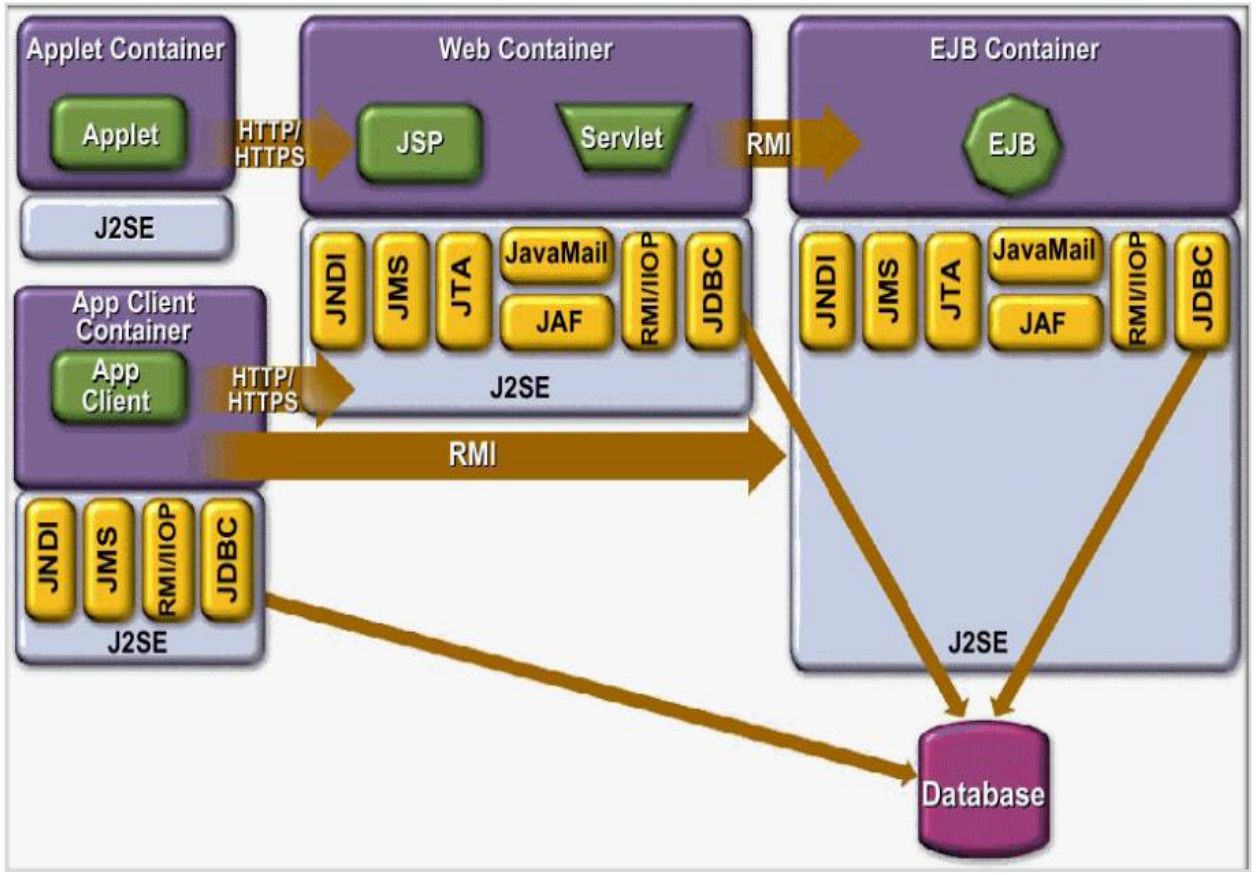
2023/2024(1)
EF234301 Web Programming

Lecture #6

Java Server Pages

Misbakhul Munir IRFAN SUBAKTI
司馬伊凡
Мисбакхул Мунир Ирфан Субакти

JSP & Servlet as the web components

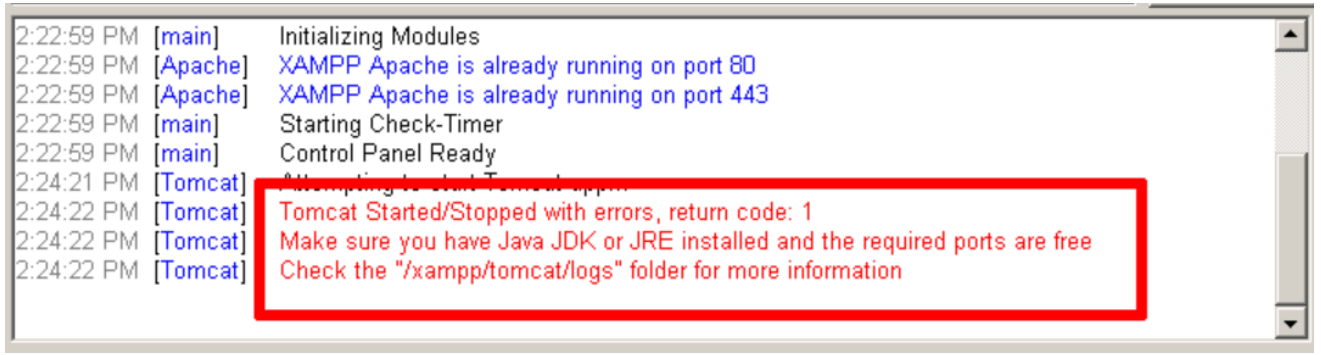


JSP page: what is it?

- *Text-based* document
  - Returning both static & dynamic content to the client browser
- Dynamic & static content can be combined
- Static content
  - HTML, XML, Text
- Dynamic content
  - Java Code
  - Displaying properties of JavaBeans
  - Invoking business logic defined in Custom tags

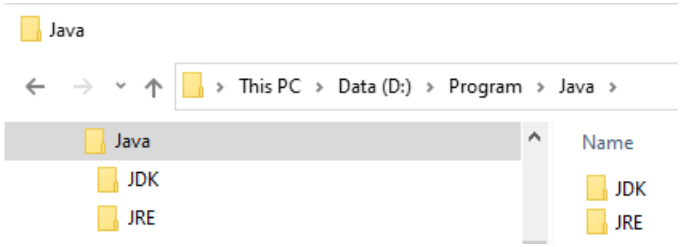
XAMPP: Apache Tomcat

- We have downloaded and installed XAMPP.
- XAMPP for Windows comes with Apache Tomcat built in, making it easy to get started with *Java-based* Web applications.
- XAMPP for Windows does not include Java, which is a pre-requisite for using Apache Tomcat. If you do not already have Java installed, you will see an error similar to the below when you attempt to start Apache Tomcat through the XAMPP control panel.

