Introduction to JSP

(Server-Side Programming using Java Server Pages)

Many HTML Pages are Mostly Static

- Servlets allow us to write dynamic Web pages
  - Easy access to request, session and context data
  - Easy manipulation of the response (cookies, etc.)
  - And lots more...
- It is very inconvenient to write and maintain long and mostly static HTML pages using Servlets (even though such pages are very common)

```java
out.println("<h1>Bla Bla</h1>" + "bla bla bla bla" + "lots more here...")
```
Introducing JSP

- The Idea:
  - Use HTML for most of the page
  - Write Servlet code directly in the HTML page, marked with special tags
- The server **automatically translates** a JSP page to a **Servlet** class and **invokes** this servlet
  - In Tomcat, you can find the generated Servlet code under $CATALINA_BASE/work/

A JSP is no more than a convenient way to write Servlets that output textual data

Relationships

- **Servlets**: HTML code is printed using Java code
- **JSP**: Java code is embedded in HTML code
- Not only for HTML!
  - JSP can be used for any textual format
  - Servlets can be used for any data!
Example

```html
<html>
<head>
<title>Hello World</title>
</head>
<body>
<h2><%= new java.util.Date() %></h2>
<h1>Hello World</h1>
</body>
</html>
```

```java
try {
  _jspxFactory = JspFactory.getDefaultFactory();
  response.setContentType("text/html");
  pageContext = _jspxFactory.getPageContext(this, request, response, null, true, 8192, true);
  _jspx_page_context = pageContext;
  application = pageContext.getServletContext();
  config = pageContext.getServletConfig();
  session = pageContext.getSession();
  out = pageContext.getOut();
  _jspx_out = out;
  out.write("<html>
      <head>
      </head>
      <body>
      <h2>\n      \n      <h1>Hello World</h1>
      </body>
      </html>\n      
      try {
        if (!out.getClass().isInstance(SkipPageException)) {
          out = _jspx_out;
          if (out != null && out.getBufferSize() != 0)
            out.clearBuffer();
          if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
        }
      } finally {
        if (_jspxFactory != null) _jspxFactory.releasePageContext(_jspx_page_context);
      }
```
The file dates.jsp is in
C:\Program Files\Tomcat 5.5\webapps\Examples\ (sCATALINA_BASE\webapps\MyApplication)

The url
http://ibm373.cs.technion.ac.il:8080/Examples/dates.jsp

Generated Servlet Hierarchy
(Tomcat 5 Implementation)
JSP Limitations and Advantages

• JSP can only do what a Servlet can do
• Easier to write and maintain HTML
• Easier to separate HTML from code
• Can be created using a "reverse engineering technique":
  • Create static HTML and then replace static data with Java code
### JSP Life Cycle

<table>
<thead>
<tr>
<th>Request #1</th>
<th>Request #2</th>
<th>Request #3</th>
<th>Request #4</th>
<th>Request #5</th>
<th>Request #6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Page first written**

**Server restarted**

**Page modified**

---

### JSP Translation

- When the JSP file is modified, JSP is translated into a Servlet
  - But only after the JSP’s url is requested by a client
  - Application is not necessarily reloaded when JSP file is modified
- Server does not generate the Servlet class after startup, if the latter already exists and isn’t too old
  - Generated Servlet acts just like any other Servlet
- The generated servlet can handle **GET, POST, HEAD** requests though it does not implement **doGet(), doPost(), doHead()** explicitly
  - Its **Servlet.service()** method calls the newly implemented main method named **HttpJspBase._jspService()**

---

JSP file named file.jsp will be translated into the Java file file_jsp.java

---

Written by Marty Hall. Core Servlets & JSP book: [www.coreservlets.com](http://www.coreservlets.com)
init() and destroy()

- init() of the generated Servlet is called every time the Servlet class is loaded into memory and instantiated
- destroy() of the generated Servlet is called every time the generated Servlet is removed
- init() and destroy() are called even when the reason for loading is a modification of the JSP file

jspInit and jspDestroy

- In JSP pages, like regular Servlets, we sometimes want to implement init and destroy
- It is illegal to use JSP declarations to override init or destroy, since they are (usually) already implemented by the generated Servlet
- Instead, override the methods jspInit() and jspDestroy()
  - The generated servlet is guaranteed to call these methods from init and destroy, respectively
  - The standard versions of jspInit and jspDestroy are empty (placeholders for you to override)
Thread Synchronization

