prepared by: pawel@kasprowski.pl

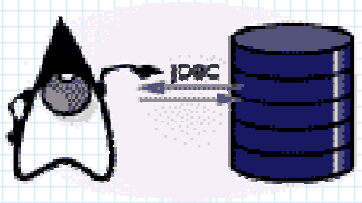
Aplikacje Internetowe

Bazy danych, JSTL

JDBC

JDBC API pozwala na:

- Ustalenie połączenia z bazą
- Wysyłanie poleceń SQL
- Przetwarzanie rezultatów



Sposób działania JDBC

Załaduj sterownik

```
Class.forName( DriverClassName);
```

Połącz się ze źródłem danych

```
Connection con = DriverManager.getConnection(
URL, Username, Password );
```

Stwórz polecenie

```
Statement stmt = con.createStatement();
```

Zapytaj bazę danych

```
ResultSet result = stmt.executeQuery("SELECT
* FROM table1");
```

Pierwszy przykład JDBC

```
import java.sql.*;
class DBExample1{
public static void main(String[] args){
  try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con = DriverManager.getConnection("jdbc:odbc:uran","lab","lab");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from pracownicy");
    while(rs.next())
      System.out.print(rs.getString(1)+"|"+rs.getString(4)+"\n");
    con.close();
    }
  catch(SQLException ec) { System.err.println(ec.getMessage()); }
  catch(ClassNotFoundException ex) {System.err.println("Cannot find driver.");}
  }
}
```

DBExample1.java

Ładowanie sterownika

- Zwykle sterownik typu 4
- Ściągnięcie pliku jar ze sterownikiem
- Umieszczenie go na ścieżce
- Zmiana nazwy sterownika i url'a w programie

Przykłady sterowników

- SQL Server (http://msdn.microsoft.com/data/ref/jdbc)

```
java -classpath ".;mssqlserver.jar" %1
```

- MySQL (http://www.mysql.com/products/connector/j/)

```
java -classpath ".;mysql-connector-java.jar" %1
```

- PostgreSQL (http://jdbc.postgresql.org)

```
java -classpath ".;postgres-8.2dev-503.jdbc2ee.jar" %1
```

run.bat

Zmiana sterownika

```
import java.sql.*;
class DBExample1{
public static void main(String[] args){
  try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con =
        DriverManager.getConnection("jdbc:odbc:uran","lab","lab");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from pracownicy");
    while(rs.next())
      System.out.print(rs.getString(1)+"|"+rs.getString(4)+"\n");
    con.close();
    }
  catch(SQLException ec) { System.err.println(ec.getMessage()); }
  catch(ClassNotFoundException ex) {System.err.println("Cannot find driver.");}
  }
}
```

DBExample1.java

Zmiana sterownika

```
import java.sql.*;
class DBExample1{
public static void main(String[] args){
  try{
    Class.forName("com.mysql.jdbc.Driver");
    Connection con =
        DriverManager.getConnection("jdbc:mysql://127.0.0.1/baza","lab","lab");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from pracownicy");
    while(rs.next())
      System.out.print(rs.getString(1)+"|"+rs.getString(4)+"\n");
    con.close();
    }
  catch(SQLException ec) { System.err.println(ec.getMessage()); }
  catch(ClassNotFoundException ex) {System.err.println("Cannot find driver.");}
  }
}
```

DBExample1.java

Użycie w AI

- Można bezpośrednio w skryplecie
- Konieczny sterownik
- Wystarczy wgrać sterownik do WEB-INF/lib
- Można go również umieścić w katalogu lib serwera

Stworzenie tabeli

```
CREATE TABLE car (
  idc int NOT NULL auto_increment,
  make varchar(50),
  model varchar(50),
  regnum varchar(8) NOT NULL,
  price double default NULL,
  PRIMARY KEY  (`idc`)
);
```

Wstawienie danych

```
INSERT INTO car
   VALUES(1,'Fiat','Brava','SH01089',20);
INSERT INTO car
   VALUES(2,'Fiat','Punto','FA34589',20);
INSERT INTO car
   VALUES(3,'Ford','Fiesta','AW29034',20);
INSERT INTO car
   VALUES(4,'Ford','Focus','KA34567',20);
INSERT INTO car
   VALUES(5,'Opel','Astra','WA10293',20);
```

