# Java Server Pages and Model View Controller architecture

# Main idea of JSP

- separate display from processing, i.e., separate html from java
  - servlet is java with some html mixed in
    - a little plain java code
    - and lots of enclosed HTML
      - out.println(“ … \” … \” … “)
    - writing and maintaining quickly becomes a headache
  - Jsp is html with a little java mixed in
    - a little java code bracketed in
      - <% …. %> , etc
    - and lots of plain HTML
- There are two types of data in a JSP page:
  - Template Data: The static part, copied directly to response
    - static HTML
  - JSP Elements: The dynamic part, translated and executed by the container
    - typically generate additional HTML portions of the page
    - can execute arbitrary Java code

# Types of JSPelements:

- Declaration:
  - `<%!` Inserts instance variable and method declarations into servlet `%>`
  - `<%!` Java declaration statements `%>`
  - `<%! int count = 0; %>`
- Scriptlet:
  - `<%` Java statements `%>`
  - `<% count = count * 10; %>`
  - `<%` inserts into the service method `%>`
- Expression:
  - expects a Java expression, which it puts inside 'out.print' in service method
  - `<%=` Java expression `%>`
  - wraps it inside a print statement
  - `<%= ++count %>` becomes … `out.print( ++count );` … in service `method`
- Directive: page level operations for the translation phase.
  - `<%@ page %>` `(page, include, taglib)`

# Types of JSPelements:

- EL Expression: more convenient and powerful than a JSP expression
  - ${ } vs <%= %>

- Action: JSP "functions" that encapsulate some commonly used functionality
  - <c:foreach />
  - <jsp:setProperty name="" property="" />
  - <jsp:include file="side.html" />

- Comment:
  - <%-- jsp comment --%>
  - contrast with HTML comment
  - <!– Comment -->
  - HTML comments get sent to the browser (part of the HTML)
  - JSP elements all processed by container and do not appear in generated HTML

  - See example: W3D3-jspelements

# JSP Element – Directive

- message from a JSP page to the JSP container that controls processing of entire page.

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

```
<HTML>
<BODY>
Going to include hello.jsp...<BR>
<%@ include file="hello.jsp" %>
</BODY>
</HTML>
```

```
<%@ taglib uri="http://www.jspcentral.com/tags" prefix="public" %>
```

# JSP Elements – Page Directives

- The page Directive:
- The page directive is used to provide instructions to the container that pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.
- Following is the basic syntax of page directive:
  - `<%@ page attribute="value" %>`
- Example:
  - `<%@ page import="java.util.Date" %>`

# JSP Elements – Include Directives

- The include directive is used to includes a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code *include* directives anywhere in your JSP page.
- The general usage form of this directive is as follows:
  - `<%@ include file="relative url" >`
- The filename in the include directive is a relative URL. If you just specify a filename with no associated path, the JSP compiler assumes that the file is in the same directory as your JSP.

# JSP Elements – Taglib Directives

- The JavaServer Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.

- The taglib directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page.

- The taglib directive follows the following syntax:
  - `<%@ taglib uri="uri" prefix="prefixOfTag" >`
  - where the `uri` attribute value resolves to a location the container understands and the prefix attribute informs the container what bits of markup are custom actions.

# JSP Demo

- Inspect the JSP demo code
- Implement and run
- Insert the scripting elements shown
- Find the generated servlet
  - %userprofile%\AppData\Local\JetBrains\.IntelliJIdeayyyy.m\tomcat\yourprojectname\work\Catalina\localhost\projectartifact\org\apache\jsp
- Find the inserted JSP statements
- Find the implicit objects

