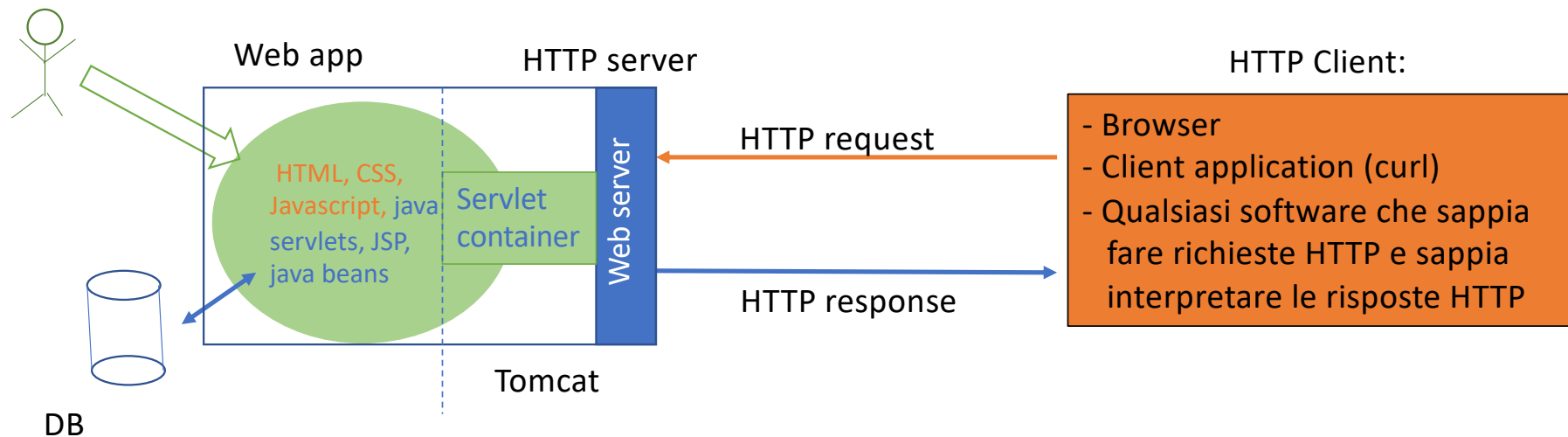


Applicazioni web dinamiche: Tomcat e modelli

Tomcat

Tu – Programmatore (sviluppo, compilazione e deploy)



Tomcat è costituito da:

1. **Web server** (accetta le richieste e restituisce pagine statiche – a volte è un software a parte come Apache)
2. **Servlet container** (permette l'esecuzione di servlet e jsp aggiunte ad esso in webapps)

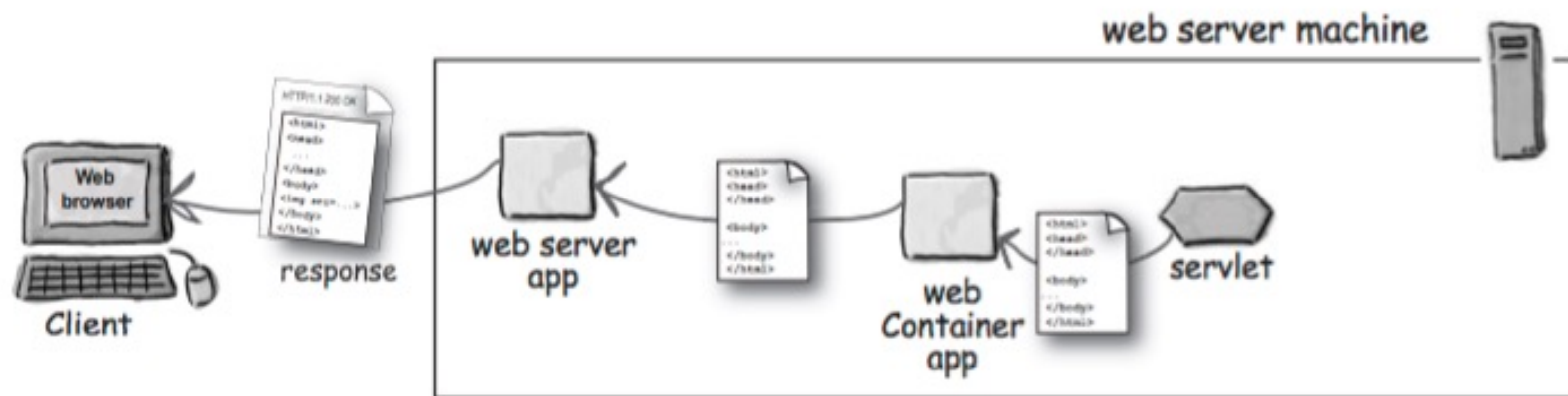
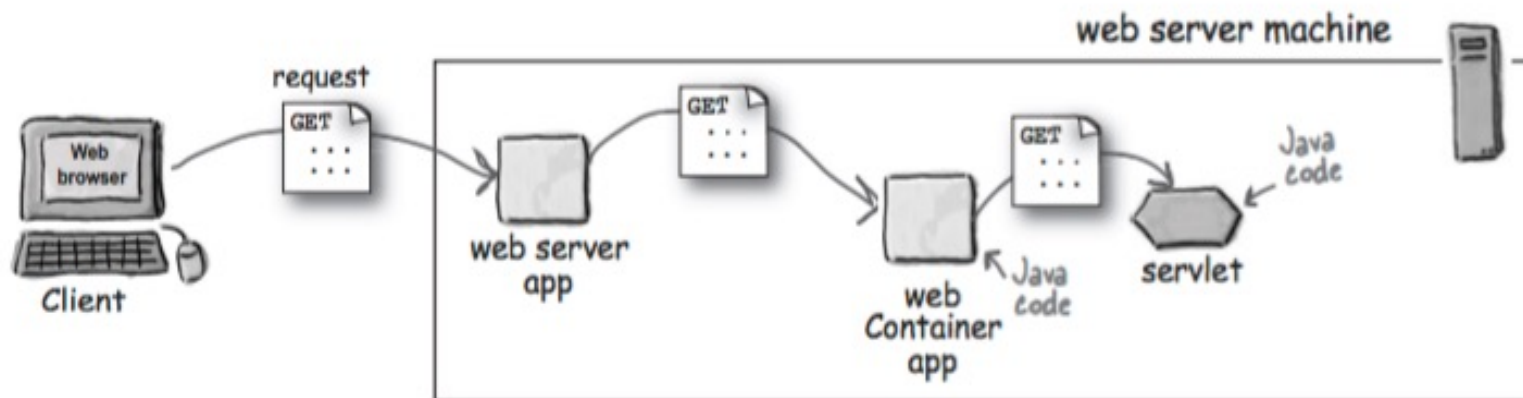
Tomcat non è (del tutto) un **Application server** in quanto anche se include un web server and un servlet container manca di altre tecnologie web più avanzate

Web server e Servlet container

- Un **server web** riceve la richiesta HTTP e controlla se la risorsa richiesta non è da eseguire sul server (per esempio, una pagina html) o altrimenti (chiamata a una servlet).
- Nel caso sia richiesta un'esecuzione sul server, il **server web** passa la richiesta al **servlet container** per l'esecuzione della servlet

What is a real Container?