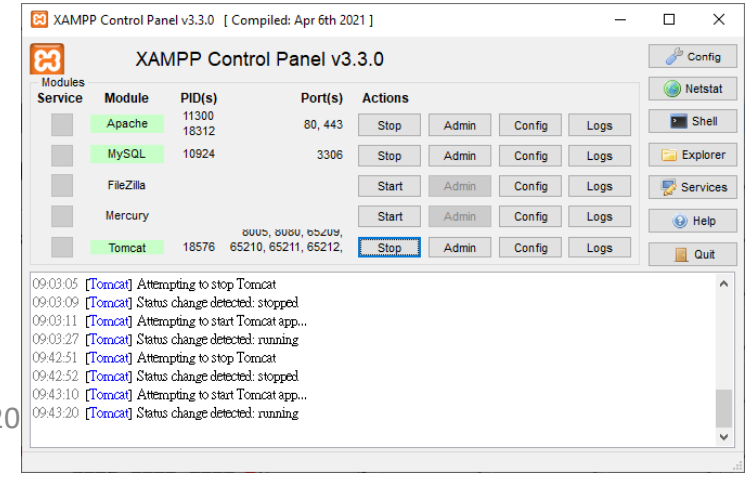


XAMPP: Apache Tomcat (continued)

- To correct that problem, install Java and then attempt to use Apache Tomcat, as below:
  - Download the latest version of the Java Runtime Environment (JRE).
  - Follow the on-screen instructions to install Java.
  - E.g., we installed JRE Java on `D:/Program/Java/JRE`

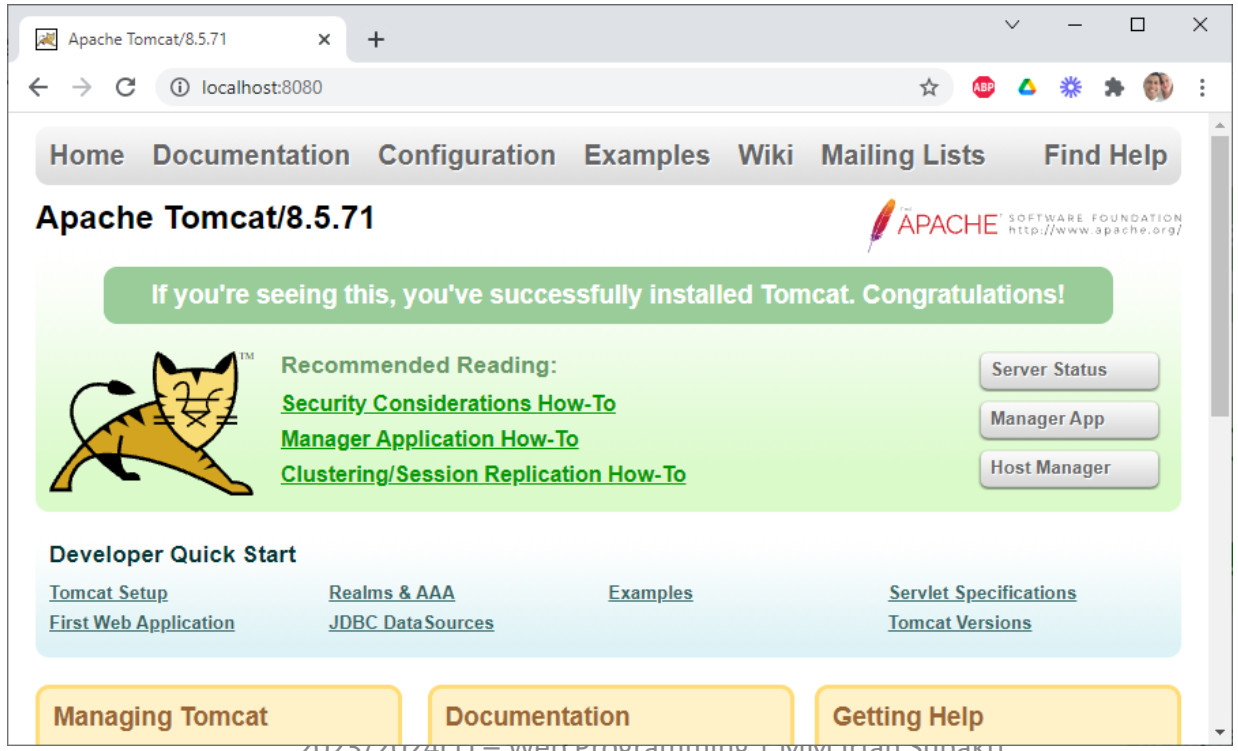


  - Start the Apache Tomcat server using the XAMPP control panel.



XAMPP: Apache Tomcat (continued)

- We should now be able to access Apache Tomcat by browsing to `http://localhost:8080` in our browser's address bar. Here's an example of what we should see:



Tomcat: Configuration

- Apache Tomcat includes two applications, the `manager` application and the `host-manager` application, that simplify management and deployment of Web applications and provide detailed information on server status. These applications can be accessed from the Apache Tomcat welcome page, as highlighted in the image above.
- Access to the above applications is `blocked` by default. To access them, we must configure one or more sets of administrator accounts and then assign the roles `manager-gui` and/or `admin-gui` to these accounts. Accounts with the `manager-gui` role would have access to the `manager` application, and those with the `admin-gui` role would have access to the `host-manager` application.

Tomcat: Configuration (continued)

- To configure these accounts, follow the steps below:
  - Edit the *tomcat-users.xml* file in the *tomcat/conf* subdirectory of our XAMPP installation directory (e.g., `D:/Program/xampp`) and add the line below, before the closing *</tomcat-users>* element:

```
<role rolename="manager-gui"/>
<role rolename="admin-gui"/>
<user username="admin" password="admin" roles="manager-gui,admin-gui" />
```

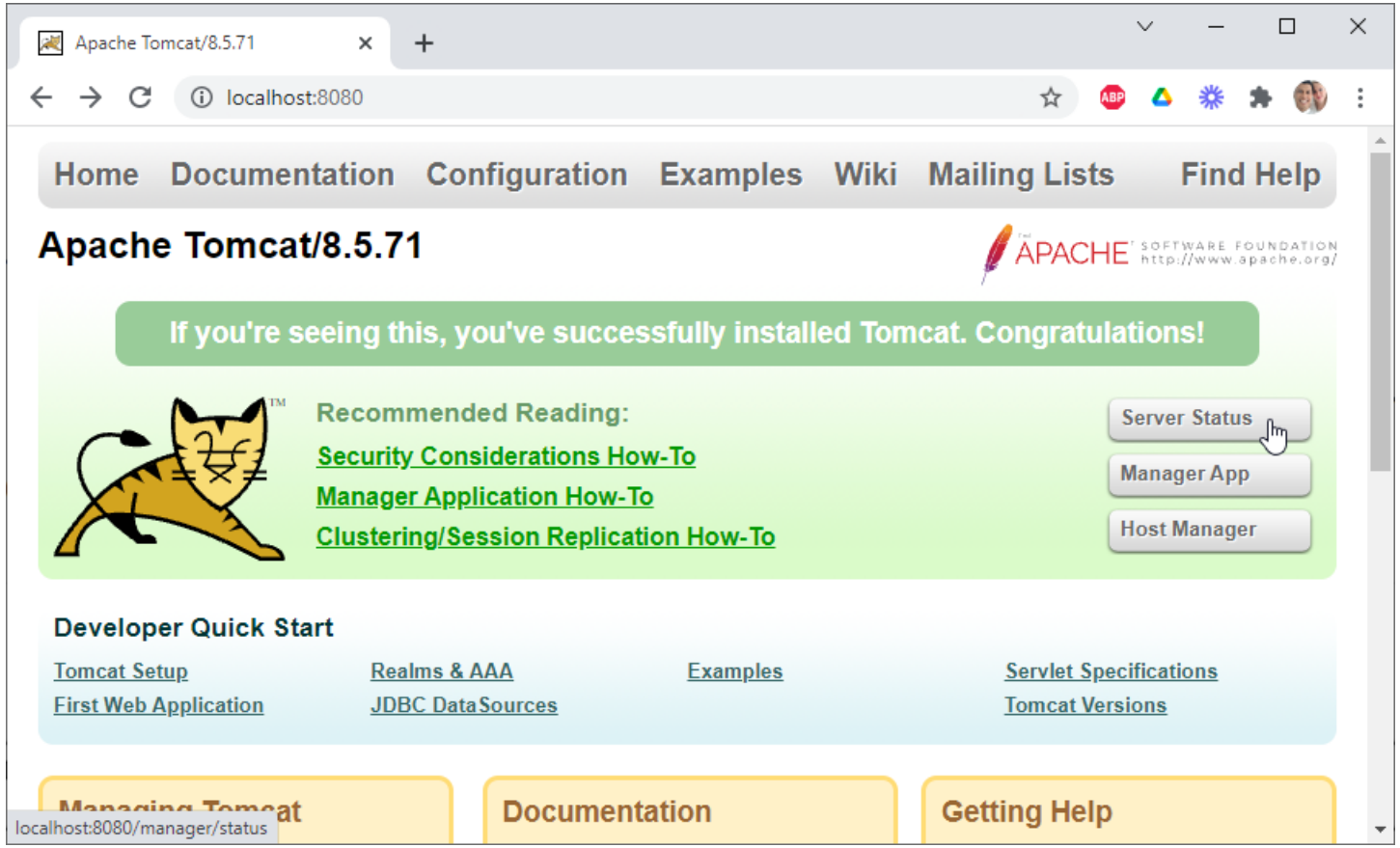
    - This configures an account with username "admin" and password "admin", with access to both the Apache Tomcat Web applications. Remember to replace the username and password with values specific to your installation.
    - If you prefer to have a separate account for each application, replace the previous configuration with this example:

```
<role rolename="manager-gui"/>
<role rolename="admin-gui"/>
<user username="manager" password="manager" roles="manager-gui" />
<user username="admin" password="admin" roles="admin-gui" />
```

