Aplikacje Internetowe

Zakres przedmiotu

- Tworzenie aplikacji internetowych
  - w języku Java
  - w środowisku Eclipse
  - z użyciem serwera Apache Tomcat
- Materiały
  - przede wszystkim Internet
  - książki na temat JSP (Java Server Pages), J2EE i ogólnie języka Java
  - oczywiście: www.kasprowski.pl ☺

Plan ramowy

- Podstawy JSP
- Użycie tagów z biblioteki JSTL
- Tworzenie servletów
- Architektura MVC2
- Komunikacja z bazą danych: JDBC, Hibernate, JPA
- Cel:
  - nabycie umiejętności tworzenia aplikacji internetowej a nie tylko zbioru dynamicznych stron WWW

Aplikacja internetowa

Definicja:

- Aplikacja uruchamiana na serwerze WWW komunikująca się z użytkownikiem za pomocą przeglądarki internetowej

Zalety aplikacji internetowej

- brak konieczności instalacji dodatkowego oprogramowania na komputerach klientów a co za tym idzie mniejsze wymagania co do jakości tych komputerów
- łatwość aktualizacji aplikacji – w przypadku zmiany wersji nie trzeba aktualizować plików na każdym komputerze a wystarczy aktualizacja na serwerze
- potencjalna możliwość zdalnego korzystania z aplikacji

Wady aplikacji internetowej

- uboższy interfejs użytkownika (nie wszystko da się zrobić w przeglądarce)
- większy nakład pracy projektowej, a co za tym idzie nieco większy koszt wykonania
- większe obciążenie serwera (kosztem mniejszego obciążenia komputerów użytkowników)

Dynamiczna generacja stron

- CGI – Common Gateway Interface
- Wejście: adres, parametry
- Wyjście: tekst strony html
- Problem: jak przekazać parametry?

Metody wysyłania danych

- Metoda GET

Odpowiednik:

http://xyz.pl/start?imie=Adam&nazwisko=Kowalski

- Metoda POST

Parametry wewnątrz wysyłanego pakietu – brak ograniczeń na wielkość parametrów

Aplikacja CGI

- Konieczność generacji całej strony
- Program drukujący tekst na standardowym wyjściu
- Niewygodne (zwłaszcza, gdy elementy dynamiczne zajmują niewielką część strony)
- Wygodniej stworzyć stronę HTML i tylko ***wstawić*** elementy generowane dynamicznie

Dynamiczne wstawki do tekstu html

Trzy najważniejsze rozwiązania:

- PHP – najpopularniejsze, własny język oparty początkowo na języku Perl
- ASP – propozycja Microsoft, wstawki w VisualBasic'u
- JSP – wstawki w języku Java

JSP – Java Server Pages

- Tworzenie dokumentów html ze wstawkami w Javie
- Dokumenty przekształcane są w servlety – klasy generujące html

Serwery JSP

- Przykładowe serwery:
  - Blazix
  - Tomcat
  - Websphere
- Standardowa funcjonalność serwera WWW
- Kompilacja JSP

Najprostszy JSP

```
<HTML>
<BODY>
Hello!  The time is now <%= new java.util.Date() %>
</BODY>
</HTML>
```

Wstawki w kodzie (scriplet'y)

```
<HTML>
<BODY>
<%
    System.out.println( "Evaluating date now");
    java.util.Date date = new java.util.Date();
%>
...
Hello!  The time is now <%= date %>
</BODY>
</HTML>
```

Scriplet tworzący tekst

```
<HTML>
<BODY>
<%
    System.out.println( "Evaluating date now");
    java.util.Date date = new java.util.Date();
%>
...
Hello!  The time is now
<%
    out.write( String.valueOf( date ));
%>
</BODY>
</HTML>
```

Obiekty dostępne w jsp

- out
  - out.write(...)
- request
  - request.getParameter(...)
  - request.getRemoteHost()
- response
  - response.sendRedirect(...)

Odczytywanie danych

- Obiekt request, metoda getParameter(„nazwa”)

```
<h1>Witaj <%=request.getParameter(”imie”)%></h1>
<p>
<%
String imie = (String)request.getParameter(”imie”);
if(imie.equals(”Paweł”))
    out.print(”Ja też mam na imię Paweł!”);
%>
</p>
```

Możliwości JSP

- Mieszanie kodu i html'a

```
<TABLE BORDER=2>
    <%
    for ( int i = 0; i < 10; i++ ) {
       %>
       <TR>
                    <TD>Numer</TD>
                    <TD><%= i+1%></TD>
            </TR>
            <%
    }
    %>
</TABLE>
```

Możliwości JSP

- Mieszanie kodu i html'a

```
<TABLE BORDER=2>
    <%
    for ( int i = 0; i < 10; i++ ) {
       %>
       <TR>
                    <TD>Numer</TD>
                    <TD><%= i+1%></TD>
            </TR>
            <%
    }
    %>
</TABLE>
```

