Презентация на данните

… чрез Servlet, JSP, JSTL и EL технологиите

Петьо Димитров

01 Април 2013

Съдържание на лекциите

Въведение в Java EE технологиите

Презентация на данните

Бизнес логика

Съхранение на данните

Съдържание

- Сървлет - същност, жизнен цикъл и особености
- JSP - същност, жизнен цикъл, JSP конструкции
- Тагове - потребителски, JSTL, таг файлове
- Expression Language - същност и употреба
- MVC pattern (JSF)

Какво е сървлет?

- сървлетът "сервира" информация като отговор от получена заявка
- програмно API позволяващо прихващане на заявка обслужвана от сървъра и генериране на отговор

- клас имплементиращ Servlet интерфейса
- може да се използва за всякакъв вид заявки (GenericServlet), но най-често се използва заедно с HTTP протокола (HttpServlet)

Приложение на сървлети

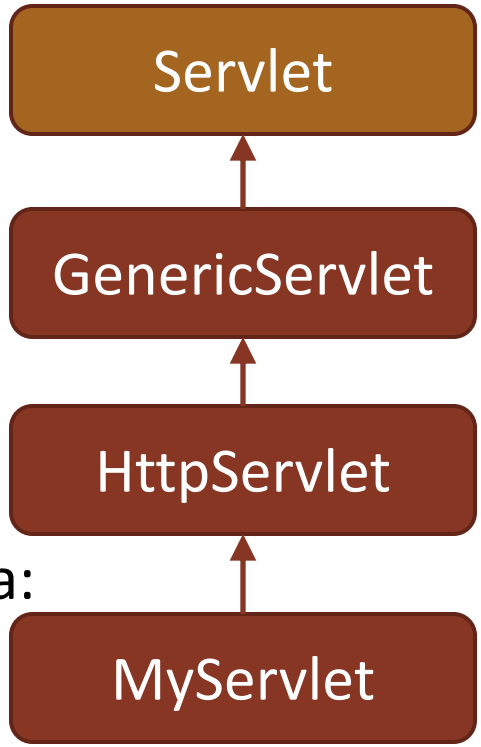
- динамично генериране на съдържание
- обработка на HTTP заявки:
  - GET – за представяне на данни
  - POST – за получаване на данни от HTML форми
- обработка на множество паралелни заявки (по-добро scalability спрямо CGI)
- платформено независимо сървърно API
- позволява препращане на заявки към други сървлети (дава възможност за load balancing)
- основа на JSP и JSF технологиите

Пример за сървлет

```
@WebServlet("/dynamic")
public class MyServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
      response.setContentType("text/html");
      Integer count = (Integer) request.getSession().getAttribute("count");
      if (count == null) {
         count = 0;
      }
      request.getSession().setAttribute("count", ++count);
      response.getWriter().println(formatDynamicContent((Integer)request.getSession().getAttribute("count")));
   }
   private String formatDynamicContent(int count) {
      StringBuilder builder = new StringBuilder();
      builder.append("<html>");
      builder.append("<head><title>").append("Dynamic Servlet").append("</title></head>");
      builder.append("<body>");
      builder.append("<h2>Timestamp: " + new Date() + "<br>").append("Times: " + count + "</h2>");
      builder.append("</body></html>");
      return builder.toString();
   }
}
```

`Servlet` интерфейс и базови класове

- `Servlet` интерфейсът дефинира базовите методи:
  - `init(ServletConfig)`
  - `service(ServletRequest, ServletResponse)`
  - `destroy()`
- `GenericServlet` класът имплементира базовата сървърна фунционалност:
  - `getInitParameter(String)`
  - `getServletContext()`
- `HttpServlet` имплементира HTTP протокола:
  - `doGet(), doPost(), doPut(), doDelete()…`
- `MyServlet` - протребителските сървлети най-често наследяват `HttpServlet`



Жизнен цикъл на сървлет

- **load** - контейнерът зарежда сървлет класа при стартиране на уеб модула или при първа заявка
- **instantiate -** контейнерът създава инстанция от сървлет класа използвайки конструктора му (например `new MyServlet()`)
- **init** - преди сървлетът да може да обслужва заявки, контейнерът го инициализира (чрез метод `init()`) и подава параметрите от web.xml-а

