Java Technologies
Web Components

“Classical” Web Components

- Servlets
- Web Filters
- Web Listeners
  - Asynchronous Servlets
- Java Beans
- *Java Server Pages (JSP)*
- *Custom Tag Libraries (CTL)*
- *Java Standard Tag Library (JSTL)*

Java Server Pages (JSP)

“Those who do not remember their past are condemned to repeat their mistakes”

George Santayana

JSP - The Context

- **The Presentation Layer of a Web App**
  - the graphical (web) user interface
  - frequent design changes
  - usually, dynamically generated HTML pages
- Should we use servlets? → **No (not directly)**
  - difficult to develop and maintain (the view)
  - lack of flexibility
  - lack of role separation: design → *designer*

Example: HelloServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Hello extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
              throws IOException {

    response.setContentType("text/html");

    PrintWriter out = new PrintWriter(response.getWriter());
    out.println("<html>" +
      "<head><title>Welcome!</title></head>" +
      "<body>" +
      "<h1><font color=\"red\"> Welcome! </font></h1>" +
      "<h2>Current date and time:" + new java.util.Date() + " </h2>" +
      "</body>" + "</html>");
    out.close();
  }
}
```

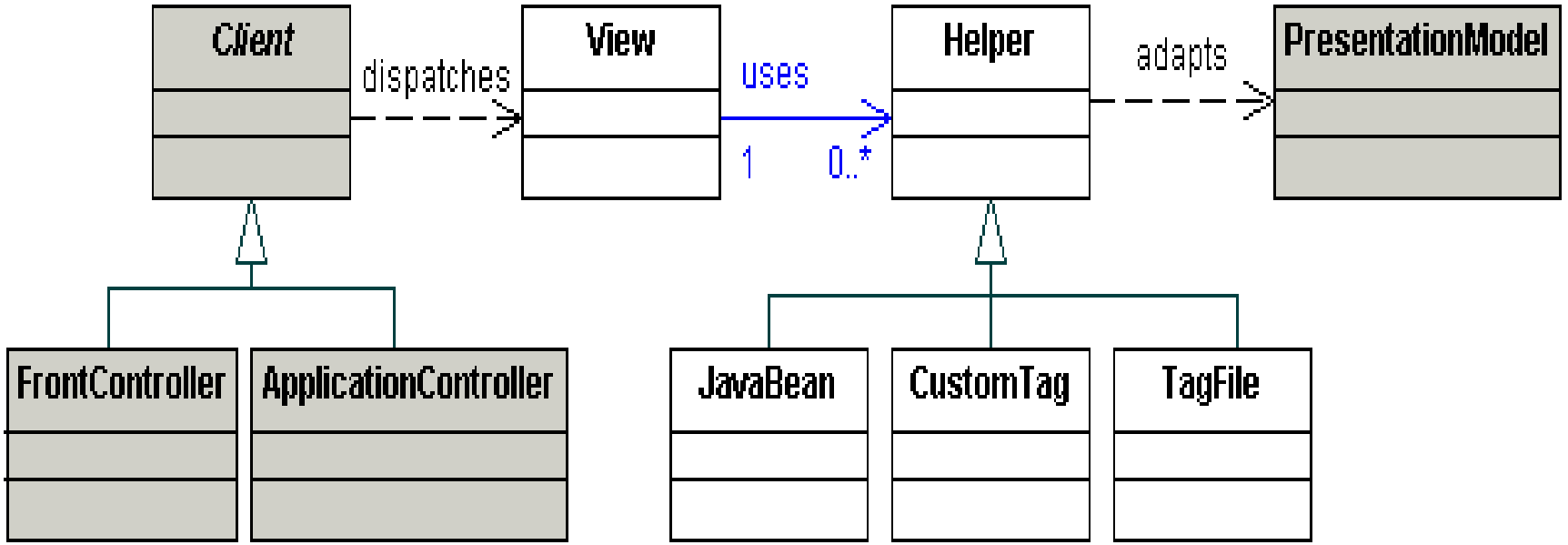
Example: hello.jsp

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1><font color="red"> Welcome! </font></h1>
  <h2>Current date and time: <%= new java.util.Date() %> </h2>
</body>
</html>
```