Użycie bazy danych

```
public String getCars() {
  try{
    Class.forName("com.mysql.jdbc.Driver");
    Connection con = DriverManager.getConnection(
                        "jdbc:mysql://127.0.0.1/baza","user","pass");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from car");
    while(rs.next())  txt += rs.getString(1)+"|"+rs.getString(4);
    con.close();
    }catch(SQLException ec) {ec.printStackTrace();}
  catch(ClassNotFoundException ex) {ex.printStackTrace();}
  }
```

Prawidłowe zamykanie połączenia

```
public String getCars() {
Connection con = null;
 try{ …
    con = DriverManager.getConnection(
                    "jdbc:mysql://127.0.0.1/baza","user","pass");
    …
}catch(SQLException ec) {ec.printStackTrace();}
finally {
    try{con.close();}catch(SQLException ec) {ec.printStackTrace();}
}
}
```

DataSource

- Zalecany zamiennik dla DriverManager
- Posiada właściwości, które można zmieniać setterami i getterami
  - każdy sterownik może mieć inne nazwy właściwości!
- Może być wpisywany do JNDI
- Może wspomagać przydział połączeń (Connection Pooling)

Zmiana w programie

- Zamiast linii:

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager
        .getConnection(" jdbc:mysql://127.0.0.1/baza ","lab","lab");
```

- Wprowadzamy (dla MySQL):

```
com.mysql.jdbc.jdbc2.optional.MysqlDataSource ds =
        new com.mysql.jdbc.jdbc2.optional.MysqlDataSource();
ds.setUser("lab");
ds.setPassword("lab");
ds.setUrl(" jdbc:mysql://127.0.0.1/baza ");
Connection con = ds.getConnection();
```

Problemy

- Utrzymywanie połączenia
- Wielowątkowość aplikacji

Utrzymywanie połączenia

- Nawiązanie połączenia jest czasochłonne – lepiej więc raz nawiązane połączenie używać do kolejnych zapytań
- Problem: jak przechować informację o połączeniu, jeśli jest ono używane w różnych obiektach
- Rozwiązanie: w każdym miejscu aplikacji dostęp do obiektu DataSource z którego można pobrać połączenie (obiekt Connection)

Wielodostęp do danych

- Aplikacje internetowe – wielu jednoczesnych użytkowników tej samej aplikacji
- Nie ma potrzeby nawiązywania połączenia dla każdego - każdy z nich może używać tego samego połączenia
- Problem: co jeśli dwóch użytkowników potrzebuje danych w tym samym czasie?
- Rozwiązanie najprostsze: blokada połączenia na czas używania
- Rozwiązanie lepsze: stworzenie wielu połączeń

Connection pooling

- Jeśli jest dużo klientów – można otworzyć kilka równoległych połączeń z bazą
- Konieczny obiekt, który przechowuje pulę połączeń i udostępnia klientom
- Gdy klient zwalnia połączenie - wraca ono do puli
- Dobra wiadomość: Apache Tomcat posiada już coś takiego – DBCP (Database Connection Pool)
- Wystarczy skonfigurować źródło danych w XMLu

Użycie DBCP

- Stosuje connection pooling
- Używa sterownika, który podamy
  - jest niezależny od serwera bazy danych
- Konfiguracja połączenia w Context
  - conf/server.xml
  - conf/Catalina/localhost/aplikacja.xml
  - aplikacja/META-INF/context.xml
- W aplikacji używamy JNDI lookup

Konfiguracja połączenia

```
<Context path="/web1" reloadable="true">
    <Resource name="jdbc/mydb" auth="Container"
    type="javax.sql.DataSource"
    maxActive="100" maxIdle="30" maxWait="10000"
    username="myuser" password="mypass"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/mydb?
                                        autoReconnect=true"/>
</Context>
```

Użycie DBCP w aplikacji

```
public Connection getConnection() {
  try{
     Context initContext = new InitialContext();
     DataSource ds =(DataSource)initContext.lookup(
                                        "java:/comp/env/jdbc/mydb");
     return ds.getConnection();
  }catch(SQLException ec) { ... }
   catch(NamingException ne) { ...}
  return null;
  }
```

Zamknięcie tak uzyskanego połączenia zwraca je do puli dostępnych

Użycie bazy danych

