



ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Web development with Java

Outline

1. **Servlet**
2. JSP – Java Server Page
3. Java Beans
4. ORM (Object Relational Mapping)

Free Servlet and JSP Engines (Servlet/JSP Containers)

❖ Apache Tomcat

- <http://jakarta.apache.org/tomcat/>

❖ IDE: NetBeans, Eclipse

- <https://netbeans.org/>
- <https://eclipse.org/>



❖ Some Tutorials:

- Creating Servlet in Netbeans: <http://www.studytonight.com/servlet/creating-servlet-in-netbeans.php>
- Creating Java Servlets With NetBeans: <http://www.higherpass.com/java/tutorials/creating-java-servlets-with-netbeans/>
- Java Servlet Example: <http://w3processing.com/index.php?subMenuId=170>
- Developing JSPs and Servlets with Netbeans: <http://supportweb.cs.bham.ac.uk/documentation/java/servlets/netbeans-webapps/>

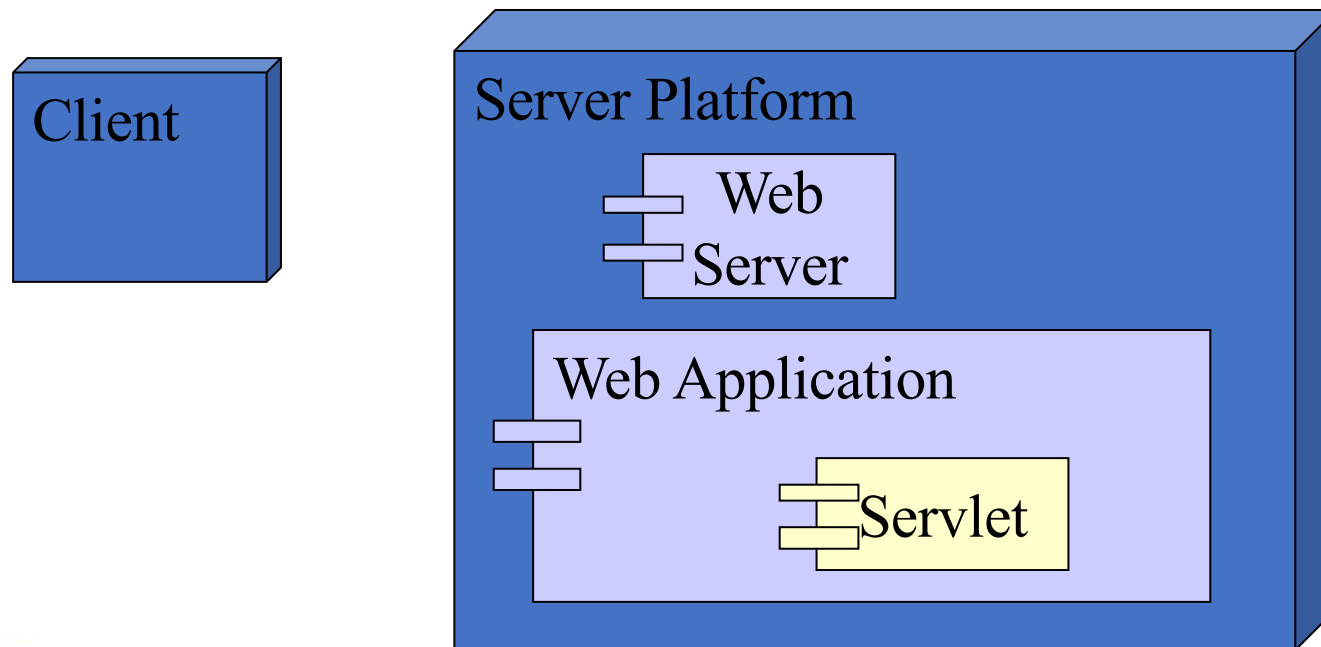
Compiling and Invoking Servlets

- ❖ Put your servlet classes in proper location
 - Locations vary from server to server. E.g.,
 - `tomcat_install_dir/webapps/ROOT/WEB-INF/classes`
- ❖ Invoke your servlets (HTTP request)
 - `http://localhost/servlet/ServletName`
 - Custom URL-to-servlet mapping (via `web.xml`)



Java Servlets

- ❖ A **servlet** is a Java program that is invoked by a web server in response to a request

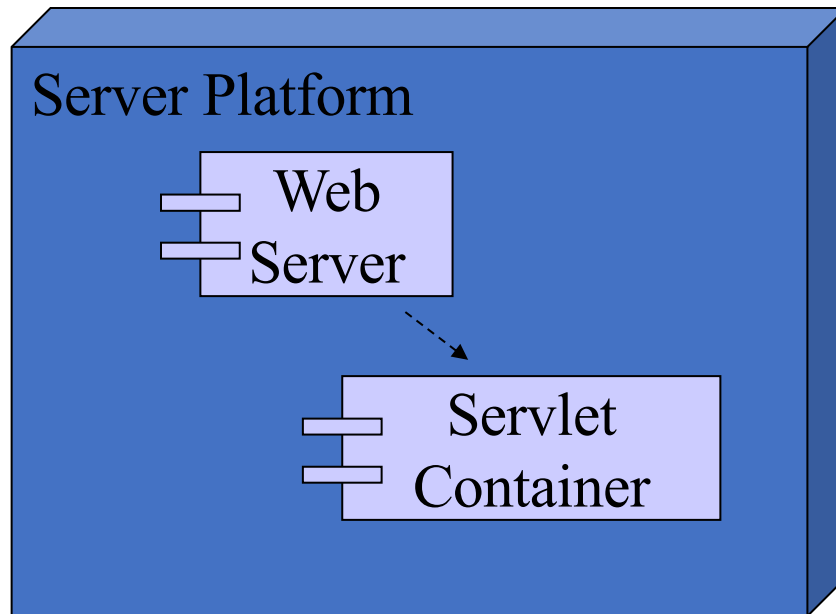


Java Servlets

- ❖ Together with web pages and other components, servlets constitute part of a web application
- ❖ Servlets can
 - create dynamic (HTML) content in response to a request
 - handle user input, such as from HTML forms
 - access databases, files, and other system resources
 - perform any computation required by an application

Java Servlets

- ❖ Servlets are hosted by a **servlet container**, such as Apache Tomcat*



The web server handles the HTTP transaction details

The servlet container provides a Java Virtual Machine for servlet execution

Environment For Developing and Testing Servlets

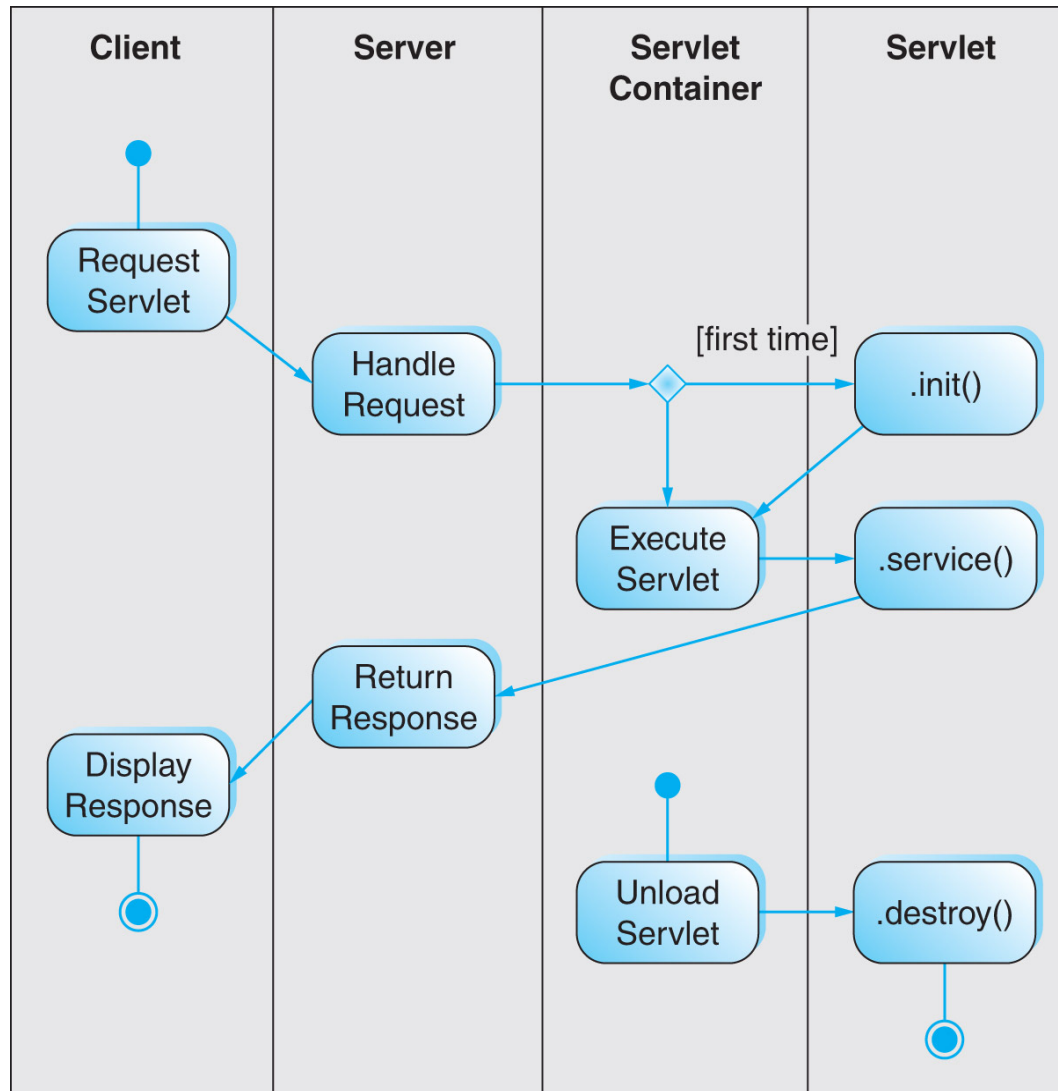
❖ Compile:

- Need Servlet.jar. Available in Tomcat package

❖ Setup testing environment

- Install and start Tomcat web server
- Place compiled servlet at appropriate location

Servlet Operation



Servlet Methods

❖ Servlets have three principal methods

`.init()`

invoked once, when the servlet is loaded by the servlet container
(upon the first client request)

`.service(HttpServletRequest req,
HttpServletResponse res)`


invoked for each HTTP request
parameters encapsulate the HTTP request and response

`.destroy()`

invoked when the servlet is unloaded
(when the servlet container is shut down)

Servlet Methods

- ❖ The default `.service()` method simply invokes method-specific methods
 - depending upon the HTTP request method

`.service()`  `.doGet()`
`.doPost()`
`.doHead()`
... etc.