# JSP Predefined Variables = *Implicit Objects*

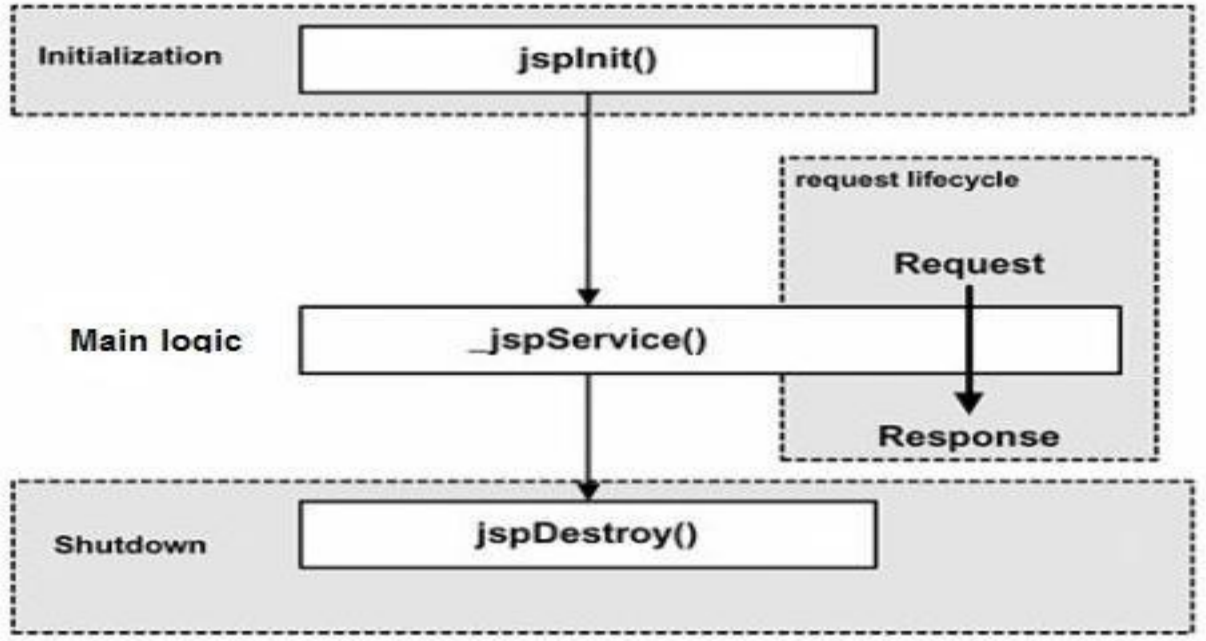
```
public void _jspService(final javax.servlet.http.HttpServletRequest request, final javax.servlet.http.HttpServletResponse response)
      throws java.io.IOException, javax.servlet.ServletException {

    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;


    try {
        response.setContentType("text/html; charset=ISO-8859-1");
        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;

       //...

    } catch (java.lang.Throwable t) {

    } finally {

    }
}
```