Жизнен цикъл на сървлет

- **service** - след успешна инициализация, сървлетът може да обслужва заявки; контейнерът създава ***отделна нишка*** за всяка заявка и вика `service()` метода на сървлета
- **destroy** - когато сървлетът вече не е нужен, контейнерът вика метод `destroy()`; подобно на `init()` това се случва веднъж в жизнения цикъл
- **unload** – сървлет класът се освобождава от JVM-а на контейнера

Споделяне на информация

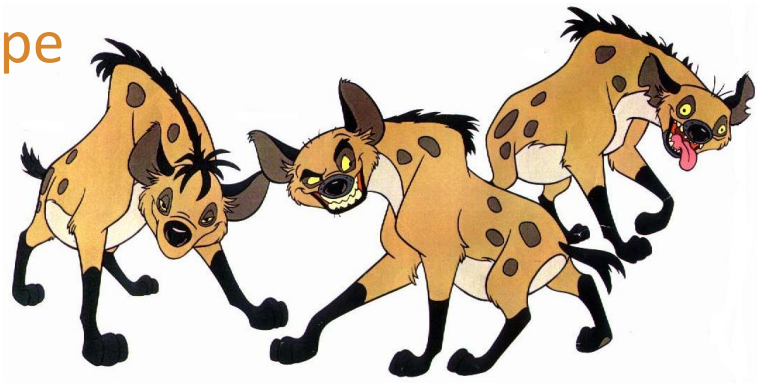
- уеб компонентите споделят информация под формата на обекти складирани като атрибути на 4 scope обекта
- те се манипулират чрез методи `setAttribute()` и `getAttribute()` на scope-а

- scope-овете са (с намаляваща видимост):
  - *application* (`javax.servlet.ServletContext`) - данните ще са видими от всички уеб компоненти в приложението
  - *session* (`javax.servlet.http.HttpSession`) - данните ще са видими за уеб компоненти в текущата сесия (може да се използва за прехвърляна на данни между заявки)
  - *request* (`javax.servlet.ServletRequest`) - данните ще са видими за уеб компоненти от текущата заявка
  - *page* (`javax.servlet.jsp.JspContext`) - само в дадената JSP страница

Concurrency при сървлети

- multithreading-а е заложен в сървлет спецификацията
  - създава се само една инстанция от даден сървлет клас
  - всяка заявка се обработва в отделна нишка
  - всички нишки работят върху един и същ обект

- паралелен достъп може да възникне в следните случаи:
  - достъп до атрибути в application scope
  - достъп до атрибути в session scope
  - достъп до член променливи на сървлета
- предпазване от грешки?



Request & Response

- `ServletRequest`-а дава достъп до:
  - HTTP хедърите на заявката
  - HTML form данни и параметри на заявката
  - други клиентски данни (cookie-та, път и т.н.)
- `ServletResponse`-а отговаря за комуникацията от сървлета обратно към клиента:
  - задаване на content length и MIME тип
  - събиране на отговора (чрез `ServletOutputStream` или `Writer`)

Извикване на други уеб компоненти

- `RequestDispatcher` – обект използван за директно и индиректно извикване на други компоненти; може да се вземе от заявката (`ServiceRequest`) или от контекста на приложението (`ServletContext`)
  - `include(ServletRequest, ServletResonse)` – вмъква съдържанието на друг статичен или динамичен компонент
  - `forward(ServletRequest, ServletResonse)` – предава изпълнението на друг компонент (не трябва да е връщан отговор преди това)
- `HttpServletResponse` – за разлика от диспечера, не прави обработка на сървъра, а връща пренасочващ отговор
  - `sendRedirect(String)` – праща HTTP 302 с новия адрес

Съхранение на състояние

- сесията се използва за корелация на поредица от заявки
- представя се като `HttpSession` обект достъпен от request обекта, чрез метод `getSession()`

- най-лесният начин за следене на сесията е чрез подаване на уникален идентификатор:
  - идентификаторът може да се подава като *cookie*
  - уеб контейнерът може да включи идентификатора като параметър на всяка заявка (URL rewriting)