HTTP Servlet

Methods of HttpServlet and HTTP requests

Methods	HTTP Requests	Comments
doGet	GET, HEAD	Usually overridden
doPost	POST	Usually overridden
doPut	PUT	Usually not overridden
doOptions	OPTIONS	Almost never overridden
doTrace	TRACE	Almost never overridden

All methods take two arguments: an `HttpServletRequest` object and an `HttpServletResponse` object.

Return a `BAD_REQUEST` (400) error by default.

Servlet Example 1

❖ This servlet will say "Hello!" (in HTML)

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

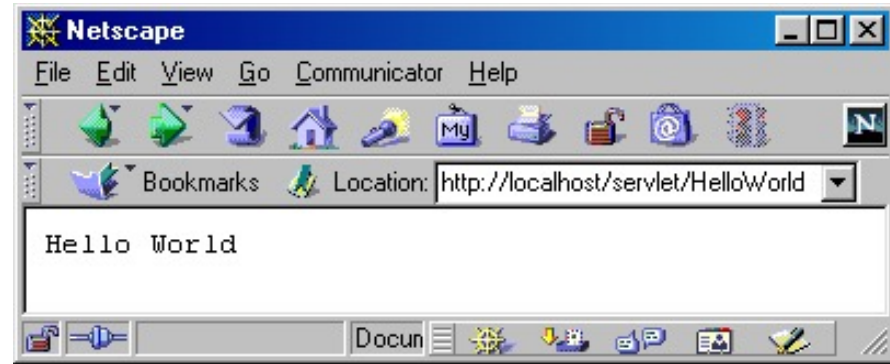
public class HelloServlet extends HttpServlet {
    public void service(HttpServletRequest req,
        HttpServletResponse res) throws IOException {
        PrintWriter htmlOut = res.getWriter();
        res.setContentType("text/html");
        htmlOut.println("<html><head><title>" +
            "Servlet Example Output</title></head><body>" +
            "<p>Hello!</p>" + "</body></html>");
        htmlOut.close();
    }
}
```

Servlet Example 2

- This servlet also will say "Hello World" (not in HTML)

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

```
public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello World");
    }
}
```



Servlet Configuration

- ❖ The web application configuration file, web.xml, identifies servlets and defines a mapping from requests to servlets

An identifying name for the servlet (appears twice)

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>servlet.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

The servlet's package and class names

The pathname used to invoke the servlet
(relative to the web application URL)

Environment Entries

❖ Servlets can obtain configuration information at run-time from the configuration file (web.xml)

- a file name, a database password, etc.

❖ in web.xml:

```
<env-entry-description>password</env-entry-  
description>  
<env-entry>  
  <env-entry-name>UserId</env-entry-name>  
<env-entry-value>Xy87!fx9*</env-entry-value>  
  <env-entry-type>java.lang.String</env-entry-type>  
</env-entry>
```


Environment Entries

❖ in the `init()` method of the servlet:

```
try {  
    Context envCtx = (Context)  
        (new InitialContext()).lookup("java:comp/env");  
    password = (String) envCtx.lookup("password");  
} catch (NamingException e) {  
    e.printStackTrace();  
} catch (ClassNotFoundException e) {  
    e.printStackTrace();  
}
```

Handling HTML Forms

- ❖ An HTML form can be sent to a servlet for processing
- ❖ The action attribute of the form must match the servlet URL mapping

```
<form method="post" action="hello" />
```

```
<servlet-mapping>  
  <servlet-name>HelloServlet</servlet-name>  
  <url-pattern>/hello</url-pattern>  
</servlet-mapping>
```

Simple Form Servlet

```
<form action="hello" method="post" >
  <p>User Id:<input type="text" name="userid" /></p>
  <p><input type="submit" value="Say Hello" /></p>
</form>
```

```
public class HelloServlet extends HttpServlet {
    public void doPost(HttpServletRequest req,
        HttpServletResponse res) throws IOException {
        PrintWriter out = res.getWriter();
        res.setContentType("text/html");
        String userId = req.getParameter("userid");
        out.println("<html><head><title>Hello</title></head>"
            + "<body><p>Hello, " + userId
            + "!</p></body></html>");
        out.close();
    }
}
```

State Management

- **session**: a series of transaction between user and application
- **session state**: the short-term memory that the application needs in order to maintain the session
 - e.g., shopping cart, user-id
- **cookie**: a small file stored by the client at the instruction of the server

Cookies

- The Set-Cookie: header in an HTTP response instructs the client to store a cookie

```
Set-Cookie: SESSIONID=B6E98A; Path=/slms;  
Secure
```

- After a cookie is created, it is returned to the server in subsequent requests as an HTTP request Cookie: header

```
Cookie: SESSIONID=B6E98A
```

Cookie Attributes

- **Name**: the unique name associated with the cookie
- **Content**: value stored in the cookie
- **Expiration Date**: cookie lifetime
- **Domain**: Defines the hosts to which the cookie should be returned
- **Path**: Defines the resource requests with which the cookie should be returned
- **Secure**: if true, cookie is returned only with HTTPS requests

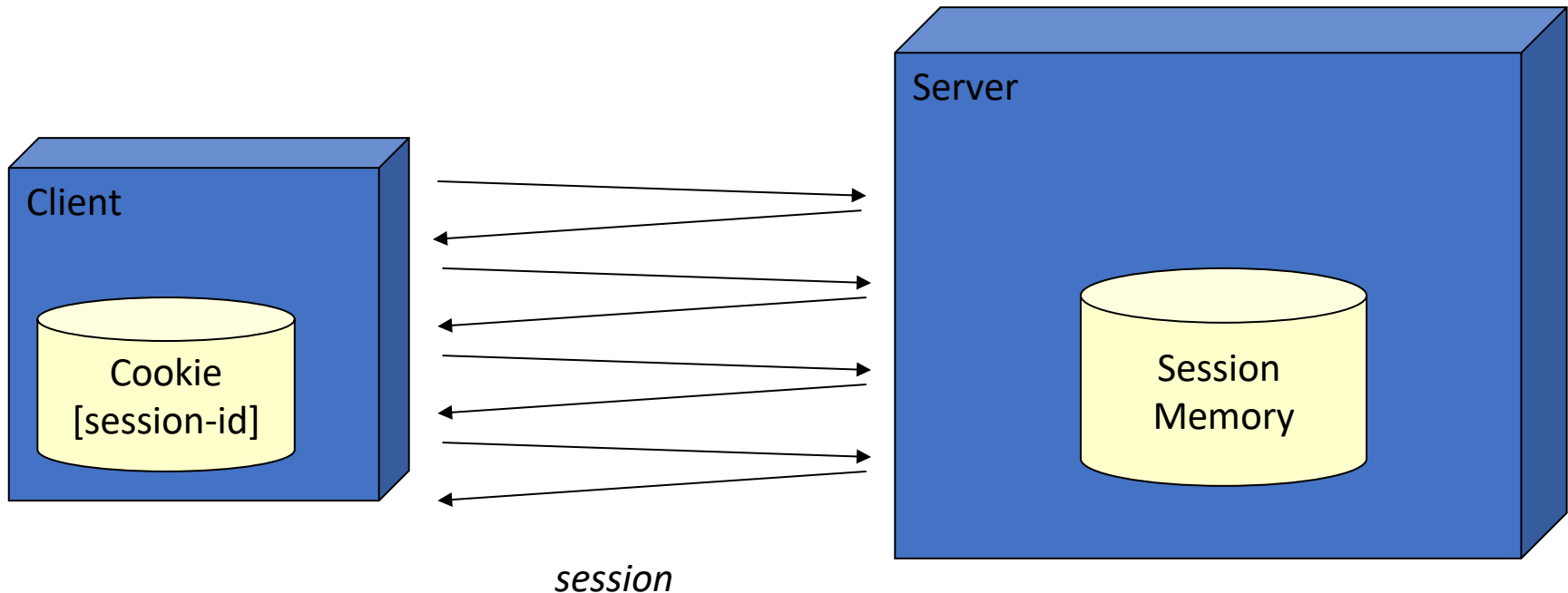
Cookie Example

- **Name:** session-id
- **Content:** 104-1898635-929144
- **Expiration Date:** Monday, June 29, 2009 3:33:30 PM
- **Domain:** .ehsl.org
- **Path:** /slms
- **Secure:** no
- This cookie will be returned with all requests matching *.ehsl.org/slms*, through the indicated expiration date

Session Management

- HTTP is inherently stateless, i.e., there is no memory between transactions
- Applications must maintain a session memory if it is required
- Cookies are used to identify sessions, by recording a unique session-id

State Management



- At the start of a new session, the server sets a new cookie containing the session-id
- With each transaction, the client sends the session-id, allowing the server to retrieve the session

Session Attributes

- The methods