- After the Servlet is generated, one instance of it serves requests in different threads, just like any other Servlet
- In particular, the service method (_,jspService_) may be executed by several concurrent threads
- Thus, as with Servlets, JSP programming requires handling concurrency

Basic JSP Elements
Basic Elements in a JSP file

- HTML code: `<html-tag>content</html-tag>`
- JSP Comments: `<%-- comment --%>`
- Expressions: `<%= expression %>`
- Scriptlets (statements): `<% code %>`
- Declarations: `<%! code %>`
- Directives: `<%@ directive attribute="value" %>`
- Actions: `<jsp:forward.../>`, `<jsp:include.../>`
- Expression-Language Expressions: `_${expression}_`

JSP Expressions

- A JSP expression is being used to insert Java values directly into the output
- It has the form: `<%= expression %>` , where expression can be a Java object, a numerical expression, a method call that returns a value, etc...
- For example:
  `<%= new java.util.Date() %>`
  `<%= "Hello"+" World" %>`
  `<%= (int)(100*Math.random()) %>`

The heading space and the following space are not created in the result.
Use ` " ` if you want a real space.
JSP Scriplets (Statements)

- JSP scriptlets let you insert arbitrary code into the Servlet service method (\_jspService)
- Scriptlets have the form: `<% Java Code %>`
- The code is inserted verbatim into the service method, according to the location of the scriptlet
- Scriptlets have access to the same automatically defined variables as expressions
  - request, response, session, out, application (servletContext), config (servletConfig)

A Divided-Code Example

Scriptlets don't have to be complete code blocks:

```jsp
<% if (Math.random() < 0.5) { %>
You <b>won</b> the game!
<% } else { %>
You <b>lost</b> the game!
<% } %>

if (Math.random() < 0.5) {
    out.write("You <b>won</b> the game!");
} else {
    out.write("You <b>lost</b> the game!");
}
```
JSP Declarations

- A JSP declaration lets you define methods or members that are being inserted into the Servlet class (outside of all methods)
- It has the following form:
  ```%
  Java Code
  %>
```
- For example:
  ```%
  private int someField = 5;
  %>
  private void someMethod(...) {...}
  %>
```
- JSPs are intended to contain a minimal amount of code so it is usually of better design to define methods in a separate Java class...

JSP Directives

- A JSP directive affects the structure of the Servlet class that is generated from the JSP page
- It usually has the following form:
  ```%
  @ directive attribute1="value1" ... 
  attributeN="valueN"
  %>
```
- Three important directives: page, include and taglib
page-Directive Attributes

- **import** attribute: A comma separated list of classes/packages to import
  
  ```jsp
  <%@ page import="java.util.*, java.io.*" %>
  ```

- **contentType** attribute: Sets the MIME-Type of the resulting document (default is text/html)
  
  ```jsp
  <%@ page contentType="text/plain" %>
  ```

---

page-Directive Attributes (cont)

- What is the difference between setting the **page contentType** attribute, and writing `<%response.setContentType("...");%>`?
  
  - In the latter case, the new servlet will call `response.setContentType()` twice
  - The first, implicit (from the JSP point of view), call will be with the default content type.
  - The second, explicit, call might even come after the buffer was flushed or after the writer was obtained…
Actions

- JSP **actions** use constructs in XML syntax to control the behavior of the Servlet engine
- Using actions, you can
  - **forward** the **request** to another resource in the application
  - dynamically **include** a resource content in the **response**
- Forward and include are translated to an invocation of the **RequestDispatcher**

The forward Action

- **jsp:forward** - Forwards the requester to a new resource

```xml
<jsp:forward page="{relativeURL|<%= expression %>}">
    <jsp:param name="parameterName" value="{parameterValue | <%= expression %}>" />
</jsp:forward>
```

You can use `%=`, `%` instead of `<%=, %> so that the code would be a legal XML.