Демонстрация

JavaServer Pages (JSP)

- технология за създаване на статични или динамични view-та
- използва HTML синтаксис и лесно се интегрира с JavaScript и CSS
- транслира се до сървлет, който се изпълнява на сървъра и връща генерираната репрезентация
- по-малко писане на Java код
- голям набор от Java EE сървъри поддържащи JSP спецификацията (дава portability)

JSP versus ASP

- много Java EE сървъри
- платформено независим
- поддържа Java (JavaScript)
- компилира се
- *модел*: JavaBeans и EJB-та
- *DB достъп*: чрез JDBC
- *разширяване*: чрез custom tag библиотеки

- Internet Information Services (IIS)
- Windows XP↑
- поддържа VBScript (JScript)
- интерпретира се*
- *модел*: Wins 32 COM
- *DB достъп*: чрез ADO
- *разширяване*: не позволява

Идеята зад JSP

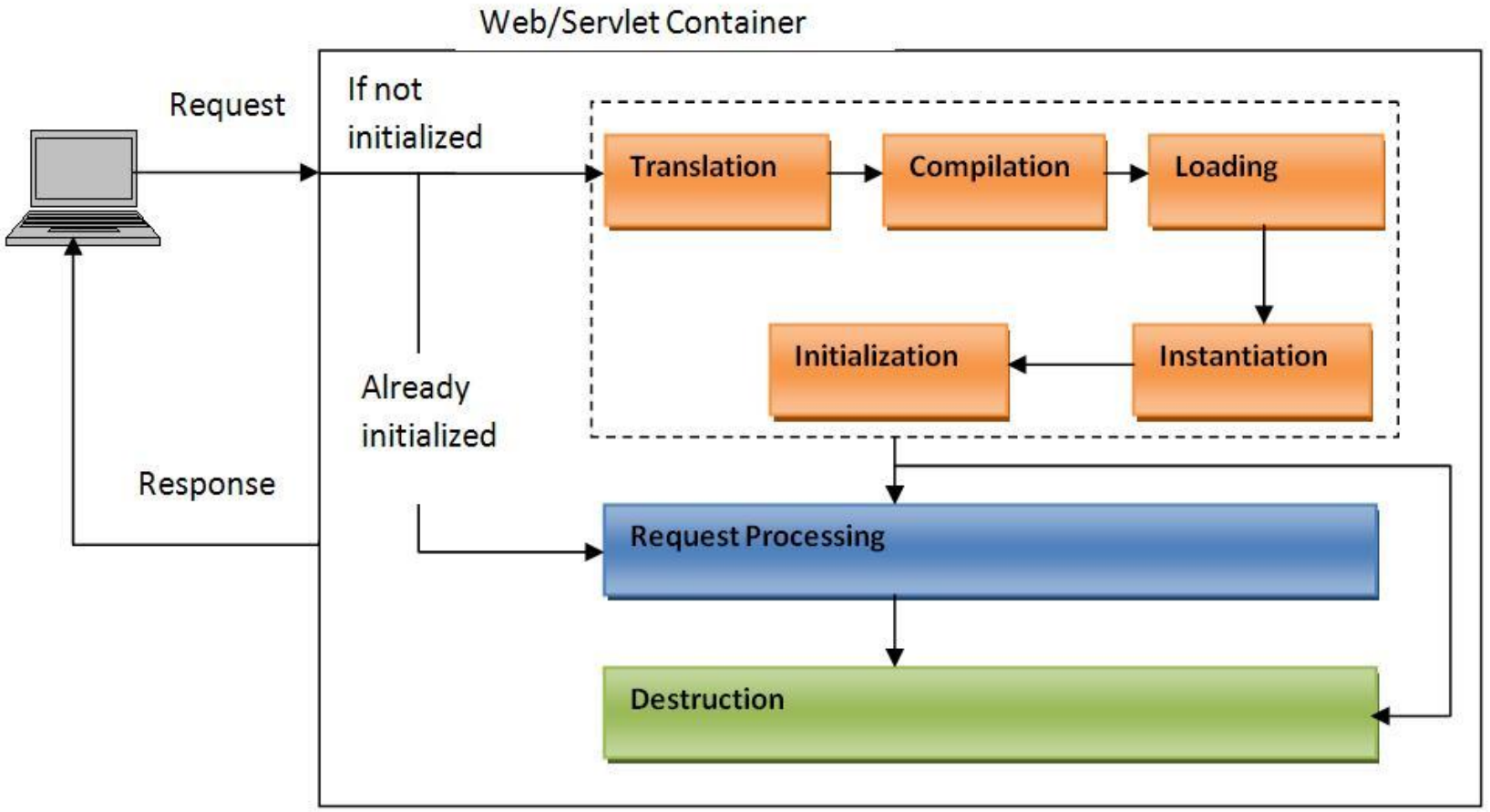
- JSP се базира на сървлет технологията
- всяко JSP в основата си е HTML страница с вложени специални JSP тагове (в които може да има Java код)
- JSP файлът има разширение .jsp (или.jspf за фрагменти)
- JSP engine-ът парсва JSP файла и създава Java сървлет. След това сървлетът се компилира до class файл.