- Le servlets non hanno un metodo `main()`. Sono sotto il controllo di un'altra applicazione Java chiamata **Container**.
- **Tomcat** (ovvero il suo componente Catalina) è un esempio di **Container**. Quando il Web Server riceve una richiesta per una Servlet, il server passa la richiesta **non alla Servlet, ma al Container** in cui la Servlet è *deployed (installata)*.
- the Container **passa** alla Servlet la **request** e la **response** come oggetti Java
- il Container **invoca i metodi della servlet** (come ***doPost()*** o ***doGet()*** o ***init()*** etc.)



Che succede se ci sono più chiamate alla stessa servlet?

The container provides (1)

- **Communications Support.** Il Container fornisce un modo semplice per le vostre servlet di parlare con il vostro server web. Non dovete costruire un ServerSocket, ascoltare su una porta, creare flussi, ecc. Il contenitore conosce il protocollo tra il server web e se stesso, così che la vostra servlet non deve preoccuparsi di un'API tra, diciamo, il server web (Apache) e il vostro codice di applicazione web. Tutto ciò di cui dovete preoccuparvi è la vostra logica di business che va nella vostra Servlet (come accettare un ordine dal vostro negozio online)
- **Lifecycle Management.** Il Container controlla la vita e la morte delle vostre Servlet, si occupa di **caricare le classi, istanziare e inizializzare** le Servlet, **invocare i metodi** della Servlet e rendere le istanze delle Servlet idonee alla **garbage collection**. Grazie al Container non ci si deve preoccupare molto della gestione delle risorse

The container provides (2)

- **Multithreading Support.** Il Container crea automaticamente un nuovo thread Java per ogni richiesta servlet che riceve
- **Declarative Security.** Con un Container, puoi usare un XML deployment descriptor per configurare (e modificare) la sicurezza senza doverla codificare nella tua servlet. Puoi gestire e cambiare la tua sicurezza senza toccare e ricompilare i tuoi file sorgente Java
- **JSP Support.** Il Container si occupa di tradurre il codice JSP in Java e poi di compilarlo in .class

Precisazione per le prossime slides

- Nelle slide che seguono facciamo riferimento ad un'applicazione elementare (**non MVC**) che usa una sola servlet in grado di creare essa stessa una pagina html (senza rimandare la risposta alla jsp)
- Inoltre il web server non è mostrato (potete immaginare che sia il rettangolo che racchiude il container e l'applicazione)
- Nella servlet il termine «javax» va sostituito con «jakarta» (per essere conforme a Tomcat 10)

In the real world, 99.9% of all servlets override either the `doGet()` or `doPost()` method.

Notice... no `main()` method. The servlet's lifecycle methods (like `doGet()`) are called by the Container.

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

99.9999% of all servlets are `HttpServlet`s.

```
public class Ch2Servlet extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException {
```

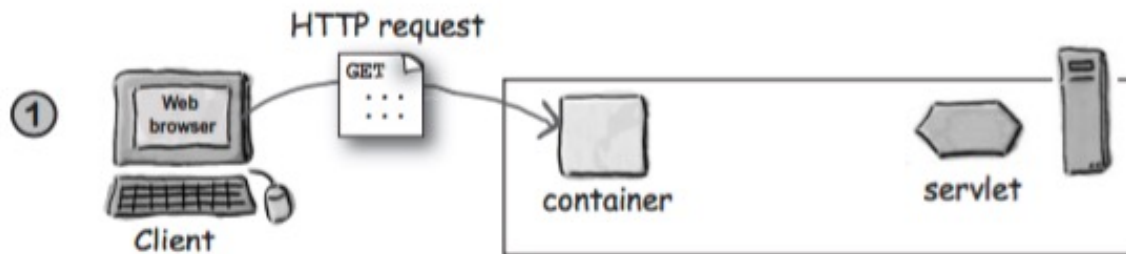
This is where your servlet gets references to the request and response objects which the container creates.

```
        PrintWriter out = response.getWriter();
        java.util.Date today = new java.util.Date();
        out.println("<html> " +
                    "<body>" +
                    "<h1 style='text-align:center>" +
                    "HF\'s Chapter2 Servlet</h1>" +
                    "<br>" + today +
                    "</body>" +
                    "</html>");
```

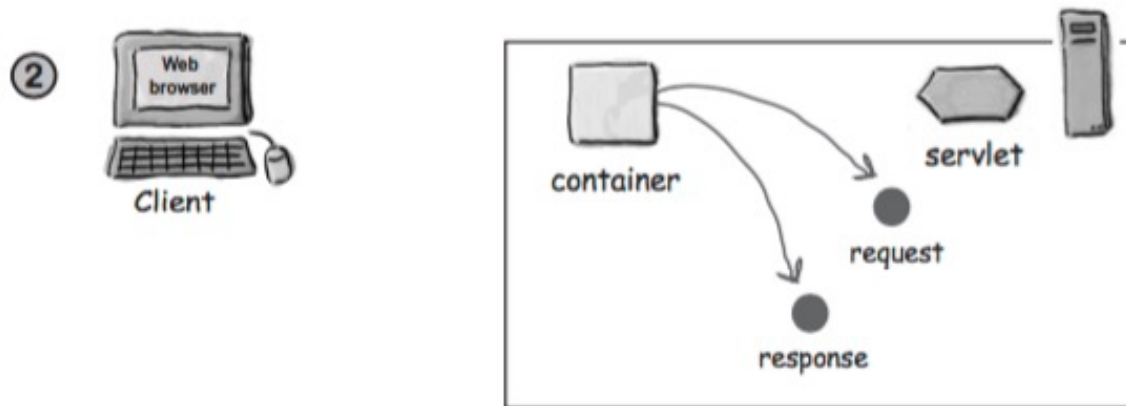
```
    }
}
```

You can get a `PrintWriter` from the response object your servlet gets from the Container. Use the `PrintWriter` to write HTML text to the response object. (You can get other output options, besides `PrintWriter`, for writing, say, a picture instead of HTML text.)