```
session.setAttribute(key, value)
```

```
session.getAttribute(key)
```

store and retrieve session memory

- key is a string; value can be any object

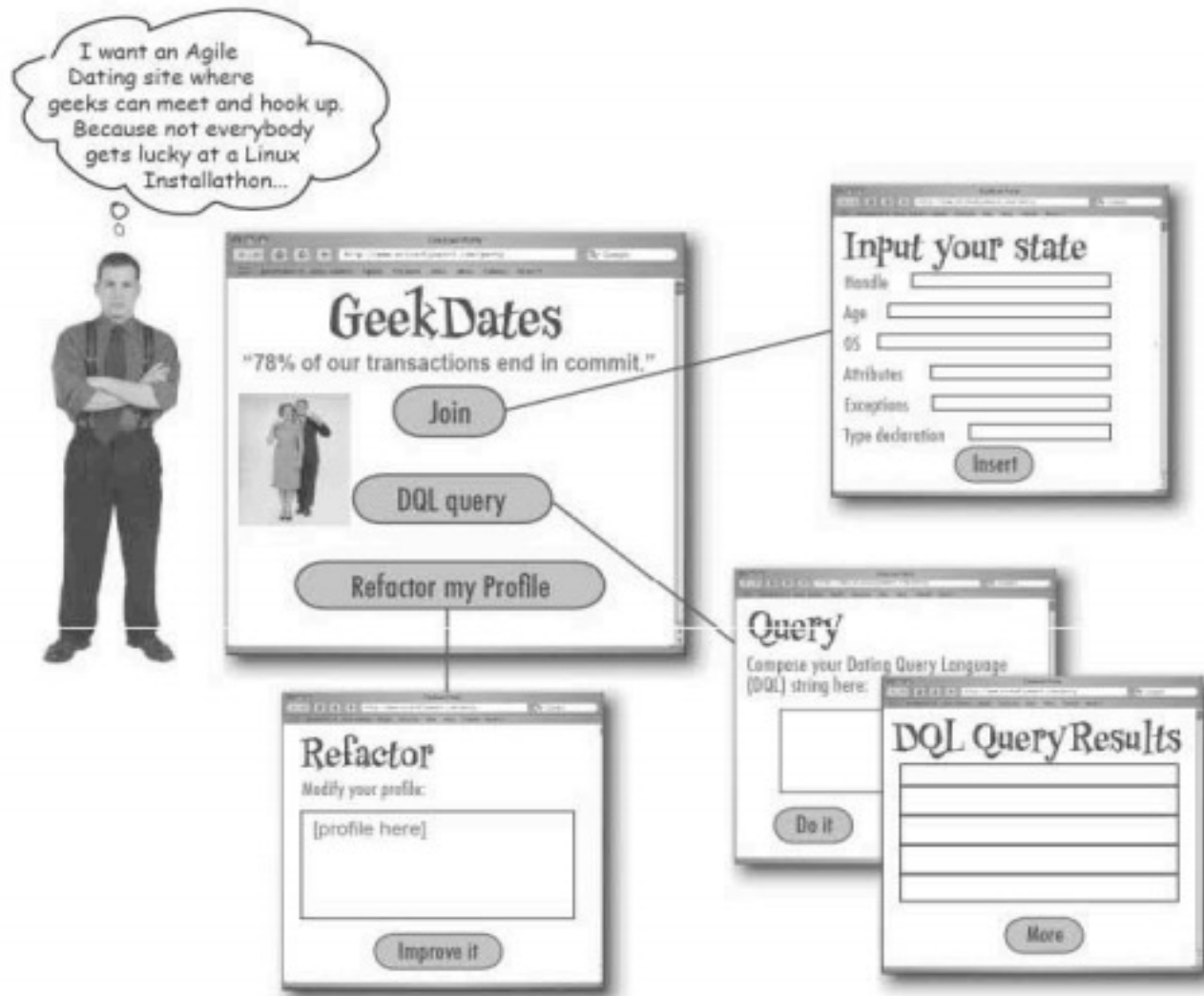
- For example,