The Concept

Standard component to create the presentation,
that conforms to

View – Helper design pattern



JavaServer Pages (JSP)

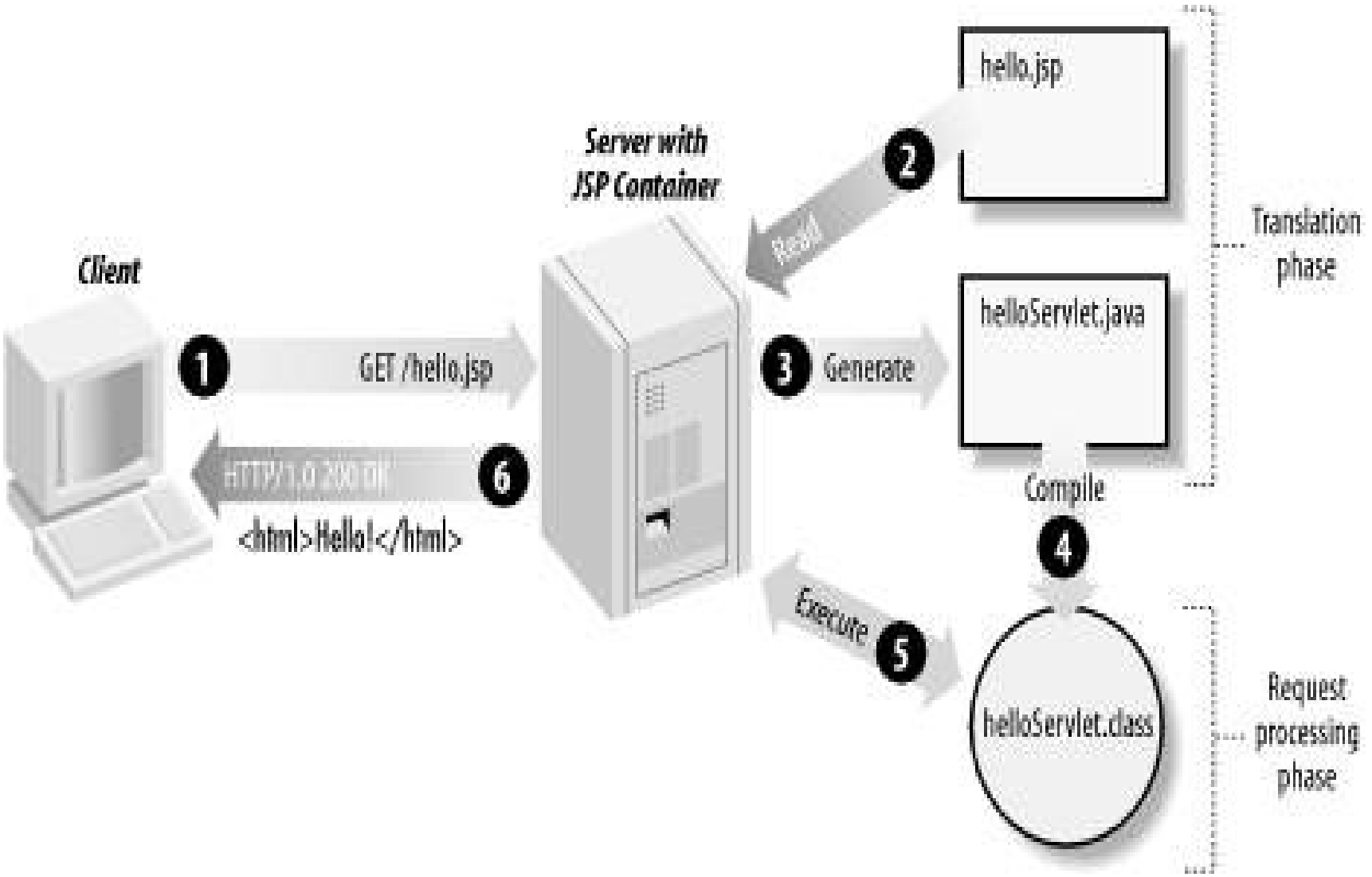
- The “classical” solution to **create web content easily** that has both static and dynamic components → Web pages
- Provides **a natural, declarative, presentation oriented** approach to creating static content → Templates
- **Are based on** and benefit from all the dynamic capabilities of **Java Servlet technology**