Примерно JSP

- при всяка заявка, ще се връща текущата дата и час

```
<html>
  <head><title>Date JSP example</title></head>
  <body>
    The date is:
    <% out.println(new java.util.Date()); %>
  </body>
</html>
```

Жизнен цикъл на JSP



- `jspInit(),jspDestroy()`
- `_jspService(HttpServletRequest, HttpServletResponse)`

JSP променливи по подразбиране

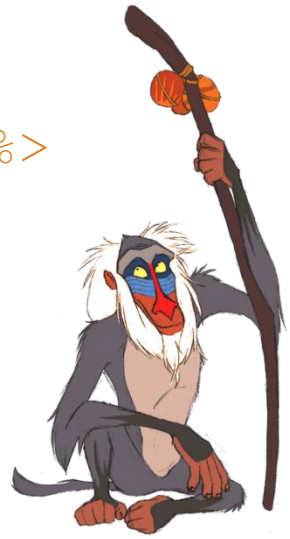
- следните променливи могат да се ползват в JSP-та:
  - `request` – за текущия `HttpServletRequest`
  - `response` – за текущия `HttpServletResponse`
  - `session` – текущата `HttpSession` (ако има такава)
  - `out` – текстов поток за JSP резултата (`PrintWriter`)
  - `application` – `ServletContext`-а на приложението
  - `exception` – за достъп до възникнала грешка
  - `config, pageContext,page`

JSP директиви (directive tags)

- JSP директивите влияят на цялостната структура на генерирания сървлет клас
- синтаксис: `<%@директива атрибут="стойност" %>`
  `<jsp:directive.име атрибут="стойност"/>`
- **page** директива - дефинира атрибути за JSP-то
  - дефинира import пакети:
    `<%@ page import="java.util.*" %>`
  - MIME тип на резултата:
    `<%@ page contentType="text/plain" %>`
  - дефиниране на имплицитна сесия:
    `<%@ page session="true" %>`
  - задаване на JSP URL за грешки:
    `<%@ page errorPage="url" %>`

JSP директиви (directive tags)

- **include** директива - позволява да включат други страници преди JSP-то да се конвертира до сървлет
  - пример: `<%@ include file="/opa/header.jsp" %>`
- **tablib** директива - декларира използването на библиотека с custom тагове
  - пример: `<%@ taglib uri="uri" prefix="op" %>`
- XML формат на директивите:
  - `<jsp:directive.page attribute="value" />`
  - `<jsp:directive.include file="url" />`
  - `<jsp:directive.taglib uri="uri" prefix="op"/>`



JSP действия (actions)

- XML елементи контролиращи поведението на сървлет engine-а
- синтаксис: **`<jsp:име атрибут="стойност" />`**

- примери:
  - `<jsp:include page="relative URL" flush="true" />`
  - `<jsp:forward page="url" />`
  - `<jsp:useBean id="opa" />`
  - `<jsp:setProperty name="opa" property="prop" value="val" />`
  - `<jsp:getProperty name="opa" property="prop" />`
  - `<jsp:text>Template data</jsp:text>`

JSP декларации (declaration tags)

