglądanie struktury wyniki wymaga użycia pętli `while` (nie wiemy ile jest wierszy) oraz skorzystania z iteratora do przeglądania w postaci metody `next()`. Każde wywołanie `next()` powoduje przejście do kolejnego wiersza/rekordu zapisanego w obiekcie `wyniki`.

W pętli tworzymy obiekt k klasy `modelKategoria`, który wykorzystamy do zapisania danych w postaci obiektów klasy `modelKategoria` w tablicy `lk`. Korzystając z obiektu `wyniki` oraz metod do pobierania danych `getInt` i `getString` pobieramy dane z aktualnie przeglądanego wiersza w obiekcie wyniki. Są to standardowe metody dla obiektów klasy Integer i String. Następnie, korzystając z metod `setIdk` i `setOpis` klasy `modelKategoria` zapisujemy dane w obiekcie `k`. Do dodawania elementu do klasy `ArrayList` służy metoda `add`. Jako jej parametr podajemy to co chcemy zapisać w tablicy, oczywiście zgodnie z typem tablicy.

Ewentualne wyjątki w części `catch` obsługujemy korzystając z ogólnej postaci wyjątku klasy `Exception`. Pomyślne lub nie zakończenie operacji wymaga zamknięcia połączenia z bazą, co najlepiej zrobić w części `finally` bloku `try`.

Pozostaje już tylko zwrócić wynik w postaci utworzonej tablicy `lk`:

```
return lk;
```

9. Przetestuj działanie metody korzystając z metody `main`. W celu przetestowania klasy `daoKsiazki` trzeba zdefiniować obiekt tej klasy np. o nazwie `obi`. Drugi niezbędny obiekt to `lista` klasy `ArrayList<modelKategoria>`, w którym można zapisać wynik działania metody `listaKategorii`, żeby następnie wypisać elementy tej listy w konsoli. Kod może mieć postać:

```
public static void main(String[] args)
{
	daoKsiazki obi=new daoKsiazki();
	ArrayList<modelKategoria> lista=obi.listaKategorii();
	String s="";
	for(int i=0; i<lista.size(); i++)
		s+=lista.get(i).getOpis()+", ";
	System.out.println(s);
}
```

Aktualną liczbę elementów w tablicy `ArrayList` można odczytać za pomocą metody `size()`, natomiast kolejny element pobieramy metodą `get`, np. pierwszy `get(0)`. Korzystanie z list tablicowych wymaga dołączenia klasy `ArrayList` z pakietu java.util.

Pamiętaj, że test z metody `main` uruchamiamy jako aplikację Java, czyli bez serwera Tomcat.

W wyniku działania testu w konsoli powinny pojawić się nazwy kategorii z tabeli kategorie: `WWW, HTML, JavaScript, Java,`

10. Analogicznie jak listę kategorii skonstruuj metodę do generowania listy informacji o wydawnictwach `listaWydawnictw`. Analogicznie skonstruuj metodę do generowania listy książek o nazwie `listaKsiazek`. W jej przypadku dołącz do funkcji listę kategorii i wydawnictw wołając utworzone metody i dodaj do listy książek obiekty z modelem wydawnictwa i kategorii. Przetestuj działanie wszystkich metod wpisując odpowiedni kod w metodzie `main`.

Kontrolery aplikacji w postaci servletów

11. W folderze `src/main/java` utwórz klasę servletu Java o nazwie `servletListaKsiazek`. Tworzone w aplikacji servlety będą pełniły rolę kontrolerów w modelu MVC czyli pośredników pomiędzy modelem, widokiem i sterownikami do bazy danych. W oknach kreatora servletu wykonaj następujące zmiany:

- w pierwszym oknie ustaw nazwę klasy servletu na `servletListaKsiazek`,
- w drugim oknie zmień mapowanie adresu URL (URL mappings | Edit) na `listaKsiazek`, pomijamy w nazwie część `servlet` ponieważ nie musi być widoczna dla użytkowników aplikacji.

12. Uzupełnij kod metody `doGet`, który odczyta dane książek i przekaże je do widoku czyli strony .jsp. Skorzystaj z obiektów klasy `daoKsiazki` oraz `HttpSession`.