# JSP Predefined Variables = *Implicit Objects*

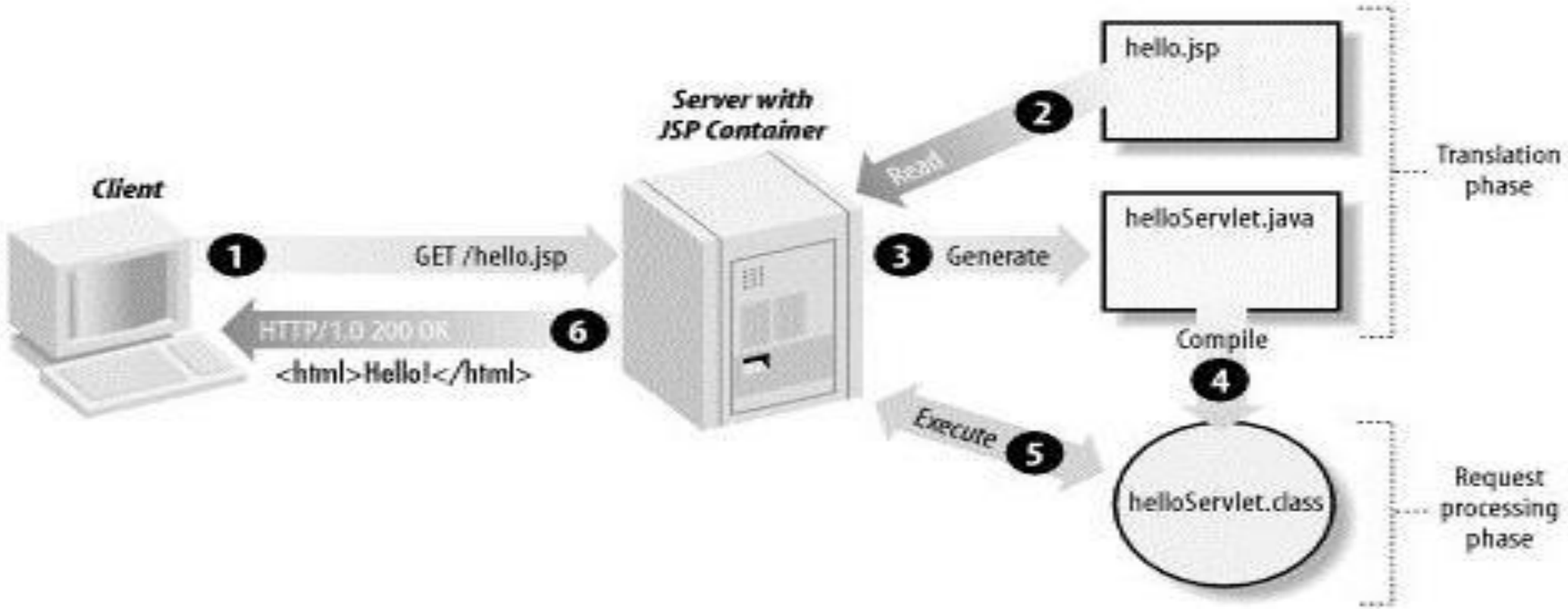
| Objects | Description |
|---|---|
| request | the HTTPServletRequest object associated with the request |
| response | the HTTPServletRespnse object associated with the response |
| out | the JspWriter used to send output to browser |
| session | the HttpSession object associated with the request |
| application | the ServletContext obtained via getServletConfig() |
| config | the ServletConfig object |
| pageContext | the PageContext object to get values from and store attributes into any of the other contexts (request, session, servletContext) |
| page | a synonym for this – the "this" of the jsp page's generated servlet; page has its own scope (elements of the page) |
| exception | The Exception object allows the exception data to be accessed by designated JSP. |

# JSP Life Cycle

- A JSP life cycle can be defined as the entire process from its creation till its destruction -- this is similar to the servlet life cycle with an additional step of compiling a JSP into servlet.
- The following are the steps of evolution of a JSP, which are typically initiated by a browser request for a jsp (e.g. index.jsp).
  - Compilation
  - Initialization
  - Execution
  - Cleanup



# JSP Life Cycle



# JSP Lifecycle

- LOADING PHASE

1. **JSP Page Translation**: A java servlet file is generated from the JSP source file. The container validates the syntactic correctness of the JSP pages and tag files. The container interprets the standard directives and actions, and generates java code for a servlet.
2. **JSP Page Compilation** : The generated java servlet file is compiled into a java servlet class.
3. **Class Loading**: The java servlet class that was compiled from the JSP source is loaded into the container.

- EXECUTION PHASE

1. **Initialization** jspInit() method is called immediately after the instance is created. It is called only once during JSP life cycle.
2. **_jspService()** This method is called for every request of this JSP during its life cycle.
3. **jspDestroy()** The servlet completes its purpose and submits itself to servlet heaven (garbage collection).

