



CORSO DI LAUREA IN INFORMATICA

# Tecnologie Software per il Web

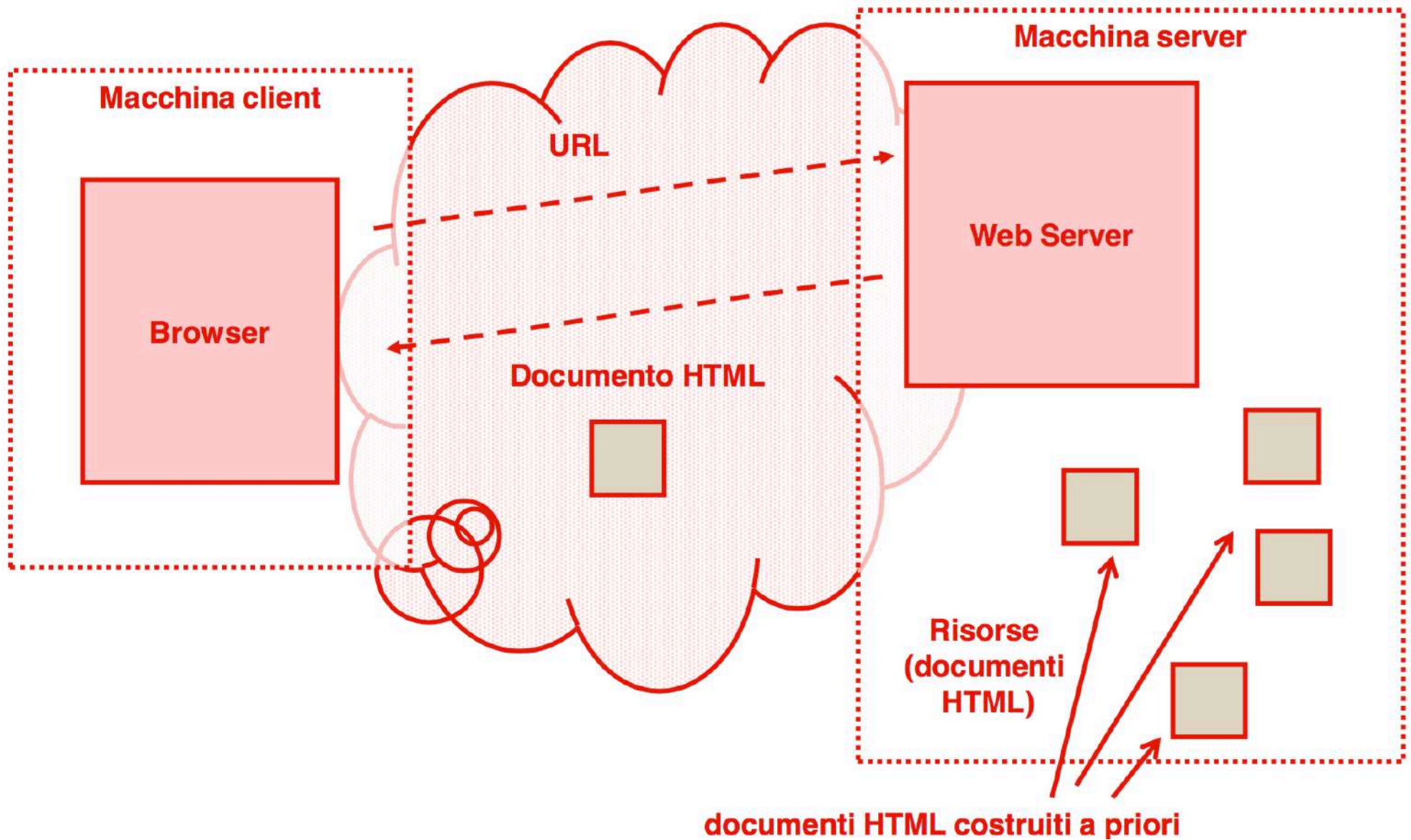
WEB DINAMICO & INTRODUZIONE ALLE SERVLET

a.a. 2019-2020

# Web Statico

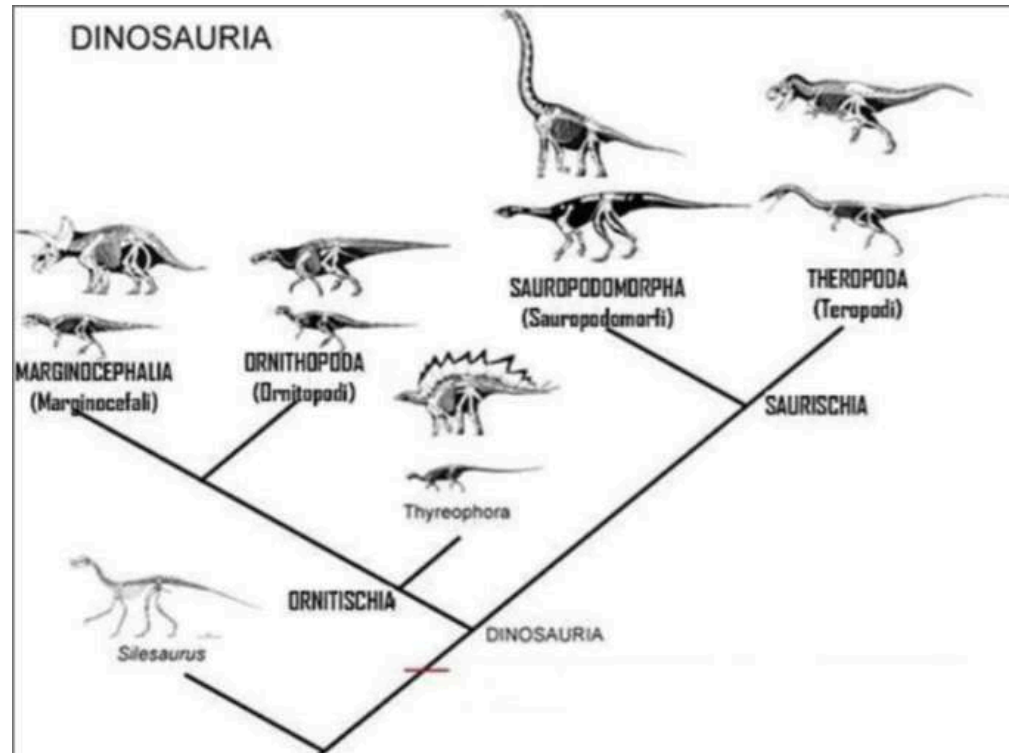
- Il modello che abbiamo analizzato finora, basato sul concetto di ipertesto distribuito, ha una natura essenzialmente statica
- Anche se l'utente può percorrere dinamicamente l'ipertesto in modi molto diversi, l'insieme dei contenuti è prefissato staticamente:
  - Pagine vengono preparate staticamente a priori
  - Non esistono contenuti composti dinamicamente in base all'interazione con l'utente
  - È un modello semplice, potente, di facile implementazione, efficiente, ma presenta **evidenti limiti**

# Modello web statico



# Limiti del modello statico

- Per capire quali sono i limiti del modello statico e come possono essere superati proviamo a ragionare su un semplice esempio
- Vogliamo costruire un'enciclopedia dei Dinosauri consultabile via Web:  
**<http://www.dino.it>**
- Possiamo creare una pagina HTML per ogni specie di dinosauro con testi e immagini
- Possiamo poi creare una pagina iniziale che fa da indice basandoci sulla classificazione scientifica
- Ogni voce è un link alla pagina che descrive un dinosauro



## Limiti del modello statico (2)

- Se vogliamo rendere più agevole l'accesso alle schede possiamo anche predisporre un'altra pagina con un indice analitico che riporta le specie di dinosauri in ordine alfabetico
- Tutto questo può essere realizzato facilmente con gli strumenti messi a disposizione dal Web statico...
- Il modello va però in crisi se proviamo ad aggiungere una funzionalità molto semplice: **ricerca per nome**
  - Quello che ci serve è una pagina con un **form**, costituito da un semplice campo di input e da un bottone, che ci consenta di inserire il nome di un dinosauro e di accedere direttamente alla pagina che lo descrive



# Ricerca

- Vediamo il codice della pagina HTML: abbiamo usato il metodo GET per semplicità

```
<html>
<head>
  <title> Ricerca dinosauri </title>
</head>
<body>
  <p>Enciclopedia dei dinosauri - Ricerca</p>
  <form method="GET" action="http://www.dino.it/cerca">
    <p>Nome del dinosauro
    <input type="text" name="nomeTxt" size="20">
    <input type="submit" value="Cerca" name="cercaBtn">
    </p>
  </form>
</body>
</html>
```

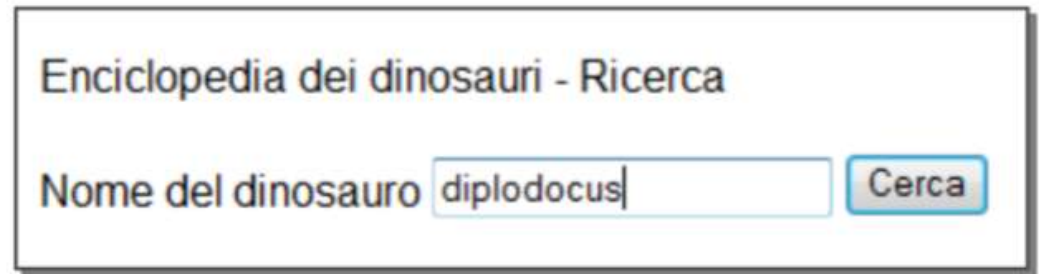
Enciclopedia dei dinosauri - Ricerca

Nome del dinosauro |

Cerca

## Eseguire la ricerca

- Se scriviamo il nome di un dinosauro e premiamo il bottone cerca, si attiva una invocazione HTTP di tipo GET con un URL di questo tipo:



Enciclopedia dei dinosauri - Ricerca

Nome del dinosauro

**`www.dino.it/cerca?nomeTxt=diplodocus&cercaBtn=Cerca`**

- Il Web Server non è in grado di interpretare immediatamente questa chiamata (URL con query) perché richiede l'esecuzione dinamica di un'applicazione legata al particolare contesto
- È quindi necessaria un'estensione specifica: *un programma scritto appositamente per l'enciclopedia che*
  1. Interpreti i parametri passati nel GET
  2. Cerchi nel file system la pagina *"diplodocus.html"*
  3. La restituisca al Web Server per l'invio al client



# Output

- Il programma elabora i dati in ingresso ed emette un output per il client in attesa di risposta
- Per passare i dati al server il programma usa funzioni di scrittura (standard output o output stream)
- Il server preleva i dati dallo standard output (o output stream) e li invia al client incapsulandoli in messaggio HTTP, ad esempio:

```
HTTP/1.0 200 OK
Date: Wednesday, 02-Feb-94 23:04:12 GMT
Server: NCSA/1.1
MIME-version: 1.0
Last-modified: Monday, 15-Nov-93 23:33:16 GMT
Content-type: text/html
...
<HTML><HEAD><TITLE>
...
```

Deve contenere almeno  
header content-type; altri  
opzionali...

Parte generata  
da Web server

Parte generata  
dal programma





# Programma e dinosauri

- Un meccanismo come quello appena descritto consente di risolvere il problema della ricerca nell'esempio dell'enciclopedia
- Il browser usando il metodo GET invia il contenuto del campo di ricerca nella parte query dell'URL:

**`www.dino.it/cerca?nomeTxt=diplodocus&cercaBtn=Cerca`**

- Il Web Server passa i parametri presenti nell'URL ad un programma denominato **cerca**
  - che usa il valore del parametro **nomeTxt** per cercare le pagine che contengono il termine inserito nel campo di ricerca (*diplodocus*)
  - che usa stdout/output stream per costruire una pagina con un elenco di link alle pagine che contengono il termine

# Altri problemi...

- La nostra enciclopedia ha anche un problema di manutenibilità...
- Se vogliamo aggiungere un dinosauro dobbiamo infatti:
  - Creare una nuova pagina con una struttura molto simile alle altre
  - Aggiungere il link alla pagina nell'indice principale (quello basato sulla classificazione delle specie)
  - Aggiungere un link nell'indice alfabetico
- La pagina di ricerca invece non richiede alcuna modifica
- Se poi volessimo cambiare l'aspetto grafico della nostra enciclopedia dovremmo rifare una per una tutte le pagine

# Soluzione

- *Una soluzione ragionevole è quella di separare gli aspetti di contenuto da quelli di presentazione*
- Ad esempio, utilizziamo un **database relazionale** per memorizzare le informazioni relative ad ogni dinosauro
- Realizziamo alcuni programmi che generano dinamicamente l'enciclopedia:
  - **scheda**: crea una pagina con la scheda di un determinato dinosauro
  - **indice**: crea la pagina di indice per specie (tassonomia)
  - **alfabetico**: crea l'indice alfabetico
  - **cerca**: restituisce una pagina con i link alle schede che contengono il testo inserito dall'utente
- Tutti e 4 i programmi usano il DB per ricavare le informazioni utili per la costruzione della pagina