```
public String getCars() {
  try{
    Connection con = getConnection();
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from car");
    while(rs.next())  txt += rs.getString(1)+"|"+rs.getString(4);
    con.close(); // nie zamyka połączenia tylko oddaje
    }catch(SQLException ec) {ec.printStackTrace();}
}
```

Użycie bazy danych

```
public String getCars() {
  Connection con = null;
  try{
    con = getConnection();
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from car");
    while(rs.next())  txt += rs.getString(1)+"|"+rs.getString(4);
}catch(SQLException ec) {ec.printStackTrace();}
finally{ try{con.close();}catch(SQLException ex) {…}
}
```

Podsumowanie

- Wystarczy skonfigurować DBCP
  - Dodanie elementu <Resource>
- Dołączyć do aplikacji sterownik
- Wywołać w programie
  - Context initContext = **new** InitialContext();
  - DataSource dataSource = (DataSource)initContext.lookup("java:/comp/env/jdbc/mydb");
- Dla obiektu dataSource wywołać:
  - Connection con = dataSource.getConnection();
  - ... tu kod korzystający z bazy…
  - Con.close();

Problemy ze skrypletami

- Mieszanie kodu HTML, Java i JavaScript powoduje nieczytelność
- Logika jest rozdzielona na różne strony
- Debugowanie jest trudne
- W rozbudowanych aplikacjach:
  - HTML developers – interfejs użytkownika
  - Java developers – logika aplikacji

Przykład strony JSP

```
<body>

<h1>Edycja odczynnika <%=request.getParameter("cas")%></h1>
<input type=button value='< powrót do karty'
    onClick='document.location="view.jsp?cas=<%=request.getParameter("cas")%>"'>
<font size=11>
<%
cas.CasEdit casedit=new cas.CasEdit(request.getParameter("cas"));
if(request.getParameter("delete")!=null) {
  casedit.delete(request.getParameter("oid"));
  %><SCRIPT language='JavaScript'>
    document.location='edit.jsp?cas=<%=request.getParameter("cas")%>&x=x';</SCRIPT><%
  }
if(request.getParameter("update")!=null && request.getParameter("sbm")!=null) {
  casedit.update(request.getParameter("oid"),request.getParameter("type"),
                request.getParameter("newdata"),request.getParameter("newdata2"),session);
  out.println(casedit.getTable("",request.getParameter("newtype"),session));
  }
else {
  if(request.getParameter("anuluj")!=null)
    out.println(casedit.getTable("",request.getParameter("newtype"),session));
  else

    out.println(casedit.getTable(request.getParameter("oid"),request.getParameter("newtype"),
    session));
}
%>
</body>
```

Użycie tagów

- Cel: uniknięcie kodu w Javie na stronie HTML
- Otwieranie i zamykanie tagu
  - <some.tag>
  - </some.tag>
- Tag pojedynczy
  - <some.tag/>
- Tagi mogą być:
  - predefinowane w specyfikacji JSP: <jsp:...>
  - ładowane z bibliotek

Tagi predefinowane

- <jsp:include page="page.jsp"/>
- <jsp:forward page="page.jsp"/>
- <jsp:param name="..." value="..."/>
- <jsp:useBean>
- <jsp:setProperty>
- <jsp:getProperty>

Biblioteki tagów

- Przykładowe biblioteki
  - http://java.sun.com/products/jsp/jstl/
  - http://jakarta.apache.org/taglibs/
  - http://jsptags.com/tags/
- Włączanie biblioteki do aplikacji
  - WEB-INF/lib/xyz.jar
  - WEB-INF/*.tld
- Załadowanie definicji (*.tld) na stronie JSP:
  - <% @taglib uri="<address>" prefix="xyz" %>
- Użycie tagu:
  - <xyz:newtag>...</xyz:newtag>

JSTL Java Standard Tag Library

- Biblioteka Sun'a
- Różne implementacje
  - najpopularniejsza: jakarta-taglibs
- Usprawnienia na stronach JSP bez użycia skrypletów (kodu w Javie)
- Ładowanie JSTL (jakarta):
  - wgrać pliki standard.jar i jstl.jar do katalogu WEB-INF/lib

Elementy JSTL

- core <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
  - Zmienne, kontrola przepływu
- fmt <%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
  - Formatowanie dat, lokalizacja
- sql <%@taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>
  - Obsługa baz danych
- XML <%@taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x"%>
  - Operacja na dokumentach XML
- functions <%@taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn"%>
  - Funkcje konwertujące i inne

Expression Language

- ${expression}
- Uproszczony dostęp do:
  - standardowych obiektów i właściwości
  - beanów
  - obiektów typu Map, List czy Array
- Włączone do JSP 2.0

Core JSTL

- c:set
- c:out
- c:if
- c:forEach
- c:choose, c:when

c:set i c:out

- ustawianie i czytanie

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<body>
<c:set var="name" value="Pawel"/>
...
My name is ${name}
</body>

</html>
```