# JSP Compilation

- When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

- The compilation process involves three steps:
  - Parsing the JSP.
  - Turning the JSP into a servlet.
  - Compiling the servlet

  You can see the generated servlet for your own JSPs in the intellij tomcat server directory:

  %userprofile%\.IntelliJIdeayyyy.m\system\tomcat\yourtomcatname\work\Catalina\localhost\projectartifact\org\apache\jsp

# JSP Initialization

- When a container loads a JSP (after translation into a servlet if necessary) it invokes the jspInit() method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit() method (your code is placed in the method body in a declaration – declarations discussed later).

```
<%! public void jspInit() {
    //your initialization code
} %>
<html>…</html>
```

- Initialization is performed only once and as with the servlet init method. You can use jspInit to initialize database connections, open files, and create lookup tables.

# JSP Execution

- This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.
- Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the `_jspService()` method in the JSP.
- The `_jspService()` method takes an `HttpServletRequest` and an `HttpServletResponse` as its parameters as follows:

```
void _jspService(HttpServletRequest request,
  HttpServletResponse response) { // Service handling
  code...
}
```

# JSP Cleanup

- The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.

- The jspDestroy() method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.

- The jspDestroy() method has the following form:

```
public void jspDestroy(){
// Your cleanup code goes here.
}
```

# Main point 1

The web container generates a servlet from a JSP file the first time the JSP is requested from a web application. Since a JSP is essentially a servlet, one should understand servlets to effectively deal with JSPs.

**Science of Consciousness:** Actions in accord with fundamental levels of knowledge promote success in dealing with more expressed values. The most fundamental level of knowledge is pure knowledge, pure consciousness.

# Why You Shouldn't Use Java in a JSP

- You can do almost everything in JSP that you can do using Java.
  - But that does not mean you should do it.
- JSP is a technology that was designed for the *presentation layer*, also known as the view.
  - In most organizations, user interface developers are responsible for creating presentation layer.
    - These developers rarely have experience writing Java code, and providing them with the ability can be dangerous.
- In a well-structured, cleanly coded application, the presentation layer is separated from the business logic, which is likewise separated from the data persistence layer.
- It's actually possible to create JSPs that display dynamic content without single line of Java inside the JSP. That's our mission!!!

# Forwarding a Request from a Servlet to a JSP

- A typical pattern when combining Servlets and JSPs is to have the Servlet accept the request, do any business logic processing and data storage or retrieval necessary, prepare a model that can easily be used in JSP, and then forward request to the JSP.

```
request.getRequestDispatcher("/WEB-
INF/jsp/welcome.jsp").forward(request, response);
```

# EL (Expression Language)

- Java code inside of JSP is not a good idea, discouraged.
- easier way to display data and perform simple operations without Java
  - something easily read,
  - familiar to both Java developers and UI developers,
  - simple set of rules and operators to make data access and manipulation easier.
- Expression Language (EL) developed to support rendering of data on JSP pages without scriptlets, declarations, or expressions.
  - inspired by ECMAScript
  - When the JSP compiler sees `${expr}`, it generates code to evaluate the expression and inserts the value of expression at that spot on the JSP page

# EL (Expression Language)

- recall JSP "expression"
  - <%= myMovieList.get(i) %>
  - evaluates the expression and writes it out to the HTML page at its location on the page
  - <%= ((Person) request.getAttribute("pers")).getDog().getName() %>
  - assumes request.setAttribute("pers", aPerson) in request scope;

- recall that attributes are where web app stores model values
  - values computed in model and then accessed in page for display
    - no model code on the JSP page, just value insertion

- EL simplifies JSP expression syntax
  - ${pers.dog.name}

# JSP Expression vs EL

```
<%=((Person) request.getAttribute("pers")).getDog().getName()%>
```

With EL it becomes:

```
${pers.dog.name}
```

# High level description of EL

${something}

- container evaluates this as follows

- checks page scope for an attribute named "something",
  - if found use it.
- otherwise check request scope for an attribute named "something",
  - if found use it.
- otherwise check session scope for an attribute named "something",
  - if found use it
- otherwise check application scope for an attribute named "something",
  - if found use it.
- otherwise ignore the expression.