One or more parameters (not attributes!) added to the original request parameters.
Forward Action Example

```jsp
<%! int even = 0; %>
<% even = (1 - even); %>
<% if (even == 0) { %>
  <jsp:forward page="/requestParams.jsp">
    <jsp:param name="sessionID" value="<%= session.getId() %>">
    <jsp:param name="even" value="true"/>
  </jsp:forward>
 <% } else { %>
  <jsp:forward page="/requestParams.jsp">
    <jsp:param name="sessionID" value="<%= session.getId() %>">
    <jsp:param name="even" value="false"/>
  </jsp:forward>
 <% } %>

<html>
<head><title>Print Request Params</title></head>
<body>
  <%@ page import="java.util.*" %>
  <% Enumeration parameterNames = request.getParameterNames(); %>
  <% while (parameterNames.hasMoreElements()) { %>
    <h2><%= name %>: <%= request.getParameter(name) %></h2>
  <% } %>
</body>
</html>
requestParams.jsp
JSP Expression Language

- JSP expression language is a comfortable tool to access useful objects in JSP
- This language provides shortcuts in a somewhat JavaScript-like syntax
- An expression in EL is written as $\{expr\}$
- For example:

  Hi, $\{user\}$. <em style="$\{style\}">Welcome</em>

Note that the EL expression does not violate the XML syntax as opposed to `<%= expression %>`

An Example

```jsp
<% response.addCookie(new Cookie("nameof","homer");
    session.setAttribute("homepage", new java.net.URL("http://www.simpsons.com");
    String[] strs = {"str1","str2");
    session.setAttribute("arr", strs);
%>

<html><head><title>JSP Expressions</title></head><body>
  <form method="get" action="el.jsp">
    <h2>Write the parameter x: <input name="x" type="text" /></h2>
    <input type="submit" value="send" /></form>
</body></html>```

elcall.jsp
Java Beans in JSP
Motivation

- Software components (e.g. objects, data structures, primitives) are extensively used in Web applications.
- For example:
  - Service local variables
  - Attributes forwarded in requests
  - Session attributes, such as user information
  - Application attributes, such as access counters
- See tutorial at http://docs.oracle.com/javase/tutorial/javabeans/
Java Beans: The Idea

- Java Beans are simply objects of classes that follow some (natural) coding convention:
  - An empty constructor
  - A readable property has a matching `getter`
  - A writable property has a matching `setter`
- Use JSP actions to access and manipulate the bean, and special action attributes to specify the properties of the bean, e.g., its scope
- JSP programmers do not wish to write cumbersome code or class files

Scopes

- Each object that is created or accessed by a JSP page has a scope
- The scope describe who can access the object and when to destroy it
- There are four scopes:
  - Page Scope
  - Request Scope
  - Session Scope
  - Application Scope
Example 1: Access Counter

In the following example, we use a Bean to maintain an access counter for requests to the pages

```java
package myUtils;

public class CounterBean {
    private int counter;

    public CounterBean() { counter = 0; }
    public int getCounter() { return counter; }
    public void setCounter(int i) { counter = i; }
    public void increment() { ++counter; }
}
```

A Bean is a concept and therefore there's no need to extend any class or implement any interface! (though it would've been very Java-ish to create an empty interface "Bean")

A Bean is created by an empty constructor

CounterBean is a concept and there's no need to extend any class or implement any interface!

A Bean must reside in a package

Counter setter and getter

Other methods can be implemented as well
<html>
<head><title>Bean Example</title></head><body>
<jsp:useBean id="accessCounter" class="myUtils.CounterBean" scope="application"/>

<% accessCounter.increment(); %>
<h1>Welcome to Page A</h1>
<h2>Accesses to this application:</h2>
<jsp:getProperty name="accessCounter" property="counter"/>
</h2>
<a href="pageB.jsp">Page B</a>
</body>
</html>

A very similar JSP

Since an instance named according to the given id can be found in the application scope, no instantiation takes place.

The default scope is page.

You could also use the type attribute in order to instantiate a data type which is either superclass of class or an interface that class implements.