# Struttura del database

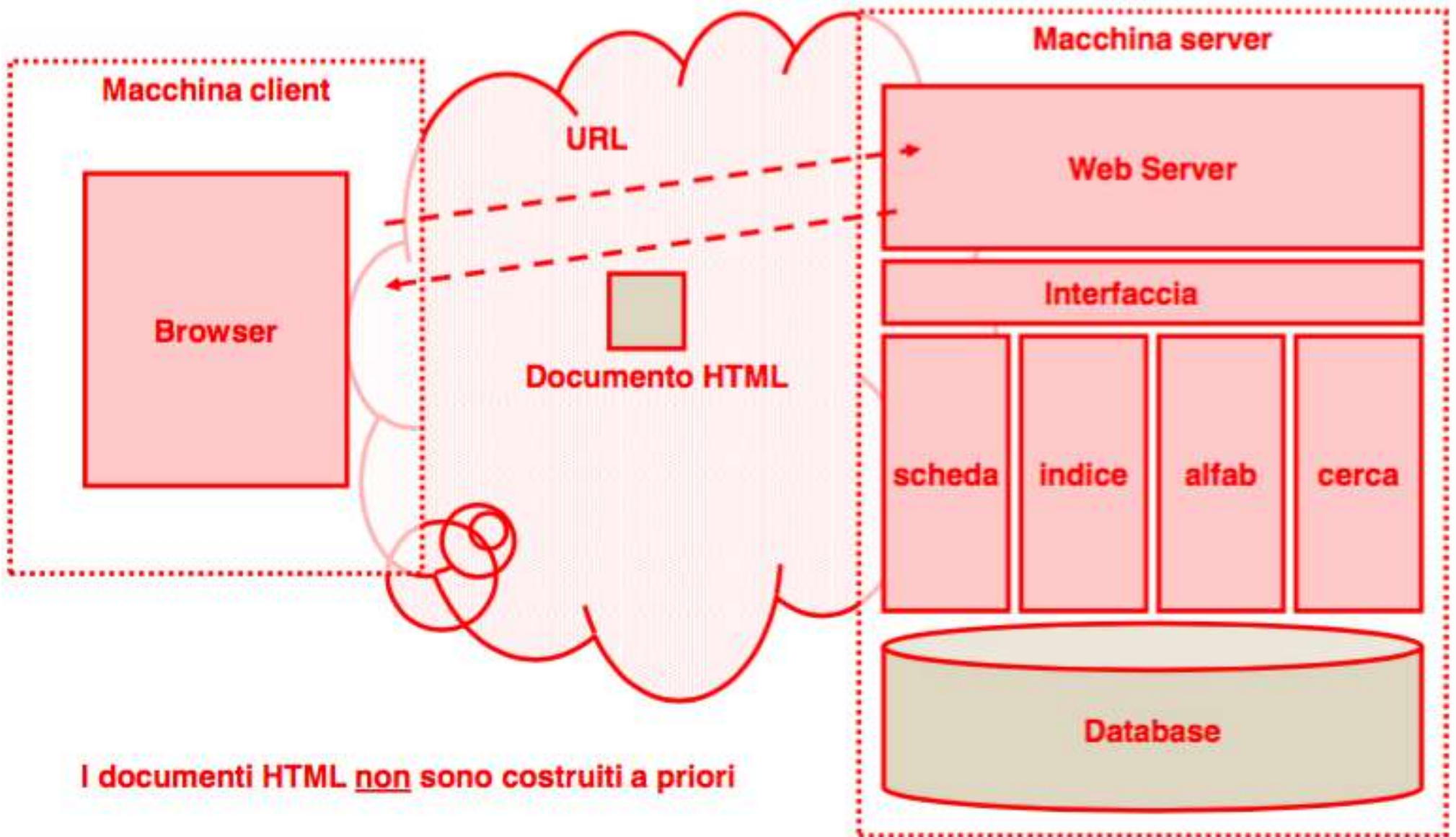
- Per semplicità possiamo utilizzare una sola tabella con la struttura sotto riportata (*non è normalizzata*)
- Il campo **testo** serve a includere il contenuto in formato HTML
- Gli altri campi permettono di costruire agevolmente l'indice basato sulla tassonomia

Specie	Ordine	Famiglia	Genere	Testo
Dicraeosaurus hansemani	Saurischia	Dicraeosauridae	Dicreaosaurus	<p>Il <b>dicreosauro</b> (gen. Dicraeosaurus) è un dinosauro erbivoro vissuto in Africa orientale nel Giurassico superiore (Kimmeridgiano, circa 150 milioni di anni fa).
...	...	...	...	...
...	...	...	...	...

# Vantaggi della nuova soluzione

- In questo modo la gestione dell'enciclopedia è sicuramente più semplice:
  - Basta inserire un record nel database per aggiungere una nuova specie
  - L'indice tassonomico e quello alfabetico si aggiornano automaticamente
  - È anche possibile cambiare agevolmente layout di tutte le pagine
    - Possibile utilizzare una pagina HTML di base (**template**) con tutte le parti fisse
    - Il programma **scheda** si limita a caricare il template e a inserire le parti variabili
    - Per cambiare l'aspetto grafico di tutte le schede è sufficiente agire una sola volta sul **template**

# Modello web dinamico



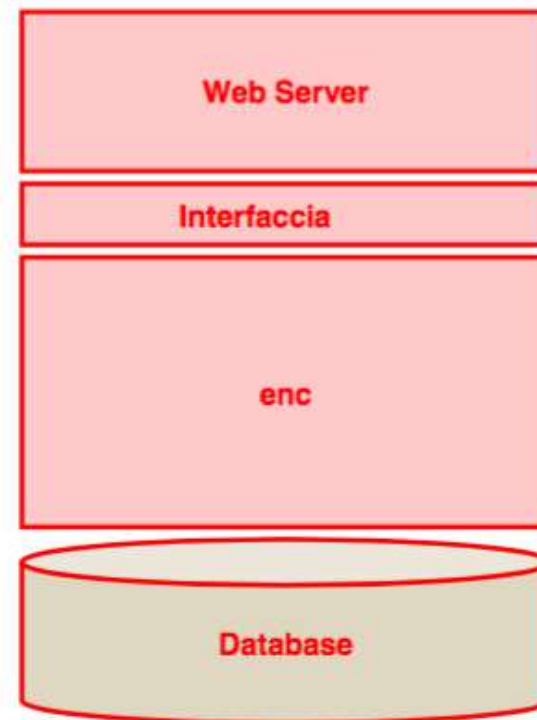
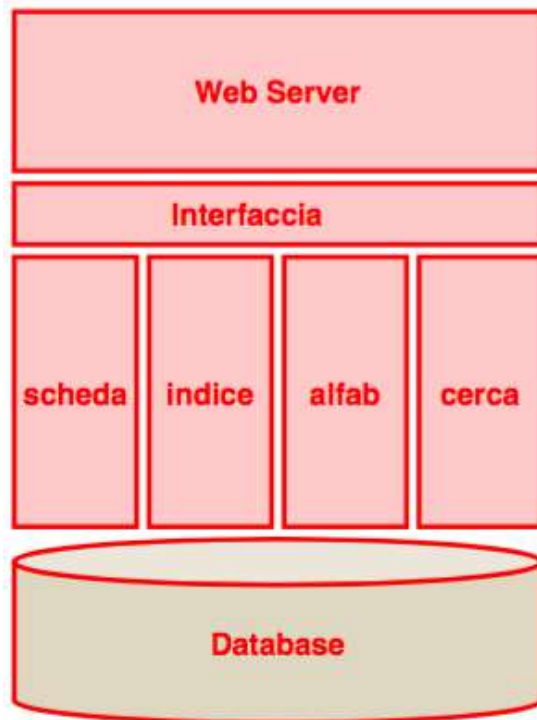


# Tutto a posto? NO!

- L'architettura che abbiamo appena visto presenta numerosi vantaggi ma soffre anche di diversi problemi
- Ci sono problemi di prestazioni:
  - *ogni volta che viene invocata un programma si crea un processo che viene distrutto alla fine dell'elaborazione*
- ❖ Ogni programma deve reimplementare tutta una serie di parti comuni (*manca di moduli di base accessibili a tutti i programmi lato server*):  
*accesso al DB, logica di interpretazione delle richieste HTTP e di costruzione delle risposte, gestione dello stato, ecc.*
- Abbiamo scarse garanzie sulla sicurezza

# Una sola applicazione

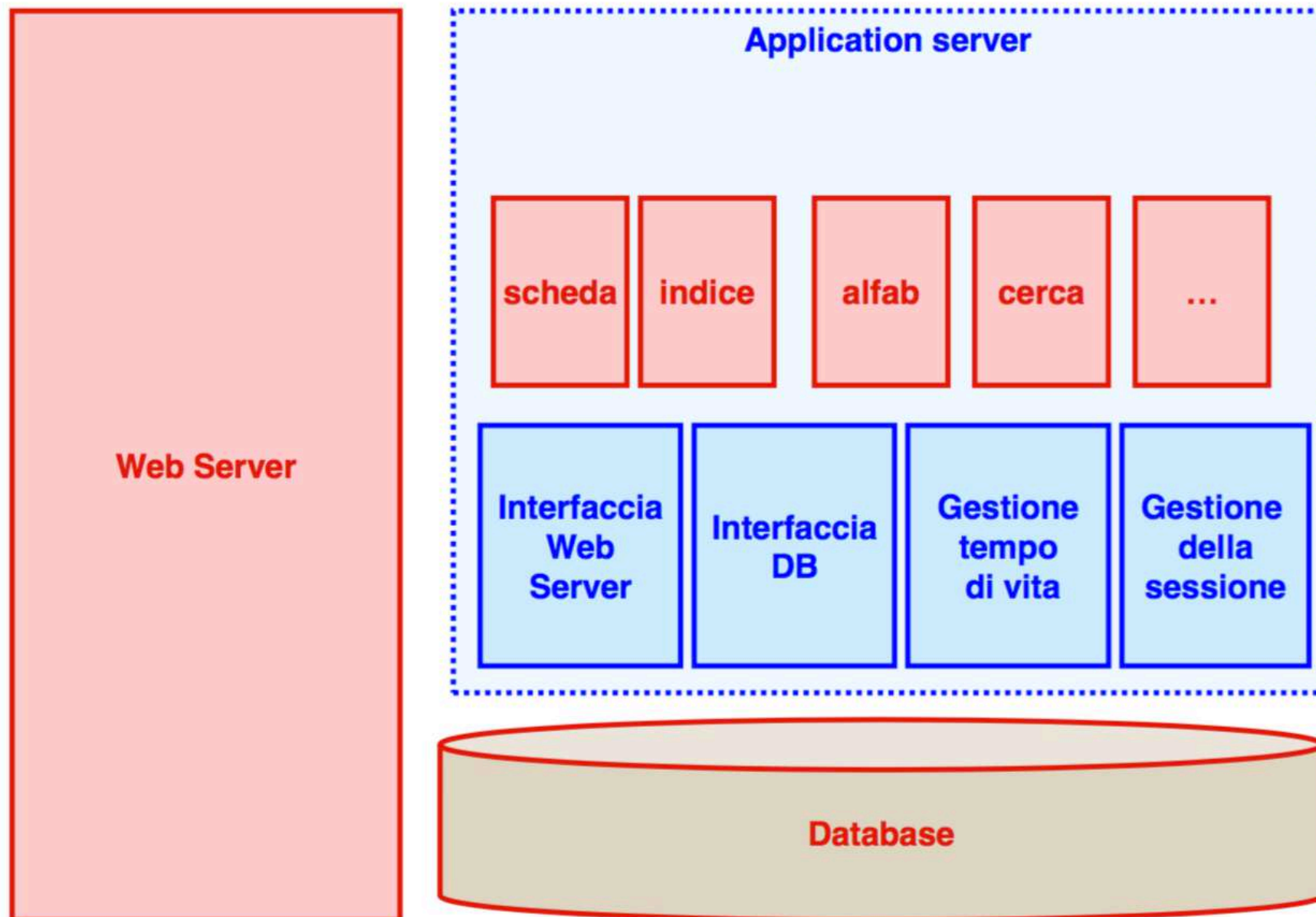
- ❖ Per ovviare al penultimo punto si potrebbe realizzare un solo programma (**enc**) che implementa tutte e quattro le funzionalità
- In questo modo però si ha un'applicazione monolitica e si perdono i vantaggi della modularità
  - Gli altri problemi rimangono invariati...