# More detailed description

${firstThing}

- if firstThing is not an implicit EL object  search page, request, session and application scopes until attribute "firstThing" is found

  ${firstThing.secondThing}

  - if firstThing is an attributed that returns a bean then secondThing is a property of the bean
  - if firstThing is an attributed that returns map then secondThing is a key of the map

  ${firstThing[secondThing]}

  - if firstThing is a bean then secondThing is a property of the bean
  - if firstThing is a map then secondThing is a key of the map
  - if firstThing is a List then secondThing is an index into the List

# Example of EL Bracket notation

- in Servlet:
  - `String[] fruit= ["Banana", "Orange", "Apple"];`
  - `request.setAttribute("myFruitList", fruit);`

- in JSP:
  - `${myFruitList[0]}` // Banana
  - Note use of myFruitList[0] versus fruit[0]

# EL is “null friendly”

- scripting languages try to fail without glaring error messages
  - helpful when used by end users
  - not so good for developers, need to be aware
  - Important for careful testing and checks
- if EL cannot find a value for the attribute it ignores it
  - caution
  - no warning or error message
- in arithmetic, treats null value as 0
  - could be a surprise
  - ${100 + myBankAccount.balance} → $100?? !!
    - “What happened to the bank account?”
- in logical expressions, nulls become “false”
- more surprises ??

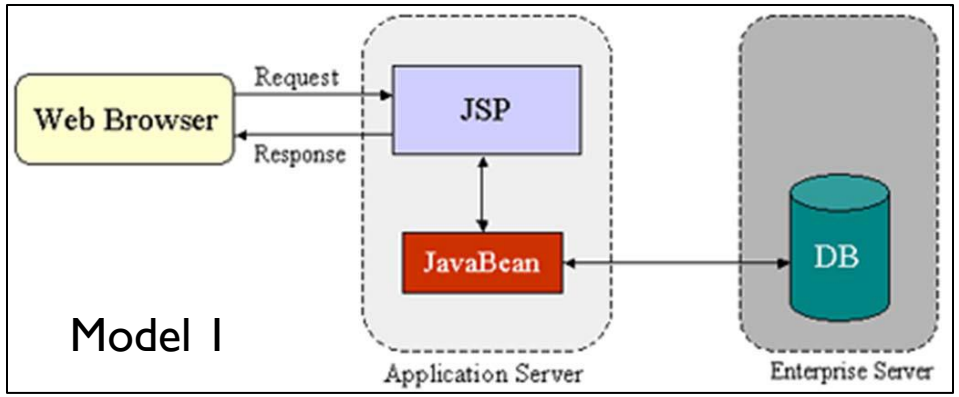
- See w3d3-expression demo

# Main point 2

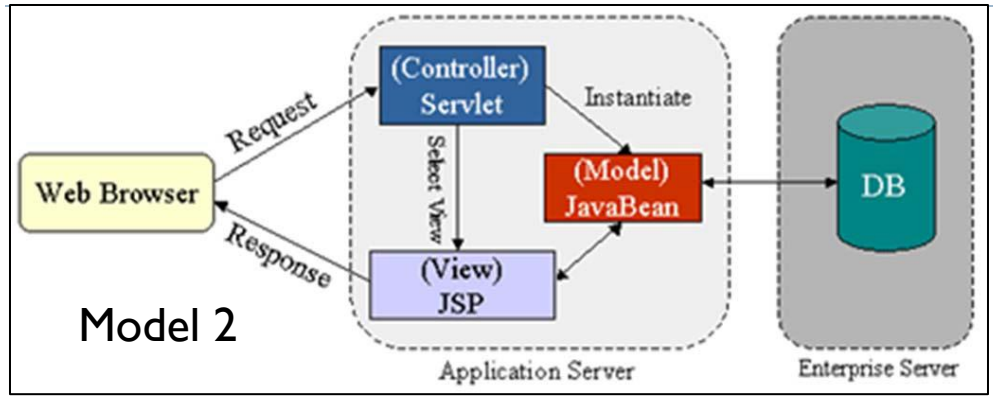
An EL expression is a compact expression of a systematic evaluation of the page, request, session and application scopes.