The request-response model



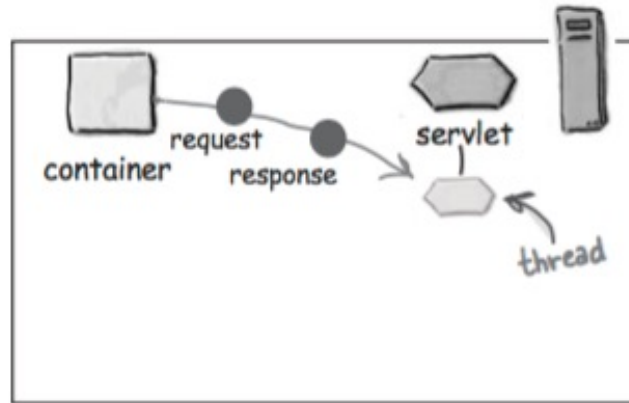
User clicks a link that has a URL to a servlet instead of a static page.



The container "sees" that the request is for a servlet, so the container creates two objects:

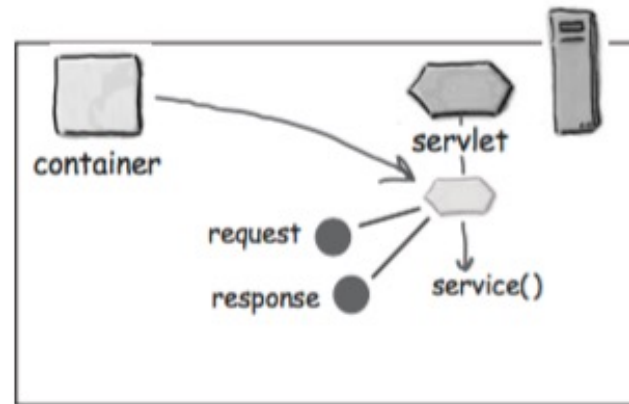
- 1) `HttpServletResponse`
- 2) `HttpServletRequest`

3



The container finds the correct servlet based on the URL in the request, creates or allocates a thread for that request, and passes the request and response objects to the servlet thread.

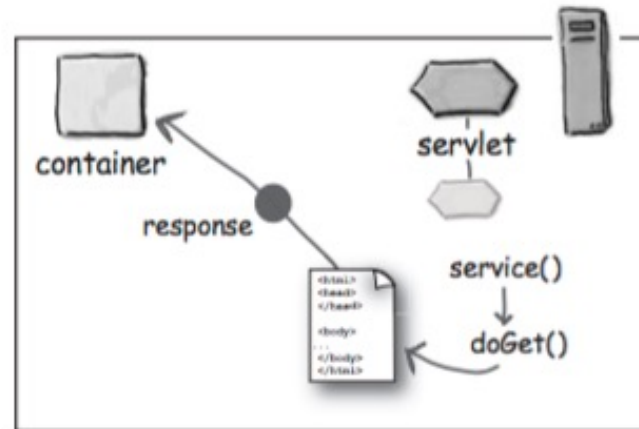
4



The container calls the servlet's service() method. Depending on the type of request, the service() method calls either the doGet() or doPost() method.

For this example, we'll assume the request was an HTTP GET.

5

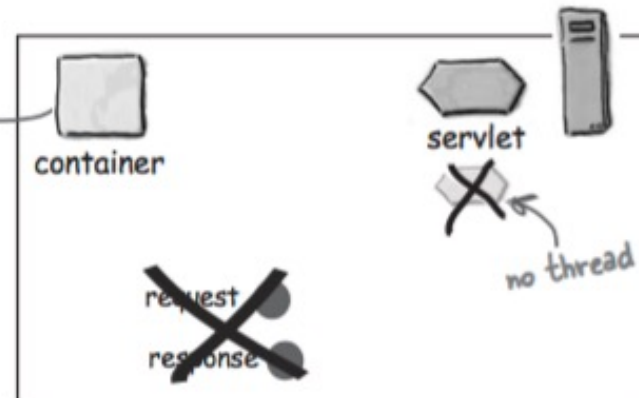


The doGet() method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!

6



HTTP response



The thread completes, the container converts the response object into an HTTP response, sends it back to the client, then deletes the request and response objects.

In the real world, 99.9% of all servlets override either the `doGet()` or `doPost()` method.

Notice... no `main()` method. The servlet's lifecycle methods (like `doGet()`) are called by the Container.

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

99.9999% of all servlets are `HttpServlet`s.

```
public class Ch2Servlet extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException {
```

This is where your servlet gets references to the request and response objects which the container creates.

```
        PrintWriter out = response.getWriter();
        java.util.Date today = new java.util.Date();
        out.println("<html> " +
                   "<body>" +
                   "<h1 style='text-align:center>" +
                   "HF\'s Chapter2 Servlet</h1>" +
                   "<br>" + today +
                   "</body>" +
                   "</html>");
```

```
    }
}
```

You can get a `PrintWriter` from the response object your servlet gets from the Container. Use the `PrintWriter` to write HTML text to the response object. (You can get other output options, besides `PrintWriter`, for writing, say, a picture instead of HTML text.)



Problemi nel mescolare Java e HTML in una servlet

- Il problema nel mescolare Java e HTML, come nella servlet precedente, è che la **logica dell'applicazione** e il modo in cui le informazioni sono **presentate nel browser** sono mescolati
 - Gli **application designers** ed i **web designers** sono persone diverse con competenze complementari e solo parzialmente sovrapposte
 - **Application designers** sono esperti in algoritmi complessi e database
 - **Web designers** si concentrano sulla composizione della pagina e sulla grafica
 - L'interfaccia può essere migrata (per esempio, verso il mobile) e la migrazione è facile se l'interfaccia e la logica dell'applicazione sono separate
- 