# Contentitore (Container)

- La soluzione migliore è quella di realizzare un contenitore in cui far “vivere” le funzioni server-side
- Il contenitore si preoccupa di fornire i servizi di cui le applicazioni hanno bisogno:
  - Interfacciamento con il Web Server
  - Gestione del tempo di vita (attivazione on-demand delle funzioni)
  - Interfacciamento con il database
  - Gestione della sicurezza
- Si ha così una soluzione modulare in cui le funzionalità ripetitive vengono portate a fattor comune
- Un ambiente di questo tipo prende il nome di **application server**

# Architettura basata su Application server



# Application server e tecnologie server side

- Le due tecnologie più diffuse nell'ambito degli **application server**:
  - .NET di Microsoft ed evoluzioni
  - Java J2EE
- Altre tecnologie interessanti:
  - Basate su Python (zope, ...)
- Altre soluzioni hanno una struttura più semplice e non sono application server a tutti gli effetti (si parla di moduli di estensione del Web Server – comunque interessanti per applicazioni Web a rapida prototipazione e basso costo):
  - PHP (Hypertext Preprocessor)

## Altri aspetti: lo stato

- L'enciclopedia dei dinosauri è un'applicazione **stateless**:
  - Il server e le applicazioni non hanno necessità di tener traccia delle chiamate precedenti
- L'interazione tra un Client e un Server può essere infatti di due tipi:
  - **Stateful**: esiste stato dell'interazione e quindi l'**n-esimo** messaggio può essere messo in relazione con gli **n-1** precedenti
  - **Stateless**: non si tiene traccia dello stato, ogni messaggio è indipendente dagli altri



# Interazione stateless

- In termini generali, un'interazione stateless è “feasible” senza generare grossi problemi solo se il protocollo applicativo è progettato con **operazioni idempotenti**
  - *Operazioni idempotenti producono sempre lo stesso risultato, indipendentemente dal numero di messaggi  $M$  ricevuti dal server stesso*
    - Ad es. un server fornisce sempre la stessa risposta  $R$  a un messaggio  $M$
- Nel nostro caso tutte le operazioni gestite dall'enciclopedia (**indice, alfabetico, scheda e cerca**) sono idempotenti
  - In generale, molto spesso abbiamo a che fare con operazioni idempotenti nelle applicazioni Web
- *Esempi di operazioni non-idempotenti?*



# Interazioni stateful

- Non tutte le applicazioni possono fare a meno dello stato
  - *Esempio banale*: se è prevista un'autenticazione, è molto comodo (necessario a fini di usabilità utente) tener traccia fra una chiamata e l'altra del fatto che l'utente si è autenticato e quindi nasce l'esigenza di avere uno stato
- In generale, tutte le volte in cui abbiamo bisogno di personalizzazione delle richieste Web, possiamo beneficiare di interazione **stateful**
  - Ad es. se estendiamo la nostra applicazione per consentire a utenti autorizzati di *modificare le schede* dei dinosauri via Web o
  - se vogliamo fornire *pagine iniziali di accesso differenziate* sulla base di storia precedente o livello di expertise utente, anche la nostra applicazione cessa di poter essere stateless

# Diversi tipi di stato

- Parlando di applicazioni Web è possibile classificare lo stato in modo più preciso:
  - **Stato di esecuzione** (insieme dei dati parziali per una elaborazione): rappresenta un avanzamento in una esecuzione; per sua natura è uno stato volatile; può essere mantenuto in memoria lato server come stato di uno o più oggetti
  - **Stato di sessione** (insieme dei dati che caratterizzano una interazione con uno specifico utente): la sessione viene gestita di solito in modo unificato attraverso l'uso di istanze di oggetti specifici (**supporto a oggetti sessione**)
  - **Stato informativo persistente** (ad esempio gli ordini inseriti da un sistema di eCommerce): viene normalmente mantenuto in una struttura persistente come un **database**

# Il concetto di sessione

- La **sessione** rappresenta lo **stato** associato ad una sequenza di pagine visualizzate da un utente:
  - Contiene tutte le informazioni necessarie durante l'esecuzione
    - Informazioni di sistema: IP di provenienza, lista delle pagine visualizzate, ...
    - Informazioni di natura applicativa: nome e cognome, username, quanti e quali prodotti ha inserito nel carrello per un acquisto, ...
- Lo **scope** di sessione è dato da:
  - **Tempo di vita** della interazione utente (lifespan)
  - **Accessibilità**: usualmente concesso alla richiesta corrente e a tutte le richieste successive provenienti dallo stesso processo browser

# Sessione come base per la conversazione

- La **conversazione** rappresenta una sequenza di pagine di senso compiuto (ad esempio l'insieme delle pagine necessarie per comperare un prodotto)
- È univocamente definita dall'insieme delle pagine che la compongono e dall'insieme delle interfacce di input/output per la comunicazione tra le pagine (**flusso della conversazione**)

# Esempio di conversazione: acquisto online

1. L'utente inserisce username e password: **inizio della conversazione**
2. Il server riceve i dati e li verifica con i dati presenti nel DB dei registrati: **viene creata la sessione**
3. L'utente sfoglia il catalogo alla ricerca di un prodotto
  - Il server lo riconosce attraverso i dati di sessione (**lettura sessione**)
4. L'utente trova il prodotto e lo mette nel carrello
  - La **sessione viene aggiornata** con informazioni del prodotto
5. L'utente compila i dati di consegna
6. L'utente provvede al pagamento, **fine della conversazione** di acquisto
  - L'ordine viene salvato nel DB (stato persistente)
  - La **sessione è ancora attiva** e l'utente può fare un altro acquisto o uscire dal sito

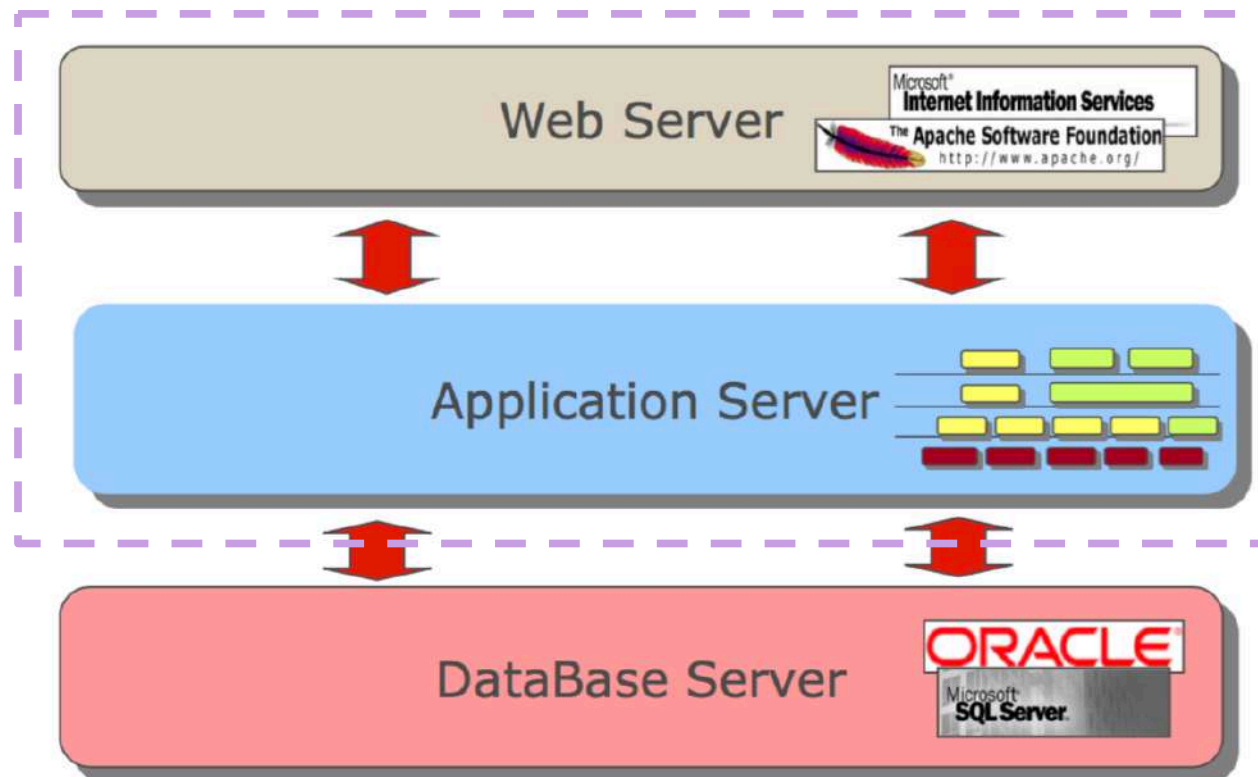


# Tecniche per gestire lo stato

- Lo **stato di sessione** deve presentare i seguenti requisiti:
  - Deve essere condiviso da Client e Server
  - È associato a una o più conversazioni effettuate da un singolo utente
  - Ogni utente possiede il suo singolo stato
- Ci sono due tecniche di base per gestire lo stato, non necessariamente alternative ma integrabili:
  1. Utilizzo del meccanismo dei **cookie** (*storage lato client*)
  2. Gestione di **uno stato sul server** per ogni utente collegato (*sessione server-side*)
- *La gestione della sessione è uno dei supporti orizzontali messi a disposizione da un application server*

# Architettura three-tier (frequente nei sistemi Web)

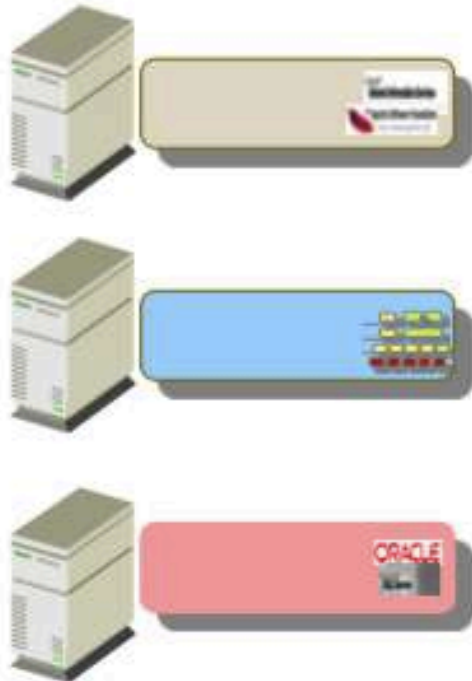
- Architettura a 3 tier, che può collassare a due in assenza di application server (sempre più raro al giorno d'oggi)



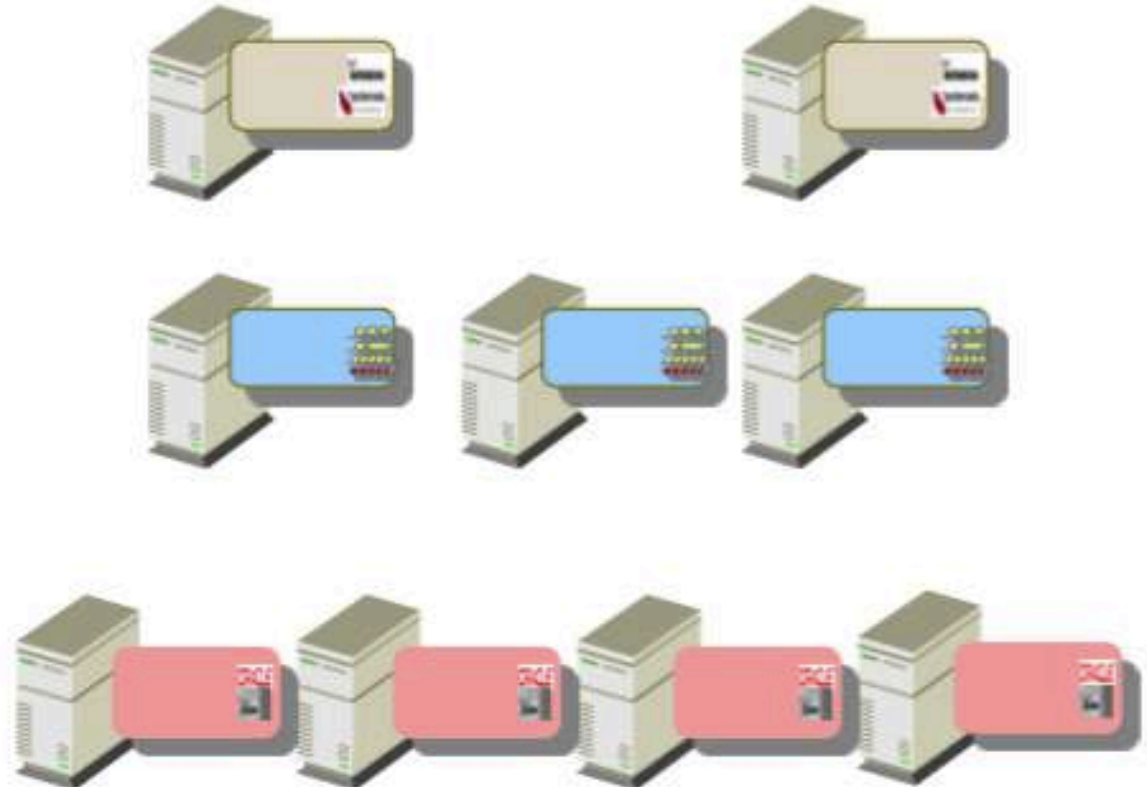
# Distribuzione verticale e orizzontale

- Obiettivo: gestione della fault tolerance e anche del bilanciamento di carico a fine di maggiori performance