**Science of Consciousness:** The laws of nature are compact expressions that control the infinite diversity of the material universe. Our actions are more in accord with the laws of nature when we experience their basis in pure consciousness.

# JSP Model 1 and Model 2 Architectures




- Simple architecture
- For small applications
- Issues
  - Pages are coupled, need to know about each other.
  - Expensive to provide another presentation for same application.
  - Java code in HTML

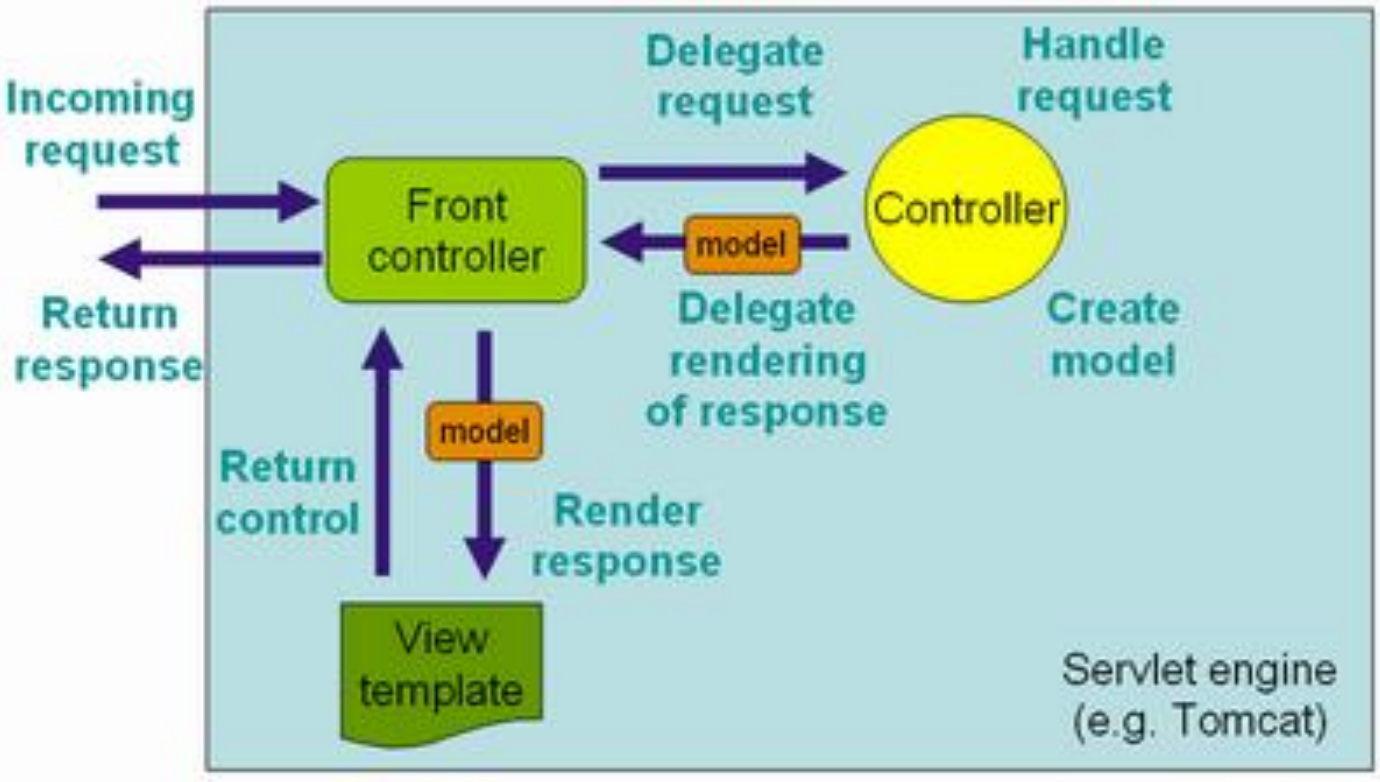



- More complex architecture
- For medium and large applications
- Advantages
- Each team can work on different
- pages. Easy to understand each page.
- - Separation of presentation from control, making it easy to change one without affecting the other.
- - no Java code in HTML