  - Save the changes.
  - Restart the Apache Tomcat server using the XAMPP control panel.

Tomcat: Configuration (continued)

- We should now be able to access the Apache Tomcat Web applications using the configured credentials. Here's an example of the management Web application, which includes controls to deploy new WAR applications.



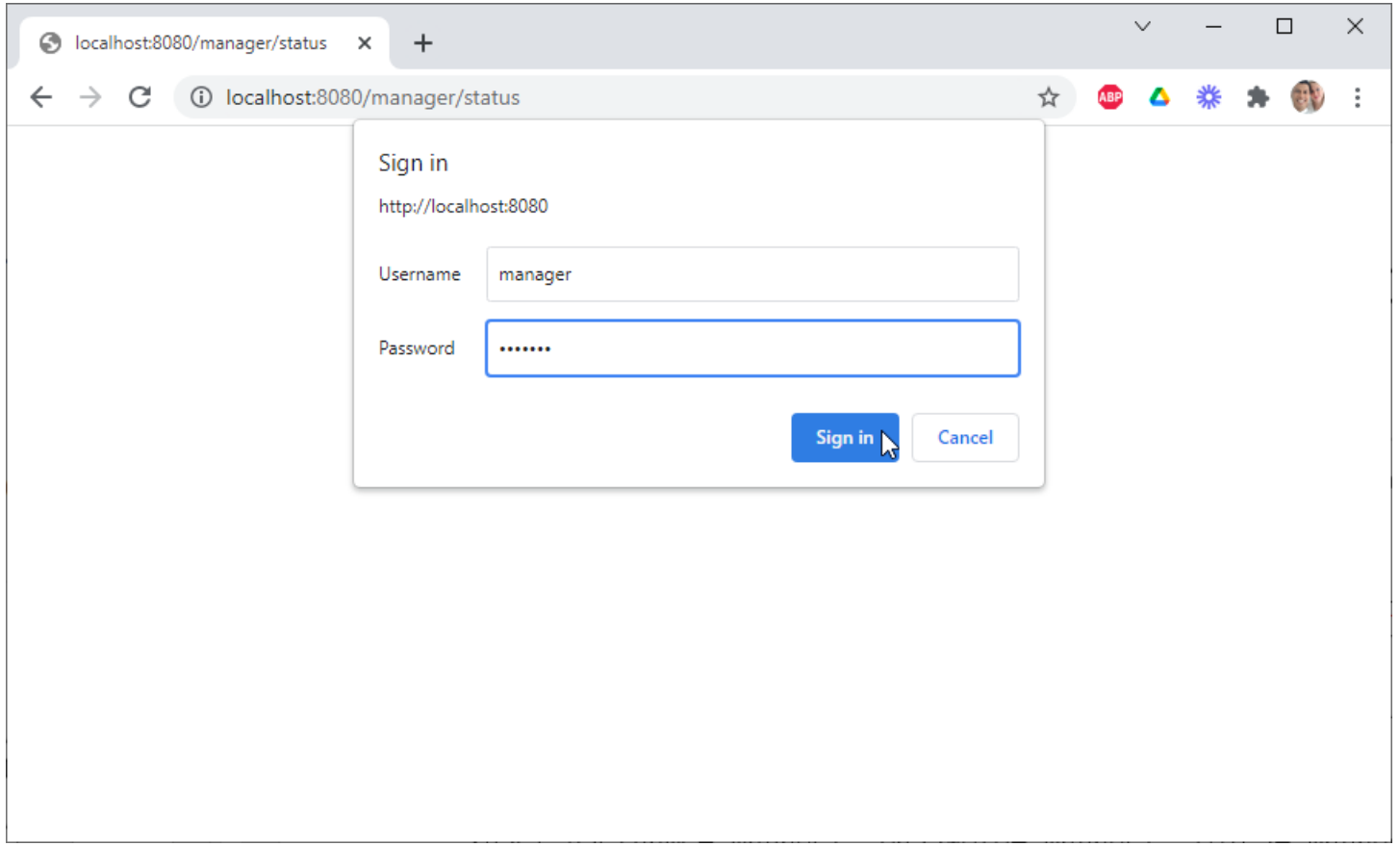
Tomcat: Configuration (continued)

- Enter the username and password, e.g., `manager` & `manager`
  - If we set:

```
<role rolename="manager-gui"/>
<role rolename="admin-gui"/>
<user username="manager"
password="manager"
roles="manager-gui,admin-gui" />
```