## Distribuzione Verticale



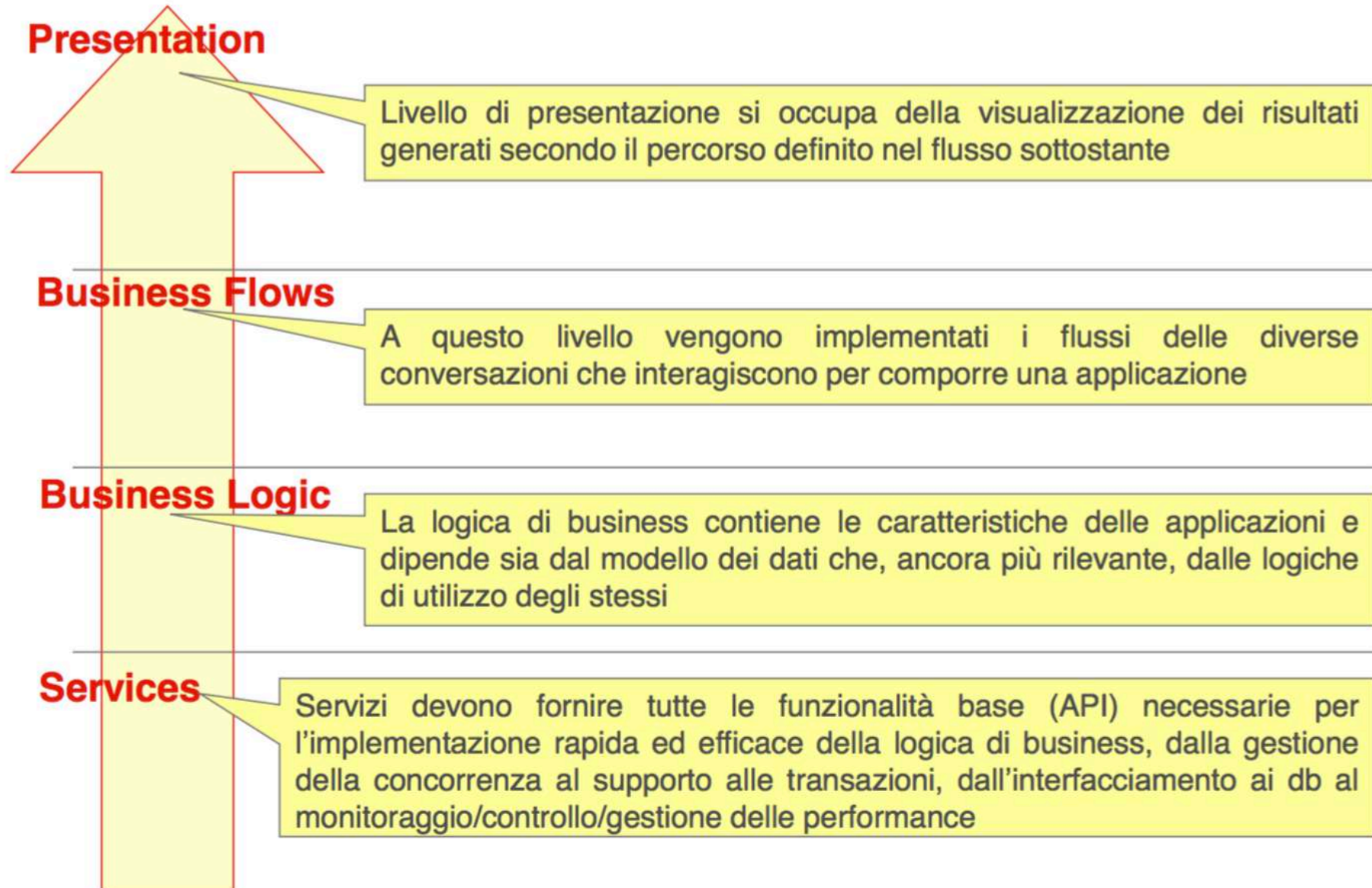
## Distribuzione Orizzontale



# Architettura multilivello

- Sviluppare applicazioni secondo una logica ad oggetti e/o a componenti significa **scomporre l'applicazione in blocchi**, servizi e funzioni
- È molto utile separare logicamente le funzioni necessarie in una struttura multilivello (*multi-tier*) al fine di fornire astrazioni via via più complesse e potenti a partire da funzionalità più elementari
- Nel tempo si è affermata una classificazione indipendente dalla implementazione tecnologica, basata su una struttura a 4 livelli principali
- *Questa struttura non fornisce dettagli implementativi, non specifica quali moduli debbano essere implementati client-side o server-side, ne nessuna altra specifica tecnica: è una architettura essenzialmente logico-funzionale*

# Schema di struttura multilivello



# Servizi

- I servizi realizzano le **funzioni di base** per sviluppo di applicazioni:
- Accesso e gestione risorse
  - Sistemi di naming
  - Sistemi di messaging e gestione code (queue)
  - Sistemi di monitoring e management
  - Sistemi legacy (termine generico che identifica applicazioni esterne per la gestione aziendale – OS390, AS400, UNIX systems, ...)
  - Stampanti, sistemi fax, sms, e-mail, dispositivi specifici, macchine automatiche, ...
- Gestione transazioni
- Gestione sicurezza
- Accesso e gestione delle sorgenti dati
  - Database locali
  - Sistemi informativi remoti



# Business logic

- È l'insieme di **tutte le funzioni** offerte dall'applicazione
- Si appoggia sui servizi per implementare i diversi algoritmi di risoluzione e provvedere alla generazione dei dati di output
- Esempi di moduli di business logic possono essere:
  - Gestione delle liste di utenti
  - Gestione cataloghi online
  - ...
- A questo livello:
  - non è significativo quali siano le sorgenti di dati (gestite dal livello dei servizi)
  - non è significativo come arrivino le richieste di esecuzione dei servizi e come vengano gestiti i risultati ai livelli superiori
- *Così modellata, la business logic presenta un elevato grado di riuso*

# Business flow

- Una conversazione è realizzata da un insieme di pagine collegate in un flusso di successive chiamate
- Il business flow raccoglie l'**insieme delle chiamate** necessarie per realizzare una conversazione
- Ogni chiamata deve:
  - a) Caricare i parametri in ingresso
  - b) Chiamare le funzioni di business logic necessarie per effettuare l'elaborazione
  - c) Generare l'output che dovrà essere visualizzato
- *L'astrazione fornita dal livello della business logic permette di definire l'esecuzione delle singole pagine in modo indipendente dalla struttura dei dati e degli algoritmi sottostanti*

# Presentazione

- Come abbiamo già detto, business flow è in grado di fornire i dati di output necessari
- Il livello di presentazione ha il compito di interpretare questi dati e generare l'**interfaccia grafica** per la visualizzazione dei contenuti (*rendering*)
- Questi due livelli sono concettualmente divisi poiché la generazione dei dati è logicamente separata dalla sua rappresentazione e formattazione
- Questo permette di avere facilmente diverse modalità/tipologie di presentazione degli stessi dati, per esempio una rappresentazione in italiano e una in inglese, o una in HTML e una in XML, o una adatta a desktop PC e una per smartphone

# Strutture semplificate

- Non tutte le tecnologie permettono di rispettare questa suddivisione:
  - In molti casi i sistemi vengono realizzati a 2 o 3 livelli
- Queste semplificazioni portano in certi casi a miglioramenti nelle performance e nella rapidità di sviluppo, ma comportano una netta riduzione della leggibilità e della manutenibilità
- *Come sempre il compromesso viene deciso in base al contesto: non esiste soluzione ideale per ogni situazione, set di requisiti applicativi, vincoli di deployment, costi, ...*
- Esempio di semplificazione a 2 livelli:



# Dynamic content

- The **Web Server application** serves only **static pages**
- A separate “helper” application that the Web Server can communicate with can build non-static, **just-in-time pages**
- Just-in-time pages don’t exist before the request comes in
- The request comes in, the helper app “writes” the HTML, and the Web Server gets it back to the client

## When instead of this:

```
<html>
<body>
The current time is
always 4:20 PM
on the server
</body>
</html>
```

## You want this:

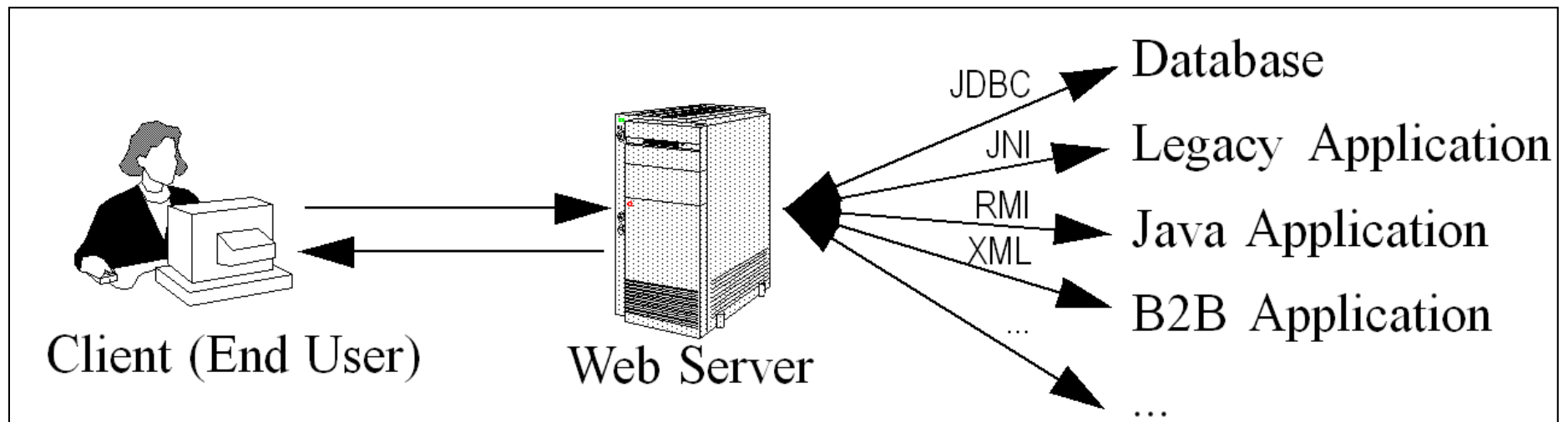
```
<html>
<body>
The current time is
[insertTimeOnServer]
on the server
</body>
</html>
```

## Saving data on the server

- The user **submits** data in a form
- The Web Server **get** the form
- To process that form data, either to save it to a file or database or even just to use it to generate the response page, you need **another app**
  - When the Web Server sees a request for a helper app, the Web Server assumes that parameters are meant for that app
- The Web Server hands over the parameters, and gives the app a way to generate a **response** to the client

# A Servlet's Job

- **Read** explicit data sent by client (form data)
- **Read** implicit data sent by client (request headers)
- Generate the results
- **Send** the explicit data back to client (HTML)
- **Send** the implicit data to client (status codes and response headers)



# Why Build Web Pages Dynamically?

- The Web page is based on **data submitted by the user**
  - Results page from search engines and order-confirmation pages at on-line stores
- The Web page is derived from **data that changes frequently**
  - A weather report or news headlines page
- The Web page uses information from databases or other server-side sources
- An e-commerce site could use a servlet to build a Web page that lists the current price and availability of each item that is for sale



# Example: firstservlet (L04 Servlet code)

- In Eclipse
- File → New → Dynamic Web Project
- Name project: FirstServlet

## Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.



Project name: FirstServlet

Project location

☒ Use default location

Location: /Users/michelerisi/Documents/workspacetsw/lez06

Browse...

Target runtime

Apache Tomcat v9.0



New Runtime...

Dynamic web module version

3.1



# Java resources → New → Servlet

**Create Servlet**  
Specify class file destination.

Project: FirstServlet

Source folder: /FirstServlet/src

Java package: it.unisa

Class name: FirstServlet

Superclass: javax.servlet.http.HttpServlet

☐ Use an existing Servlet class or JSP

Class name: FirstServlet

**Create Servlet**  
Enter servlet deployment descriptor specific information.

Name: FirstServlet

Description:

Initialization parameters:

Name
------

URL mappings:












/firstservlet
---------------

☐ Asynchronous Support

**URL Mappings**

Pattern: /firstservlet

# Servlet...

- ▼  FirstServlet
  - ▶  Deployment Descriptor: FirstServlet
  - ▶  JAX-WS Web Services
  - ▼  Java Resources
    - ▼  src
      - ▼  it.unisa
        - ▶  FirstServlet.java
  - ▶  Libraries
  - ▶  JavaScript Resources
  - ▶  build
  - ▶  WebContent
  - ▶  Servers

# FirstServlet.java (Hello World!)

```
package it.unisa;
```

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

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

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

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

```
@WebServlet("/firstservlet") //Java annotation
```

```
public class FirstServlet extends HttpServlet {
```

```
...
```

```
}
```

# FirstServlet.java (2)

**protected void doGet(**

**HttpServletRequest request,**

**HttpServletResponse response)**

**throws ServletException, IOException {**

response.setContentType("text/html");

PrintWriter out = response.getWriter();

java.util.Date today = **new java.util.Date();**

out.println("<html>" + "<body>" +

"<h1 align=center> First Servlet </h1>" +

"<br>" + today +

"</body>" + "</html>");

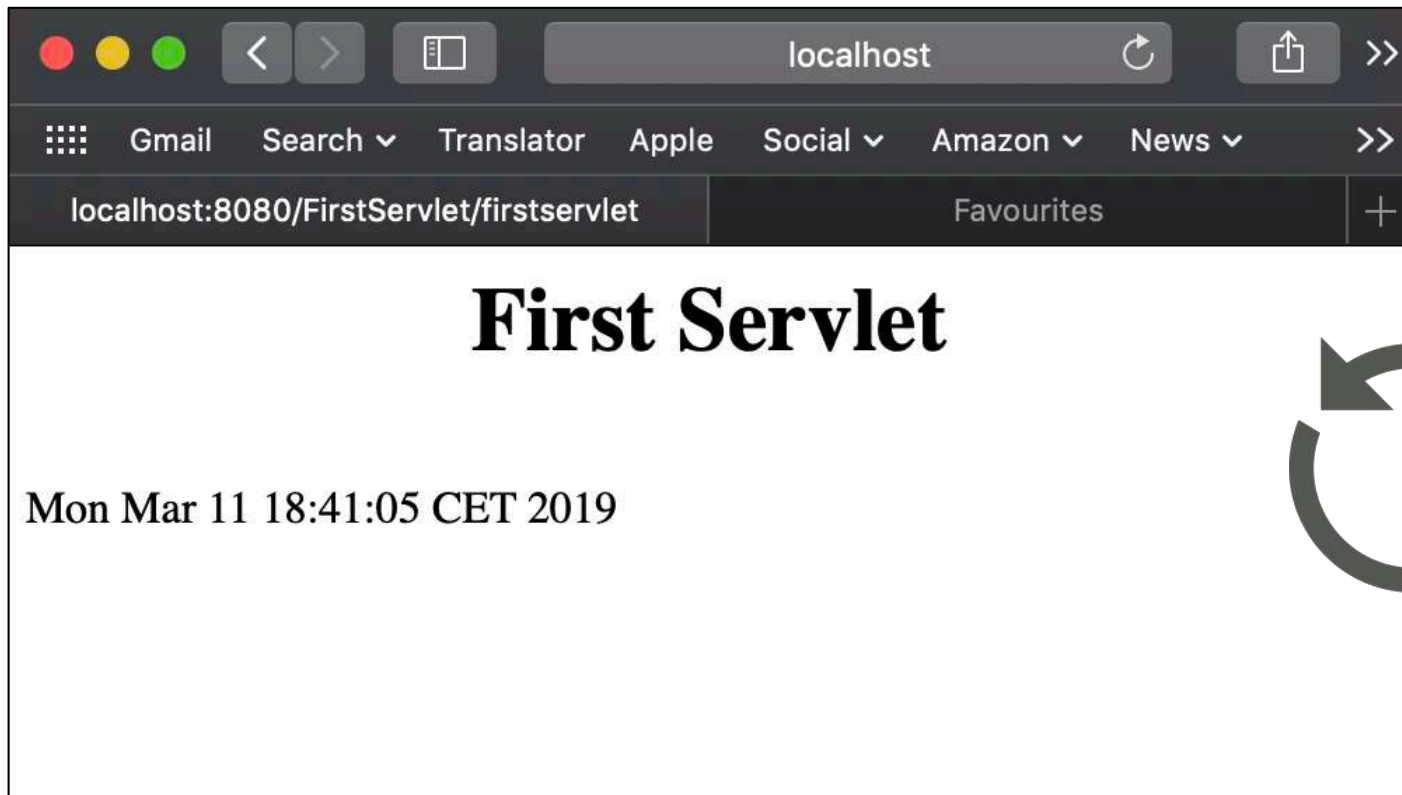
}

}

import  
java.io.PrintWriter;