  → Every JSP is actually a servlet
- A first (small) step towards the role separation

The main features of JSP

- A JSP page is a text document that contains:
  - **static data (template)** (HTML, SVG, WML, etc.)
  - **JSP elements**, which construct the content
- JSP elements may be:
  - **JSP tags**
  - **scriptles** (code sequences written in Java...)
- JSP uses an **expression language** for accessing server-side objects
- The source files may be **.jsp, .jspf, .jspx**

The Life Cycle of a JSP Page



Translation Phase

```
//Example taken from Apache Tomcat application server
//This is the source file of the servlet generated for hello.jsp

package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class hello_jsp
    extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
  ...
  public void _jspService(HttpServletRequest request,
                          HttpServletResponse response)
              throws java.io.IOException, ServletException {
  ...
  out.write("<HTML>");
  ...
  out.write("</HTML>");
  }
}
```

implements javax.servlet.Servlet

JSP Syntax

```
<%-- Hello World example, not again ...  --%>
```

JSP Comment

```
<%@ page language="java" contentType="text/html;
    pageEncoding="UTF-8"%>

<html>
<head> <title>First JSP</title> </head>

<%@ page import="java.util.Date" %>
<%@ include file="header.html" %>

<%! String message = "Hello World"; %>

<body>
<%
   for(int i=1; i<=4; i++) {
     out.println("<H" + i + ">" + message + "</H" + i + ">");
   }
%>
<!-- Page generated at: <%= new Date() %> -->
</body>
</html>
```