Wygeneruj obiekt klasy `daoKsiazki`, wywołaj jego metodę `listaKsiazek`, która generuje dynamiczną tablicę z danymi `ArrayList`, zapamiętaj tą tablicę w zmiennej. Kod może mieć postać:

```
daoKsiazki dao=new daoKsiazki();
ArrayList<modelKsiazka> tk= dao.listaKsiazek();
```

Wygeneruj obiekt sesji do przekazywania danych do widoku czyli strony listaKsiazek.jsp.

```
HttpSession sesja=request.getSession();
```

Przekaż całą tablicę z danymi jako atrybut sesji, można nadać mu taką samą lub inną nazwę:

```
sesja.setAttribute("tk", tk);
```

Wygeneruj obiekt dyspozytora żądania klasy `RequestDispatcher` dla odpowiedniego adresu URL i przekaż obiekty żądania i odpowiedzi korzystając z jego metody `forward`.

```
String nextURL="/listaKsiazek.jsp";
RequestDispatcher rd=getServletContext().getRequestDispatcher(nextURL);
rd.forward(request, response);
```

Uzupełnij brakujące odwołania do bibliotek.

Dodaj adnotację z nazwą odwołania do servletu `@WebServlet("/listaKsiazek")` i usuń plik web.xml jeśli został utworzony.

Widok strony czyli JSP, HTML i CSS oraz znaczniki JSTL i język EL

13. W folderze `webapp` projektu wygeneruj plik listaKsiazek.jsp korzystając z szablonu HTML5. Ponieważ będziemy w niej korzystać ze znaczników JSTL dołącz do pliku pakiet z biblioteką tych znaczników:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Prefix definiuje przedrostek, który podajemy przed nazwą każdego znacznika z tej biblioteki. Umożliwia to dołączanie znaczników z wielu różnych bibliotek bez konieczności sprawdzania czy ich nazwy się nie powtarzają. Jest to standardowa procedura w języku XML, z którego tu właśnie korzystamy.

Korzystając za znaczników JSTL Wyświetl liczbę książek oraz ich tytuły w kolejnych akapitach. Kod może mieć postać:

```
<p>Liczba książek: ${tk.size()}</p>
<h2>Lista książek</h2>
<c:forEach var="el" items="${tk}">
      <p>${el.tytul}, ${el.wyd.nazwa}</p>
</c:forEach>
```

Odwołujemy się do atrybutu `tk` przekazanego z servletu, jest to tablica `ArrayList`. W pierwszym akapicie wywołujemy metodę `size()` obiektu `ArrayList`, która podaje w wyniku liczbę elementów tablicy.

Biblioteka JSTL oraz język wyrażeń EL umożliwiają dostęp do obiektów, takich jak nasza tablica oraz wykonywanie instrukcji nazywanych tutaj akcjami. Są to m.in. akcje pętli, iteracji i warunkowe.

Znacznik `forEach` z biblioteki `core` w kodzie powyżej jest odpowiednikiem pętli `foreach`, czyli umożliwia dostęp do elementów struktury złożonej np. tablicy. Wpisujemy znacznik `forEach` z

prefiksem biblioteki `core` czyli c w postaci `<c:forEach> </c:forEach>`. Znacznik może mieć kilka atrybutów/parametrów, tutaj wykorzystane są dwa:

- `items` do przekazania do przetwarzania struktury złożonej, w naszym przypadku podajemy tam odwołanie do listy tablicowej `tk`,
- `var` definiującej nazwę zmiennej elementu tej tablicy.

Podane ustawienie umożliwia przetworzenie wszystkich elementów tablicy `tk` po kolei. Wewnątrz, pomiędzy znacznikiem początku i końca `forEach` wpisujemy fragment do powtarzania. Tutaj jest to akapit html, w którym odwołujemy się do zdefiniowanego elementu tablicy czyli `el`. Jak pamiętamy elementami tablicy są obiekty klasy `modelKsiazka`, które posiadają właściwość `tytul`, stąd zapis `el.tytul`.

Sprawdź działanie napisanego fragmentu aplikacji uruchamiając `servletListaKsiazek.java` na serwerze Tomcat.

Strona główna i menu dla aplikacji

14. Skonstruuj stronę główną aplikacji razem z menu. Ponieważ menu będzie dołączane do każdej strony, najlepiej umieścić je w osobnym pliku. W przypadku bardziej złożonych witryn stronę główną można złożyć z kilku części np. nagłówka, stopki, menu i zawartości zapisanych w osobnych plikach.