# Run on a server

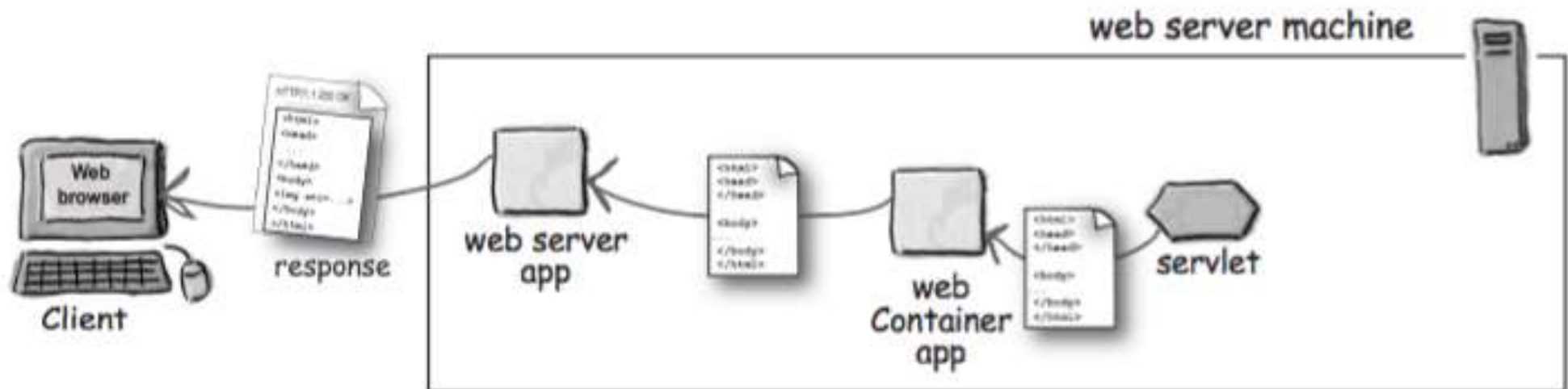
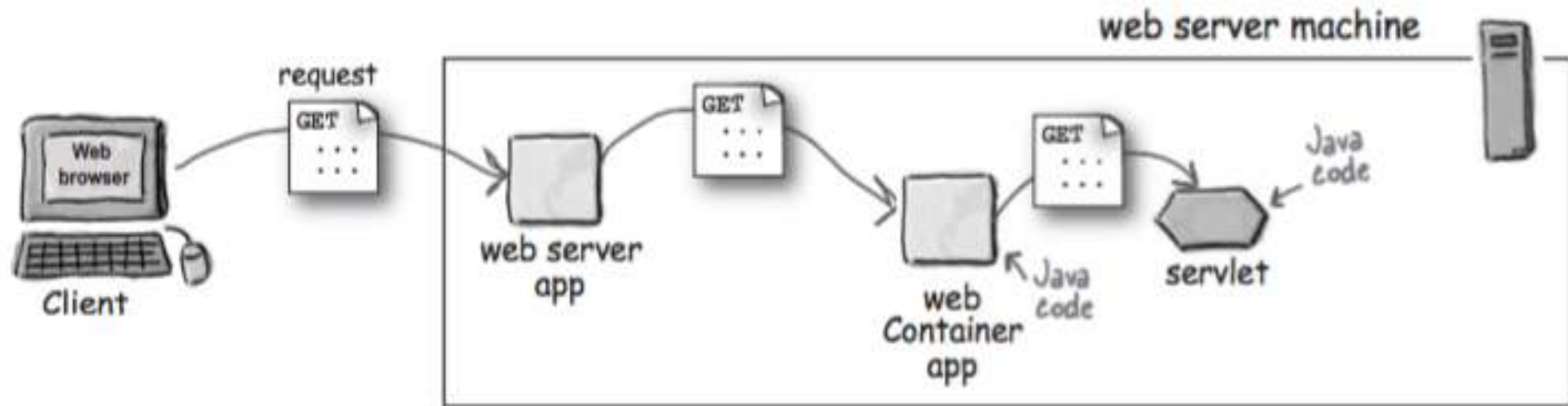
- **selezionare FirstServlet.java – tasto destro – run as → run on a server**
- **oppure fare il deploy dell'applicazione FirstServlet, e lanciare Tomcat**
- `http://localhost:8080/FirstServlet/firstservlet`



# web.xml (API 4.0)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" id="WebApp_ID" version="4.0">
  <display-name>FirstServlet</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

  <servlet>
    <servlet-name>firstservlet</servlet-name>
    <servlet-class>it.unisa.FirstServlet</servlet-class> <!-- Java class-->
  </servlet>
  <servlet-mapping>
    <servlet-name>firstservlet</servlet-name>
    <url-pattern>/firstservlet</url-pattern> <!-- URL -->
  </servlet-mapping>
</web-app>
```





# What is a real Container?

- **Servlets don't have a `main()` method. They're under the control of another Java application called a **Container****
- **Tomcat** is an example of a **Container**. When your Web Server application (like Apache) gets a request for a *Servlet*, the server hands the request not to the Servlet itself, but to the Container in which the Servlet is *deployed*
- the Container **gives** the Servlet the HTTP **request** and **response**
- the Container **that calls the servlet's methods** (like **`doPost()`** or **`doGet()`**)

# The container provides (1)

- **Communications Support.** The container provides an easy way for your servlets to talk to your Web Server. You don't have to build a ServerSocket, listen on a port, create streams, etc.
- The Container knows the protocol between the Web Server and itself, so that your servlet doesn't have to worry about an API between, say, the Apache Web Server and your own Web application code. All you have to worry about is your own business logic that goes in your Servlet (like accepting an order from your online store)
- **Lifecycle Management.** The Container controls the life and death of your Servlets
- It takes care of *loading* the classes, *instantiating* and *initializing* the Servlets, *invoking* the Servlet methods, and making servlet instances eligible for garbage collection. With the Container in control, you don't have to worry as much about resource management

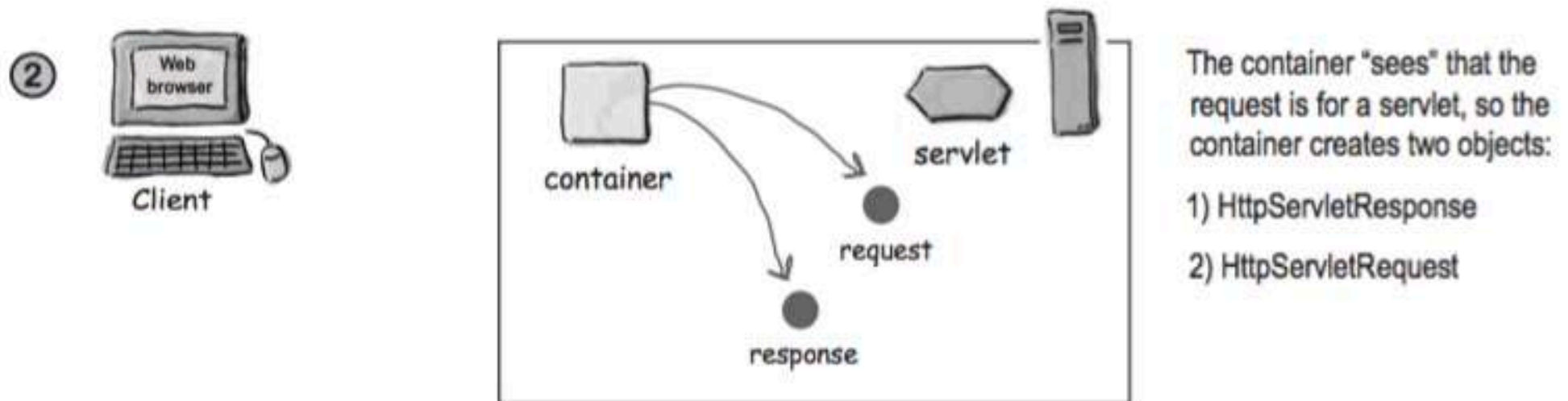
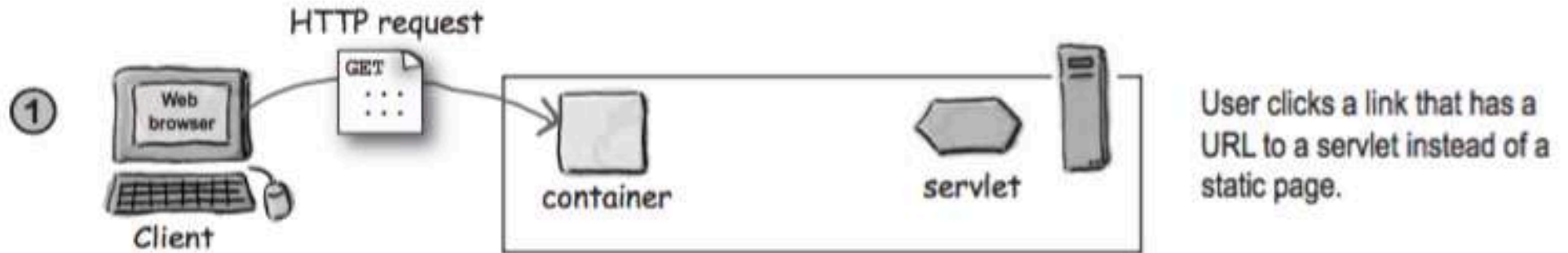
## The container provides (2)

- **Multithreading Support.** The Container automatically creates a new Java thread for every servlet request it receives
- When the Servlet's done running the HTTP service method for that client's request, the thread completes (i.e., dies). This doesn't mean you're off the hook for thread safety - you can still run into synchronization issues. But having the server create and manage threads for multiple requests still saves you a lot of work

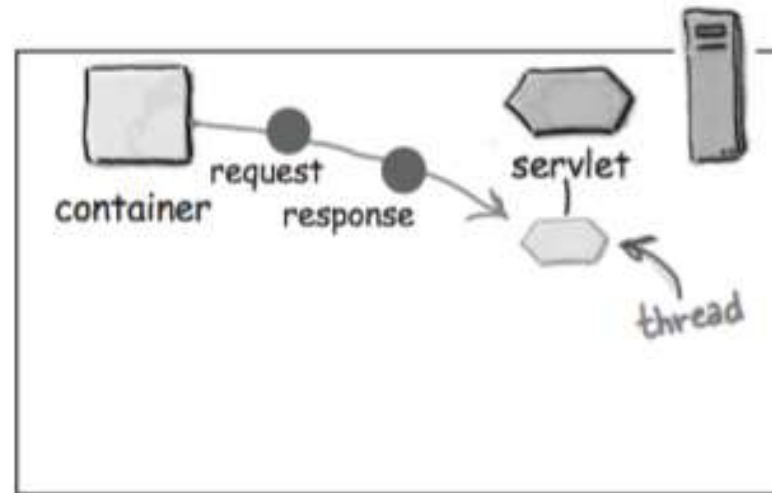
## The container provides (3)

- ❖ **Declarative Security.** With a Container, you get to use an XML deployment descriptor to configure (and modify) security without having to hard-code it into your servlet (or any other) class code
- You can manage and change your security without touching and recompiling your Java source files
- **JSP Support.** The container takes care of translating the JSP code into real Java

# The request-response model

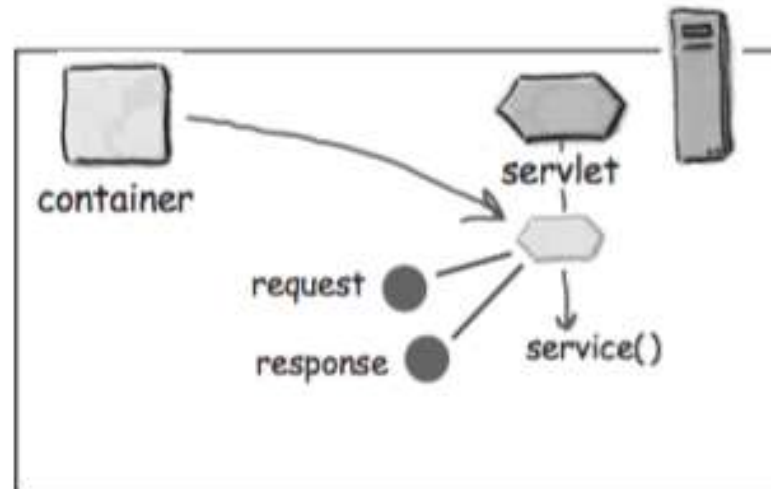


3



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.

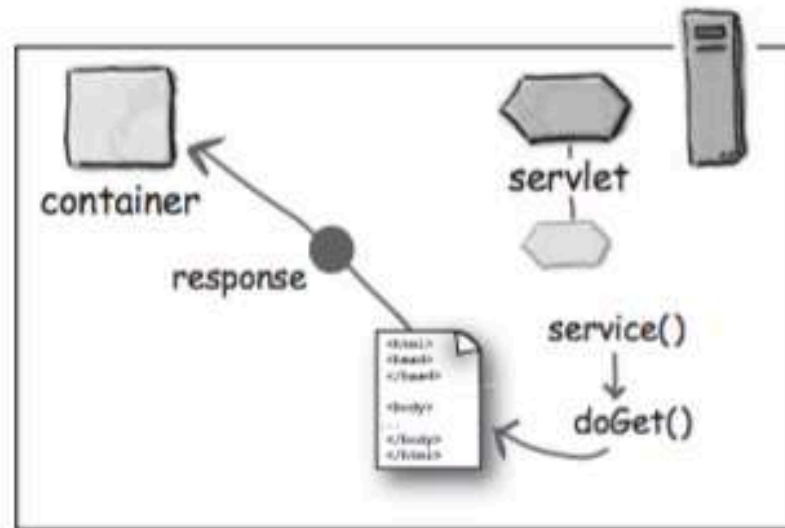
4



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

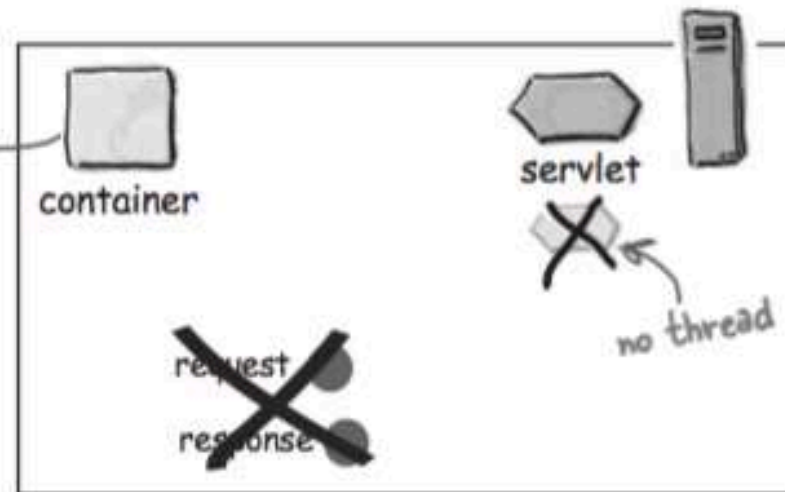


The `doGet()` method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!

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.

99.9999% of all servlets are `HttpServlet`s.

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