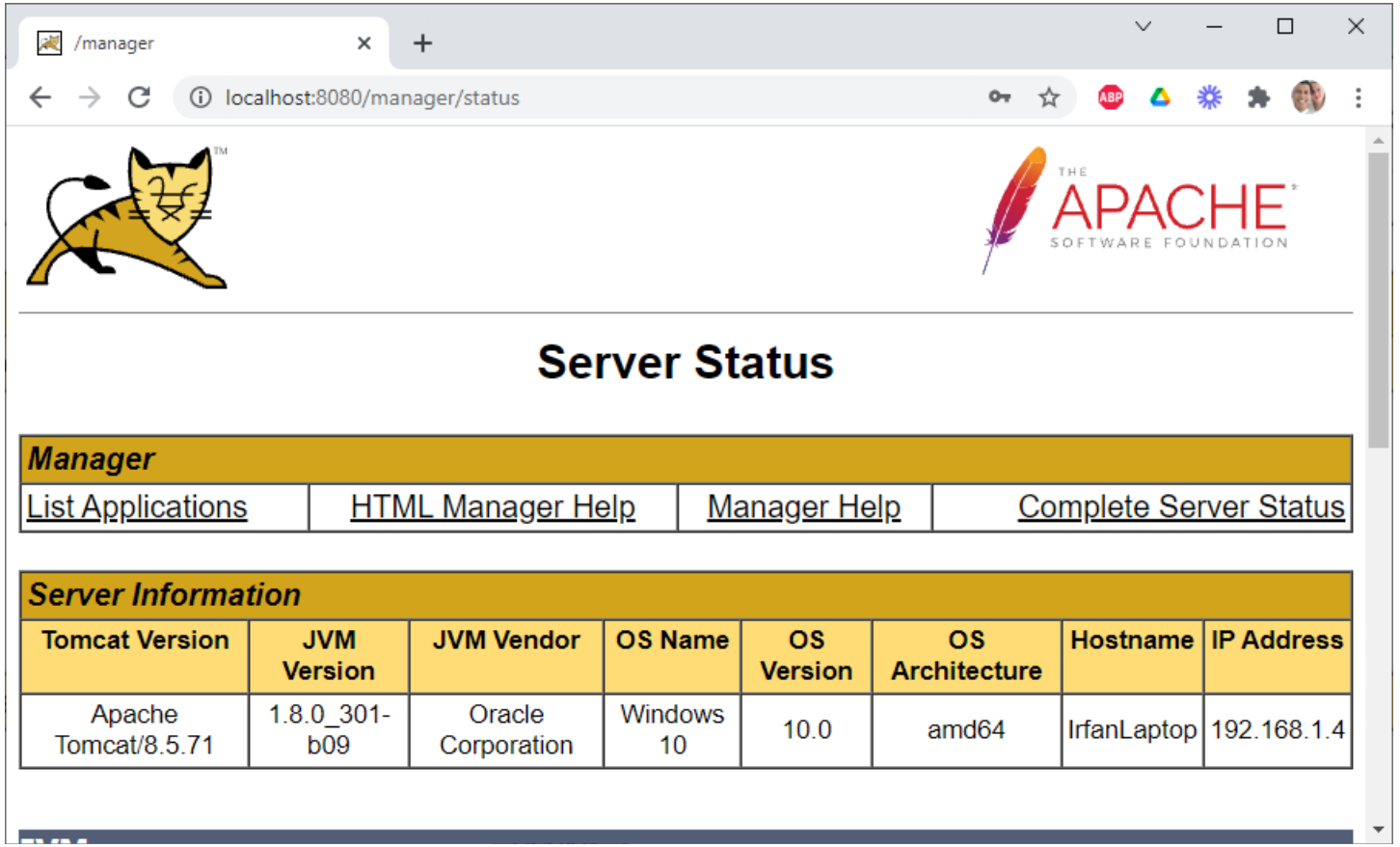
- Enter the username and password, e.g., `admin` & `admin`
  - If we set:

```
<role rolename="manager-gui"/>
<role rolename="admin-gui"/>
<user username="admin" password=
"admin" roles="manager-gui,admin-
gui" />
```



Tomcat: Configuration (continued)

- … an voila! We're able to see the server status!



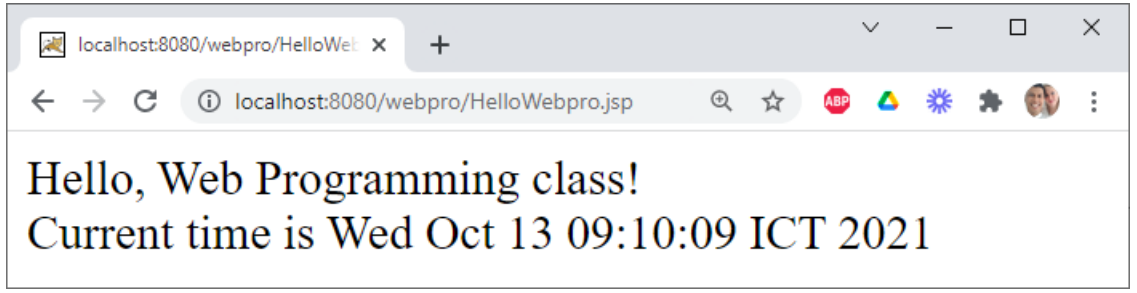
JSP page: A simple one

- Colour: black = static, red = dynamic contents

```
<html>
   <body>
      Hello, Web Programming class!
      <br>
      Current time is <% =new java.util.Date() %>
   </body>
</html>
```

HelloWebpro.jsp

```
1 <html>
2     <body>
3         Hello, Web Programming class!
4         <br>
5         Current time is <%= new java.util.Date() %>
6     </body>
7 </html>
```



JSP & Servlet: Comparison

Servlets

- HTML code in Java
- Not easy to author

JSP

- Java-like code in HTML
- Very easy to author
- Code is compiled into a servlet

JSP: Benefits

- Content and display logic are separated
- Simplify web application development with JSP, JavaBeans and Custom tags
- Support software reuse through the use of components (JavaBeans, Custom tags)
- Automatic deployment
  - Recompile automatically when changes are made to JSP pages
- Easier to author web pages
- Platform-independent

JSP over Servlet: Why?

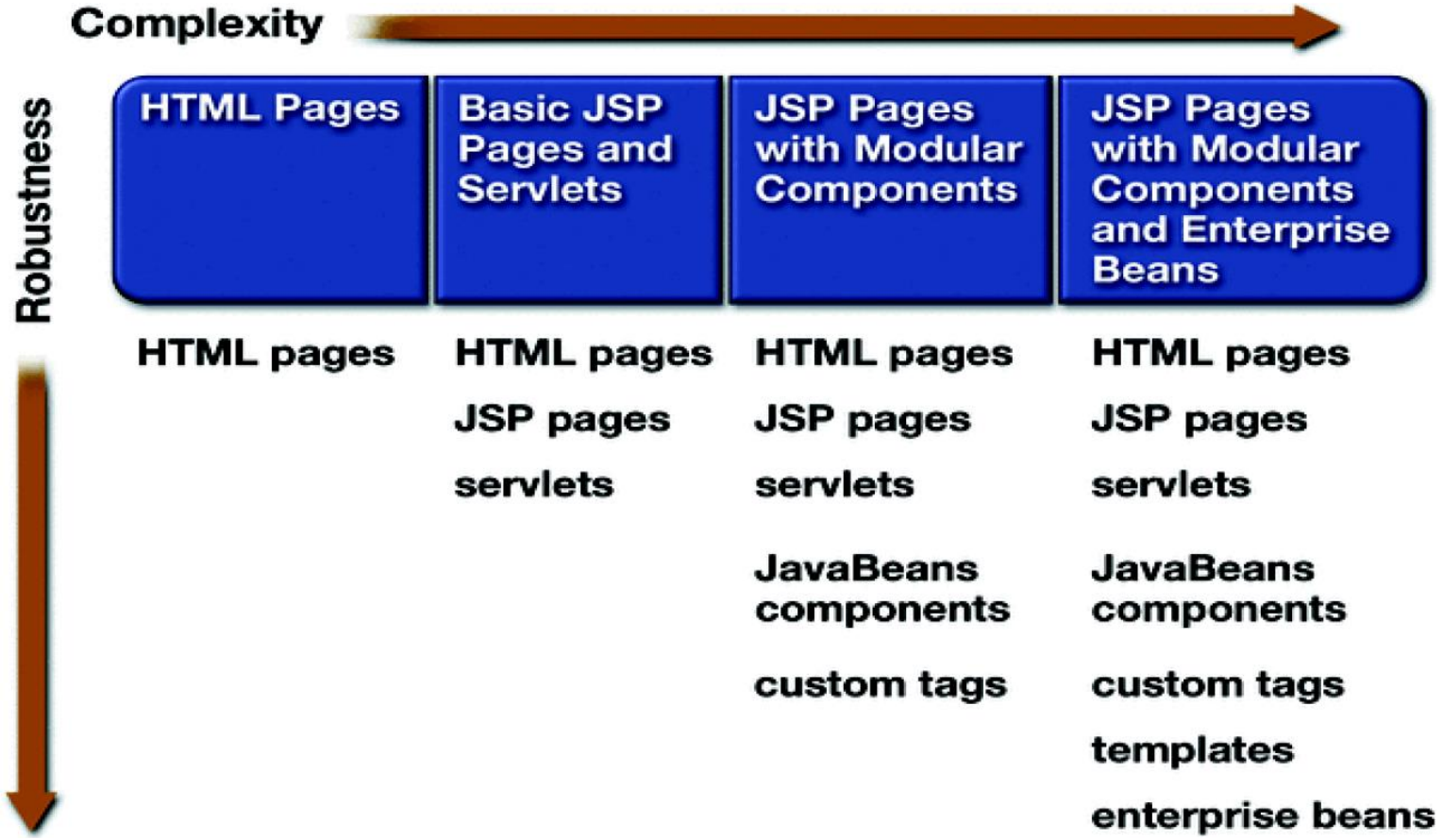
- Servlets can do a lot of things, but it is pain to
  - Use those `println()` statements to generate HTML page
  - Maintain that HTML page
- No need for compiling, packaging, CLASSPATH setting