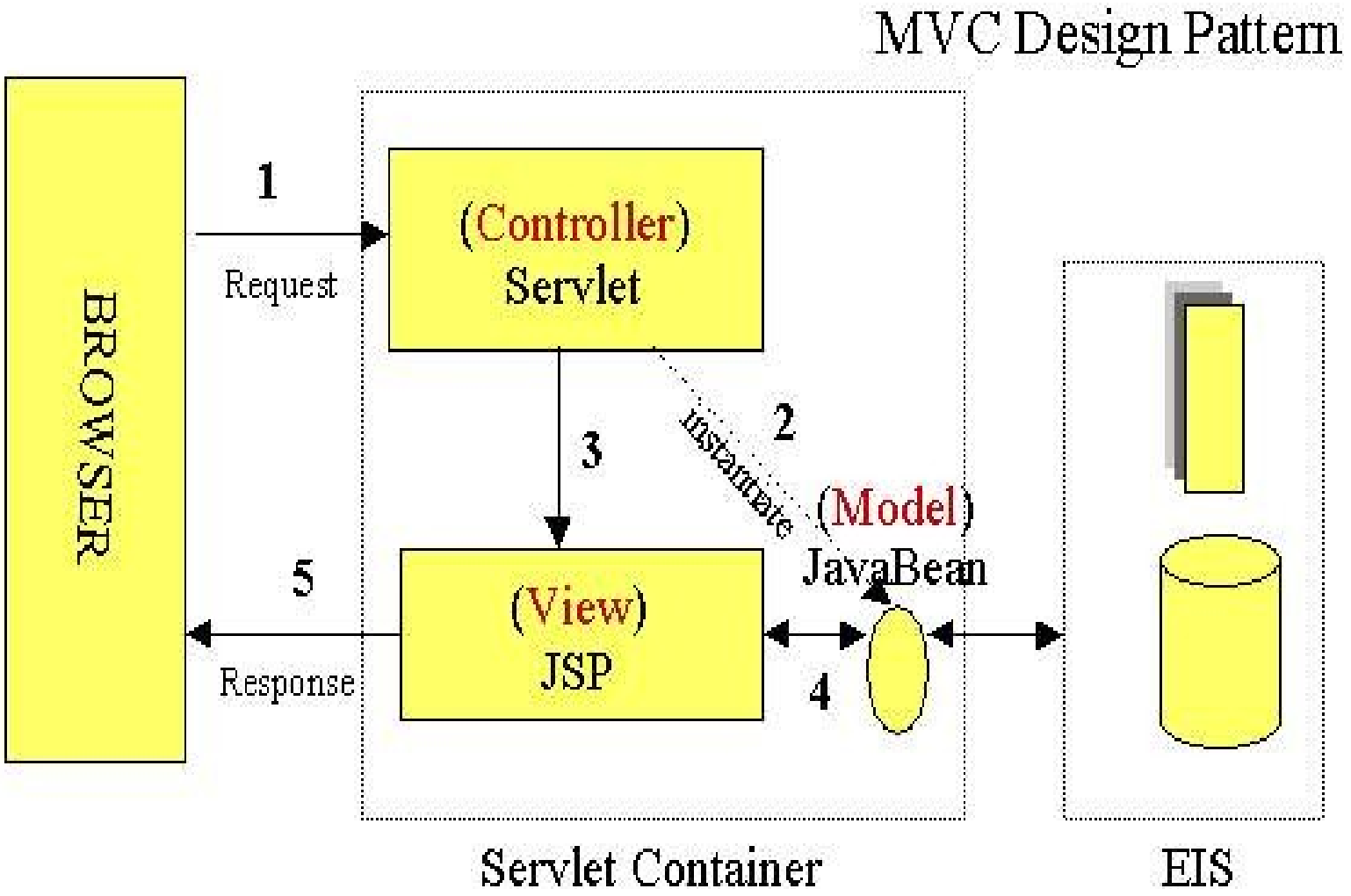
JSP Directives

JSP Declaration

JSP Scriptlet

JSP Expression

Model – View – Controller



Custom Tag Libraries (CTL)

“Code reuse is the Holy Grail of Software Engineering.”

Douglas Crockford

CTL - The Context

- JavaServer Pages offer a standard solution to **create web content** dynamically using **JSP elements**: JSP tags, scriptles, etc.
- JSP are used to create **the view** of the application (the presentation layer)
- Presentation → Designer, ~~not Programmer~~
- **How can we generate dynamic content without writing Java code?**
  - The first step: **JSP Standard Actions**

    (jsp:useBean, jsp:forward, jsp:include, etc)

The Concept

We need a component that:

- **Encapsulates** various types of non-standard dynamic functionalities such as:
  - generating an HTML table with available products
  - extracting some data from an XML document, etc.
- Can be used inside JSP in a **similar** manner **to the standard actions**
  - the programmer writes the functionality
  - the designer acceses the functionality in a declarative fashion (using a tag)
- Promotes code **reusability** (libraries)

Separation of Concerns (Soc)

- A design principle for **separating a system into distinct sections**, such that:
  - each section addresses a separate concern
  - the overlapping should be minimal
- Edsger W. Dijkstra: "*On the role of scientific thought*" (1974)
  - "focusing one's attention upon some aspect"
  - “the only available technique for effective ordering of one's thoughts”

Custom Tags in JSP

- A custom tag is a **user-defined JSP element**
- The object that implements a custom tag is called a **tag handler**
  - *class* (programmatically)
  - *tag file* (JSP fragment)
- A tag is invoked in a JSP file using XML syntax:

```
<prefix:tagName>
   Body
</prefix:tagName>
```

- When a JSP is translated into a servlet, the tag is converted to operations on the tag handler.