Numer	1
Numer	2
Numer	3
Numer	4
Numer	5
Numer	6

Dyrektywa page

- Użycie bibliotek Java

```
<HTML>
<BODY>
<%
    System.out.println( "Evaluating date now" );
    java.util.Date date = new java.util.Date();
%>
Hello! The time is now <%= date %>
</BODY>
</HTML>
```

Dyrektywa page

- Użycie bibliotek Java

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

<HTML>
<BODY>
<%
    System.out.println( "Evaluating date now" );
    Date date = new Date();
%>
Hello! The time is now <%= date %>
</BODY>
</HTML>
```

Dyrektywa page

- Użycie bibliotek Java

```
<%@ page import="java.util.*,java.text.*" %>

<HTML>
<BODY>
<%
    System.out.println( "Evaluating date now" );
    Date date = new Date();
%>
Hello! The time is now <%= date %>
</BODY>
</HTML>
```

Pierwsza aplikacja

- Kalkulator kredytowy
- Obliczanie raty kredytu
- Część prezentacyjna (JSP)
- Część "logiki biznesowej" (java)

Kalkulator kredytowy

- Obliczenie wysokości miesięcznej raty spłaty kredytu na podstawie jego kwoty, oprocentowania rocznego i liczby rat miesięcznych

$$rata = kwota * \frac{oproc/12}{1 - \frac{1}{1+(oproc/12)^{liczba\_rat}}}$$

Parametryzacja

- Dokument index.jsp zawiera formularz w którym użytkownik wpisuje parametry kredytu
  - kwota
  - na ile lat
- Formularz wysyłany jest do dokumentu result.jsp, który oblicza ratę

Formularz

```
<form action="result.jsp">
    kwota: <input type="text" name="kwota"><br/>
    ile lat: <input type="text" name="lat"><br/>
    <input type="submit"/>
</form>
```

Plik result.jsp

- Odczytanie parametrów:

```
String strKwota = request.getParameter("kwota");
String strLat = request.getParameter("lat");
```

- Konwersja na liczbę (w Javie niezbędna!):

```
double kwota = Double.parseDouble(strKwota);
int lat = Integer.parseInteger(strLat);
```

- Wykonanie obliczenia:

```
double rata = kwota * (procent/12)/
    (1-(1/Math.pow(1.0+procent/12,lat*12)));
```

Nasze narzędzie pracy

Historia Eclipse

- Założyciele (2001): Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft, Webgain (Board of Stewards)
- Kolejni członkowie to m.in.: HP, Oracle, SAP, Ericsson, Intel
- Licencja CPL (Common Public License)
- Aktualna wersja: 3.4 (Ganymede)
- Podstawowe narzędzie pracy w wielu firmach programistycznych!

Rozszerzalność

- Eclipse jest narzędziem uniwersalnym:
  - Eclipse IDE for Java Developers
  - Eclipse IDE for C/C++ Developers
  - PHPEclipse
- Istnieją dedykowane wersje Eclipse'a (komercyjne):
  - Red Hat Developer Studio
  - IBM Data Studio
  - Borland JBuilder
- Są to Eclipse'y rozszerzone o własne wtyczki firmy

Przykładowe wtyczki

- Amateras HTMLEditor
- Sysdeo TomcatPlugin
- DBViewer Plugin
- AzzuriClay – modelowanie baz danych
- HibernateTools – dostęp do baz danych
- Omondo – EclipseUML
- Visual Editor – tworzenie GUI
- PHPEclipse
- Tworzone z użyciem: Plug-In Development Environment

Instalacja wtyczek

- Help->Software Updates
- Przegranie pakietu do katalogu plugins

Tworzenie środowiska

- Instalacja Java JDK
- Instalacja Apache Tomcat (darmowy)
  - katalog główny: webapps/ROOT
  - aplikacje: webapps/<nazwa_aplikacji>
  - konfiguracja: conf/server.xml
- Instalacja Eclipse
- Instalacja plug-in'u Sysdeo do Eclipse'a

Tworzenie projektu Eclipse

- New Tomcat project: kredyty
- Katalog /webapp
- Plik index.jsp
- Start Tomcat'a
- http://localhost:8080/kredyty
- Stworzenie formularza startowego (index.jsp)
- Stworzenie result.jsp

Wady rozwiązania

- Oprocentowanie jest określone na stałe wewnątrz pliku HTML – zmiany muszą być dokonywane bezpośrednio w nim
- Sposób obliczenia raty także jest w pliku HTML
- Zmiana wyglądu strony – zmiana w pliku HTML
- A więc: łatwo popełnić błąd, który spowoduje, że aplikacja przestanie działać prawidłowo
- Dalsza rozbudowa aplikacji (np. ściąganie oprocentowania z bazy danych) bardzo utrudnione

Rozwiązanie dwuwarstwowe

- Rozdzielenie części „prezentacyjnej” od części obliczeniowej
- Programista (grafik) tworzący interfejs WWW nie musi znać wzoru na obliczanie kredytu
- Programista tworzący kod obliczający kredyt nie musi wiedzieć gdzie będzie on używany

Klasa Kredyt