```
session.setAttribute("userid", userId);
```

```
String userId =
```

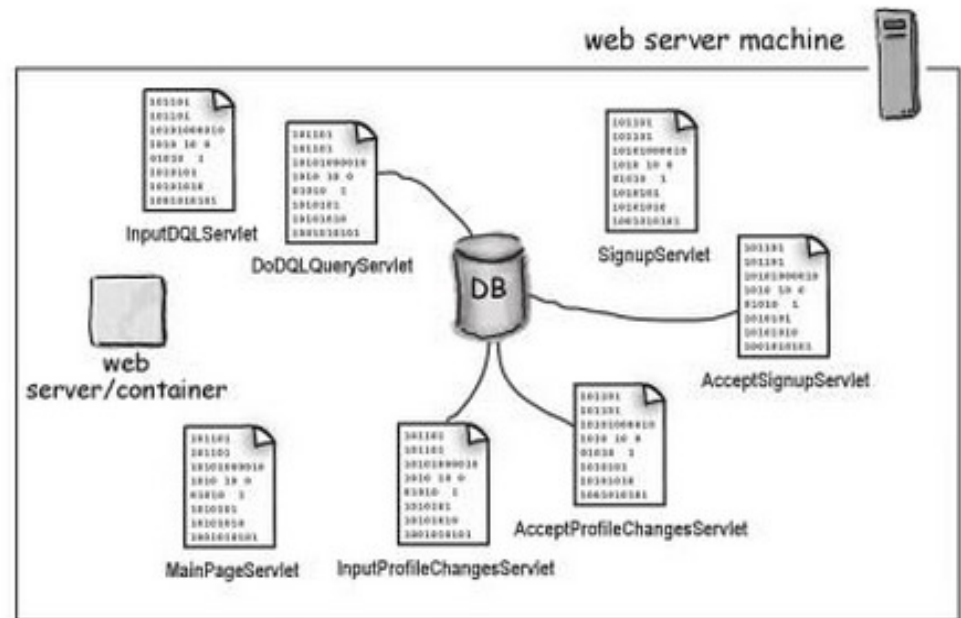
```
(String) session.getAttribute("userid");
```

Problem



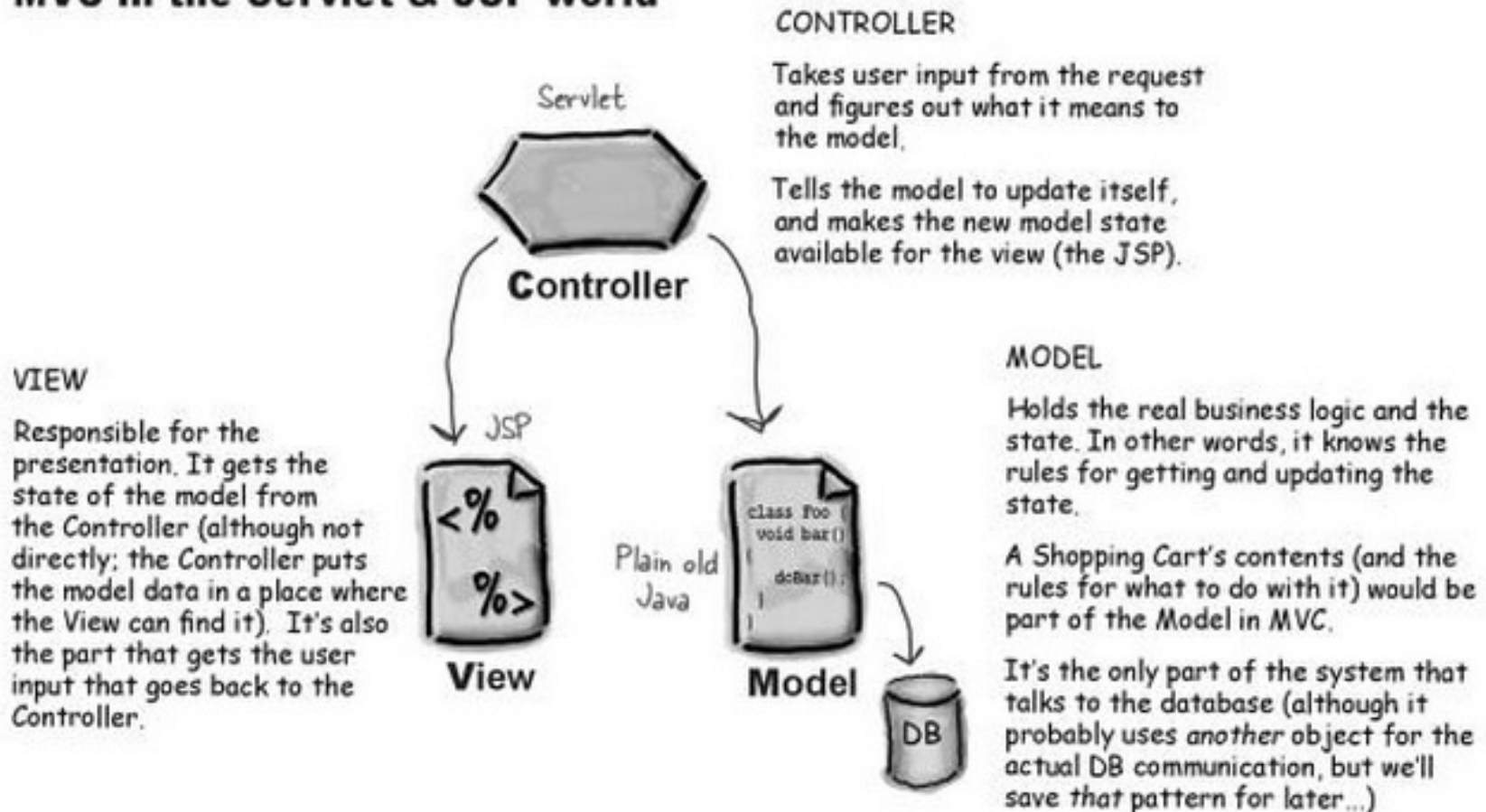
Initial Solution

- Develop a number of servlets
- Each servlet plays the role of one function (a.k.a business logic)



Better Solution: Using MVC

MVC in the Servlet & JSP world



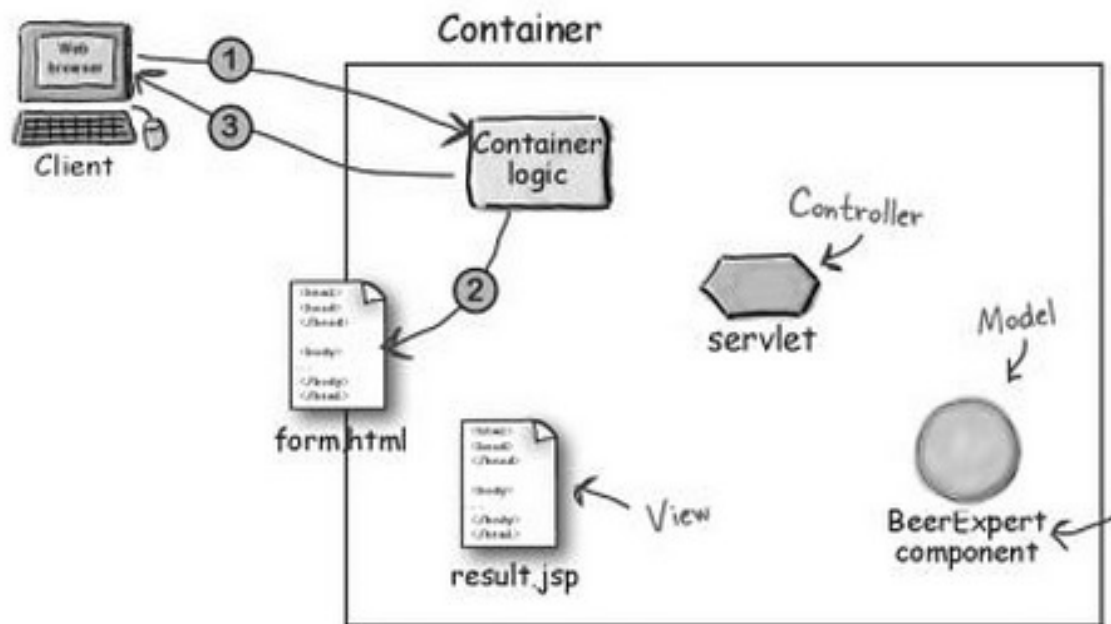
Example 1: Beer Recommendation



HTML
page

JSP page





1 - The client makes a request for the *form.html* page.

2 - The Container retrieves the *form.html* page.

3 - The Container returns the page to the browser, where the user answers the questions on the form and...

4 - The browser sends the request data to the container.

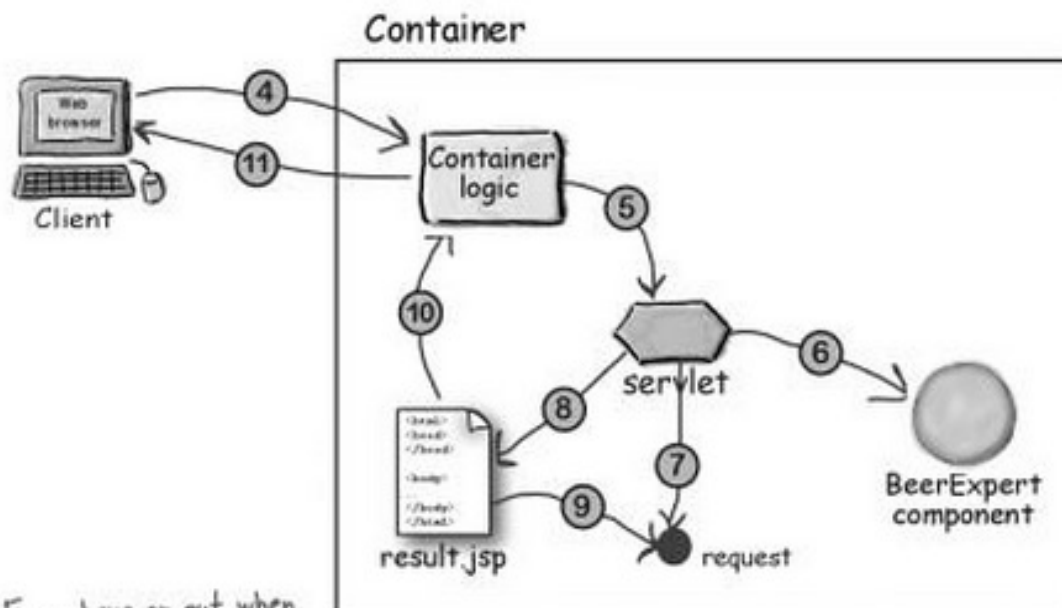
5 - The Container finds the correct servlet based on the URL, and passes the request to the servlet.

6 - The servlet calls the BeerExpert for help.

7 - The expert class returns an answer, which the servlet adds to the request object.

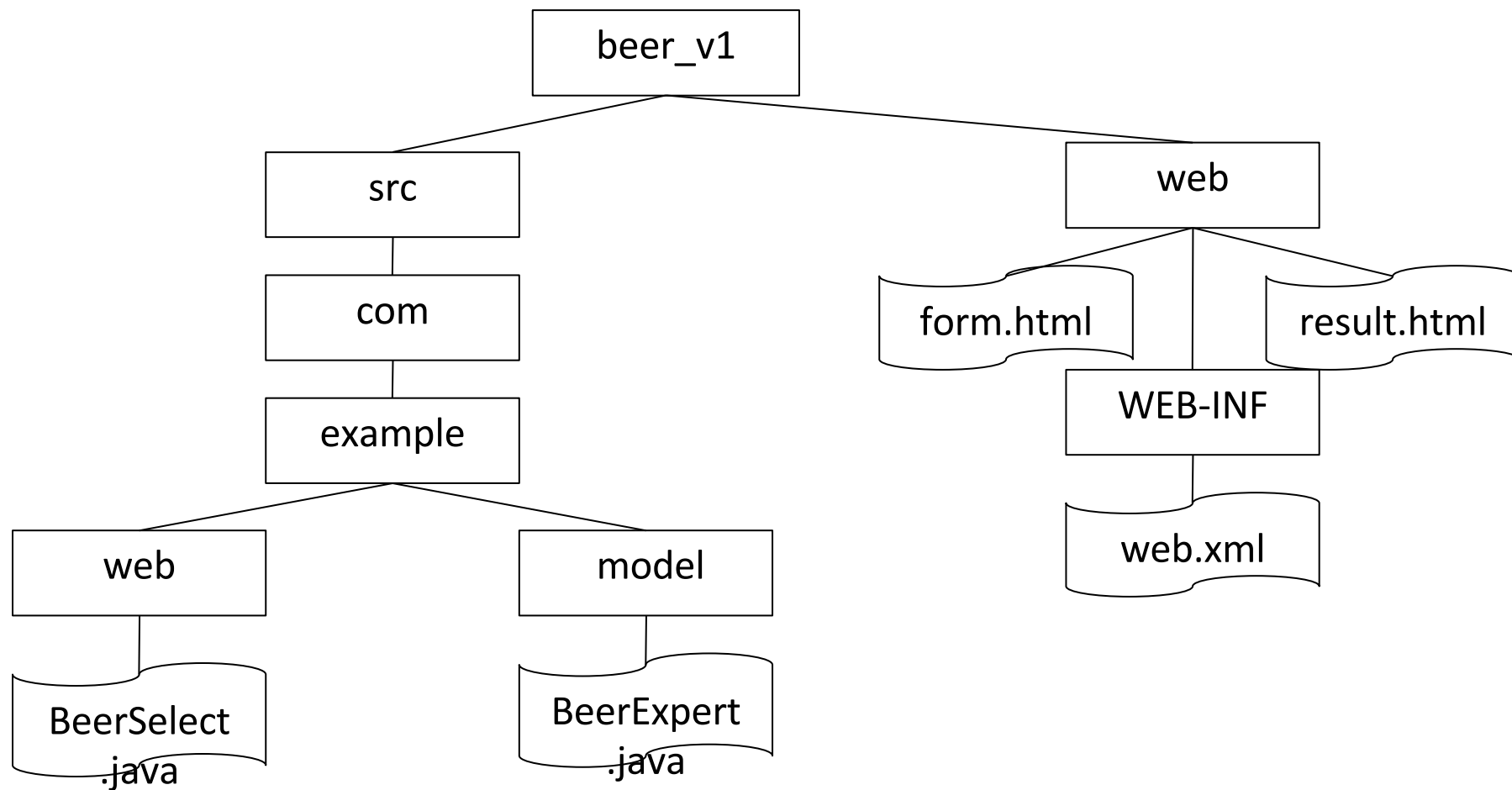
8 - The servlet forwards the request to the JSP.

9 - The JSP gets the answer from the request object.

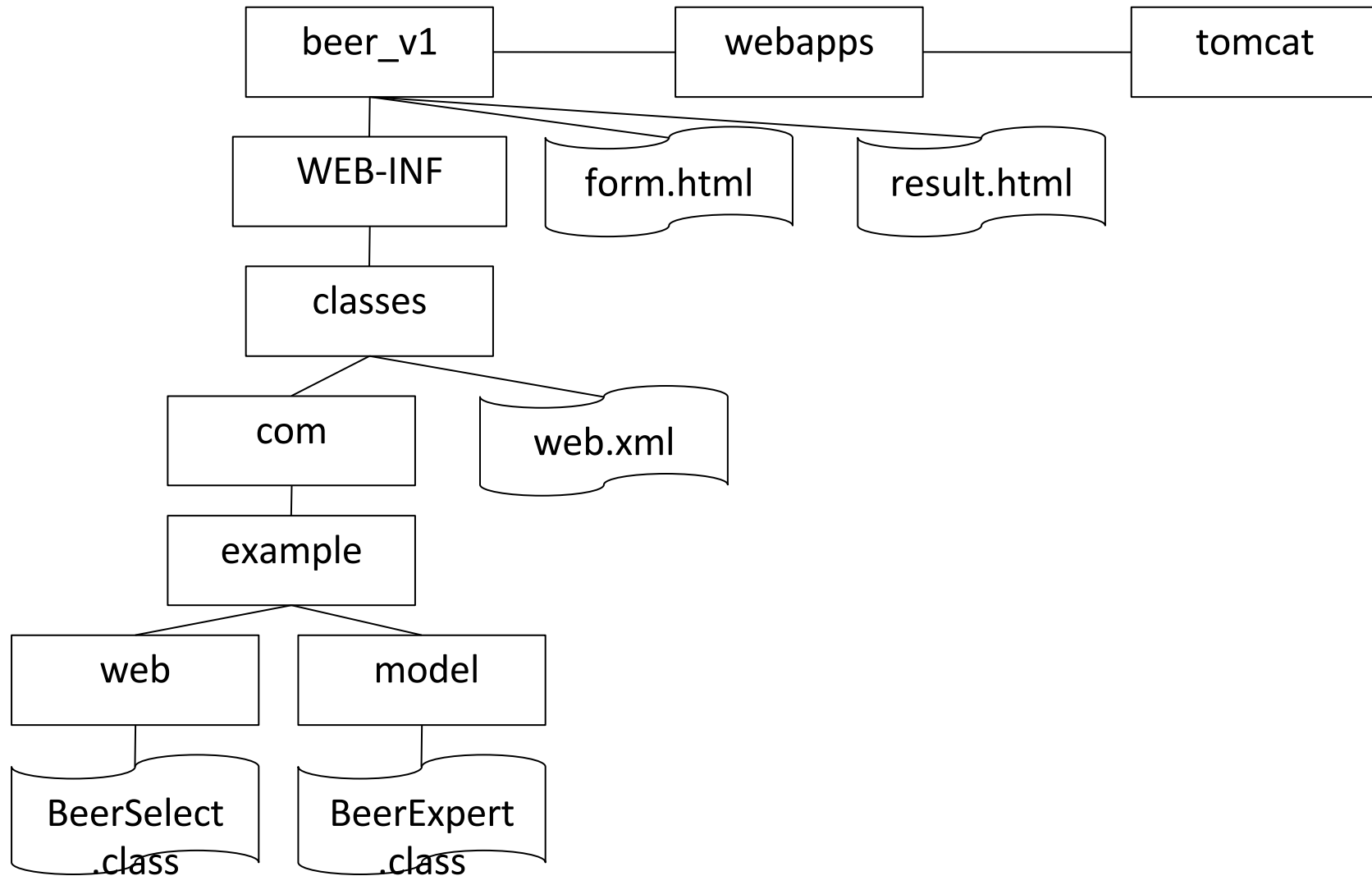


From here on out when

Application Programming Structure



Structure of Folder Development



form.html