Custom Tag Features

- Customized by means of attributes passed from the calling page
- Pass variables back to the calling page
- Access all the objects available to JSP pages
- Communicate with each other
- Be nested within one another and communicate by means of private variables
- Distributed in a tag library

Creating a Tag Handler

```
public class HelloTagHandler extends SimpleTagSupport {

    /**
     * Called by the container to invoke this tag. The
       implementation of this method is provided by the tag
       library developer, and handles all tag processing
     */

    @Override
    public void doTag() throws JspException, IOException {

        // Create dynamic content
        JspWriter out = getJspContext().getOut();
        out.print("Hello World from Infoiasi!");
    }

}
```

The Tag Library Descriptor (TLD)

Defines a mapping between tag handlers (classes) and tag names

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <short-name>mylibrary</short-name>

  <uri>/WEB-INF/tlds/mylibrary</uri>

  <tag>
    <name>hello</name>
    <tag-class>HelloTagHandler</tag-class>
    <description> Displays the Hello World message (again) </description>
    <body-content>empty</body-content>
  </tag>

</taglib>
```

Using the Custom Tag

```
<b>Somewhere, inside a JSP</b>

<p>


<%@ taglib uri="/WEB-INF/tlds/mylibrary"
            prefix="say" %>


<say:hello/>
```

Use the *taglib directive* to specify the tag library

Use the custom tag

Java Standard Tag Libraries (JSTL)

"I believe in standardizing tag libraries :)
I do not believe in standardizing human beings".
Albert Einstein

JSTL - The Context

- JSP are used to create **the view**
- Custom tags are **user-defined JSP elements:**
  - encapsulate functionalities
  - promote reusability and role separation
  - implemented using
    - **classes (handlers):** by programmers
    - **JSP → tag files: by designers...**
- **How can we generate dynamic content in a tag file without writing Java code?**

The Concept

We need a solution to:

- **Allow the designer to implement custom tags** in the form of tag files.
- **Simplify** the creation of JSP pages
  - accessing the model (data stored in beans)
  - controlling the execution of a page
  - etc.
- **Standardize** the design elements.
- **Optimize** the execution of JSP pages.

Example

In a JSP (page or tag file), we verify if the user has a specific role, *using a scriptlet*:

```
<%
  User user = (User) session.getAttribute("user");
  if (user.getRole().equals("member"))) {
%>
  <p>Welcome, member!</p>
<% } else { %>
  <p>Welcome, guest!</p>
<% } %>
```

awful...

It is difficult to provide a custom tag handler for every situation.

Example (cont.)

A more appealing solution would be:

```
<c:choose>
  <c:when test="${user.role == 'member'}">
    <p>Welcome, member!</p>
  </c:when>
  <c:otherwise>
    <p>Welcome, guest!</p>
  </c:otherwise>
</c:choose>
```

Expression Languge

Standard Tags

Expression Language (EL)

- **Access application data** stored in JavaBeans components
- **Create expressions** *arithmetic* and *logical* in a intuitive manner:
  - `${ (10 % 5 == 0) or ( 2 > 1) ?  "yes" : 'no' }`
  - `${ header["user-agent"] }`
- **No programming** skills required
- When the JSP compiler “sees” the **${}** form in an attribute, it generates code to evaluate the expression and substitues the resulting value.

EL Syntax

- **Literals**: true, false, null, “a”, 'b', 123, 9.99
- **Variables**
  - `PageContext.findAttribute(`*`variable`*`)`
  - `page → request → session → application`
  - **`variable.property`** `or` **`variable[property]`**
- **Operators**: as usual plus:
  - *eq, ne, ge, le, lt, gt, mod, div, and, or, not, empty*
- **Implicit Objects**:
  - *param, request, response, session, servletContext*
  - *pageScope, requestScope, sessionScope, applicationScope*
  - *header, cookie, initParam, etc.*

Standard Tag Libraries (JSTL)

- **A collection of useful JSP tags** which encapsulates functionalities common to many JSP applications.
- JSTL has support for:
  - **Core Tags**
  - **Formatting tags**
  - **XML tags**
  - **SQL tags**
  - **JSTL Functions**

Core Tags (c)

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

Core tags are the most frequently used JSTL tags.