```
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.)

```
    }
}
```



Oh if only there were a way to put Java inside an HTML page instead of putting HTML inside a Java class.





## JSP: introducing Java into HTML

```
<html>
<body>
<h1>Skyler's Login Page</h1>
<br>
<%= new java.util.Date() %>
</body>
</html>
```

Whoa! This looks like a little Java, right in the middle of HTML!?

skylerlogin.jsp

- A JSP page looks just like an HTML page, except you can put Java and Java-related things inside the page
- So it really is like inserting a variable into your HTML

# Problems with mixing Java and HTML

- “The key problem with mixing Java and HTML, as in “Hello World!”, is that the **application logic and the way the information is presented** in the browser are mixed”
- The *business application designers* and the *Web-page designers* are different people with complementary and only partially overlapping skills
  - Business application designers are experts in complex algorithms and databases
  - Web designers focus on page composition and graphics
- The interface can be migrated (e.g., towards mobile) and the migration is easy if the interface and the application logic are separated
- The architecture of your JSP based applications should reflect this distinction

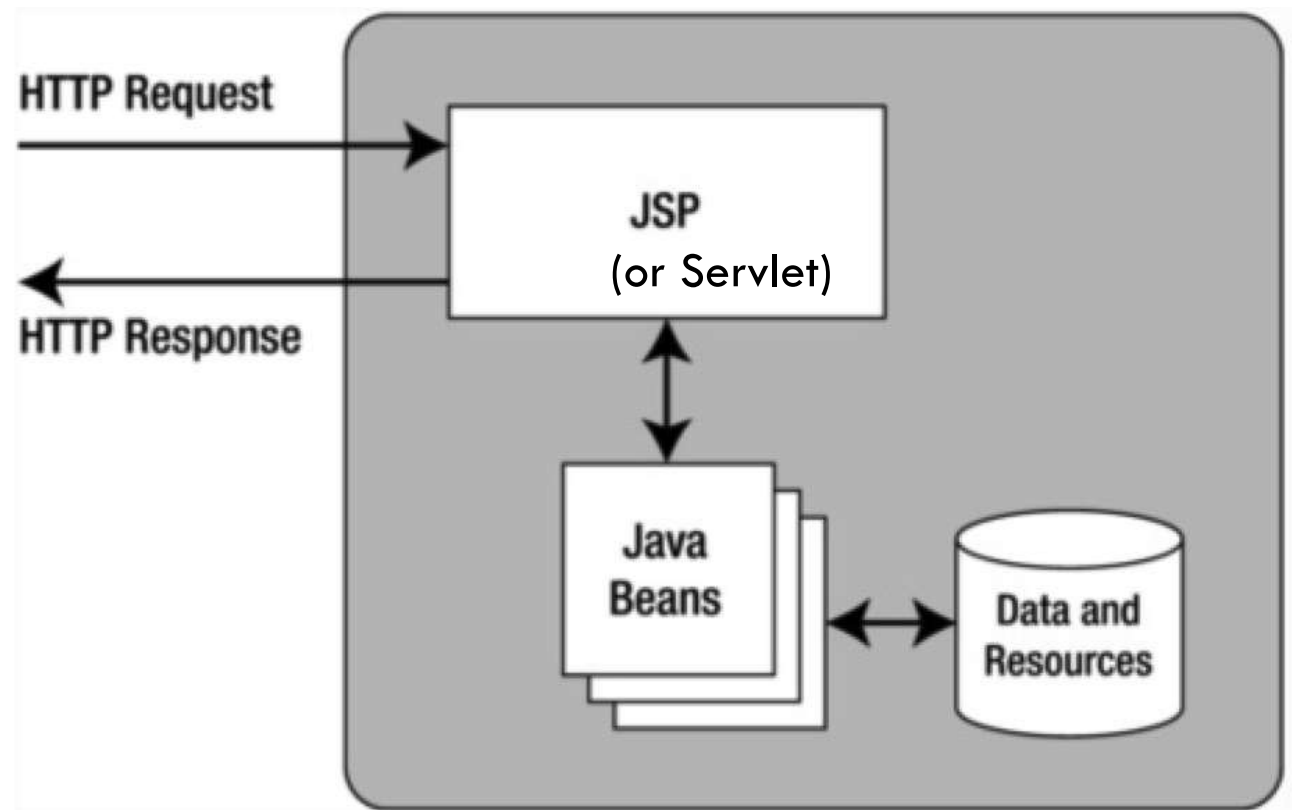
# Java Model

- 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** 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”

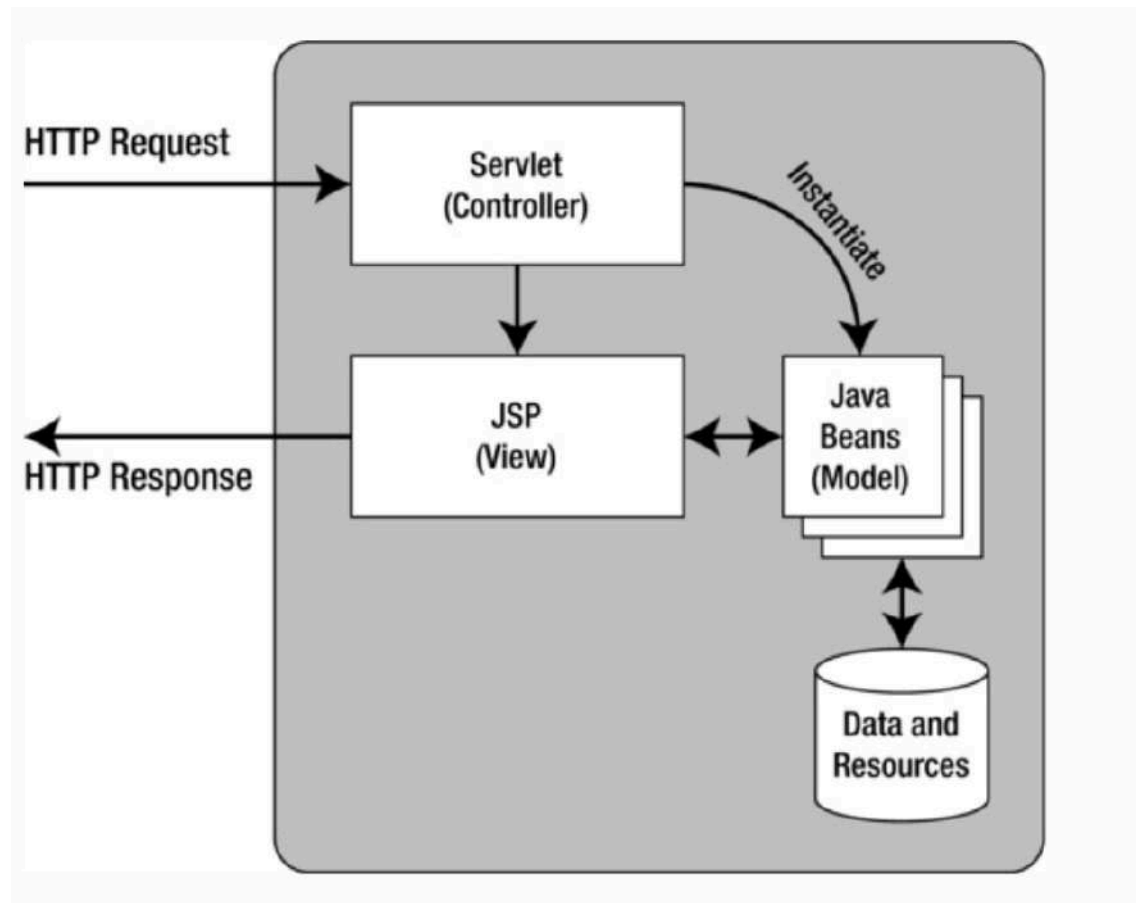
**“Move the bulk of the application logic from JSP/Servlet to Java classes (i.e., Java beans), which can then be used within JSP/Servlet. This is called the JSP/Servlet Model 1 architecture. JSP/Servlet has to handle HTTP request”**

- **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



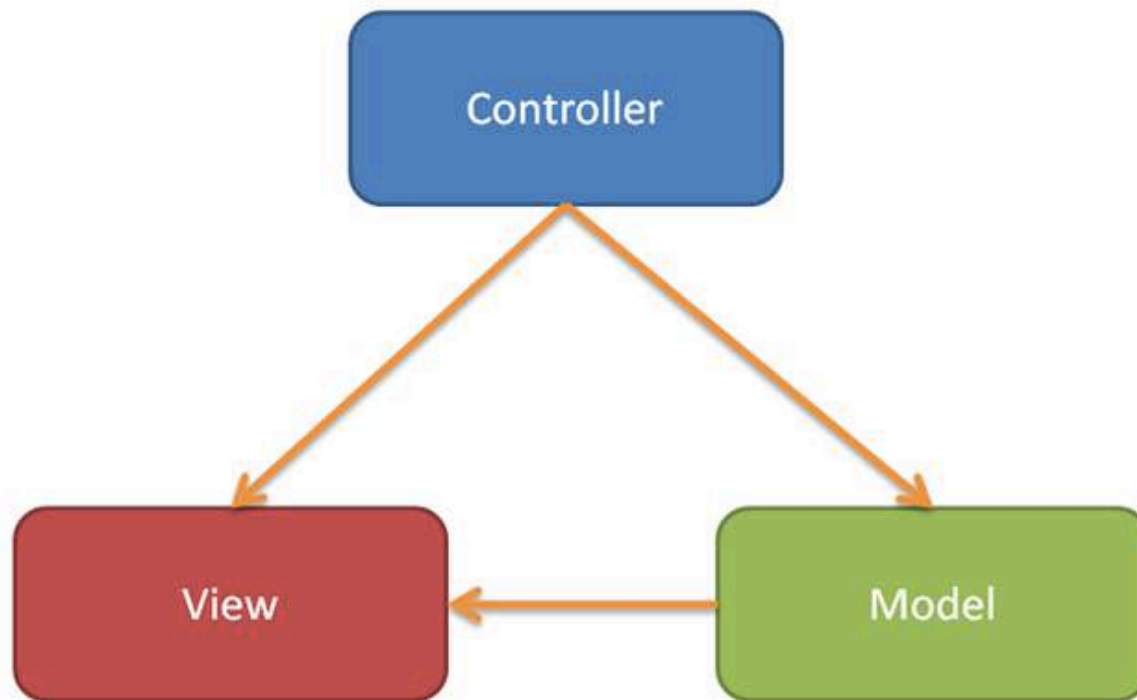
# “The Model 2 architecture”

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



# MVC

- Visto che Model 2 preme per separazione netta fra logica di business e di presentazione, usualmente associato con paradigma **Model-View-Controller (MVC)**



# 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 del rendering dei 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
  - gira l'input dell'utente verso il controller
- **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
- Il **Controller** (implementato da Servlet) 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 del 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

### CONTROLLER

Takes user input from the request and figures out what it means to the model.

Tells the model to update itself, and makes the new model state available for the view (the JSP).