```
<form method="POST"
action="SelectBeer.do">
  Select beer characteristics
  <p>Color:
    <select name="color" size="1">
      <option value="light">light</option>
      <option value="amber">amber</option>
      <option value="brown">brown</option>
      <option value="dark">dark</option>
    </select>
    <center> <input type="SUBMIT"> </center>
</form>
```



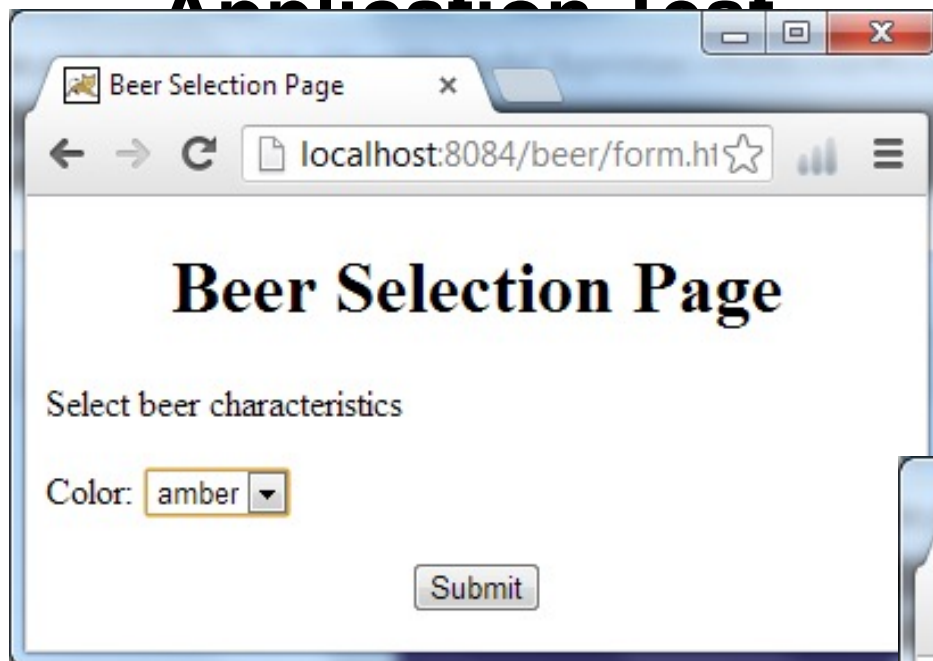
web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  <servlet>
    <servlet-name>ServletBeer</servlet-name>
    <servlet-class>com.example.web.BeerSelect
      </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>ServletBeer</servlet-name>
    <url-pattern>/SelectBeer.do</url-pattern>
  </servlet-mapping>
</web-app>
```

Servlet BeerSelect – Version 1

```
public class BeerSelect extends HttpServlet {  
    @Override  
    protected void doPost(HttpServletRequest request,  
        HttpServletResponse response) throws  
        ServletException, IOException {  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        out.println("Beer Selection Advice<br>");  
        String c = request.getParameter("color");  
        out.println("<br>Got beer color "+c);  
    }  
}
```

Application Test

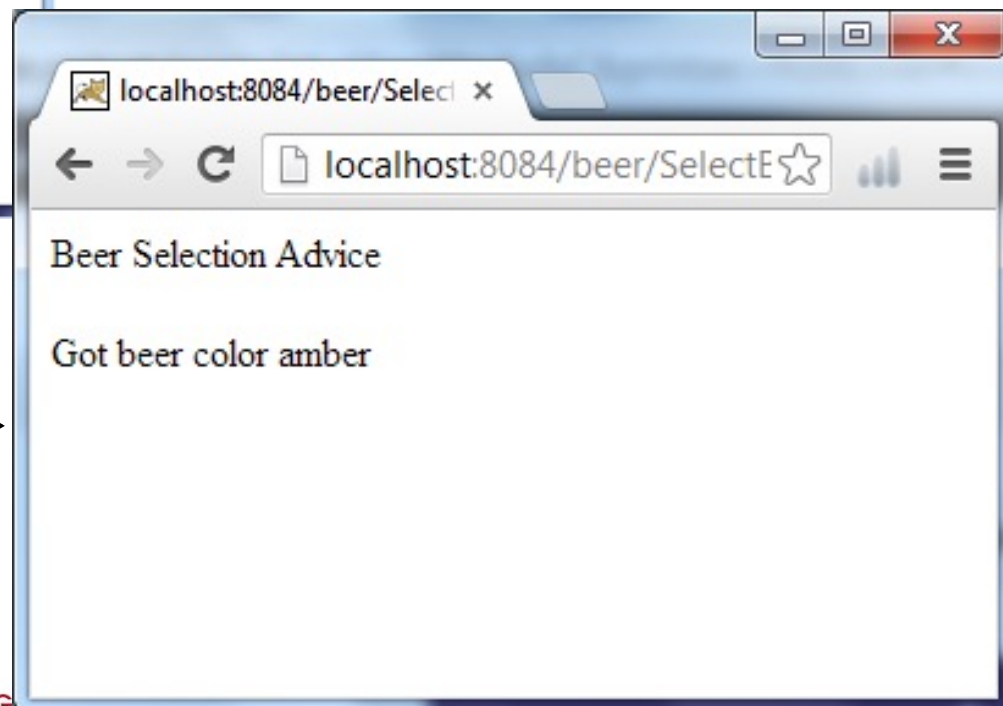


A screenshot of a web browser window titled "Beer Selection Page". The address bar shows "localhost:8084/beer/form.html". The page content includes the title "Beer Selection Page", the instruction "Select beer characteristics", a label "Color:" followed by a dropdown menu showing "amber", and a "Submit" button.

Beer Selection Page

Select beer characteristics

Color:



A screenshot of a web browser window titled "localhost:8084/beer/SelectE". The address bar shows "localhost:8084/beer/SelectE". The page content includes the title "Beer Selection Advice" and the text "Got beer color amber". An arrow from the "Submit" button in the first screenshot points to this window.

Beer Selection Advice

Got beer color amber

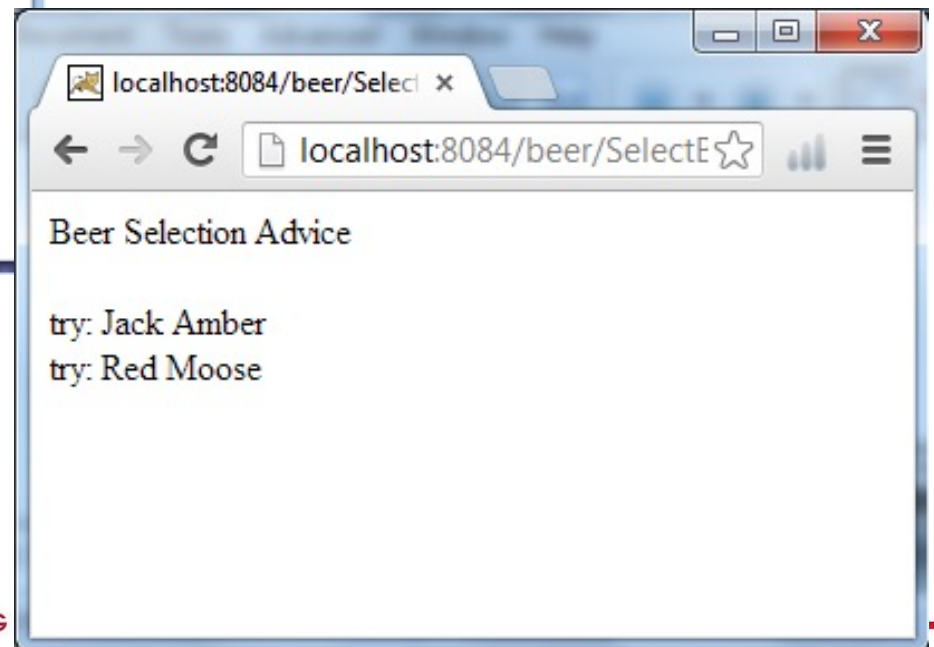
Model BeerExpert

```
public class BeerExpert {  
    public List getBrands(String color) {  
        List brands = new ArrayList();  
        if(color.equals("amber")) {  
            brands.add("Jack Amber");  
            brands.add("Red Moose");  
        }  
        else{  
            brands.add("Jail Pale Ale");  
            brands.add("Gout Stout");  
        }  
        return brands;  
    }  
}
```

Servlet BeerSelect – Version 2

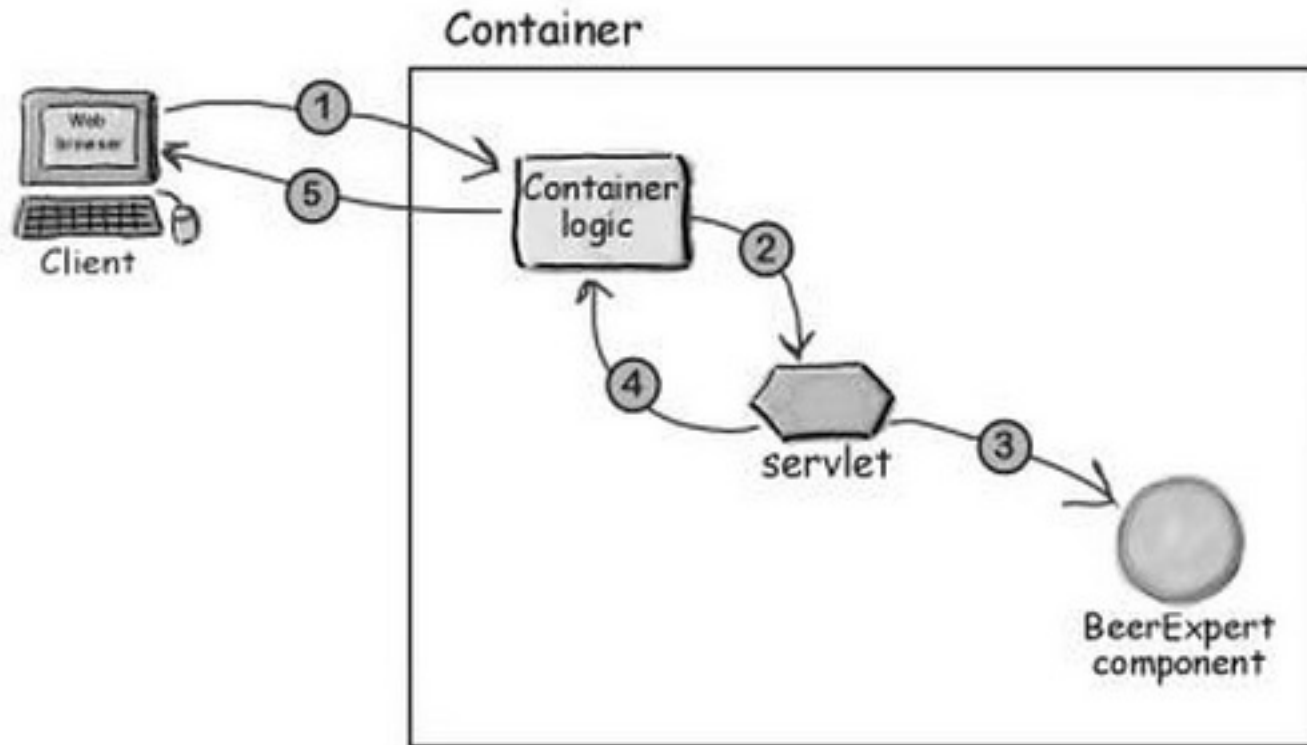
```
import package com.example.web;
...
protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    String c = request.getParameter("color");
    BeerExpert be = new BeerExpert();
    List result = be.getBrands(c);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("Beer Selection Advice<br>");
    Iterator it = result.iterator();
    while(it.hasNext()) {
        out.print("<br>try: "+it.next());
    }
}
```

Application Test



Current Architecture of the Application

What's working so far...