JSP over Servlet: Or Servlet over JSP?

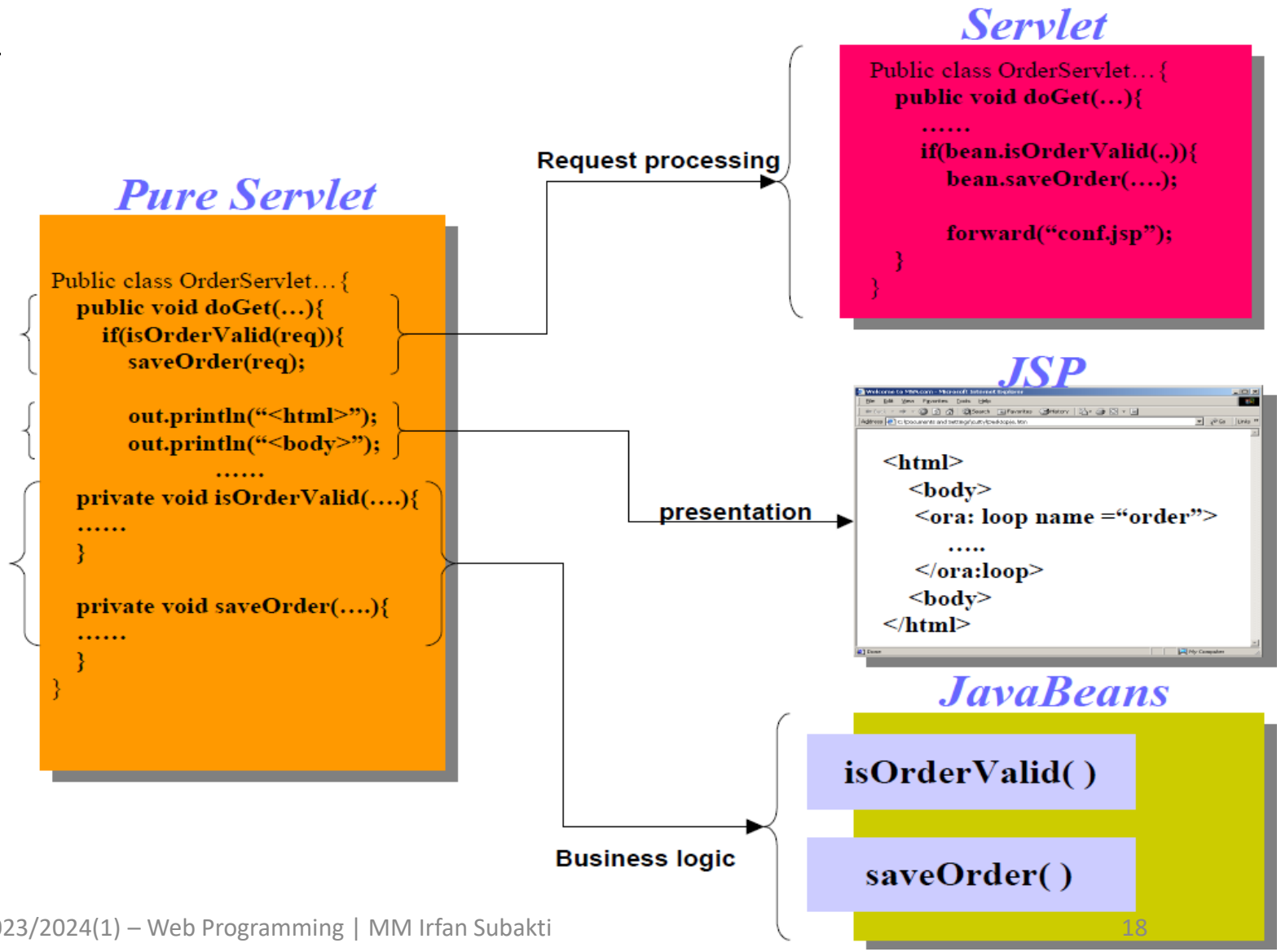
- No, we want to use both!
- It's for leveraging the strengths of each technology
  - Servlet's strength: *controlling* and *dispatching*
  - JSP's strength: *displaying*
- In a typically production environment, both servlet and JSP are used in a so-called MVC (Model-View-Controller) pattern
  - Servlet handles **controller** part
  - JSP handles **view** part