### VIEW

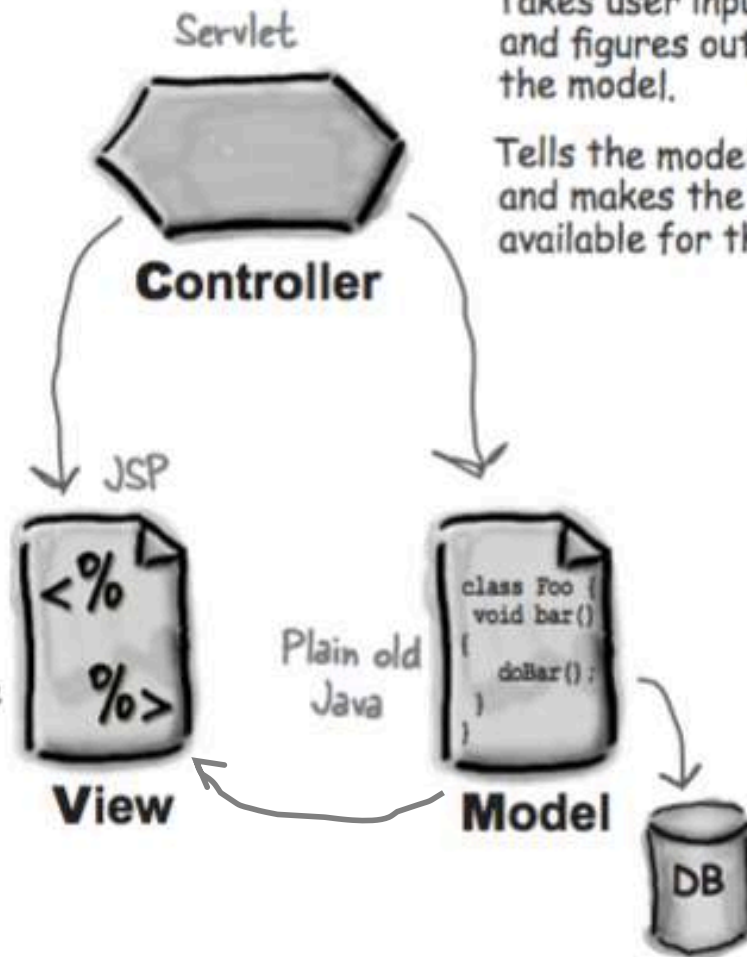
Responsible for the presentation. It gets the state of the model from the Controller (although not directly; the Controller puts the model data in a place where the View can find it). It's also the part that gets the user input that goes back to the Controller.

### MODEL

Holds the real business logic and the state. In other words, it knows the rules for getting and updating the state.

A Shopping Cart's contents (and the rules for what to do with it) would be part of the Model in MVC.

It's the only part of the system that talks to the database (although it probably uses *another* object for the actual DB communication, but we'll save that pattern for later...)



# Create and deploy an MVC Web app

1. Review the **user's views** (what the browser will display), and the high level architecture
2. Create the **development environment** (in Eclipse) that we will use for this project
3. Perform **iterative development and testing** on the various components of our web application

## Example: Beer Advisor

Users will be able to:

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

A screenshot of a web browser window titled "form.html". The browser's address bar is empty, and the search bar contains the text "Google". The browser's menu bar includes "Apple", ".Mac", "Amazon", "eBay", "Yahoo!", and "News". The main content area of the browser displays the "Beer Selection Page". The page has a title "Beer Selection Page" in a large, bold, black font. Below the title, the text "Select beer characteristics" is displayed. Underneath, there is a label "Color:" followed by a dropdown menu showing the word "light". At the bottom of the page, there is a "Submit" button.

← 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 is empty, and the search bar contains the text "Google". The browser's menu bar includes "Apple", ".Mac", "Amazon", "eBay", "Yahoo!", and "News". The main content area of the browser displays the "Beer Recommendations JSP". The page has a title "Beer Recommendations JSP" in a large, bold, black font. Below the title, the text "try: Jack's Pale Ale" is displayed, followed by "try: Gout Stout" on the next line.

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

# Iterative approach

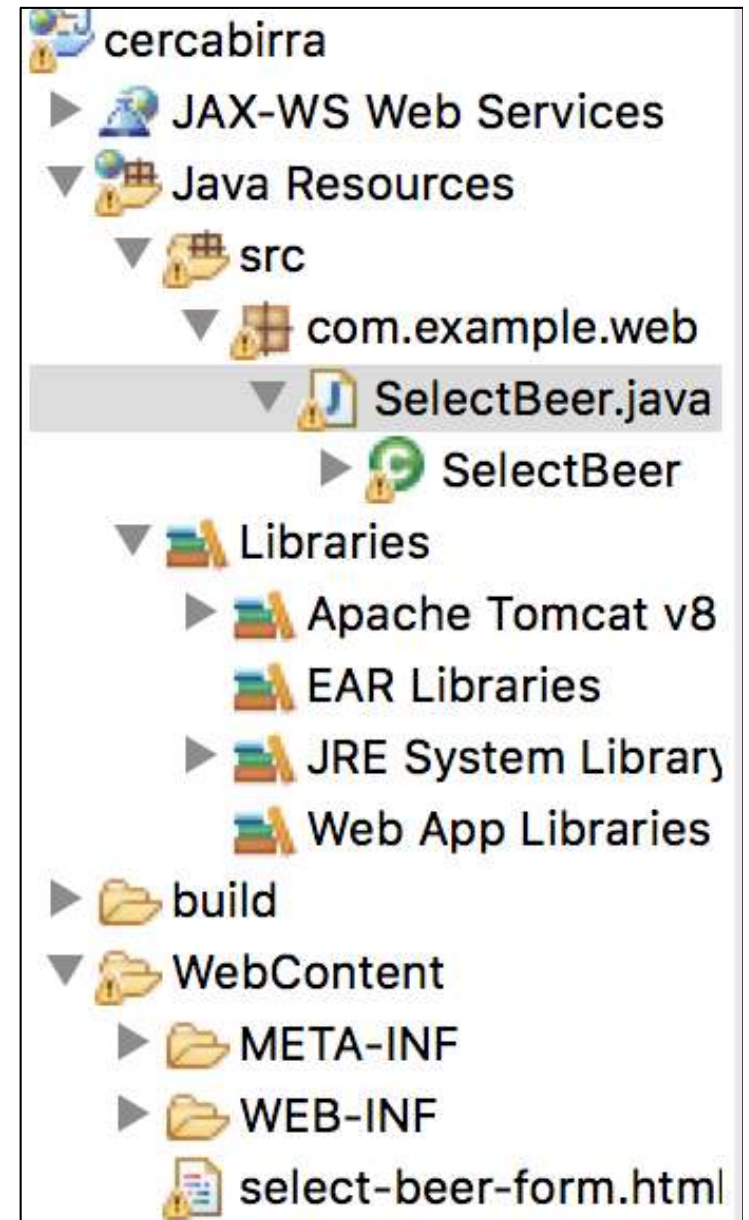
- Our plan is to build the Servlet in stages, testing the various communication links as we go

## MVC

- *In the end the Servlet will accept a parameter from the request, invoke a method on the model, save information in a place the JSP can find, and forward the request to the JSP*
- But **for this first version**, our goal is just to make sure that the HTML page can properly invoke the Servlet, and that the servlet is receiving the HTML parameter correctly

# First version (cercabirraV1.zip)

- **Create the Dynamic Web Project in your development environment**
- Create the *select-beer-form.html* file into WebContent
- Create the SelectBeer.java Servlet





# 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

- Right-click the page → Run on a server

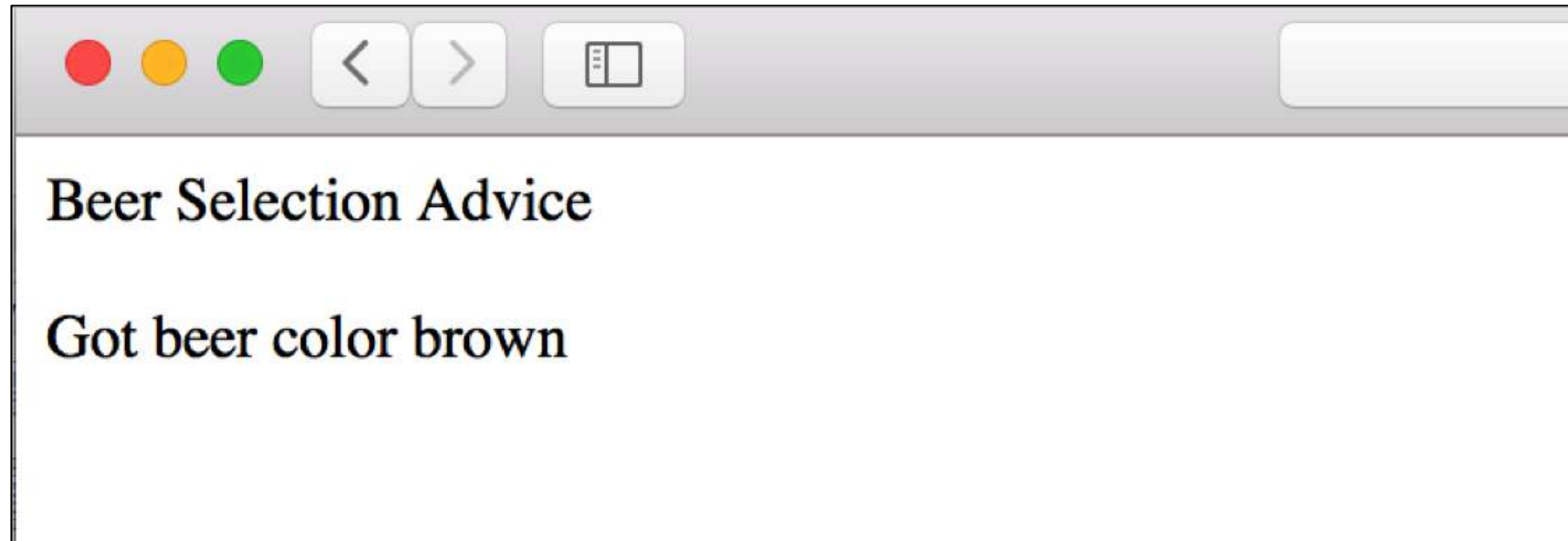
# Beer Selection Page

Select beer characteristics

Color:  

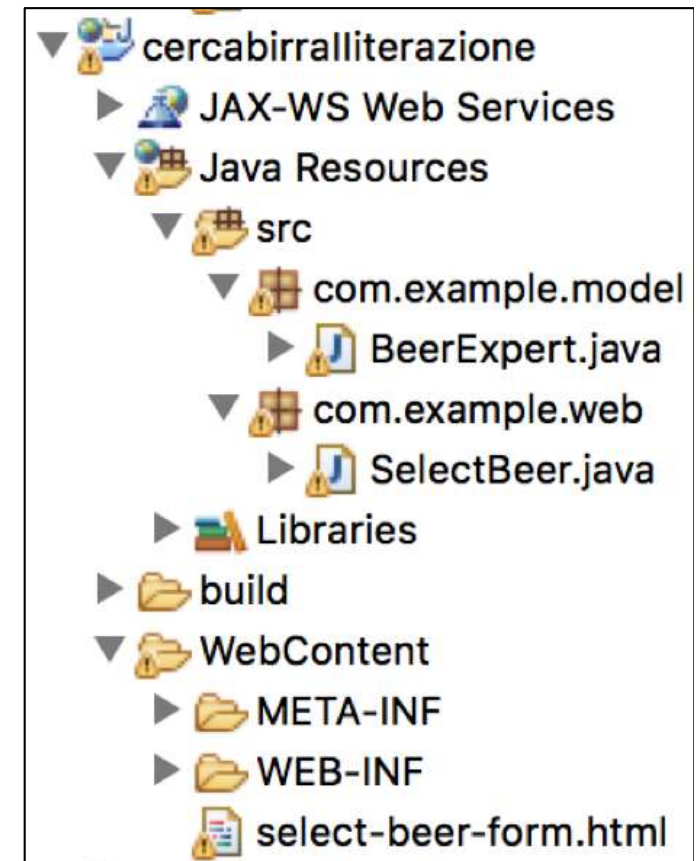
light  
amber  
✓ brown  
dark

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



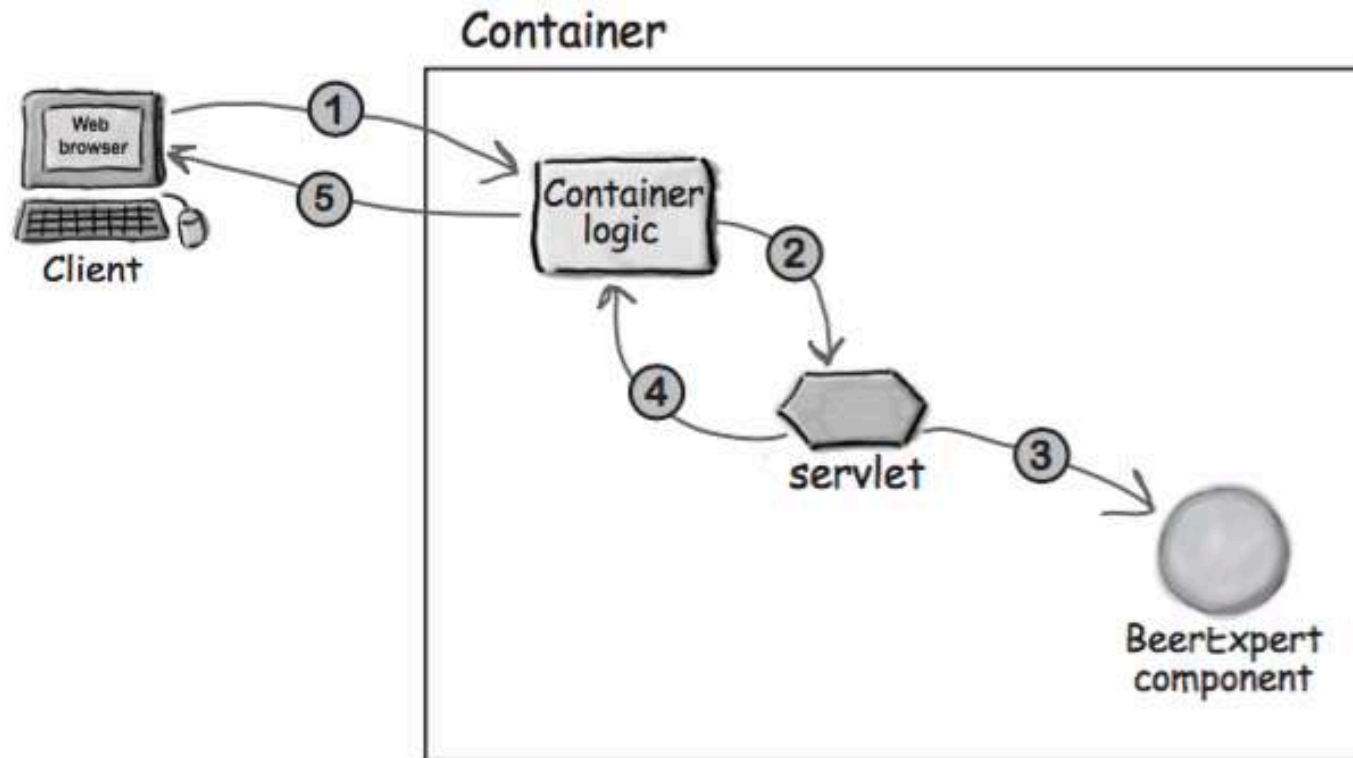
## Building and testing the model class (cercabirraV2.zip)