1 - The browser sends the request data to the Container.

2 - The Container finds the correct servlet based on the URL, and passes the request to the servlet.

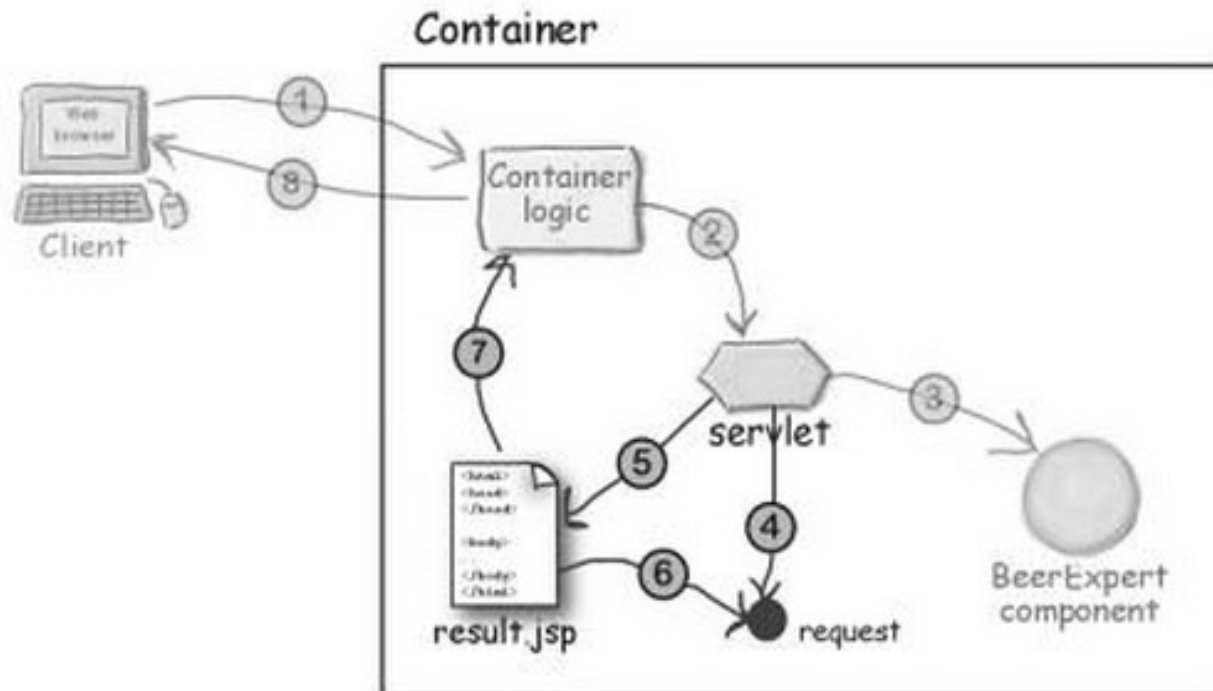
3 - The servlet calls the BeerExpert for help.

4 - The servlet outputs the response (which prints the advice).

5 - The Container returns the page to the happy user.

Desired Application Architecture

What we WANT...



1 - The browser sends the request data to the Container.

2 - The Container finds the correct servlet based on the URL, and passes the request to the servlet.

3 - The servlet calls the BeerExpert for help.

4 - The expert class returns an answer, which the servlet adds to the request object.

5 - The servlet forwards the request to the JSP.

6 - The JSP gets the answer from the request object.

7 - The JSP generates a page for the Container.

8 - The Container returns the page to the happy user.

Result.jsp

```
<%@ page import="java.util.*"%>
<!DOCTYPE html>
<html>
<body>
    <h1 align="center">Beer Recommendation </h1> <p>
        <%
            List styles=(List)
request.getAttribute("styles");
            Iterator it = styles.iterator();
            while(it.hasNext()) {
                out.print("<br>try: "+it.next());
            }
        %>
    </body>
</html>
```

Servlet BeerSelect – Version 3

```
import package com.example.web;
...
protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    String c = request.getParameter("color");
    BeerExpert be = new BeerExpert();
    List result = be.getBrands(c);

    request.setAttribute("styles", result);
    RequestDispatcher view =
request.getRequestDispatcher("result.jsp");
view.forward(request, response);
}
```

Application Test

The image displays two browser windows illustrating a web application test. The left window, titled 'Beer Selection Page', shows a form with the heading 'Beer Selection Page' and the instruction 'Select beer characteristics'. A dropdown menu for 'Color' is set to 'amber', and a 'Submit' button is visible. An arrow points from the 'Submit' button to the right window. The right window, titled 'localhost:8084/beer/SelectE', displays the 'Beer Selection Advice' with the following text:

try: Jack Amber
try: Red Moose

At the bottom left, there is a logo for SOICT (Viện Công Nghệ Thông Tin và Truyền Thông) celebrating its 25th anniversary, with the text 'ĐẠI HỌC BÁCH KHOA' and '25 YEARS ANNIVERSARY'.

Outline

1. Servlet
2. **JSP – Java Server Page**
3. Java Beans
4. ORM (Object Relational Mapping)

Introduction and Overview

❖ Server-side java:

■ Scheme 1:

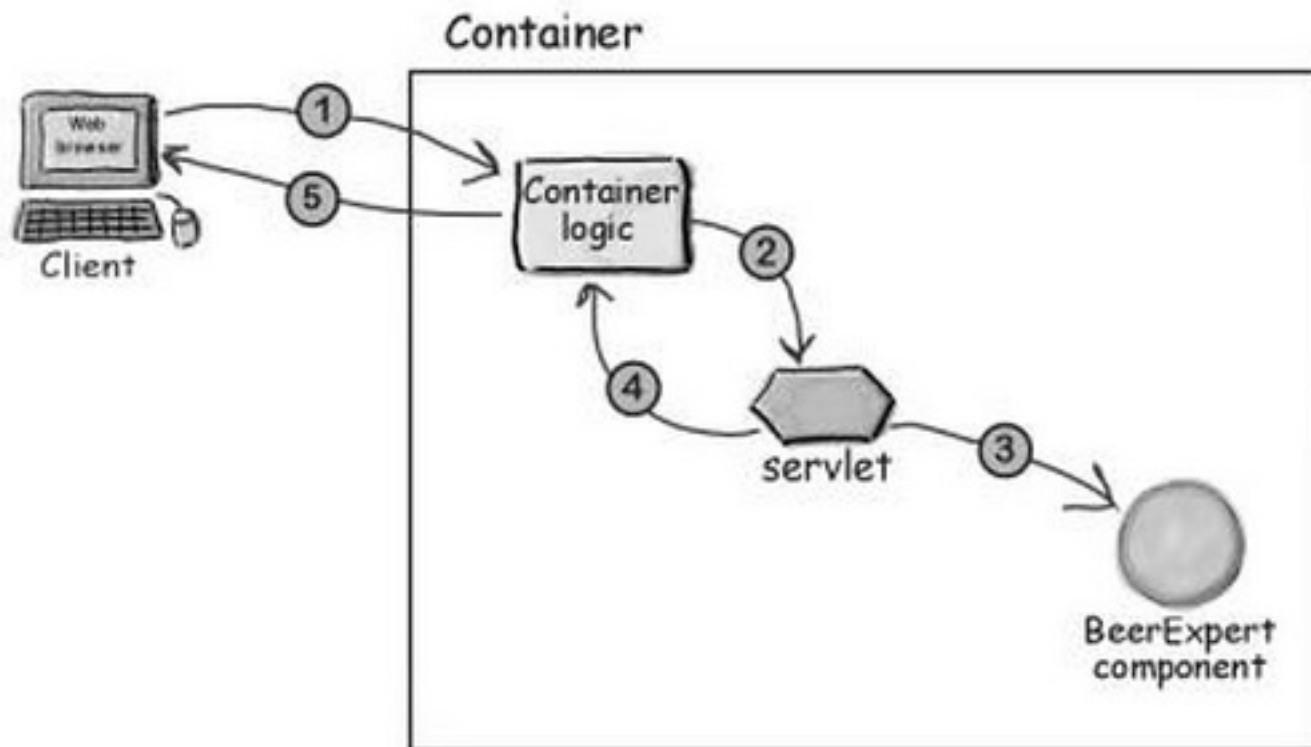
- HTML files placed at location for web pages
- Servlets placed at special location for servlets
- Call servlets from HTML files

■ Scheme 2:

- JSP: HTML + servlet codes + jsp tags
- Placed at location for web pages
- Converted to normal servlets when first accessed

Scheme 1

What's working so far...



1 - The browser sends the request data to the Container.

2 - The Container finds the correct servlet based on the URL, and passes the request to the servlet.