An instance named according to the given id is either found in the relevant scope or is created.
Part of the Generated Servlet

```java
myUtils.CounterBean accessCounter = null;

synchronized (application) {
    accessCounter = (myUtils.CounterBean)
        _jspx_page_context.getAttribute("accessCounter",
            PageContext.APPLICATION_SCOPE);
    if (accessCounter == null) {
        accessCounter = new myUtils.CounterBean();
        _jspx_page_context.setAttribute("accessCounter",
            accessCounter, PageContext.APPLICATION_SCOPE);
    }
}
```

Advantages of Java Beans

- Easy and standard management of data
  - Automatic management of bean sharing and lots more
- Good programming style
  - Allow standard but not direct access to members
  - You can add code to the setters and getters (e.g. constraint checks) without changing the client code
  - You can change the internal representation of the data without changing the client code
- Increase of separation between business logic (written by programmers) and HTML (written by GUI artists)
Listeners

- A **listener** is a class that listens for various events from the lifecycle of a web application.
- Listeners use an event handling model.
  - Events are raised by the container and listener classes are registered to be notified of particular events.
- How you register a listener?
Servlet Context Listeners

- **ServletContextListener**
  - Notified when the context is initialized (the application is being started)
  - Notified when the context is destroyed (the application is being shut down)
- **ServletContextAttributeListener**
  - Notified when an attribute on the web application context (ServletContext) has been added, removed or replaced.

**Typical Uses of ServletContextListener**

- **Resource initialization**
  - Works well for initializing resources like logging, application properties & database connections
- **Resource cleanup**
  - Cleaning up resources, logging application state before shut down
Http Session Listeners

- HttpSessionBindingListener
  - Notifies objects when they are bound to a session
  - Example:
    - Suppose customer info is stored in a database, and there is a desire to retrieve the information to the application when the customer logs in.
    - If you add the customer to the session on log-in, and the customer has a HttpSessionBindingListener, it can be notified when the session starts and go look up the info.

Http Session Listeners

- HttpSessionListener
  - This listener is notified when a session is created or destroyed.
  - Want to keep a count of how many sessions are out there? This would be a good way to do it.

HttpSessionAttributeListener

- This type of listener is notified when any type of attribute is added to, removed from, or replaced on a session.
  - May be useful if the class can not be changed to implement HttpSessionBindingListener
Http Session Listeners

- HttpSessionActivationListener
  - Notified when a session is about to be migrated (activated/deactivated)
  - This comes into play with distributed applications – the application runs on multiple VM’s potentially on multiple servers.

```java
package Session;
import javax.servlet.http.*;

public class SessionCount implements HttpSessionListener {
    private static int numberOfSessionsCount = 0;

    public void sessionCreated (HttpSessionEvent evt) {
        numberOfSessionsCount++;
    }
    public void sessionDestroyed (HttpSessionEvent evt) {
        numberOfSessionsCount--;
    }

    public static int getNumberOfSessionsCount() {
        return numberOfSessionsCount;
    }
}
```
Session Listener Review

- 4 Types of Listeners
  - HttpSessionActivationListener
  - HttpSessionListener
  - HttpSessionBindingListener
  - HttpSessionAttributeListener

- 2 Types of Events
  - HttpSessionEvent
  - HttpSessionBindingEvent

Two Other Listener Types

- ServletRequestListener
  - Notified when a request is initialized
  - Notified when a request is destroyed

- ServletRequestAttributeListener
  - Notified when a request attribute is added, removed, or replaced.
Filters

- A filter can intercept an HTTP request and execute code before or after the requested servlet or JSP is executed.
- This makes them idea for cross-cutting concerns, which are aspects of an application that are more used across the entire application such as logging, localization, authentication, compression, caching, etc.
Benefits of filters

- **Modular code** – Filters allow you to store code in a single location but be applied to multiple parts of the application.
- **Flexible code** – You can use the web.xml file to control when filters are executed. This allows you to apply the filters to different parts of an application and to turn them on or off easily.
- **Chaining** – You can chain multiple filters together to perform multiple tasks.

```java
public class LogFilter implements Filter {

    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        //Get the IP address of client machine.
        String ipAddress = request.getRemoteAddr();
        //Log the IP address and current timestamp.
        System.out.println("IP "+ipAddress + ", Time "+ new Date().toString());
        chain.doFilter(req, res);
    }
```
<filter>
  <filter-name>LogFilter</filter-name>
  <filter-class>
    myPackage.LogFilter
  </filter-class>
  <init-param>
    <param-name>test-param</param-name>
    <param-value>This parameter is for testing.</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/</url-pattern>
</filter-mapping>

public void init(FilterConfig config) throws ServletException {

  //Get init parameter
  String testParam = config.getInitParameter("test-param");

  //Print the init parameter
  System.out.println("Test Param: "+ testParam);
}

public void destroy() {
  //add code to release any resource
}

Types of filters

- Filters can be applied in different times:
  - Intercept and process request **before** they are sent to the servlet.
  - Process responses **after** the servlet but before responses are sent back to the client.
- Both cases are written the same way: **there is only one filter interface**.