- JSP декларациите позволяват дефинирането на методи или член променливи в тялото на генерирания сървлет клас
- синтаксис: **<%! Java декларации; %>**
- обикновено не генерират съдържание, а се използват заедно с JSP изрази и скриплети

- пример:

```
<%!
  long counter = 0;
  public void getCounter() {return counter;}
%>
```

JSP изрази (expression tags)

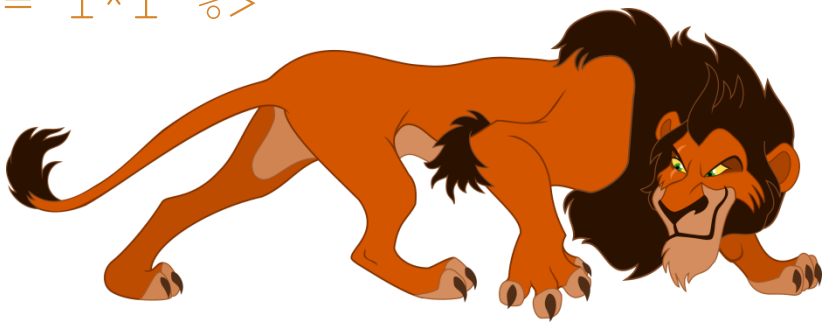
- JSP изразите се използват за директно вмъкване на резултата от Java израз в изходния резултат от JSP-то
- синтаксис: **`<%= Java израз %>`**

- примери:
  - `<%= new java.util.Date() %>`
  - `<%= Math.PI %>`
  - `<%= request.getRemoteHost() %>`
  - `<%= session.getMaxInactiveInterval() %>`

JSP скриплети (scriplets)

- JSP скриплетите позволяват директното влагане на Java код в `_jspService()` метода на JSP-то
- синтаксис: **`<% Java код; %>`**
- скриплетите имат достъп до стандартните JSP променливи (request, response, session, …)
- пример:

```
<% for (int i=0; i<10; i++) { %>
    <%= i %> * <%= i %> = <%= i*i %>
    <br>
<% } %>
```



Демонстрация

Потребителски тагове (custom tags)

- компоненти за многократно използване позволяващи компонентно-ориентирана разработка в Java уеб приложенията
- всеки разработчик може да създаде custom tag
- скриват сложността на визуализацията
- изглеждат като HTML тагове, което улеснява уеб дизайнерите и разработчиците
- пример:

```
<mytags:opa someattr="value" />
```

Библиотека от тагове (tag library)

- съвкупност от пакетирани потребителски тагове
- всяка библиотека има префикс и URI идентификатор
- всяка таг библиотека се състои от:
  - TLD дескриптор описващ таговете
  - JAR с компилираните класове и ресурси на таговете
- използване:
  - TLD дескрипторът се слага в /WEB-INF
  - JAR-ът се поставя в /WEB-INF/lib
  - регистриране библиотеката в JSP-то:
    `<%@ taglib prefix="mytags" uri="WEB-INF/my.tld" %>`
  - използване на тага: `<mytags:opa someattr="value" />`

JavaServer Pages Standard Tag Library (JSTL)

- представлява набор от потребителски тагове (custom tags) за многократно използване
- стандартна библиотека с тагове част от JSP 2.0
- имплементира често използвани функционалности:
  - *JSTL Core* - основни функции за работа с променливи, условия, цикли, операции за вход/изход, …
  - *JSTL Format* - форматиране и интернационализация
  - *JSTL XML* - четене на XML данни , XSL трансформации
  - *JSTL SQL* - изпълнение на SQL заявки (!)
  - *JSTL Functions* - извикване на стандартни функции

Предимства на JSTL

- позволява JSP страниците да съдържат само XML
  - избягваме влагане на Java код в view-тата (чрез скриплети)
  - кодът става по лесен за четене и поддръжка
- спестяват се усилия на разработчика, защото може наготово да се използва често срещана функционалност
- лесно използване:

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

JSTL Core тагове

- работа с променливи:
  - `<c:set var="n1" value="v1" scope="page" />`
  - `<c:set var="n2" value="v2" scope="request" />`
  - `<c:set var="n3" value="v3" scope="session" />`
  - `<c:set var="n4" value="v4" scope="application"/>`
  - `<c:remove var="n1" scope="page" />`
- извеждане на данни:
  - `<c:out value="${user.age}" escapeXml="false"/>`
- прихващане на грешки:
  -
```
<c:catch var="DivideByZeroException">
    <% int x = 10/0; %>
</c:catch>
```

JSTL Core тагове (продължение)

- условни операции:
  - ```
    <c:if test='${param.p == "val"}'>
    Пропърти p на param е равно на "val"
    </c:if>
    ```
  - ```
    <c:choose>
      <c:when test='${param.p=="val"}'>оп</c:when>
      <c:otherwise>няма оп</c:otherwise>
    </c:choose>
    ```
- цикли:
  - ```
    <c:forEach var='item' begin='1' end='10'>
      <c:out value='${item}'/>
    </c:forEach>
    ```
  - ```
    <c:forEach var='item' items='${itemsList}'>
      <li><c:out value='${item}'/></li>
    </c:forEach>
    ```

Таг файлове

- файл съдържащ JSP фрагмент, който може да се използва многократно като потребителски таг
- препоръчва се използването на разширение .tag

- използване:
  - за разделяне на JSP-то на отделни модули
  - за повторно използване на логика

- видове:
  - самостоятелни (без тяло): `<m:opa />`
  - поддържащи други тагове (с тяло): `<m:opa>X</m:opa>`

Примерен таг файл

- файл с име opa.tag (самостоятелен):

```
<%@ tag body-content="empty" %>
<%@ attribute name="firstName" required="true"%>
<h1>Hello, <%= firstName %> !</h1>
```

- поставяне на opa.tag в /WEB-INF/tags

- деклариране:

```
<%@ taglib prefix="m" tagdir="/WEB-INF/tags" %>
```

- употреба:

```
<m:opa firstName="Penn"/>
```

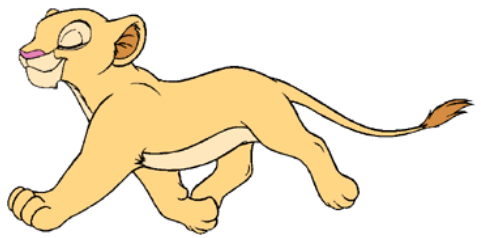
Unified Expression Language (UEL)

- позволява лесното достъпване на обекти, техните пропъртита и методи от JSP и JSF страници
- синтаксис: **`${ el израз }`**

- замества JSP action таговете:
  - `<jsp:useBean id="user" scope="request" />`
  - `<jsp:setProperty name="user" property="name" value="Penn" />`
  - `<jsp:getProperty name="user" property="name"/>`
- пример:
  - `${opa.prop = "Penn"}`
  - `${opa.prop}`

Предимства на UEL

- дава кратка нотация за достъп до данни в scope-ове page, request, session и application
  `${course}`
- лесен достъп до елементи на колекция
  `${students[2]}`
- лесен достъп до пропъртита на bean обекти:
  `${course.presenter.name}`
- достъп до параметри на заявката, хедъри, cookies, …
  `${param["studentCount"]}`
  `${cookie["dateOfLastVisit"]}`



Предимства на UEL (продължение)

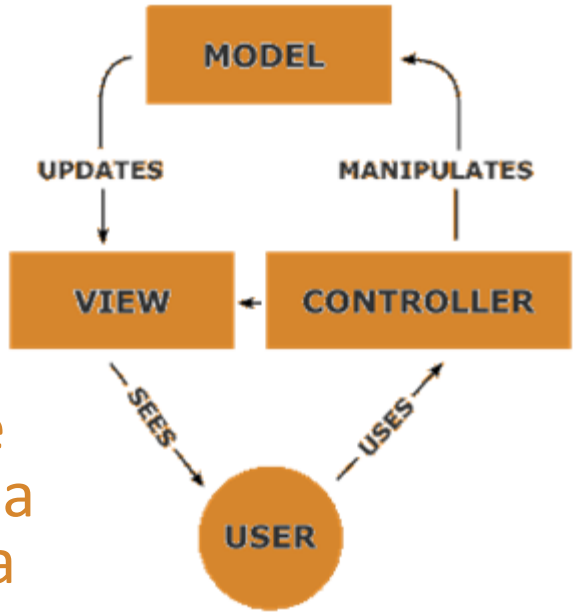
- набор от прости оператори (+, -, *, /, =, <, >, ==, &&, ||, ?:, empty, not):
  ```
  ${(2 + 5) * 3}
  ${course.studentCount - 1}
  ${not empty presenter ? "тук съм" : "успах се" }
  ```
- автоматично конвертиране на типове към стринг
- възможност за извикване на функции:
  ```
  ${fn:length("It's the cycle of life…")}
  ```
- връща празен стринг вместо exception:
  ```
  ${thisIsNull.prop} → ""
  ```