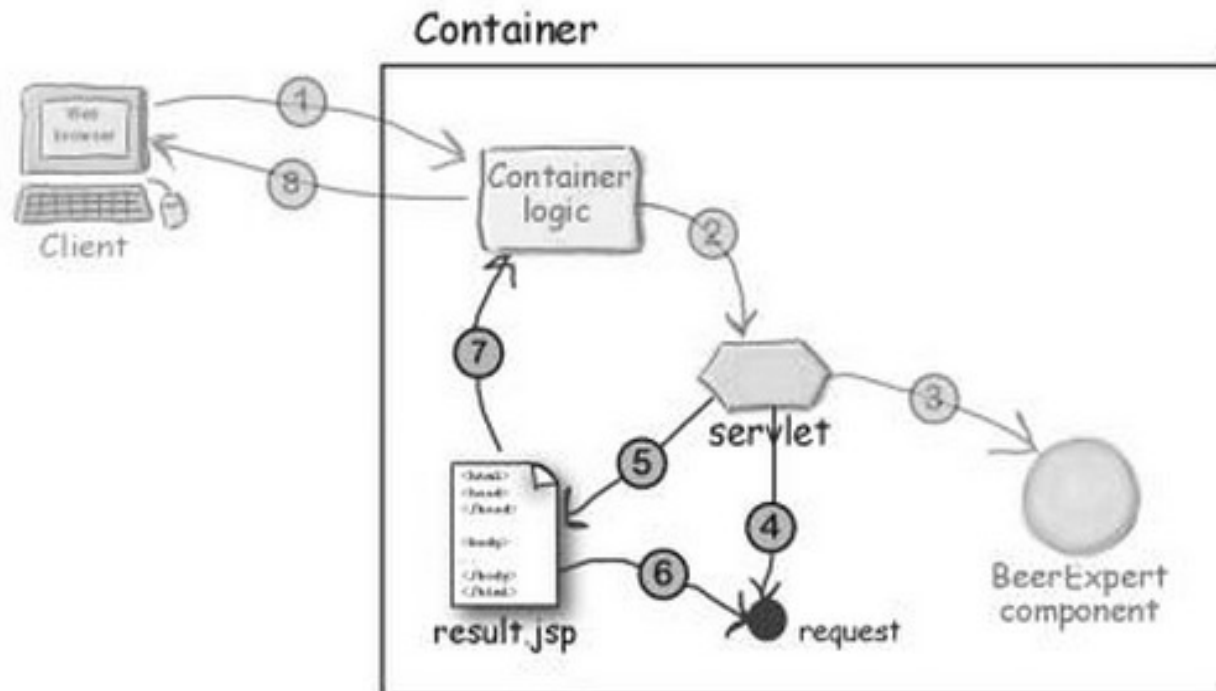
3 - The servlet calls the BeerExpert for help.

4 - The servlet outputs the response (which prints the advice).

5 - The Container returns the page to the happy user.

Scheme 2

What we WANT...



1 - The browser sends the request data to the Container.

2 - The Container finds the correct servlet based on the URL, and passes the request to the servlet.

3 - The servlet calls the BeerExpert for help.

4 - The expert class returns an answer, which the servlet adds to the request object.

5 - The servlet forwards the request to the JSP.

6 - The JSP gets the answer from the request object.

7 - The JSP generates a page for the Container.

8 - The Container returns the page to the happy user.

Introduction and Overview

❖ Example: Hello.jsp

- `<HTML>`
 - `<HEAD><TITLE>JSP Test</TITLE></HEAD>`
 - `<BODY BGCOLOR="#FDF5E6">`
 - `<H1>JSP Test</H1>`
 - `Time: <%= new java.util.Date() %>`
 - `</BODY>`
 - `</HTML>`
-
- `<H1>JSP Test</H1>`: normal HTML
 - `<%, %>`: special JSP tags
 - `new java.util.Date()`: java code
 - Placed at: tomcat/webapps/ROOT/jsp

Introduction and Overview

❖ Ingredients of a JSP

- Regular HTML
 - Simply "passed through" to the client by the servlet created to handle the page.
- **JSP constructs**
 - **Scripting elements** let you specify Java code that will become part of the resultant servlet,
 - **Directives** let you control the overall structure of the servlet
 - **Actions** let you specify existing components that should be used, and control the behavior of the JSP engine
 - JavaBeans: a type of components frequently used in JSP

JSP constructs - Scripting Elements

- ❖ JSP converted to Servlet at first access
- ❖ JSP scripting elements let you insert Java codes into the servlet results
 - **Expressions:**
 - Form `<%= expression %>`
 - Evaluated and inserted into the output
 - **Scriptlets**
 - Form `<% code %>`
 - Inserted into the servlet's service method
 - **Declarations:**
 - Form `<% ! code %>`
 - Inserted into the body

JSP constructs - Scripting Elements

❖ JSP Expressions:

- Form: `<%= expression %>`
- Example
 - Time: `<%= new java.util.Date() %>`
- Processing
 - Evaluated, converted to a string, and inserted in the page.
 - At run-time (when the page is requested)

JSP constructs - Scripting Elements

❖ JSP Expressions:

- Several variables predefined to simply jsp expressions
 - **request**, the HttpServletRequest;
 - **response**, the HttpServletResponse;
 - **session**, the HttpSession associated with the request (if any);
 - **out**, the PrintWriter (a buffered version of type JspWriter) used to send output to the client.
- Example:
 - **Your hostname:** `<%= request.getRemoteHost() %>`

JSP constructs - Scripting Elements

❖ JSP Scriptlets

- Form: `<% code %>`
- Example:
 - `<% String queryData = request.getQueryString();`
 - `out.println("Attached GET data: " + queryData); %>`
- Inserted into the servlet's service method EXACTLY as written
- Can access the same predefined variables as JSP expressions

JSP constructs - Scripting Elements

❖ JSP Declarations:

- Form: `<% ! code %>`
- Example: `<% ! private int accessCount = 0; %>`
- Inserted into the main body of the servlet class (outside of the service method processing the request)
- Normally used in conjunction with JSP expressions or scriptlets.
 - `<% ! private int accessCount = 0; %>`
 - Accesses to page since server reboot:
 - `<%= ++accessCount %>`

JSP constructs - JSP Directives

❖ Affect the overall structure of the servlet class.

- Form:

```
<%@ directive attribute1="value1"
                        attribute2="value2"
                        ...
                        AttributeN="valueN" %>
```

❖ Two commonly used types of directives

- Page directives
 - ```
<%@ page import="java.util.*" %>
```
- Include directives
  - ```
<%@ include file="/navbar.html" %>
```

JSP constructs - Directives

❖ Examples of Page directives

- `<%@ page import="java.util.*" %>`
`<%@ page language="java" import="java.util.*" %>`
- `<%@ page contentType="text/plain" %>`
 - Same as : `<% response.setContentType("text/plain"); %>`
- `<%@ page session="true" %>`

JSP constructs - Directives

❖ Include Directive

- lets you include files at the time the JSP page is translated into a servlet (static including).
- Form: `<%@ include file="relative url" %>`
- Example situation where it is useful:
 - Same navigation bar at bottom of many pages.
 - Usage
 - Keep content of the navigation bar in an URL
 - Include the URL in all the pages

JSP constructs - **Actions**

❖ JSP *actions* control the behavior of the servlet engine. Let one

- Dynamically insert a file
- Forward the user to another page
- Reuse JavaBeans components
- ..

JSP constructs - Actions

❖ The include action

- Form:

- `<jsp:include page="relative URL" flush="true" />`

- Inserts the file at the time the page is requested.

- Differs from the include directive, which inserts file at the time the JSP page is translated into a servlet.

- Example: IncludeAction.jsp

JSP constructs - **Actions**

❖ The forward action:

- Form: `<jsp:forward page="relative URL" />`
`<jsp:forward page="<%= someJavaExpression %>" />`
- Forward to the page specified.
- Example: ForwardAction.jsp

❖ Several actions related to reuse of JavaBeans components

- Discuss next

JSP vs Servlet

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

❖ Servlets:

- Using `println()` to create HTML pages
 - → Whenever developers make a change, they have to recompile and redeploy, which is not really convenient

❖ JSP:

- correct the problem of Servlet

Benefits of using JSP...

- Contents and display logic (or presentation logic) are separated.
- Web application development can be simplified because business logic is captured in the form of JavaBeans or custom tags while presentation logic is captured in the form of HTML template.
- Because the business logic is captured in component forms, they can be reused in other Web applications.
- And again for web page authors, dealing with JSP page is a lot easier than writing Java code.
- And just like Servlet technology, JSP technology runs over many different platforms.

Benefits of Using JSP over Servlet

- ❖ Exploit both two technologies
 - The power of Servlet is “controlling and dispatching”
 - The power of JSP is “displaying”
- ❖ In practice, both Servlet and JSP are very useful in MVC model
 - Servlet plays the role of Controller
 - JSP plays the role of View

Outline

1. Servlet
2. JSP – Java Server Page
3. **Java Beans**
4. ORM (Object Relational Mapping)

JavaBeans

❖ Beans

- Objects of Java classes that follow a set of simple naming and design conventions
 - Outlined by the JavaBeans specification
- Beans are Java objects
 - Other classes can access them and their methods
 - One can access them from jsp using scripting elements.
- Beans are special Java objects
 - Can be accessed using JSP actions.
 - Can be manipulated in a builder tool
 - Why interesting?
 - Programmers provide beans and documentations
 - Users do not have to know Java well to use the beans.

JavaBeans

❖ Naming conventions:

- Class name:
 - Often include the word Bean in class name, such as UserBean
- Constructor:
 - Must implement a constructor that takes no arguments
 - Note that if no constructor is provided, a default no-argument constructor will be provided.