Filters in Requests and Responses

- For the request side processing, do the filter logic first, then call the FilterChain doFilter() method.
- For the response side, call the FilterChain doFilter() method, then do your filter logic.
Facelets

MVC Pattern
MVC in Traditional JSP

Presentation Layer Models
Facelets

- Web template system
- Replacing JSP as the default view technology in apache products
  - Supported by Java Server Faces
- Supports Expression Language (EL)

### Facelets Templating Tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui:component</td>
<td>Defines a component that is created and added to the component tree.</td>
</tr>
<tr>
<td>ui:composition</td>
<td>Defines a page composition that optionally uses a template. Content outside of this tag is ignored.</td>
</tr>
<tr>
<td>ui:debug</td>
<td>Defines a debug component that is created and added to the component tree.</td>
</tr>
<tr>
<td>ui:decorate</td>
<td>Similar to the composition tag but does not disregard content outside this tag.</td>
</tr>
<tr>
<td>ui:define</td>
<td>Defines content that is inserted into a page by a template.</td>
</tr>
<tr>
<td>ui:fragment</td>
<td>Similar to the component tag but does not disregard content outside this tag.</td>
</tr>
<tr>
<td>ui:include</td>
<td>Encapsulate and reuse content for multiple pages.</td>
</tr>
<tr>
<td>ui:insert</td>
<td>Inserts content into a template.</td>
</tr>
<tr>
<td>ui:param</td>
<td>Used to pass parameters to an included file.</td>
</tr>
<tr>
<td>ui:repeat</td>
<td>Used as an alternative for loop tags, such as c:forEach or h:dataTable.</td>
</tr>
<tr>
<td>ui:remove</td>
<td>Removes content from a page.</td>
</tr>
</tbody>
</table>
A Template with Three Sections

Composite page using template
Other features

- Referencing to another page using ui:include (e.g. header and footer)
- Custom tags
- Composite components

Java Server Faces (JSF)

- A server-side UI Framework adopted by apache
  - Officially released in 2004
  - Current version 2.2
- Event driven component model (using Facelets)
- HTTP stateless is concealed
  - The components are stateful
- Rapid UI development on server-side java
  - no need to deal with browsers and web servers
  - Reuse UI components
JSF Main Features

- UI development
  - bind UI components to data model
  - abstract request processing into event-driven model
- Navigation – flexible rules drive the flow of pages
- Session and Object management
- Validation and Error feedback
- Internationalization
- Custom components

Events

- Flow is dominated by events
  - events fired by JSF UI components
    - added to events list
    - notifies all listeners
  - system events are fired when the application starts or stopping
- Event types:
  - Value change – changes in input components (e.g., select element)
  - Action - clicks on a button or link component
  - Application – JSF lifecycle: PostConstructApplicationEvent, PreDestroyApplicationEvent, PreRenderViewEvent
- Handling events:
  - method binding – handle the event in the relevant JavaBean
  - with Listener – implement new class for handling the event
We continue to see teams run into trouble using JSF -- JavaServer Faces -- and are recommending you avoid this technology.

Teams seem to choose JSF because it is a J2EE (sic)standard without really evaluating whether the programming model suits them. We think JSF is flawed because it tries to abstract away HTML, CSS and HTTP, exactly the reverse of what modern web frameworks do. JSF, like ASP.NET webforms, attempts to create statefulness on top of the stateless HTTP protocol and ends up causing a whole host of problems involving shared server-side state.

We are aware of the improvements in JSF 2.0 but think the model is fundamentally broken. We recommend teams use simple frameworks and embrace and understand web technologies including HTTP, HTML and CSS.