Достъп до обекти при UEL

- атрибутите могат да се търсят в конкретен scope:
  `${session.course}`
- ако името на атрибута не е валиден Java идентификатор, се използва алтернативната нотация:
  `${session[course.with.dots]}`
- ако не се зададе scope, търсенето на обекта се извършва от най-малкия към най-големия scope:
  - pageContext
  - request
  - session
  - application

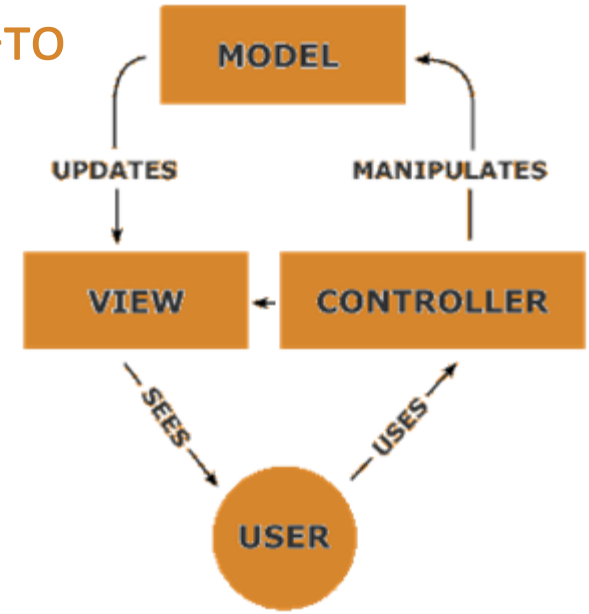
Model-View-Controller (MVC)

- софтуерен *pattern* разделящ презентацията на информация от работата на потребителя с нея
- използва се при визуални приложения
- разделя приложението на:
  - **model** - съхранява данните, състоянието и бизнес логиката; информира view-то при промяна на състоянието *
  - **view** - представя визуално данните от модела
  - **controller** - приема потребителските команди и на тяхна база манипулира данните в модела; праща сигнали за промяна на view-то



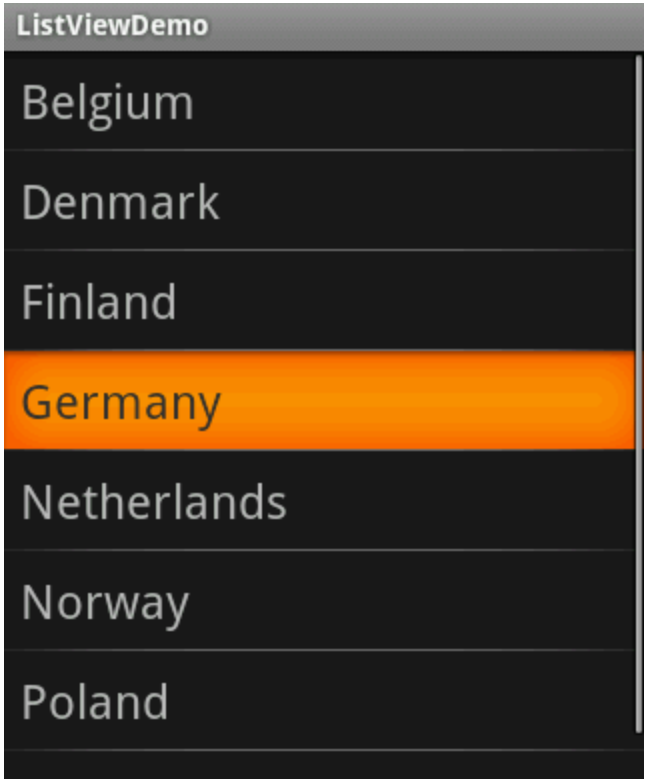
Model-View-Controller (продължение)