JSP: Architecture

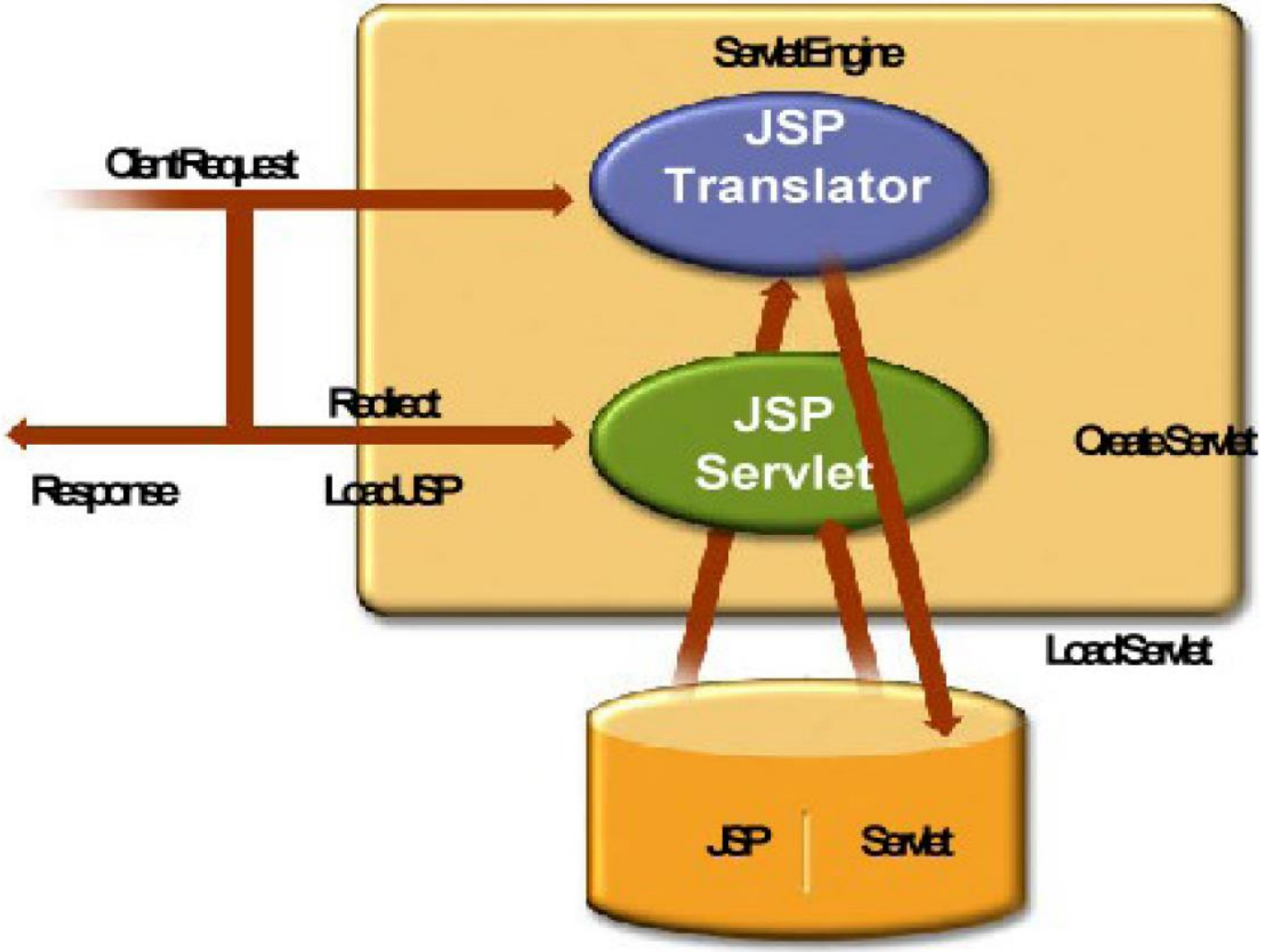
- Web application designs



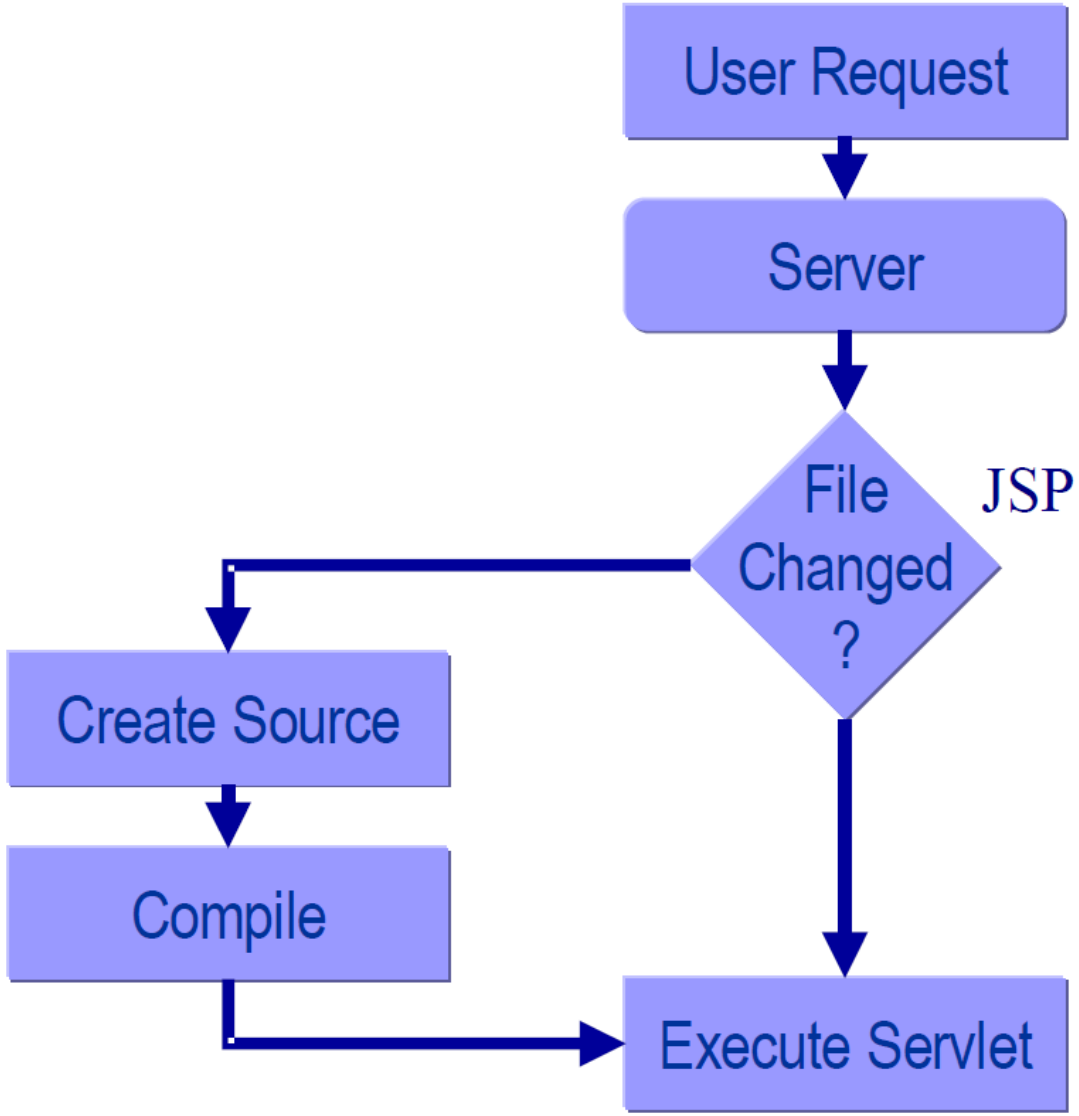
Separate `request` processing from `presentation`



JSP: Architecture diagram



JSP: How does it work?