```
<c:set var="message" scope="page" value="Hello JSTL!"/>

<c:out value="${message}" default="Hello World!"/>
```

```
<c:forEach var="item" items="${sessionScope.cart.items}">
    <c:out value="${item}"/> <br/>
</c:forEach>
```

```
<c:import url="someFile.csv" var="content" />
<c:forTokens var="item" items="${content}" delims=",">
    <c:out value="${item}"/> <br/>
</c:forTokens>
```

```
<c:if test="${empty session.user}">
    <c:redirect url="login.jsp"/>
</c:if>
```

Formatting Tags (fmt)

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

Used to format and display text, the date, the time, and numbers for internationalized Web sites.

```
<jsp:useBean id="now" class="java.util.Date" />
<fmt:setLocale value="ro-RO" />
<fmt:formatDate value="${now}"
                type="both"
                dateStyle="full" timeStyle="full"/>
```

```
<fmt:setLocale value="ro"/>
<fmt:setBundle basename="somepackage.Messages"
               var="msg"
               scope="page"/>
<fmt:message key="hello"
             bundle="${msg}"/>
```

I18N *L10N*

XML Tags (x)

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

Create and manipulate XML documents: parsing XML, transforming XML data, and flow control based on XPath expressions.

```
<c:import url="agenda.xml" var="xml" />
<x:parse doc="${xml}" var="agenda" scope="application" />
<x:set var="friend"
       select="$agenda/friends/person[@id=$param:personId]" />
<x:out select="$friend/name"/></h2>
```

```
<x:forEach var="friend"
           select="$agenda/friends/person" />
  <x:out select="$friend/name" />
</x:forEach>
```

```
<c:import url="agenda.xml" var="xml" />
<c:import url="style.xsl"  var="style" />
<x:transform source="${xml}" xslt="${style}"/>
```

SQL Tags (sql)

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

Tags for interacting with relational databases: connect, read (query), update, delete.

```
<sql:setDataSource var="timtable"
    url="jdbc:sybase:Tds:localhost:2638?ServiceName=TimetableDB"
    driver="com.sybase.jdbc4.jdbc.SybDataSource"
    user="DBA" password="sql"/>
<sql:setDataSource var="timetabe"
    dataSource="jdbc/TimetableDB" />
```

```
<c:set var="roomId" value="C401"/>
<sql:query var="rooms" dataSource="${timetable}"
           sql="select * from rooms where code = ?" >
   <sql:param value="${roomId}" />
</sql:query>
<c:out value="${rooms.rowCount}"/>
```