c:if tag

- <c:if test="${some test}">
  - jakiś kod
- </c:if>

```
<c:if test="${x>5}">
x is greater than 5!
</c:if>
```

Zakresy widoczności obiektów

- page
  - tylko na stronie
- request
  - dla następnej strony przetwarzającej parametry z aktualnej
- session
  - dla wszystkich stron w ramach sesji
- application
  - dla całej aplikacji

Zakres (scope) zmiennych

- Domyślnie: request (inne: session, application, page)

```
<c:if test="${empty sessionScope.x}">
    <c:set var="x" value="1" scope="session"/>
</c:if>

<p>Current value: ${x}</p>

<c:set var="x" value="${x + 1}" scope="session"/>
```

c:choose tag

```
<p>
<c:choose>
    <c:when test="${x> 0 and x<= 5}">
            "x" value in range (0, 5>
    </c:when>
    <c:when test="${x> 5 and x <= 10}">
            "x" value in range (5, 10>
    </c:when>
    <c:otherwise>
            "x" value over 10 or below 0!
    </c:otherwise>
</c:choose>
</p>
```

c:foreach tag

Kod z użyciem JSTL:

```
<ul>
        <c:forEach var="i" begin="1" end="10">
                <li>${i}
        </c:forEach>
</ul>
```

To samo jako scriplet:

```
<ul>
<%
        for(int i=1; i<=10; i++) {
                out.println("<LI>"+i);
        }
%>
</ul>
```

Inne przykłady c:forEach

- <c:forEach var="i" begin="0" end="1000" step="100">
- <c:forEach var="word" items="${words}">
- <c:forEach var="country" items="Australia,Canada,Japan,Philippines,USA">
- <c:forTokens var="color" items="(red (orange) yellow)(green)((blue) violet)" delims="()">

Moduł Functions

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

<c:set var="str" value="Here we have a simple text"/>
<p>Text: ${str}
<p>To upper case: ${fn:toUpperCase(str)}
<p>Text length: ${fn:length(str)}</p>
<p>Does it contain "have" word?: ${fn:contains(str,"have")}</p>
<p>Number of words (space separated): ${fn:length(fn:split(str," "))}</p>
<p>Letter "s" in at position: ${fn:indexOf(str, "s")}</p>
<p>Replace "s" with "%": ${fn:replace(str, "s", "#")}</p>
```

Funkcje formatujące

- <fmt:formatNumber>
- <fmt:parseNumber>
- <fmt:formatDate>
- <fmt:parseDate>
- <fmt:setLocale>
- <fmt:bundle>, <fmt:setBundle>
- <fmt:message>
- <fmt:param>

Funkcje modułu sql

```
<sql:setDataSource
driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/mydb"
user="" password=""/>

<sql:query var="results" sql="SELECT * FROM car"/>

<c:forEach var="row" items="${results.rows}">
    ${row.make}  ${row.model} <br/>
</c:forEach>
```

Uniwersalny showRS

```
<table border="1"> <tr>
 <%-- Get the column names for the header of the table --%>
 <c:forEach var="columnName" items="${results.columnNames}">
  <th>${columnName}</th>
 </c:forEach>
 <%-- Get the value of each column while iterating over rows --%>
 <c:forEach var="row" items="${results.rows}">
  <tr>
   <c:forEach var="column" items="${row}">
    <td>${column}</td>
   </c:forEach>
 </c:forEach>
</table>
```

Podsumowanie

- W JSP można używać tagów
- Jest kilka użytecznych bibliotek tagów
- Główna idea: bez kodu w Javie na stronie JSP!
- Problem:
  - obsługa wyjątków, kontrola parametrów
- Krok dalej:
  - sprawdzenie parametrów i przygotowanie danych w klasach Javy
  - wyświetlanie przygotowanych danych za pomocą JSP