JSP-based web app: Developing steps

1. Write (and compile) the Web component code (Servlet or JSP) and helper classes referenced by the web component code
2. Create any static resources, e.g., images or HTML pages
3. Create deployment descriptor → `web.xml`
4. Build the Web application → `*.war` file or deployment-ready directory
5. Install or deploy the Web application into a Web container
   - Clients (browsers) are now ready to access via URL

1. Write & compile the Web component code

- Create development tree structure
- Write either servlet code and/or JSP pages along with related helper code
- Create `build.xml` for Ant-based build (and other application life-cycle management) process

Development tree structure

- Keep Web application source separate from compiled files
  - Facilitate iterative development
- Root directory
  - `build.xml`: Ant build file
  - `context.xml`: Optional application configuration file
  - `src`: Java source of servlets and JavaBeans components
  - `web`: **JSP pages** and HTM pages, images

2. Create any static resources

- HTML pages
  - Custom pages
  - Login pages
  - Error pages
- Image files that are used by HTML pages or JSP pages

3. Create deployment descriptor (`web.xml`)

- Deployment descriptor contains deployment runtime instructions to the Web container
  - URN (Uniform Resource Name) that the client uses to access the Web component
- Every Web application has to have it, i.e., `web.xml`

4. Build the Web application

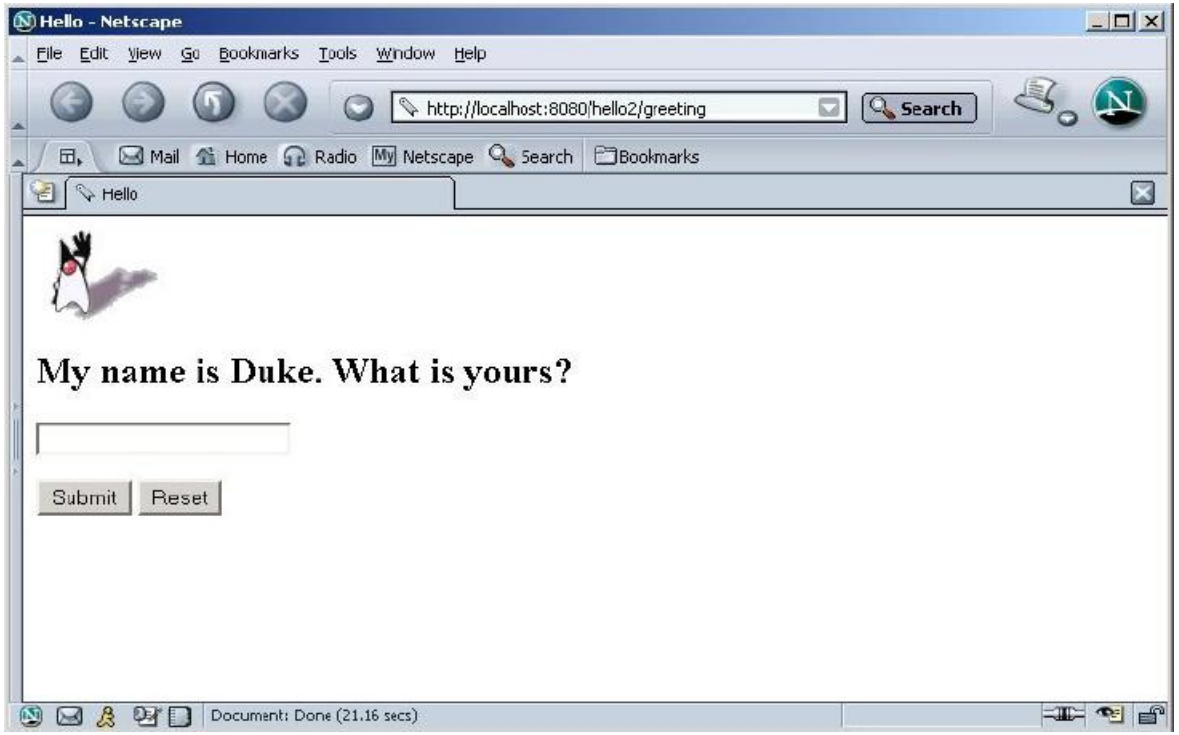
- Either `*.WAR` file or unpacked form of `*.WAR` file
- Build process is made of
  - Create `build` directory (if it is not present) and its subdirectories
  - Copy `*.jsp` files under `build` directory
  - Compile Java code into `build/WEB-INF/classes` directory
  - Copy `web.xml` file into `build/WEB-INF` directory
  - Copy image files into `build` directory

5. Install or deploy Web application

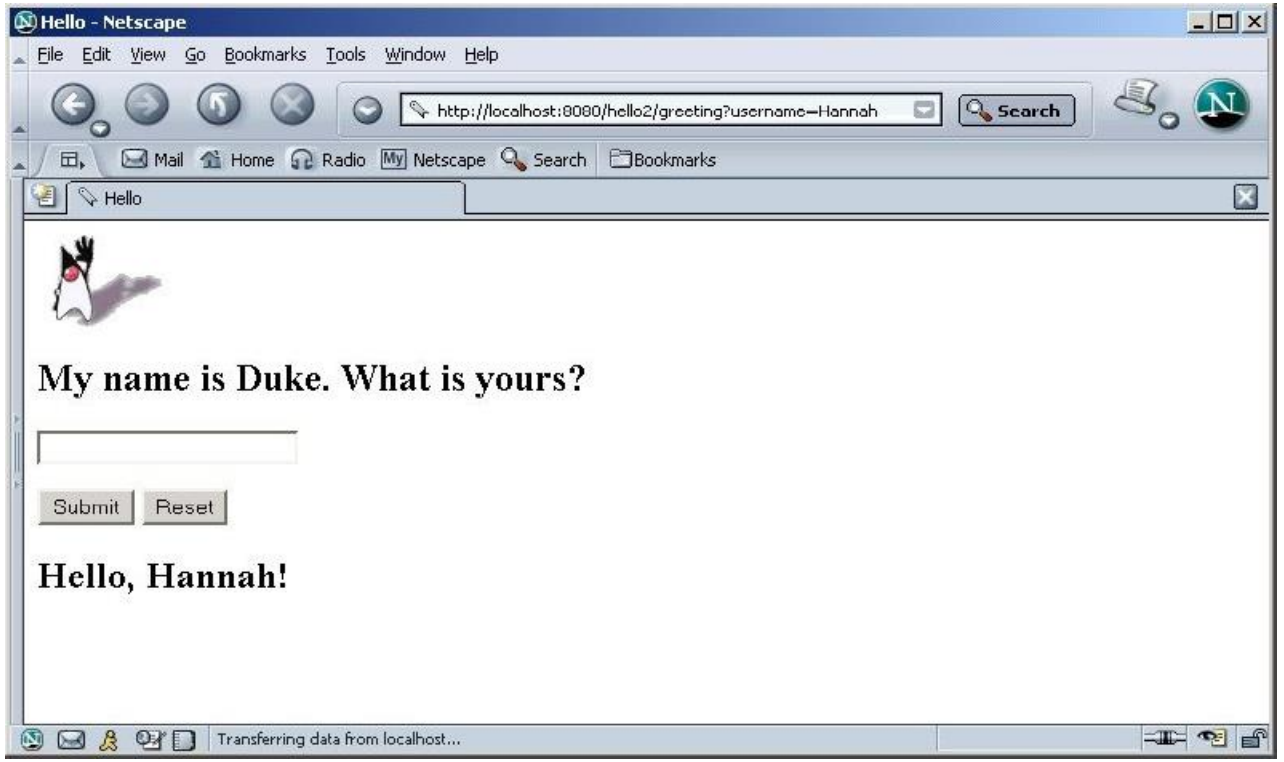
- There are two different ways to install/deploy Web application
  - *By asking Tomcat Manager* via sending a command to it (Tomcat Manager is just another Servlet app that is always running)
    - We need proper credential to perform this
      - This is why we need `build.properties` file with proper userid and password, otherwise we will experience `HTTP 401 error`
    - `ant install` for temporary deployment
    - `ant deploy` for permanent deployment
  - *By manually* copying files to Tomcat's webapps directory and then restarting Tomcat
    - Copying `*.war` file or unpacked directory to `<tomcat-install>/webapps/` directory manually and then restart Tomcat