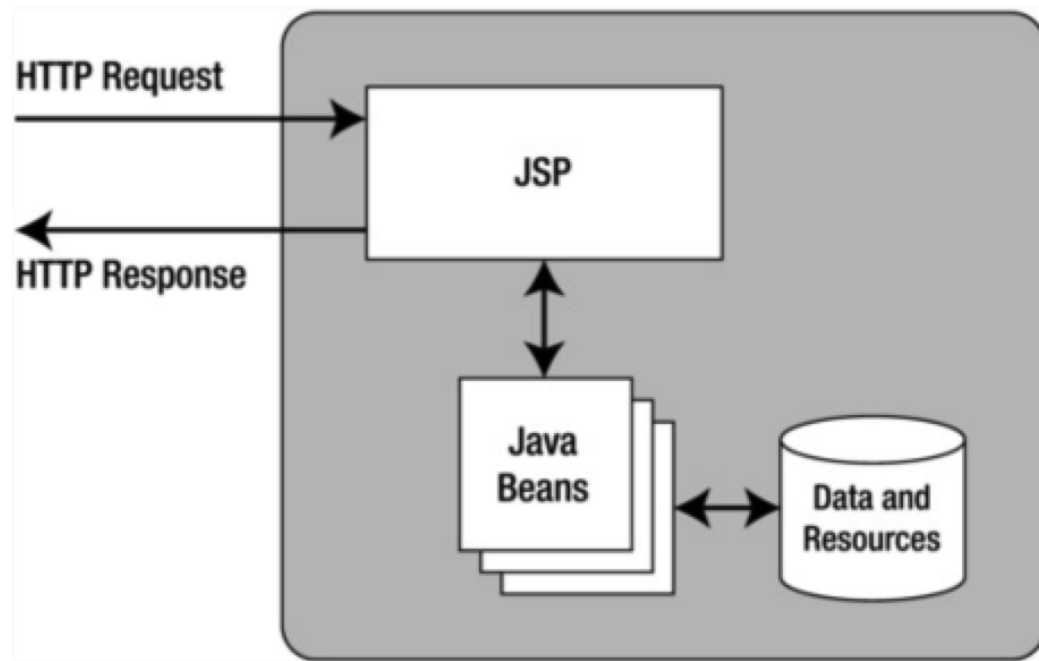
Java Model

- Nel progetto di applicazioni Web in Java, due modelli di ampio uso e di riferimento: **Model 1** e **Model 2**
- **Model 1** è un pattern semplice in cui codice responsabile per presentazione contenuti è **mescolato** con logica di business
 - Suggerito solo per piccole applicazioni (*sta diventando obsoleto nella pratica industriale*)
- **Model 2 (MVC)** come design pattern più complesso e articolato che **separa chiaramente** il livello **presentazione** dei contenuti dalla **logica** utilizzata per manipolare e processare i contenuti stessi
 - Suggerito per applicazioni di medio-grandi dimensioni

The Model 1 architecture

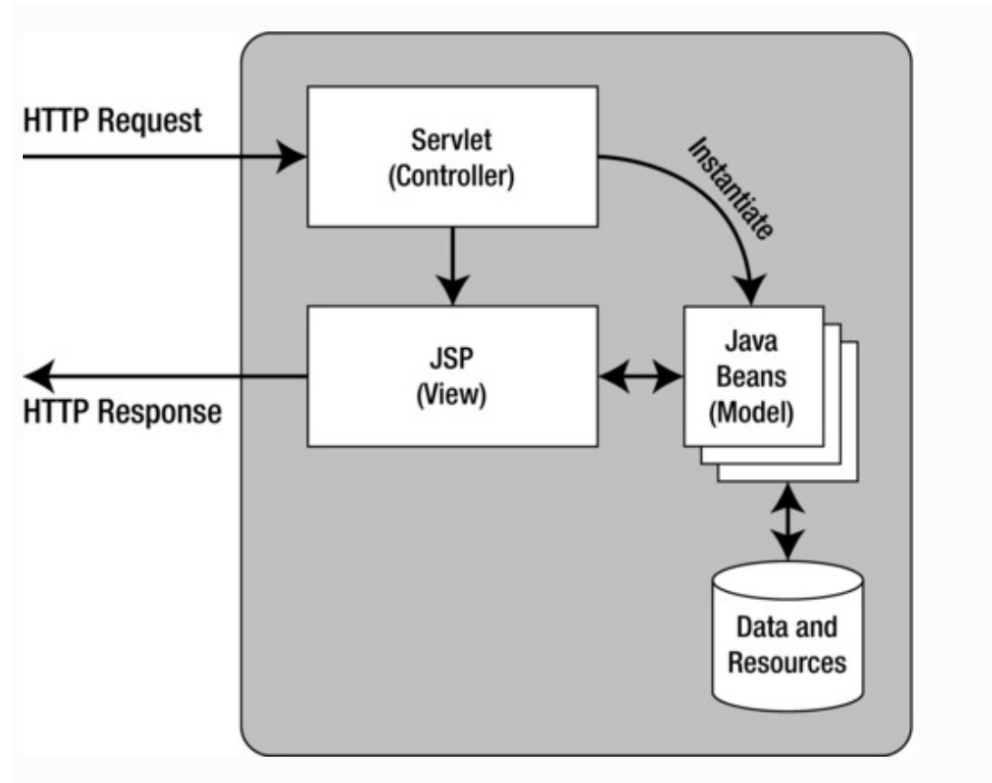
“Move the bulk of the application logic from JSP to Java classes (i.e., Java beans), which can then be used within JSP. This is called the JSP Model 1 architecture. JSP has to handle HTTP request”

- **JavaBeans** are classes that encapsulate many objects into a single object (the bean). They are serializable, have a zero-argument constructor, and allow access to properties using **getter** and **setter** methods.
- The name "Bean" was given to encompass this standard, which aims to create reusable software components for Java



The Model 2 architecture (MVC)

- A better solution, more suitable for larger applications, is to split the functionality further and **use JSP exclusively to format the HTML pages**



Architettura MVC (generale)

- Architettura adatta per applicazioni Web interattive
- **Model** – rappresenta **livello dei dati**, incluse operazioni per accesso e modifica. Model deve notificare le view associate quando viene modificato e deve supportare:
 - possibilità per la view di interrogare stato di model
 - possibilità per il controller di accedere alle funzionalità incapsulate dal model
- **View** – si occupa di visualizzare i contenuti del model. Accede ai dati tramite il model e specifica come dati debbano essere presentati
 - aggiorna la presentazione dei dati quando il model cambia
 - Fornisce i moduli per l'invio dell'input dell'utente verso il controller da parte del browser
- **Controller** – definisce il comportamento dell'applicazione
 - fa dispatching di richieste utente e seleziona la view per la presentazione
 - interpreta l'input dell'utente e lo mappa su azioni che devono essere eseguite da model (in una GUI stand-alone, input come click e selezione menu; in una applicazione Web, richieste HTTP GET/POST)