- видимост между компонентите:
  - моделът не знае за останалите компоненти
  - view-то знае само за модела
  - контролерът знае за модела и view-то
- съставни части на *pattern-а*:
  - composite pattern (view)
  - strategy pattern (view и контролер)
  - observer pattern (view и модел)



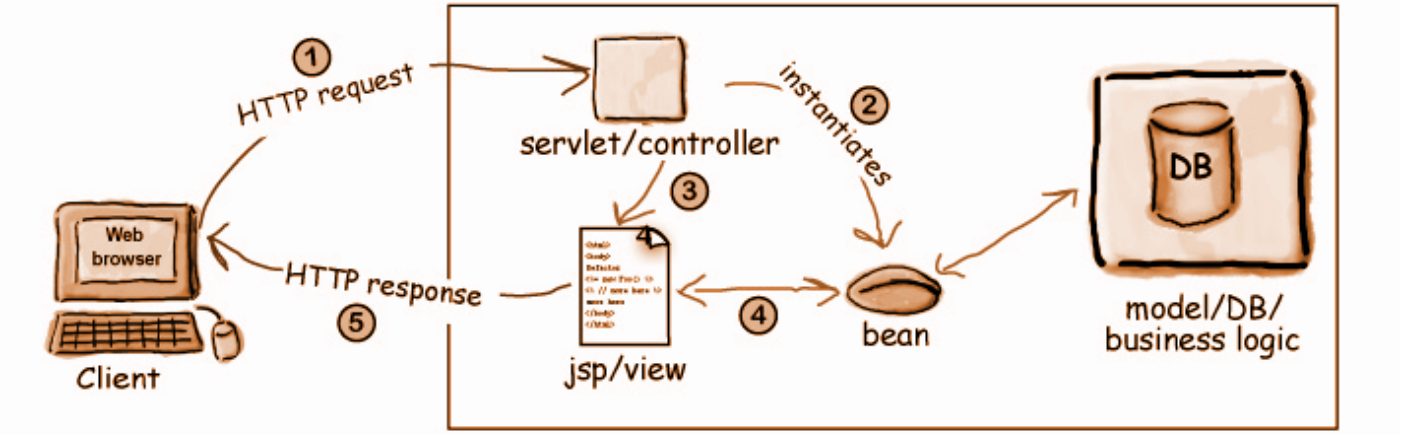
Model-View-Controller - пример

- потребителят изпраща команда за добавяне на елемент в списъка:
  - контролерът получава командата и модифицира модела
  - моделът информира view-то за промяната
  - view-то визуализира новия елемент
- потребителят изпраща команда за скролиране:
  - контролерът я получава и праща сигнал до view-то
  - view-то скролира списъка



MVC и мрежата (model 2)

1. потребителят праща заявка на даден URL
2. контролерът извиква модела и получава данни*
3. контролерът асоциира данните с view-то и предава контрола на view-то
4. view-то използва данните и изгражда презентация
5. получената презентация се връща на потребителя



Демонстрация

JavaServer Faces (JSF)

- Java framework за създаване на компонентно-базирани уеб приложения

- използва request-driven MVC:
  - модел - managed bean-ове, ejb-та и entity-та (beans)
  - view - JSF UI тагове и Facelets (xml файлове)
  - контролер - рутира заявки и избира view, което да се визуализира; контролера е стандартен - `FacesServlet`

Out of scope

- За сървлети: filter-и, listener-и, web.xml дескриптор, сървлет анотации, изпращане на файлове, асинхронни сървлети, частични дескриптори и други
- За JSP-та: детайли за JSP директивите и JSTL тагове (Format, SQL, XML, функции), създаване на потребителски тагове, deferred UEL изрази
- За JSF технологията: почти всичко ☺

Въпроси

Благодаря за вниманието