JavaBeans

❖ Naming conventions: Methods

- Semantically, a bean consists of a collection of properties (plus some other methods)
- The signature for property access (getter and setter) methods
 - `public void setPropertyName(PropertyType value) ;`
 - `public PropertyType getPropertyName ()`
- Example:
 - Property called **rank**:
`public void setRank(String rank) ;`
`public String getRank () ;`
 - Property called **age**:
`public void setAge(int age) ;`
`public int getAge () ;`

JavaBeans

❖ Property name conventions

- Begin with a lowercase letter
- Uppercase the first letter of each word, except the first one, in the property name.
- Examples: **firstName**, **lastName**

❖ Corresponding setter and getter methods:

- **setFirstName**, **setLastName**
- **getFirstName**, **getLastName**
- Note the case difference between the property names and their access method

JavaBeans

❖ Indexed properties

- Properties whose values are sets
- Conventions:
 - `public PropertyType[] getProperty()`
 - `public PropertyType getProperty(int index)`
 - `public void setProperty(int index, PropertyType value)`
 - `public void setProperty(PropertyType[])`
 - `public int getPropertySize()`

JavaBeans

❖ Bean with indexed properties

```
import java.util.*;

public class StatBean {
    private double[] numbers;

    public StatBean() {numbers = new double[0];    }

    public double getAverage() {...}

    public double getSum() { .. }

    public double[] getNumbers()
    {
        return numbers;    }

    public double getNumbers(int index)
    {
        return numbers[index];    }

    public void setNumbers(double[] numbers)
    {
        this.numbers = numbers;    }

    public void setNumbers(int index, double value)
    {
        numbers[index] = value;    }

    public int getNumbersSize()
    {
        return numbers.length;    }
}
```


JavaBeans

❖ Boolean Properties

- Properties that are either true or false
- Setter/getter methods conventions
 - `public boolean isProperty() ;`
 - `public void setProperty(boolean b) ;`

 - `public boolean isEnabled() ;`
 - `public void setEnabled(boolean b) ;`

 - `public boolean isAuthorized() ;`
 - `public void setAuthorized(boolean b) ;`

Using Beans in JSP

❖ JSP actions for using beans:

- `jsp:useBean`
 - Find or instantiate a JavaBean.
- `jsp:setProperty`
 - Set the property of a JavaBean.
 - Call a setter method
- `jsp:getProperty`
 - Get the property of a JavaBean into the output.
 - Call a getter method

Using Beans in JSP

❖ Example: The bean

- `package jspBean201;`

```
public class SimpleBean {  
    private String message = "No message  
specified";  
  
    public String getMessage() {  
        return (message) ;  
    }  
    public void setMessage(String message) {  
        this.message = message;  
    }  
}
```

❖ Compile with javac and place in regular classpath

- In Tomcat, same location as servlets. (can be different on other web servers)

Using Beans in JSP

❖ Use SimpleBean in jsp: ReuseBean.jsp

- Find and instantiate bean

```
<jsp:useBean id="test" class="jspBean201.SimpleBean" />
```

- Set property

```
<jsp:setProperty name="test" property="message"  
  value="Hello WWW"/>
```

- Get property: call the getMessage method and insert what it returns to web page

```
<H1>Message: <|>
```

```
  <jsp:getProperty name="test" property="message" />  
</|></H1>
```

Using Beans in JSP

❖ The jsp:useBean action:

- Format

- Simple format: `<jsp:useBean .../>`

```
<jsp:useBean id="test"  
class="jspBean201.SimpleBean" />
```

- Container format: body portion executed only when bean first instantiated
- `<jsp:useBean ...>`
 Body
 `</jsp:useBean>`

Using Beans in JSP

❖ The jsp:useBean action:

▪ Attributes:

```
<jsp:useBean id=.." scope=..", type=..", beanName=..", class=.." />
```

```
<jsp:useBean id="table" scope="session" class="jspBean201.TableBean" />
```

- Scope: Indicates the context in which the bean should be made available
 - `page` (default): available only in current page
 - `request`, available only to current request
 - `session`, available only during the life of the current HttpSession
 - `Application`, available to all pages that share the same ServletContext
- `id`: Gives a name to the variable that will reference the bean
 - New bean not instantiated if previous bean with same id and scope exists.
- `class`: Designates the full package name of the bean.
- `type` and `beanName`: can be used to replace the class attribute

Using Beans in JSP

❖ The jsp:setProperty action:

- Forms:

```
<jsp:setProperty name=".." property=".." value=".." />
```

```
<jsp:setProperty name=".." property=".." param=".." />
```

- If the value attribute is used
 - String values are automatically converted to numbers, boolean, Boolean, byte, Byte, char, and Character
- If the param attribute is used
 - No conversion

Outline

1. Servlet
2. JSP – Java Server Page
3. Java Beans
4. **ORM (Object Relational Mapping)**

The Object-Oriented Paradigm

- ❖ The world consists of objects
- ❖ So we use object-oriented languages to write applications
- ❖ We want to store some of the application objects (a.k.a. persistent objects)
- ❖ So we use a Object Database?

The Reality of DBMS

- ❖ Relational DBMS are still predominant
 - Best performance
 - Most reliable
 - Widest support
- ❖ Bridge between OO applications and relational databases
 - CLI and embedded SQL (JDBC)
 - **Object-Relational Mapping (ORM)** tools

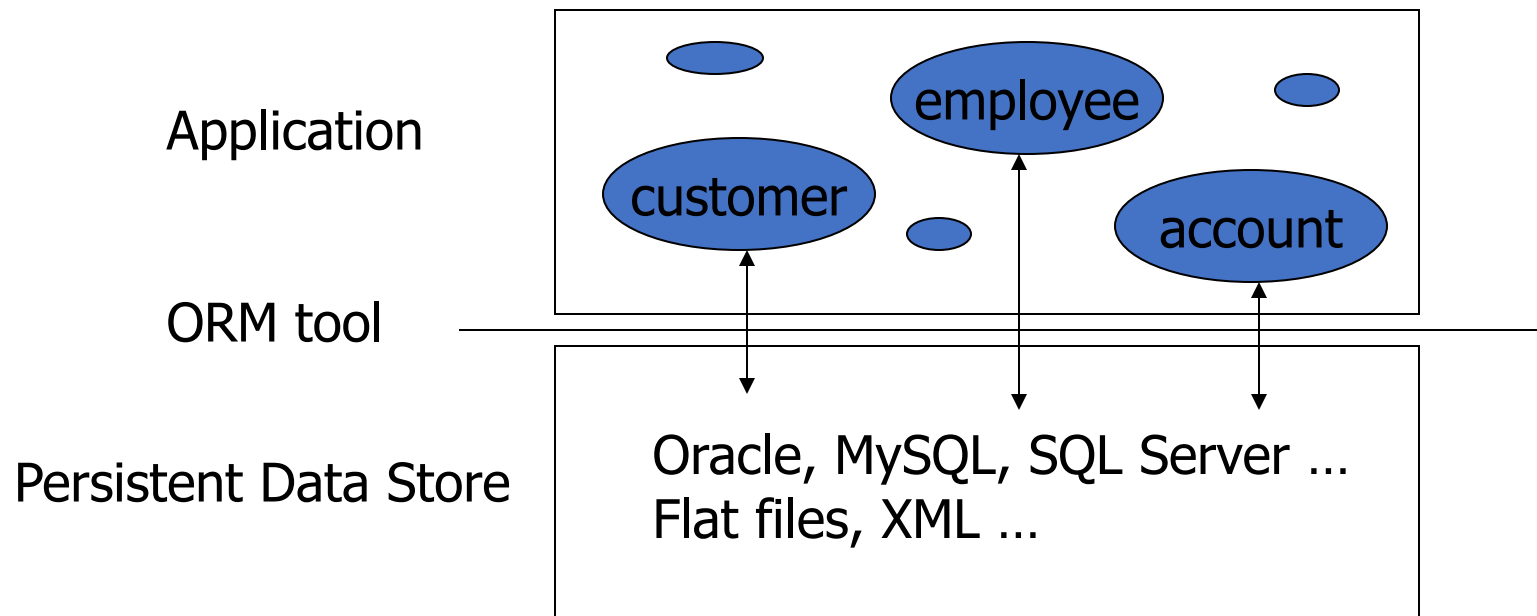
Object-Relational Mapping

- ❖ It is a programming technique for converting object-type data of an object oriented programming language into database tables.
- ❖ Hibernate is used convert object data in JAVA to relational database tables.

What is Hibernate?

- ❖ It is an object-relational mapping (ORM) solution that allows for persisting Java objects in a relational database
- ❖ Open source
- ❖ Development started late 2001

The ORM Approach



O/R Mapping Annotations

- ❖ Describe how Java classes are mapped to relational tables

@Entity	Persistent Java Class
@Id	Id field
@Basic (can be omitted)	Fields of simple types
@ManyToOne @OneToMany @ManyToMany @OneToOne	Fields of class types

Basic Object-Relational Mapping

- ❖ Class-level annotations
 - `@Entity` and `@Table`
- ❖ Id field
 - `@Id` and `@GeneratedValue`
- ❖ Fields of simple types
 - `@Basic` (can be omitted) and `@Column`
- ❖ Fields of class types
 - `@ManyToOne` and `@OneToOne`

persistence.xml

❖ <persistence-unit>

- name

❖ <properties>

- Database information
- Provider-specific properties

❖ No need to specify persistent classes

Access Persistent Objects

- ❖ EntityManagerFactory
- ❖ **EntityManager**
- ❖ Query **and** TypedQuery
- ❖ Transaction
 - A transaction is required for updates

Some EntityManager Methods

- ❖ `find(entityClass, primaryKey)`
- ❖ `createQuery(query)`
- ❖ `createQuery(query, resultClass)`
- ❖ `persist(entity)`
- ❖ `merge(entity)`
- ❖ `getTransaction()`

<http://sun.calstatela.edu/~cysun/documentation/jpa-2.0-api/javax/persistence/EntityManager.html>

Persist() vs. Merge()

Scenario	Persist	Merge
Object passed was never persisted	<ol style="list-style-type: none">1. Object added to persistence context as new entity2. New entity inserted into database at flush/commit	<ol style="list-style-type: none">1. State copied to new entity.2. New entity added to persistence context3. New entity inserted into database at flush/commit4. New entity returned
Object was previously persisted, but not loaded in this persistence context	<ol style="list-style-type: none">1. EntityExistsException thrown (or a PersistenceException at flush/commit)	<ol style="list-style-type: none">1. Existing entity loaded.2. State copied from object to loaded entity3. Loaded entity updated in database at flush/commit4. Loaded entity returned
Object was previously persisted and already loaded in this persistence context	<ol style="list-style-type: none">1. EntityExistsException thrown (or a PersistenceException at flush or commit time)	<ol style="list-style-type: none">1. State from object copied to loaded entity2. Loaded entity updated in database at flush/commit3. Loaded entity returned

<http://blog.xebia.com/2009/03/jpa-implementation-patterns-saving-detached-entities/>

Java Persistence Query Language (JPQL)

- ❖ A query language that looks like SQL, but for accessing *objects*
- ❖ Automatically translated to DB-specific SQL statements
- ❖ `select e from Employee e where e.id = :id`
 - From all the Employee objects, find the one whose id matches the given value

Advantages of ORM

- ❖ Make RDBMS look like ODBMS
- ❖ Data are accessed as objects, not rows and columns
- ❖ Simplify many common operations. E.g.
`System.out.println(e.supervisor.name)`
- ❖ Improve portability
 - Use an object-oriented query language (OQL)
 - Separate DB specific SQL statements from application code
- ❖ Object caching

Q&A



25 YEARS ANNIVERSARY
SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

**Thank you
for your
attentions!**



soict.hust.edu.vn/



fb.com/groups/soict