Mapping possibile su applicazioni Web Java-based

- In applicazioni Web conformi al Model 2, richieste del browser cliente vengono passate a Controller (implementato da Servlet)
- Il **Controller** si occupa di eseguire logica business necessaria per ottenere il contenuto da mostrare. Il Controller mette il contenuto nel **Model** (implementato con JavaBean o Plain Old Java Object - POJO) in un messaggio e decide a quale **View** (implementata da JSP) passare la richiesta
- La View si occupa del rendering contenuto (ad es. stampa dei valori contenuti in struttura dati o bean, ma anche operazioni più complesse come invocazione metodi per ottenere dati)




Mapping possibile su applicazioni Web Java-based (2)

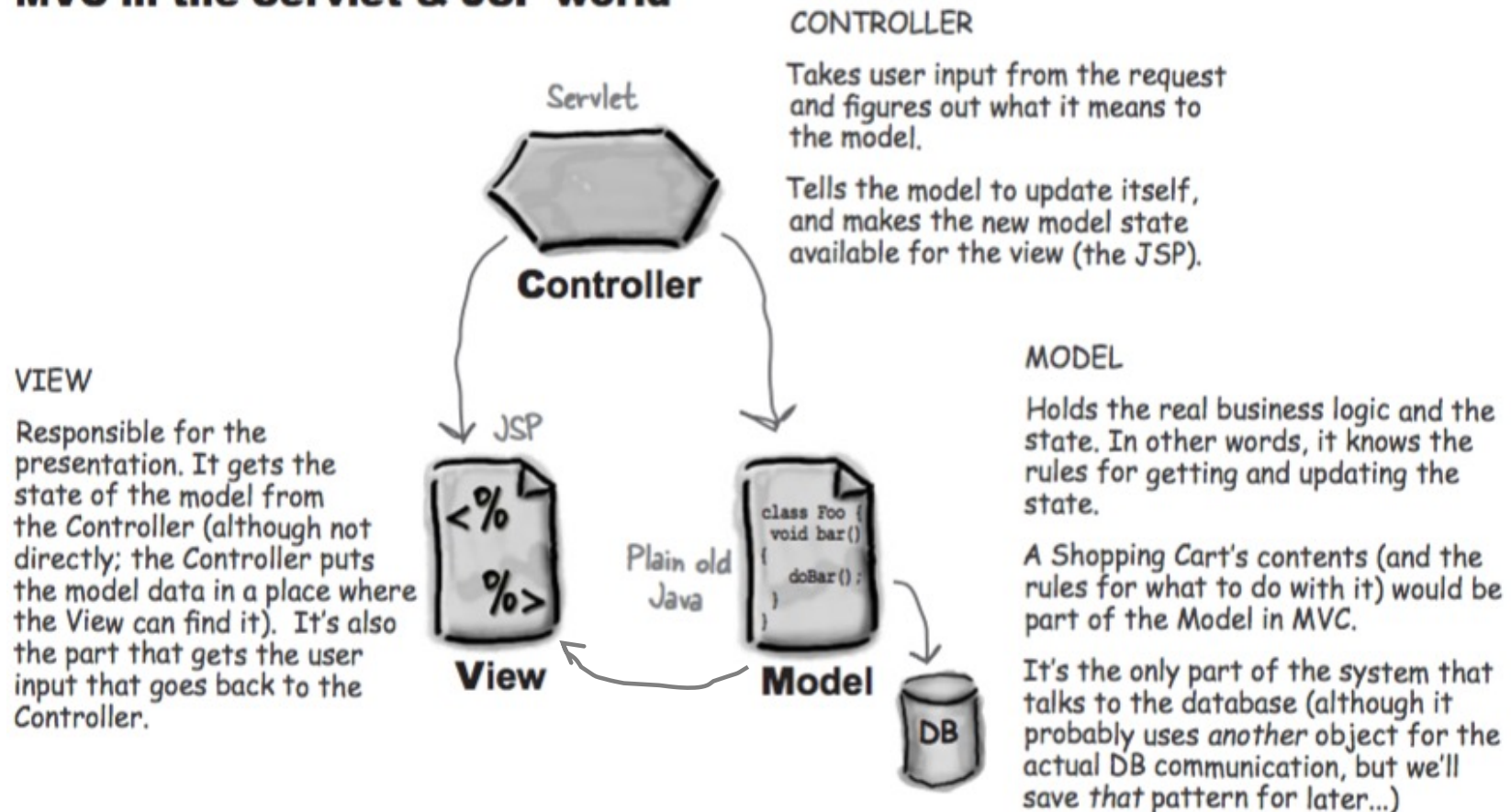
Servlet (**Controller**):

- processes the request
- handles the application logic
- instantiates Java beans (**Model**)

JSP (**View**):

- obtains data from the beans
 - format the response without having to know anything about what's going on behind the scenes
- 

MVC in the Servlet & JSP world





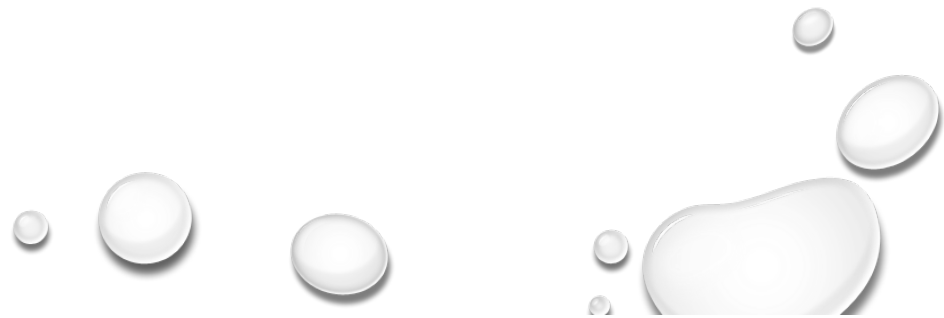
Example: Create and deploy an MVC Web app





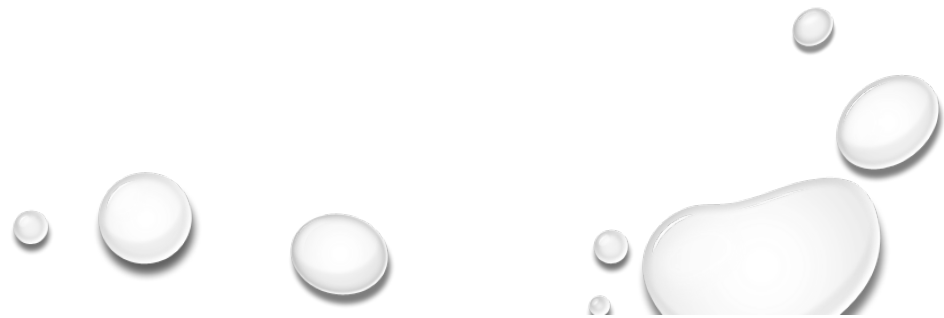
Example: Beer Advisor

Users will be able to:

- surf to Web app
 - answer a question
 - get back stunningly useful beer advice
- 

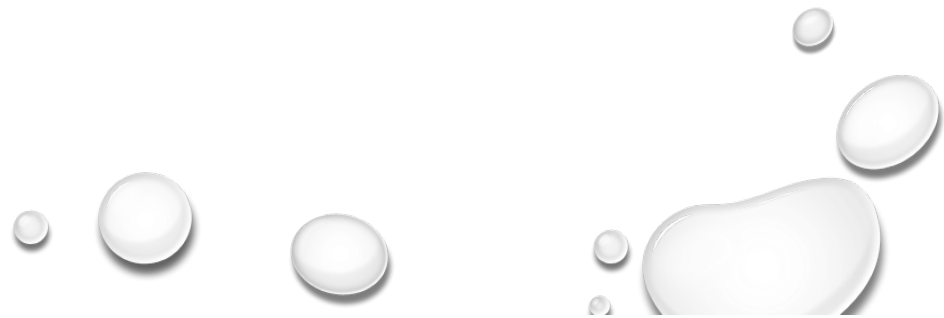


Iterative approach

- Il nostro piano è di costruire la Servlet per gradi, testando i vari collegamenti di comunicazione man mano che andiamo avanti
 - Alla fine la Servlet accetterà un parametro dalla richiesta, invocherà un metodo sul modello, salverà le informazioni in un posto che la JSP possa trovare, e inoltrerà la richiesta alla JSP
 - Ma **per questa prima versione**, il nostro obiettivo è solo quello di assicurarci che la pagina HTML possa invocare correttamente la Servlet, e che la Servlet riceva correttamente il parametro HTML
- 



First version (cercabirraV1.zip)

- Crea l'HTML nel tuo ambiente di sviluppo (IntelliJ)
 - Metti una copia del file form.html nella cartella
- 

A screenshot of a web browser window titled "form.html". The browser's address bar shows a Google search engine. The page content includes a title "Beer Selection Page", a label "Select beer characteristics", a color selection dropdown menu currently set to "light", and a "Submit" button at the bottom.

This page will be written in HTML, and will generate an HTTP Post request, sending the user's color selection as a parameter.

A screenshot of a web browser window titled "form.html". The browser's address bar shows a Google search engine. The page content includes a title "Beer Recommendations JSP", followed by two lines of text: "try: Jack's Pale Ale" and "try: Gout Stout".

This page will be a JSP that gives the advice based on the user's choice.

select-beer-form.html

```
SelectBeer.java  select-beer-form.html ✖
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Collecting Three Parameters</title>
5  </head>
6  <body bgcolor="#FDF5E6">
7      <H1 align="CENTER">Beer Selection Page</H1>
8
9      <form ACTION="SelectBeer" method="get">
10         Select beer characteristics
11         <p>
12             Color:
13             <select name="color" size="1">
14                 <option value="light">light</option>
15                 <option value="amber">amber</option>
16                 <option value="brown">brown</option>
17                 <option value="dark">dark</option>
18             </select>
19             <input type="submit">
20         </form>
21
22 </body>
23 </html>
```

Code for Servlet first version

```
package com.example.web;

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

/** Servlet that prints out the Beers of the selected color</a>. */
@WebServlet("/SelectBeer")
public class SelectBeer extends HttpServlet {

    @Override
    public void doGet(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);

    }
}
```

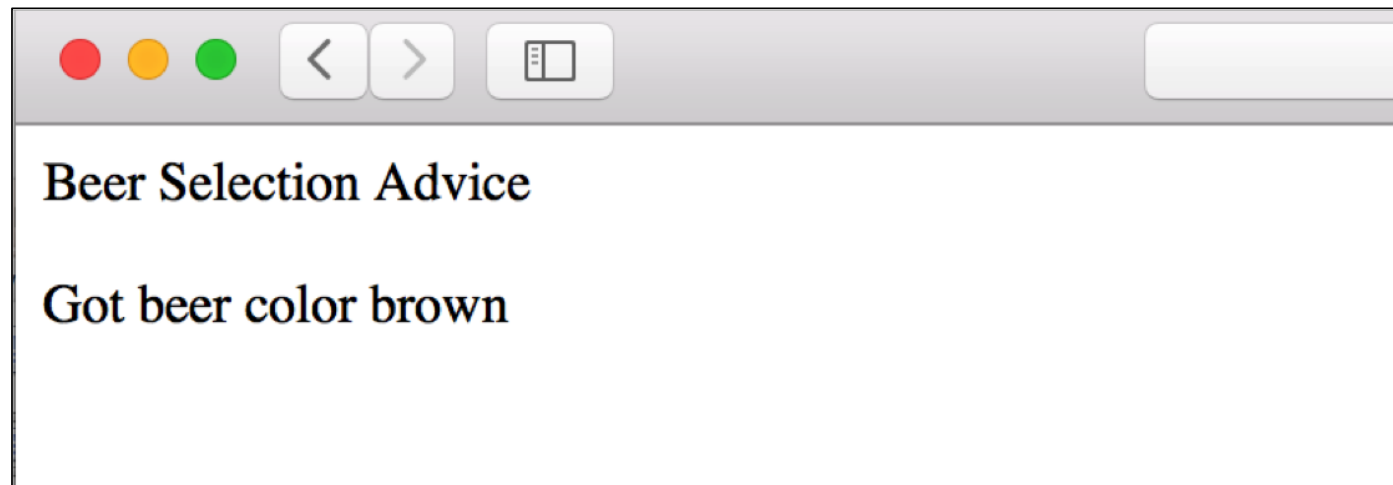
Test the form

Beer Selection Page

Select beer characteristics

Color:

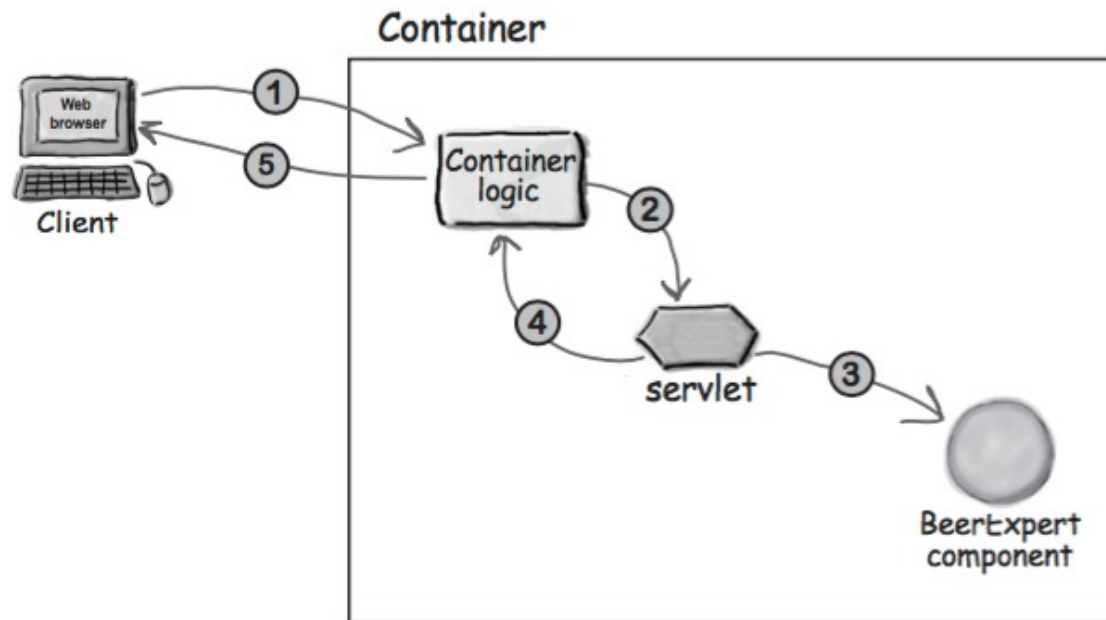
<http://localhost:8080/cercabirra/select-beer-form.html>



Building and testing the model class (cercabirraV2.zip)

- In MVC, the **model** tends to be the “**back-end**” of the application
- It's often the **legacy system** that's now being exposed to the Web
- In most cases it's just plain old Java code, with no knowledge of the fact that it might be called by Servlets. The model shouldn't be **tied down** to being used by only a single Web app, so it should be in its own utility packages
- **The specs for the model**
 - It exposes one method, **getBrands()**, that takes a preferred beer color (as a String), and returns an ArrayList of recommended beer brands (also as Strings)

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.

Build the test class for the model

- Create the test class for the model
- The model it's just like any other Java class, and you can test it without Tomcat

```
package com.example.model;

import java.util.*;

public class BeerExpert {

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

Enhancing the Servlet (version 2) to call the model

```
package com.example.web;
```

```
import com.example.model.*;
```

```
import javax.servlet.*;
```

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

```
import java.io.*;
```

```
import java.util.*;
```

```
public class BeerSelect extends HttpServlet {
```

```
    doGet
```

```
    public void doPost(HttpServletRequest request,
```

```
                        HttpServletResponse response)
```

```
        throws IOException, ServletException {
```

Don't forget the import for the package that BeerExpert is in.

We're modifying the original servlet, not making a new class.

```
String c = request.getParameter("color");
```

```
BeerExpert be = new BeerExpert();
```

```
List result = be.getBrands(c);
```

Instantiate the BeerExpert class and call getBrands().

```
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());
```

```
}
```

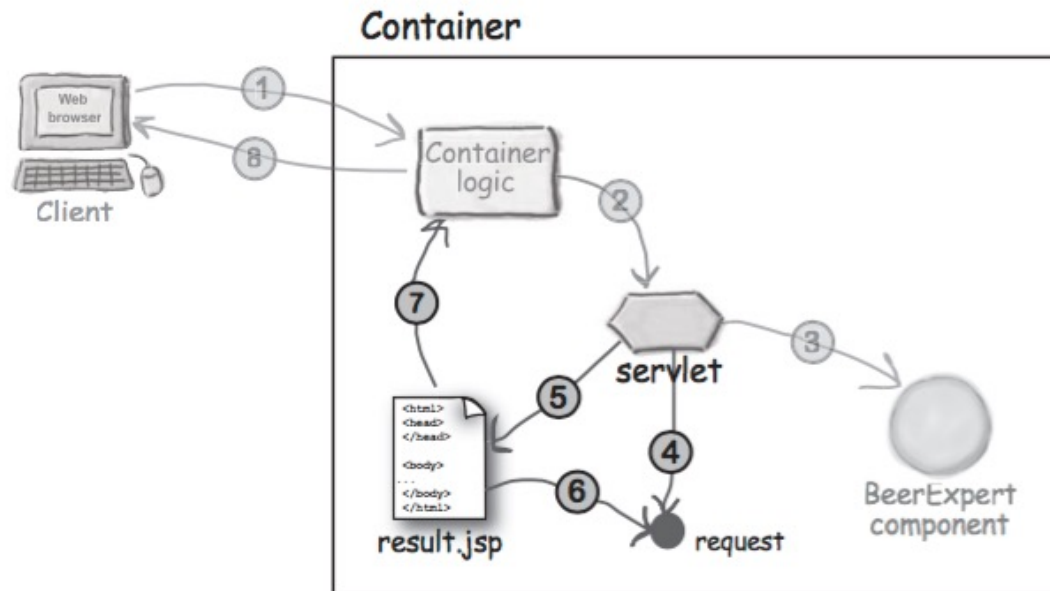
```
}
```

Print out the advice (beer brand items in the ArrayList returned from the model). In the final (third) version, the advice will be printed from a JSP instead of the servlet.

Enhancing the servlet to “call” the JSP (cercabirraV3.zip)

- In questo passo modificheremo la servlet per "chiamare" la JSP per produrre l'output (View)
- Il Container fornisce un meccanismo chiamato "request dispatching" che permette ad un componente gestito dal Container di chiamarne un altro, ed è quello che useremo: la servlet otterrà le informazioni dal modello, le salverà nell'oggetto request, poi invierà la richiesta alla JSP
- Le modifiche importanti che dobbiamo fare al servlet:
 - Aggiungere la risposta del componente del modello all'oggetto della richiesta, in modo che la JSP possa accedervi (passo 4 del processo 2*** nella prossima figura)
 - Chiedere al contenitore di inoltrare la richiesta a "result.jsp" (Passo 5 nel Processo 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.

Code for Servlet version 3

```
package com.example.web;

import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class BeerSelect extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {

        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>");
```


Code for servlet version 3 (2)

```
request.setAttribute("styles", result);
```

← Add an attribute to the request object for the JSP to use. Notice the JSP is looking for "styles".