- 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**
  - Its package should be **com.example.model** (in **src**)
  - 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)



# Process 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.



# 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 two) to call the model

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

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

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

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

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

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

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

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

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

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



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

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

Instantiate the BeerExpert class and call getBrands().

```
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());
```

```
}
```

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.

```
}
```

# Test the form

## Beer Selection Page

Select beer characteristics

Color:  

Beer Selection Advice

try: Jail Pale Ale  
try: Gout Stout

## Beer Selection Page

Select beer characteristics

Color:  

Beer Selection Advice

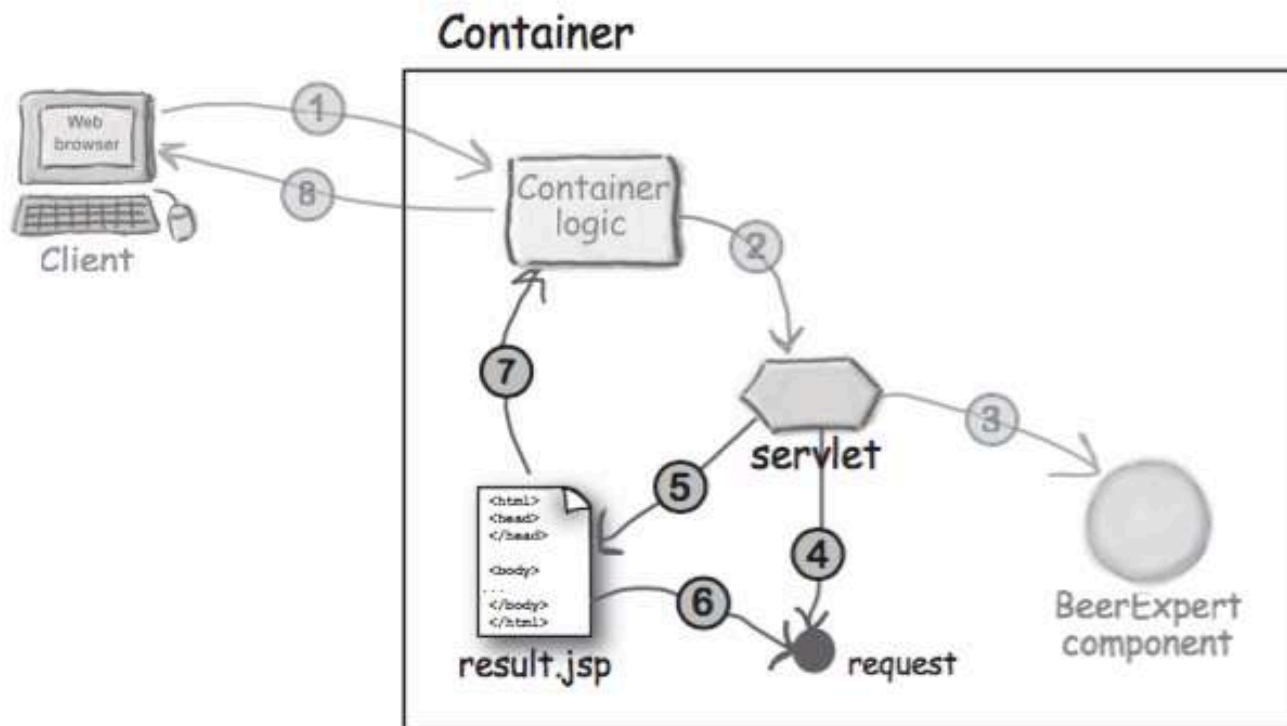
try: Jack Amber  
try: Red Moose

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

- In this step we're going to modify the servlet to “call” the JSP to produce the output (view)
- The Container provides a mechanism called “**request dispatching**” that allows one Container-managed component to call another, and that's what we'll use - the servlet will get the info from the model, save it in the request object, then *dispatch the request to the JSP*
- **The important changes we must make to the servlet:**
  1. Add the model component's answer to the request object, so that the JSP can access it (Step 4 in Process 2\*\*\*)
  2. Ask the Container to forward the request to “result.jsp” (Step 5 in Process 2\*\*\*)

# Process 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 three

```
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>");
```

Not needed!!!

## Code for servlet version three (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.

```
import javax.servlet.RequestDispatcher;
```



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

## Here's the JSP...

```
<%@ page import="java.util.*" %>
<html>
<body>
<h1 align="center">Beer Recommendations JSP</h1>
<p>

<%
    List styles = (List)request.getAttribute("styles");
    Iterator it = styles.iterator();
    while(it.hasNext()) {
        out.print("<br>try: " + it.next());
    }
%>

</body>
</html>
```

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

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

← 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...

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

## Test the form

### Beer Selection Page

Select beer characteristics

Color:  

### Beer Recommendations JSP

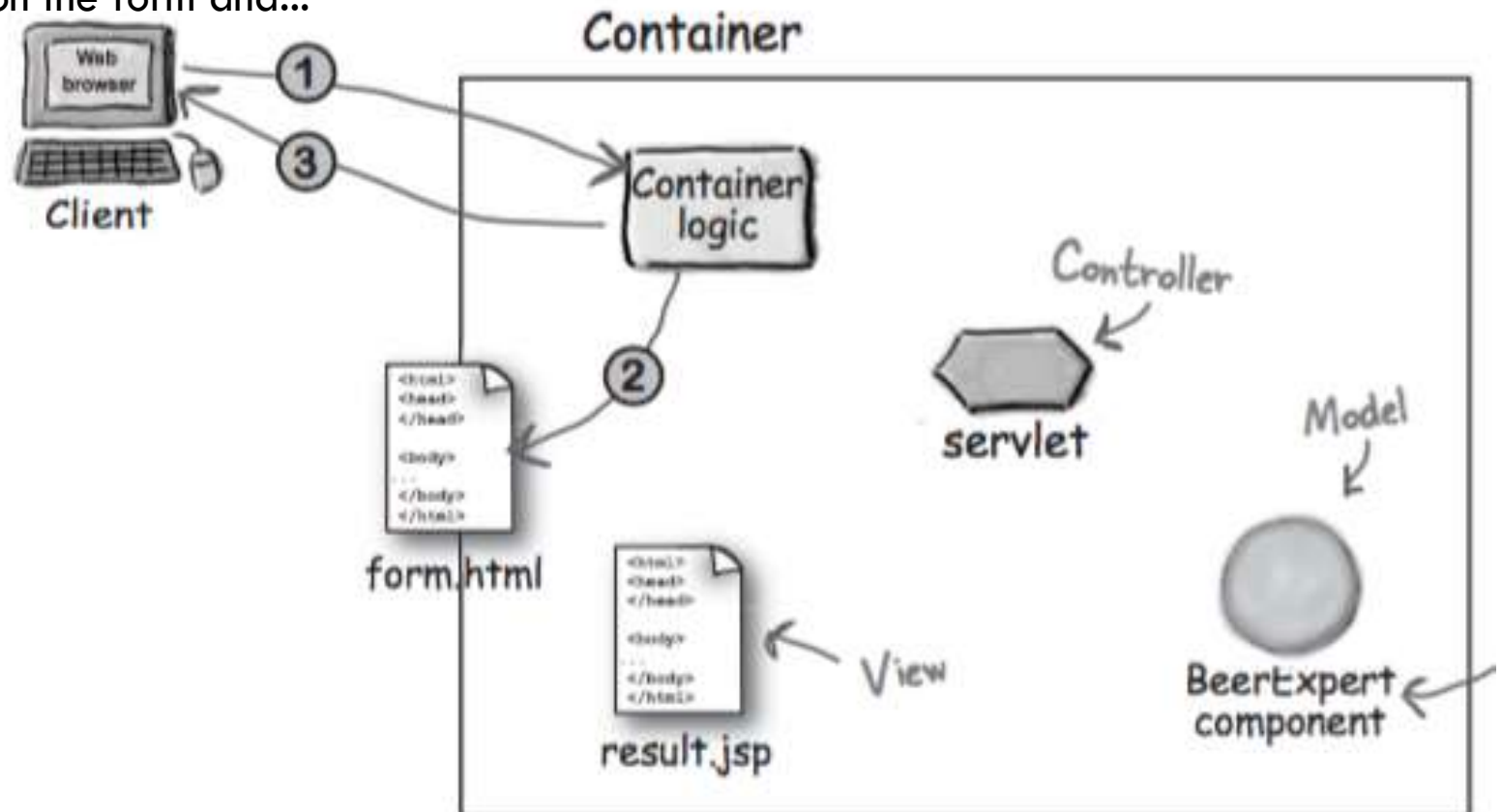
try: Jail Pale Ale

try: Gout Stout



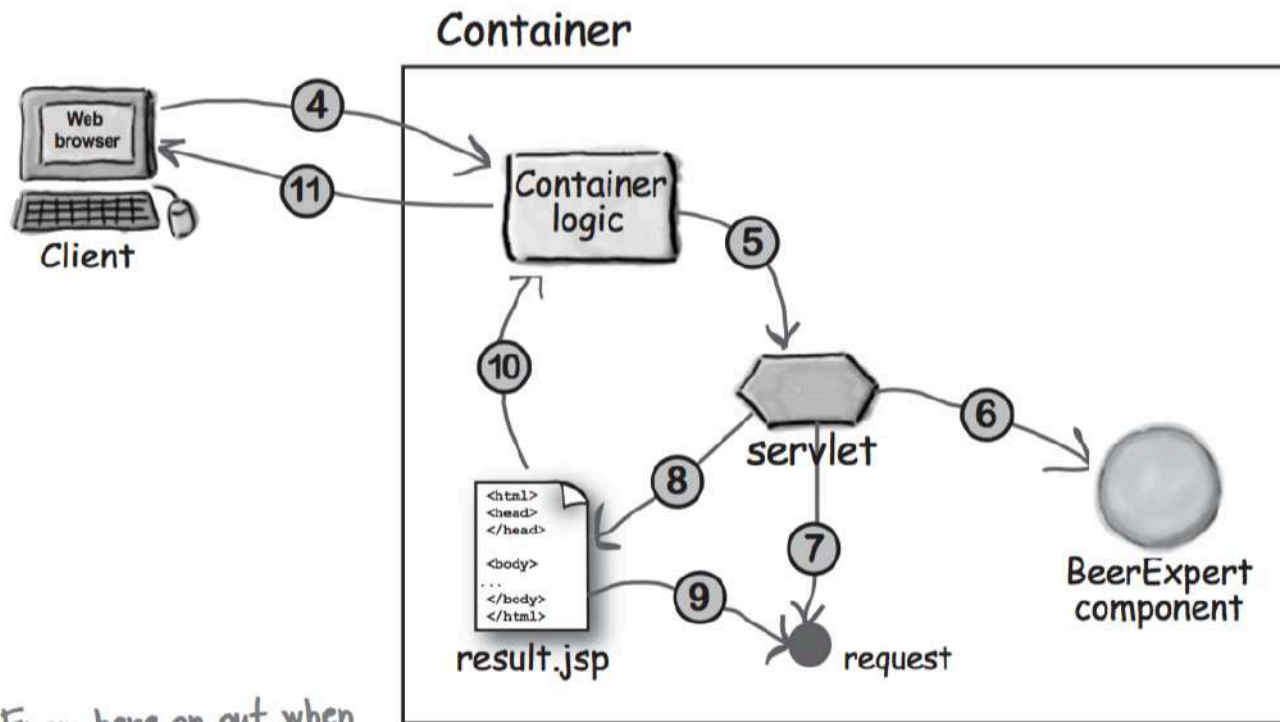
## Overall Process 2\*\*\*

1. The client makes a request for the `select-beer-form.html` page
2. The Container retrieves the `select-beer-form.html` page
3. The Container returns the page to the browser, where the user answers the questions on the form and...



## Overall Process 2\*\*\*

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



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