Wygeneruj w odpowiednim miejscu plik o nazwie index.jsp i umieść w nim następujący kod:

```
<c:import url="/menu"/>
<div id="zawartosc">
<p>Witamy w naszej aplikacji, która ma pomagać w wyborze książek. Obecnie możesz tu znaleźć informacje na temat ciekawych książek z informatyki.</p>
</div>
```

Ponieważ korzystamy tu z biblioteki JSTL, na początku pliku trzeba umieścić znaczniki XML dołączające bibliotekę jstl-core analogicznie, jak w pliku `listaKsiazek.jsp`. Można również umieścić ten kod na stałe w szablonach stron JSP środowiska Eclipse.

Znacznik import umożliwia włożenie pliku do pliku, w naszym przypadku będzie to menu, które za chwilę skonstruujemy. Za menu został wydzielony blok `div` z zawartością (`div` o identyfikatorze `zawartosc`). Wkładamy tam zwykły akapit z tekstem informującym o witrynie.

15. Utwórz menu aplikacji w postaci pliku `menu.jsp` oraz `servletMenu.java`. Umieść je w odpowiednich katalogach aplikacji.

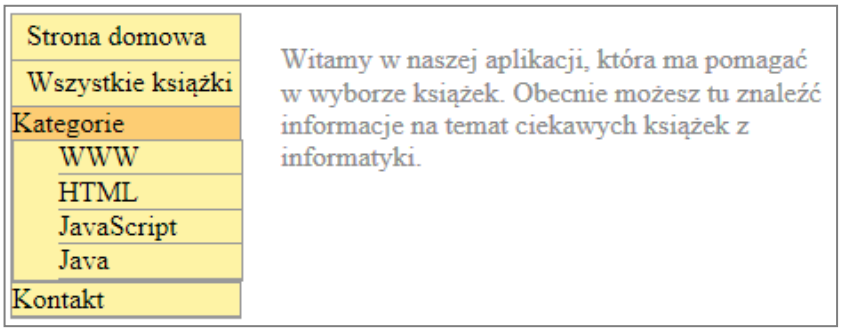
W servlecie ustaw jego adres na `"/menu"`: (`@WebServlet("/menu")`).

W metodzie `doGet` servletu umieść kod generujący tablicę `ArrayList` z obiektami klasy `modelKategoria` korzystając z odpowiedniej metody obiektu klasy `daoKsiazki`, analogicznie jak w przypadku listy książek. Przekaż tablicę jako atrybut żądania URL w sesji. Na końcu przekieruj żądanie do pliku `menu.jsp`.

W pliku `menu.jsp` usuń strukturę dokumentu HTML i umieść tylko `div` o identyfikatorze `menu` oraz listę nienumerowaną z odnośnikami w postaci opcji menu. Jedną z opcji opisz jako Kategorie i umieść tam listę kategorii odebraną z servletu w postaci tablicy. Każdą kategorię zapisz w postaci znacznika `li` podlisty `ul`. Ogólną strukturę przedstawiono w kodzie poniżej.

Katedra Systemów Informacyjnych
Uniwersytet Morski w Gdyni

```
<div id="menu">
<ul>
      <li><a href="index.jsp">Strona domowa</a></li>
      <li><a href="ListaKsiazek">Wszystkie książki</a></li>
      <li class="podmenu">Kategorie
            <ul>
            <!--  tutaj umieść elementy li z nazwami kategorii-->
            </ul>
      </li>
      <li>Kontakt</li>
</ul>
</div>
```