# Model 1 vs 2 architecture (cont)

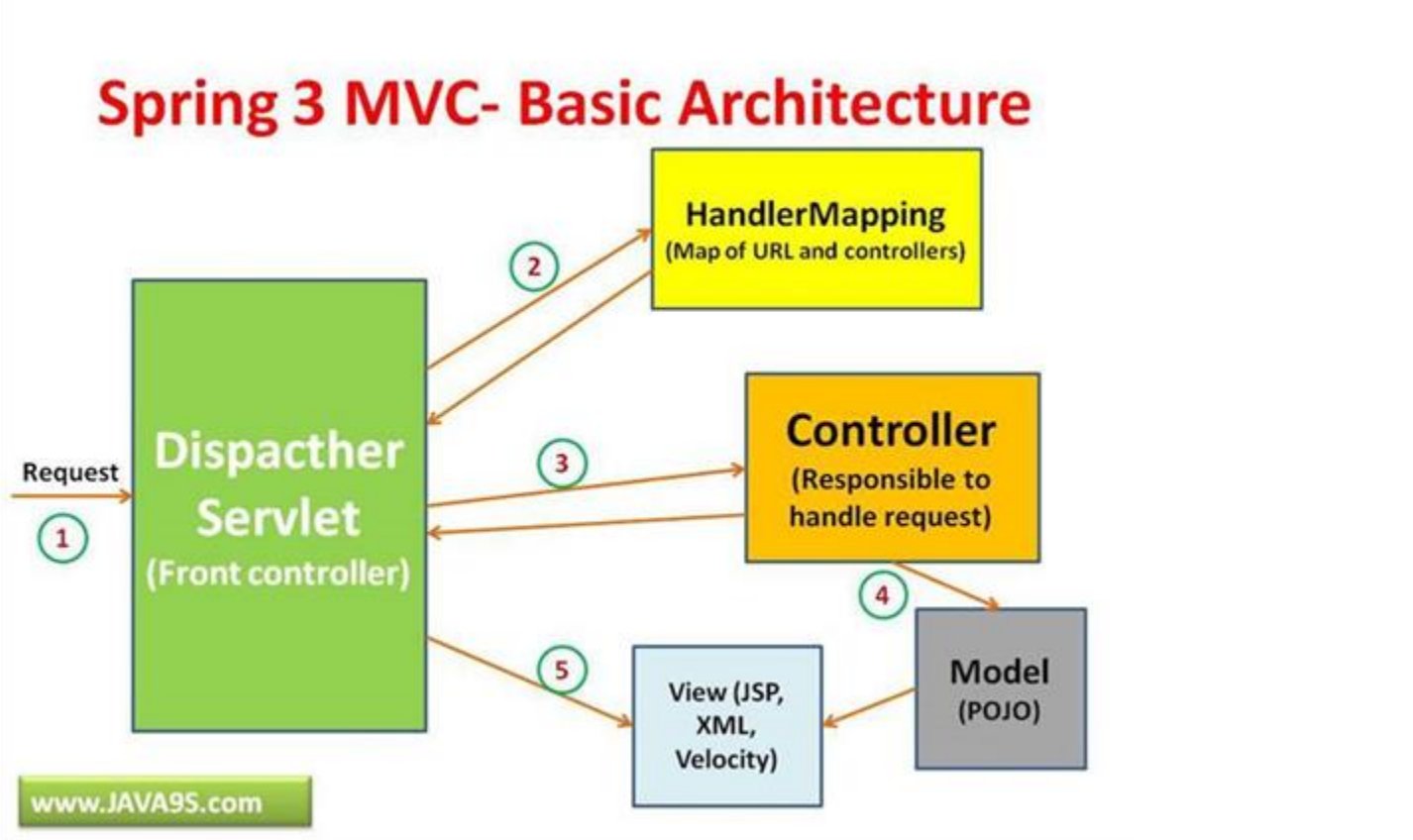
- Model 1:
  - JSP acts as both controller and view
  - JavaBean (POJO) is model
  - Problems:
    - JSPs became very complicated,
    - JSPs contains page navigation logic,
    - JSPs perform validation and conversion of string parameters
- Model 2 – Spring MVC (Front Controller)
  - Have a single controller servlet that takes requests
  - Gets request parameters
  - Converts and validates them
  - Calls object with business logic to do processing
  - Forwards results to jsp page for display
  - NO SCRIPTING on JSP pages
- http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html