```
class Kredyt {
  double procent = 0.05;
  double kwota;
  double lat;

  public Kredyt(double k, double l) {
    kwota = k;
    lat = l;
  }

  public double getRata() {
    double rata = kwota * (procent/12)/
                  (1-(1/Math.pow(1.0+procent/12,lat*12)));
    return rata;
  }
}
```

Kredyt.java

Klasa Kredyt

```
package pl.kasprowski.utils;

public class Kredyt {
  double procent = 0.05;
  double kwota;
  double lat;

  public Kredyt(double k, double l) {
    kwota = k;
    lat = l;
  }

  public double getRata() {
    double rata = kwota * (procent/12)/
                  (1-(1/Math.pow(1.0+procent/12,lat*12)));
    return rata;
  }
}
```

Kredyt.java

Nowy result.jsp

```
<%
Kredyt kredyt = new Kredyt(
        Double.parseDouble(request.getParameter("kwota")),
        Double.parseDouble(request.getParameter("lat"))
        );
    out.write("<p>Rata wynosi: "+kredyt.getRata()+"</p>");
%>
```

Nowy result.jsp [2]

```
<%
Kredyt kredyt = new Kredyt(
        Double.parseDouble(request.getParameter("kwota"),
        Double.parseDouble(request.getParameter("lat"))
        );
%>
<p>Rata wynosi: <%=kredyt.getRata()%></p>
```

Dodanie konstruktora

```
package pl.kasprowski.utils;
public class Kredyt {
  double procent = 0.05;
  double kwota;
  double lat;

  public Kredyt(double k, double l) {
    kwota = k;
    lat = l;
  }

  public Kredyt(String k, String l) {
    kwota = Double.parseDouble(k);
    lat = Double.parseDouble(l);
  }

  public double getRata() {
    double rata = kwota * (procent/12)/
                (1-(1/Math.pow(1.0+procent/12,lat*12)));
    return rata;
  }
}
```

Kredyt.java

Dodanie konstruktora

```
package pl.kasprowski.utils;
public class Kredyt {
  double procent = 0.05;
  double kwota;
  double lat;

  public Kredyt(double k, double l) {
    kwota = k;
    lat = l;
  }

  public Kredyt(String k, String l) {
    this(Double.parseDouble(k),Double.parseDouble(l));
  }

  public double getRata() {
    double rata = kwota * (procent/12)/
                  (1-(1/Math.pow(1.0+procent/12,lat*12)));
    return rata;
  }
}
```

Kredyt.java

Nowy result.jsp [3]

```
<%
Kredyt kredyt = new Kredyt(
        request.getParameter("kwota"),
        request.getParameter("lat"))
        );
%>
<p>Rata wynosi: <%=kredyt.getRata()%></p>
```

Brak jakiegokolwiek kodu obliczającego cokolwiek!

Obsługa wyjątków

- Gdy coś nie zadziała tak jak trzeba – należy to przewidzieć i "wyłapać" w programie
- Jeśli nie wyłapiemy wyjątku – rezultaty mogą być niespodziewane
- Wyłapywanie wyjątków:

```
try{
    ...tu kod, który może wygenerować wyjątek...
}catch(Exception ex) {...tu kod obsługujący...}
```

Miejsce obsługi wyjątków

- Strona HTML
  - zaleta – można bezpośrednio wypisywać komunikaty
  - wada – strona staje się nieczytelna
- Kod klasy
  - zaleta – kod HTML jest czytelny
  - wada – w kodzie HTML często i tak trzeba zareagować na wyjątek
  - wada podstawowa – nie da się przechwycić wyjątku w naszym konstruktorze ☹

try-catch w result.jsp

```
<%
try{
    Kredyt kredyt = new Kredyt(
            request.getParameter("kwota"),
            request.getParameter("lat"))
            );
    %>
    <p>Rata wynosi: <%=kredyt.getRata()%></p>
    <%
}catch(NumberFormatException ex) {out.write("Nie udało się
    obliczyć raty!");
%>
```

try-catch w result.jsp

```
<%
try{
    Kredyt kredyt = new Kredyt(
            request.getParameter("kwota")),
            request.getParameter("lat"))
            );
    out.write("<p>Rata wynosi: "+kredyt.getRata()+"</p>");
}catch(NumberFormatException ex) {out.write("Nie udało się
    obliczyć raty!");
%>
```

Dyrektywa errorPage

- Ustawienie strony do obsługi błędu

  <%@ page errorPage=”/error.jsp” %>
- Strona error.jsp – dostęp do obiektu exception

  <%@ page isErrorPage="true" %>

  <% response.setStatus(500); %>

  <%= exception.getClass().getName() %>

Dyrektywa include

- Dołączanie innych plików do dokumentu

```
<HTML>
<BODY>
Tutaj będzie zawartość pliku hello.jsp:<br/>
<%@ include file="hello.jsp" %>
A teraz znów jesteśmy w naszym dokumencie
</BODY>
</HTML>
```

Dziękuję za uwagę