```
<sql:update dataSource="${timetable}">
  update rooms set address='la subsol' where code='C112'
</sql:update>
```

Alternatives to JSP

“The absence of alternatives clears the mind marvelously.”

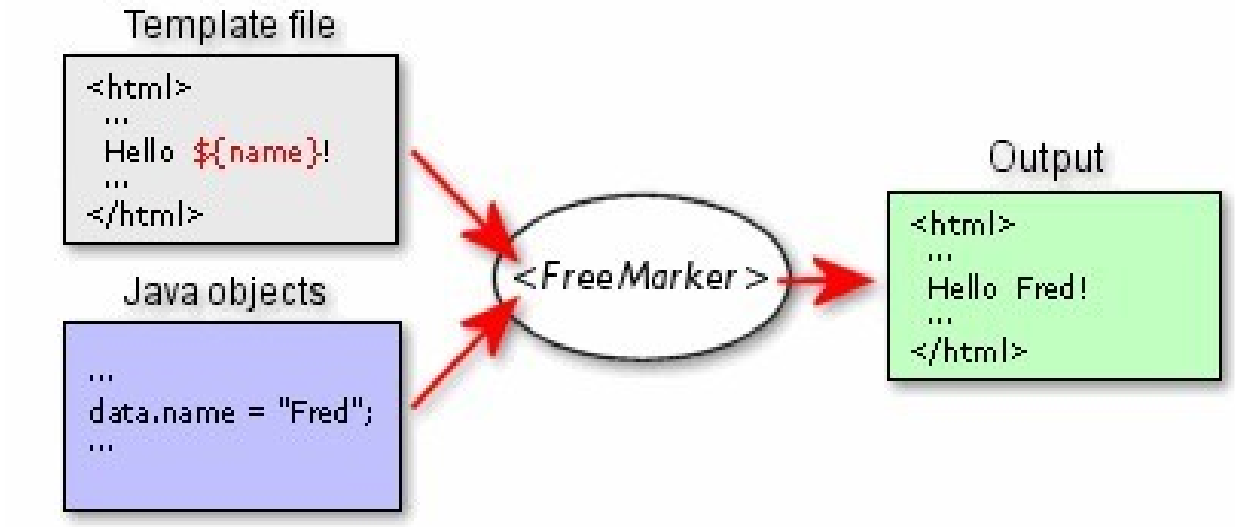
Henry Kissinger

Template Engines

- The Context → **generate documents**
  - reports, emails, sql scripts, source files, etc.
  - web pages

- We need a **generic** solution to:
  - specify the **template**
  - specify the **data**
  - **generate** the document

MVC Frameworks



- Template Language → View
- Data (Beans) → Model
- Runtime Engine → Controller

Example: The Template File

Static text + Template Languge (simple syntax)

```
<html>
  <head> <title> Welcome </title> </head>
  <body>

    <h1>Welcome ${user}!</h1>

    <p>Our latest product:
    <a href="${latestProduct.url}">${latestProduct.name}</a>!

    <p>All the products:
    <#list products as product>
      <li>${product.name}, ${product.price}

      <#if product.stock == 0>
        Empty stock!
      <#if>

    </#list>
  </body>
</html>
```

Example: The Model

```
(root)
  |
  +- user = "Big Joe"
  |
  +- latestProduct
     |
     +- url = "products/greenmouse.html"
     |
     +- name = "green mouse"
    ...
```

```
Map<String, Object> data = new HashMap<String,Object>();
User user = new User("Big Joe");
data.put("user", user);

Product product = new Product();
product.setName("green mouse");
product.setUrl("products/greenmouse.html");
data.put("latestProduct", product);

data.put("today", new java.util.Date());
```

Merging the View and the Model

```
// Initialization: where are my templates?
Configuration cfg = new Configuration();
cfg.setDirectoryForTemplateLoading(new File("someFolder"));
// set global variables if you need to
cfg.setSharedVariable("version", "0.0.1 beta");

// Prepare the data
Map<String,Object> data = ... ;

// Choose a template
Template template = cfg.getTemplate("someTemplate.ftl");

// Specify the output stream
String filename = "someFile.html"
Writer out = new BufferedWriter(new FileWriter(filename));

//Do it: process, merge, etc.
template.process(data, out);

out.close();
```

Using FreeMarker in a Web App

- Register the FreeMarker Servlet

```
<servlet>
  <servlet-name>freemarker</servlet-name>
  <servlet-class>
    freemarker.ext.servlet.FreemarkerServlet
  </servlet-class>
  <init-param>
    <param-name>TemplatePath</param-name>
    <param-value>/</param-value>
  </init-param> ...
</servlet>
```

- Map the requests

```
<servlet-mapping>
  <servlet-name>freemarker</servlet-name>
  <url-pattern>*.ftl</url-pattern>
</servlet-mapping>
```

- Any request (.ftl) goes to the servlet

```
http://localhost:8080/myapp/products.ftl
```