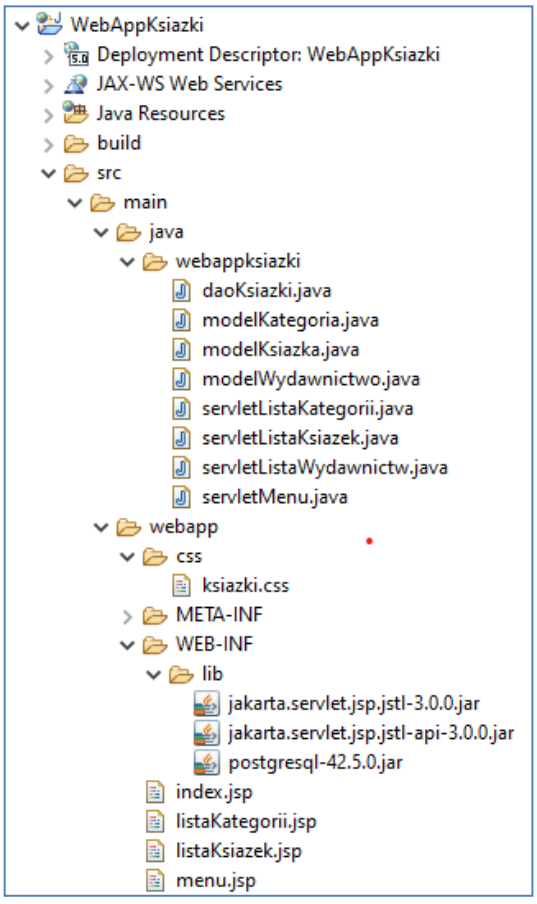


W części `head` dokumentu index.jsp umieść znacznik `link` z poleceniem dołączenia do pliku arkusza stylów zapisanego w pliku `ksiazki.css`. Plik ten został dołączony do materiałów. W projekcie zwykle umieszcza się go w folderze `css` utworzonym jako podkatalog `webapp` (patrz struktura projektu na obrazku poniżej). Znacznik w pliku `.jsp` powinien więc mieć postać:

```
<link rel="stylesheet" href="css/ksiazki.css">
```

Przykład strony głównej aplikacji razem z menu widać na obrazku powyżej.

Struktura projektu powinna być teraz taka jak na obrazku.



Po uzupełnieniu kodu przetestuj działanie aplikacji uruchamiając plik `index.jsp` na serwerze Tomcat. Zauważ, że po wybraniu opcji Wszystkie książki menu znika – nie ma go na stronie z listą książek. Dodaj menu oraz div zawartosc do strony `listaKsiazek.jsp` i ponownie przetestuj działanie aplikacji.

16. Wygeneruj stronę `kontakt.jsp`. Umieść na niej wymyślony adres własnej firmy oraz listę wydawnictw, z którymi współpracuje utworzoną na podstawie informacji z bazy danych.

17. Uzupełnij aplikację o możliwość wyświetlania książek z wybranej kategorii.

Do klasy `daoKsiazki` dodaj metodę np. `listaKsiazekKategorii(int id)`, która pobierze

z bazy dane książek z wybranej kategorii. Kategorię podaj w postaci identyfikatora (liczby całkowitej) jako parametr metody.

W pliku `menu.jsp` zamień nazwy kategorii na linki z parametrem w postaci identyfikatora kategorii.

W servlecie `servletListaKsiazek` odbierz parametr żądania w postaci `id` kategorii korzystając z metody `getParameter` obiektu `request`. Następnie, wywoła dodaną do `daoKsiazki` metodę `listaKsiazekKategorii` podając odebrane `id` jako jej parametrem. Jeśli parametr nie istnieje, tzn. `getParameter` oddaje w wyniku `null` wyświetl listę wszystkich książek.

Sprawdź działanie aplikacji po zmianach.

18. Zmodyfikuj działanie aplikacji generując strony do usuwania i dopisywania książek, kategorii i wydawnictw do bazy.

19. (Zadanie dodatkowe) Zmodyfikuj działanie aplikacji dodając do tabeli ksiazki kolumnę rok wydania oraz zamieniając kolumnę adres w tabeli wydawnictwo na miasto i dodając nową kolumnę państwo.

20. (Zadanie dodatkowe) Zmodyfikuj działanie aplikacji według własnych pomysłów wykorzystując technologie Java.

21. Gotową aplikację uruchom na swoim zdalnym szkolnym serwerze Tomcat.

Jeśli pracowałeś na lokalnej bazie danych połącz się z bazą danych na szkolnym serwerze foka i na swoim koncie załóż schemat ksiazki i tabele analogicznie jak na serwerze lokalnym. Konieczne będzie połączenie szyfrowane przez tunel.

W aplikacji zmień dane dotyczące połączenia z bazą danych na szkolną bazę danych i przetestuj działanie aplikacji.

Wygeneruj plik **.war** aplikacji i umieść go na swoim szkolnym serwerze Tomcat. Sprawdź działanie aplikacji.

W **sprawozdaniu** w systemie **Sprawer** prześlij:

- **plik .zip** z **projektem (tylko katalog src)**,
- **link do aplikacji** uruchomionej na szkolnym serwerze foka.