# SpringMVC Architecture




- http://stackoverflow.com/questions/20314098/spring-mvc-without-servlets
- SpringMVC site

# SpringMVC Architecture




- http://java9s.com/spring-3-mvc/spring-3-mvc-introduction-spring-3-mvc-architecture

# Demo: Model 1

```
<h1>Time JSP</h1>

<%
   //the parameter "zone" shall be equal to a number between 0 and 24 (inclusive)
   TimeZone timeZone = TimeZone.getDefault(); //returns the default TimeZone  if
(request.getParameterValues("zone") != null)
   /*
   since we're basing our time from GMT, we'll set our Locale to
   Brittania, and get a Calendar.
   */
   Calendar myCalendar = Calendar.getInstance(timeZone, Locale.UK);
%>

The current time is:
<%=myCalendar.get(Calendar.HOUR_OF_DAY)%>:
<%=myCalendar.get(Calendar.MINUTE)%>:
<%=myCalendar.get(Calendar.SECOND)%>
```

# Demo: Model 2 - MVC

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/html");

    String name = request.getParameter("name");
    String password = request.getParameter("password");

    User bean = new User();
    bean.setName(name);
    bean.setPassword(password);
    request.setAttribute("user", bean);

    boolean status = bean.validate();

    if (status) {
        RequestDispatcher rd = request.getRequestDispatcher("login-success.jsp");
        rd.forward(request, response);
    } else {
        request.setAttribute("errormsg", "Sorry, your username and password are not
correct!");
        RequestDispatcher rd = request.getRequestDispatcher("index.jsp");
        rd.forward(request, response);
    }
}
```

```
${errormsg }
<form action="ControllerServlet"
method="post">
   Name:<input type="text"
name="name"><br />
   Password:<input type="password"
name="password"><br />
   <input type="submit" value="login">
</form>
```

# Main point 3

When you use JSP pages according to a Model 2 architecture, there is a servlet that acts as a controller (process of knowing) that sets attribute values based on computations and results from a business model (knower), then dispatches the request to the servlet generated by the JSP page (known). The JSP servlet then retrieves the attribute values and inserts them into the designated places in the HTML being sent to the browser.

**Science of Consciousness:** Complete knowledge is the wholeness of knower, known, and process of knowing.

# CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

JSP: Knower, Known, and Process of Knowing

1. Java Server Pages make it easy for HTML authors to interact with servlets.

2. The JSP Expression Language is designed to promote easy access to dynamic content contained in the MVC data model.

---

**3. Transcendental consciousness** is the knower knowing Himself or Herself.

4. **Impulses within the Transcendental Field** Model 2 architectures are successful with large systems because they maintain the integrity of the knower (model), known (view), and process of knowing (controller). Similarly, when developers maintain the integrity of their own Self then their actions will be successful even when developing large complex systems under demanding conditions.

5. **Wholeness moving within itself:** In unity consciousness, one appreciates that knower, known, and process of knowing are all expressions of the same underlying unified field of pure intelligence, one's own Self, pure bliss consciousness.