6. Perform client access to Web application

- From a browser, go to URN of the Web application
  - E.g., `http://localhost:8080/hello2/greeting`



response.jsp



JSP scripting elements

- Let’s insert Java code into the servlet that will be generated from JSP page
- Minimise the usage of JSP scripting elements in our JSP pages if possible
- There are three forms
  - Expression: `<% =Expressions %>`
  - Scriptlets: `<% Code %>`
  - Declarations: `<% !Declarations %>`

Expressions

- During execution phase
  - Expression is evaluated and converted into a `String`
  - The `String` is then inserted into the servlet's output stream directly
  - Results in something like `out.println(expression)`
  - Can use predefined variables (implicit objects) within expression
- Format
  - `<% =Expression %>`
  - `<jsp:expression> Expression </jsp:expression>`
  - Semi-colons "*;*"are not allowed for expressions

Expressions: Example

- Display current time using `Date` class
  - `Current time: <% =new java.util.Date() %>`
- Display random number using `Math` class
  - `Random number: <% =Math.random() %>`
- Use implicit objects
  - `Your hostname: <% =request.getRemoteHost() %>`
  - `Your parameter: <% =request.getParameter("yourParameter") %>`
  - `Server: <% =application.getServerInfo() %>`
  - `Session ID: <% =session.getId() %>`

Scriptlets

- Used to insert arbitrary Java code into servlet's `jspService()` method
- Can do things expression alone cannot do
  - Setting response headers and status codes
  - Writing to a server log
  - Updating database
  - Executing code that contains loops, conditionals
- Can use predefined variables (implicit objects)
- Format
  - `<% Java code %>` or
  - `<jsp:scriptlet> Java </jsp:scriptlet>`

Scriptlets: Example

- Display query string

```
<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>
```

- Setting response type

```
<% response.setContentType("text/plain"); %>
```

Scriptlets: Loop example

```
<%
    Iterator i = cart.getItems().iterator();
    while (i.hasNext()) {
        ShoppingCartItem item = (ShoppingCartItem) i.next();
        BookDetails bd = (BookDetails) item.getItem();
%>
        <tr>
            <td align="right" bgcolor="#ffffff">
                <%=item.getQuantity()%>
            </td>
        <td bgcolor="#ffffaa">
            <strong><a href="
                <%=request.getContextPath()%>/bookdetails?bookId=
                <%=bd.getBookId()%>"><%=bd.getTitle()%></a>
            </strong>
        </td>
...
<%
    // End of while
    }
%>
```

Scriptlet: Example

Declarations

- Used to define variable or methods that get inserted into the main body of servlet class
  - Outside of `_jspService()` method
  - Implicit objects are not accessible to declarations
- Usually used with Expression or Sciptlets
- For initialisation and cleanup in JSP pages, use declarations to override `jspInit()` and `jspDestroy()` methods
- Format
  - `<% !method or variable declaration code %>`
  - `<jsp:declaration> method or variable declaration code </jsp:declaration>`

JSP Page fragment: Example

```
<H1>Some heading</H1>
<%!
  private String randomHeading() {
    return("<H2>" + Math.random() + "</H2>");
  }
%>
<% =randomHeading() %>
```

Resulting Servlet Code: Example

```
public class xxxx implements HttpJSPPage {
    private String randomHeading() {
        return("<H2>" + Math.random() + "</H2>");
    }
    public void _jspService(HttpServletRequest request,
                            HttpServletResponse response)
                        throws ServletException, IOException {
        response.setContentType("text/html");
        HttpSession session = request.getSession(true);
        JSPWriter out = response.getWriter();
        out.println("<H1>Some heading</H1>");
        out.println(randomHeading());
        ...
    }
...
}
```

Declaration: Example

```
<%!
   private BookDBAO bookDBAO;
   public void jspInit() {
   ...
   }
   public void jspDestroy() {
   ...
   }
%>
```

XML Syntax: Why?

- From JSP 1.2
- Examples
  - `<jsp:expression> Expression </jsp:expression>`
  - `<jsp:scriptlet> Java code </jsp:scriptlet>`
  - `<jsp:declaration> declaration code </jsp:declaration>`
- We can leverage
  - XML validation (via XML Schema)
  - Many other XML tools
  - Editor
  - Transformer
  - JAVA APIs

Including contents in a JSP Page

- Two mechanisms for including another Web resource in a JSP page
  - `include` directive
  - `jsp:include` element

`include` directive

- Is processed when the JSP page is translated into a servlet class
- Effect of the directive is to insert the text contained in another file - either static content or another JSP page - in the including JSP page
- Used to include banner content, copyright information, or any chunk of content that you might want to reuse in another page
- Syntax and Example
  - `<%@ include file="filename" %>`
  - `<%@ include file="banner.jsp" %>`

`jsp:include` element

- Is processed when a JSP page is executed
- Allows you to include either a static or dynamic resource in a JSP file
  - static: its content is inserted into the calling JSP file
  - dynamic: the request is sent to the included resource, the included page is executed, and then the result is included in the response from the calling JSP page
- Syntax and example
  - `<jsp:include page="includedPage"/>`
  - `<jsp:include page="date.jsp"/>`

Which one to use it?

- Use `include` directive if the file changes rarely
  - It is faster than `jsp:include`
- Use `jsp:include` for content that changes often
- Use `jsp:include` if which page to include cannot be decided until the main page is requested

Forwarding to another Web component

- Same mechanism as in Servlet
- Syntax
  - `<jsp:forward page="/main.jsp"/>`
- Original request object is provided to the target page via `jsp:parameter` element
  - `<jsp:forward page="...">`
  - `<jsp:param name="param1" value="value1"/>`
  - `</jsp:forward>`

Directives

- Directives are messages to the JSP container in order to affect overall structure of the servlet
- Do not produce output into the current output stream
- Syntax
  - `<%@ directive {attr=value}* %>`

Directives: Three types

- page: Communicate page dependent attributes and communicate these to the JSP container
  - `<%@ page import="java.util.*" %>`
- include: Used to include text and/or code at JSP page translation-time
  - `<%@ include file="header.html" %>`
- Taglib: Indicates a tag library that the JSP container should interpret
  - `<%@ taglib uri="mytags" prefix="codecamp" %>`

Page directives

- Give high-level information about the servlet that results from the JSP page.
- Control
  - Which classes are imported
    - `<%@ page import="java.util.*" %>`
  - What MIME type is generated
    - `<%@ page contentType="MIME-Type" %>`
  - How multithreading is handled
    - `<%@ page isThreadSafe="true" %> <%!--Default --%>`
    - `<%@ page isThreadSafe="false" %>`
  - What page handles unexpected errors
    - `<%@ page errorPage="errorpage.jsp" %>`

Implicit Objects

- A JSP page has access to certain implicit objects that are always available, without being declared first
- Created by container
- Corresponds to classes defined in Servlet

Implicit Objects (continued)

- request (HttpServletRequest)
- response (HttpServletResponse)
- session (HttpSession)
- application(ServletContext)
- out (of type JspWriter)
- config (ServletConfig)
- pageContext