```
RequestDispatcher view =
```

```
    request.getRequestDispatcher("result.jsp");
```

← Instantiate a request dispatcher for the JSP.

```
view.forward(request, response);
```

← Use the request dispatcher to ask the Container to crank up the JSP, sending it the request and response.

Create the JSP “view” that gives the advice (result.jsp)

Here's the JSP...

```
<%@ page import="java.util.*" %>
```

← This is a “page directive”
(we’re thinking it’s pretty
obvious what this one does).

```
<html>
```

```
<body>
```

```
<h1 align="center">Beer Recommendations JSP</h1>
```

← Some standard HTML (which is known as
“template text” in the JSP world).

```
<p>
```

```
<%
```

```
    List styles = (List)request.getAttribute("styles");
```

```
    Iterator it = styles.iterator();
```

```
    while(it.hasNext()) {
```

```
        out.print("<br>try: " + it.next());
```

```
    }
```

```
%>
```

Some standard Java sitting
inside <% %> tags (this is
known as scriptlet code).

← Here we’re getting an attribute
from the request object. A
little later in the book, we’ll
explain everything about
attributes and how we managed
to get the request object..

```
</body>
```

```
</html>
```


Test the app

4 - Test the app via form.html

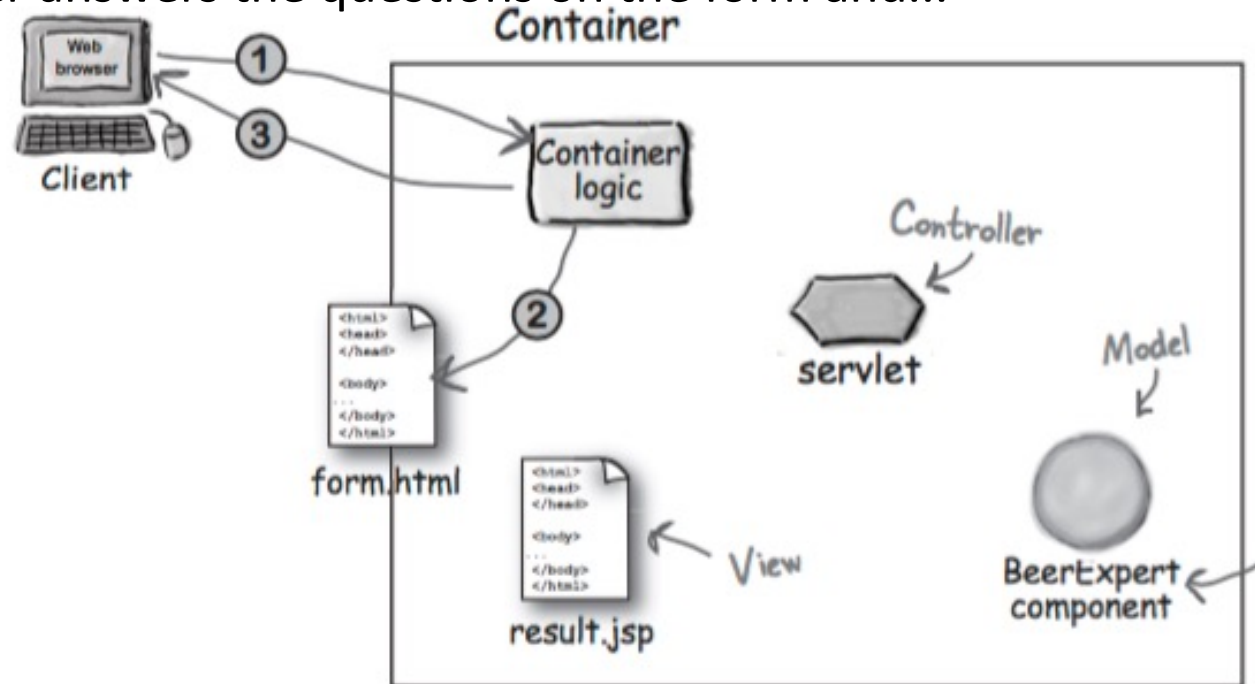


Here's what you should see! →

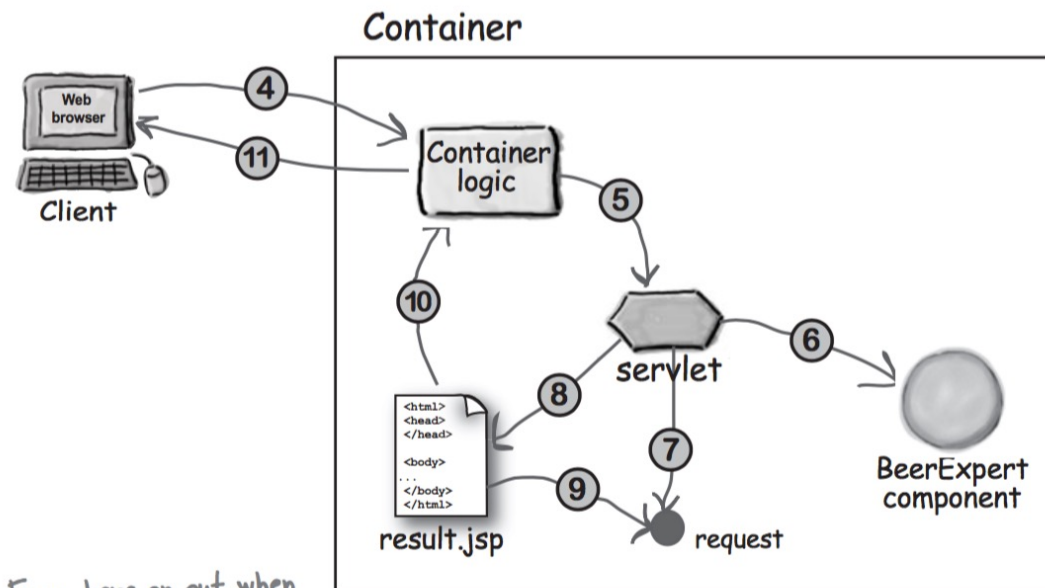


Overall Process

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...



Overall Process



From here on out when you don't see the web server, assume it's there.

4. The browser sends the request data to the Container
5. The Container sends 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
10. The JSP generates a page for the Container
11. The Container returns the page to the user