Java Enterprise Edition

Gabriele Tolomei

DAIS – Università Ca' Foscari Venezia

Programma del Corso

- 09/01 – Introduzione
- 10/01 – Java Servlets
- 16-17/01 – JavaServer Pages (JSP)
- 23-24/01 – Lab: Applicazione “AffableBean”
- 30-31/01 – Enterprise JavaBeans (EJB) + Lab

Modulo 2: Java Servlets

- Tecnologie web server-side
  - Applicazioni client/server su Web (HTTP)
- Java Servlets
  - Ruolo all'interno della piattaforma Java EE
- Esercitazione
  - Creazione di un progetto web dinamico su Eclipse

Il Web

- Il Web nasce per consentire la condivisione di risorse distribuite su hosts collegati tra loro tramite Internet
- Definisce 2 ruoli:
  - **Client** → esegue richieste di accesso alle risorse
  - **Server** → memorizza le risorse ed evade le richieste verso i clients

Struttura del Web

- Si basa sul noto stack di protocolli di rete **TCP/IP** e su di esso definisce 3 concetti fondamentali:
  - un sistema per l'identificazione univoca di risorse distribuite (**URL**)
  - un protocollo di richiesta/risposta tramite cui le risorse vengono trasferite tra il client ed il server (**HTTP**)
  - un linguaggio (**HTML**) per la rappresentazione di particolari risorse: pagine Web connesse tra loro da hyperlinks (grafo del Web)

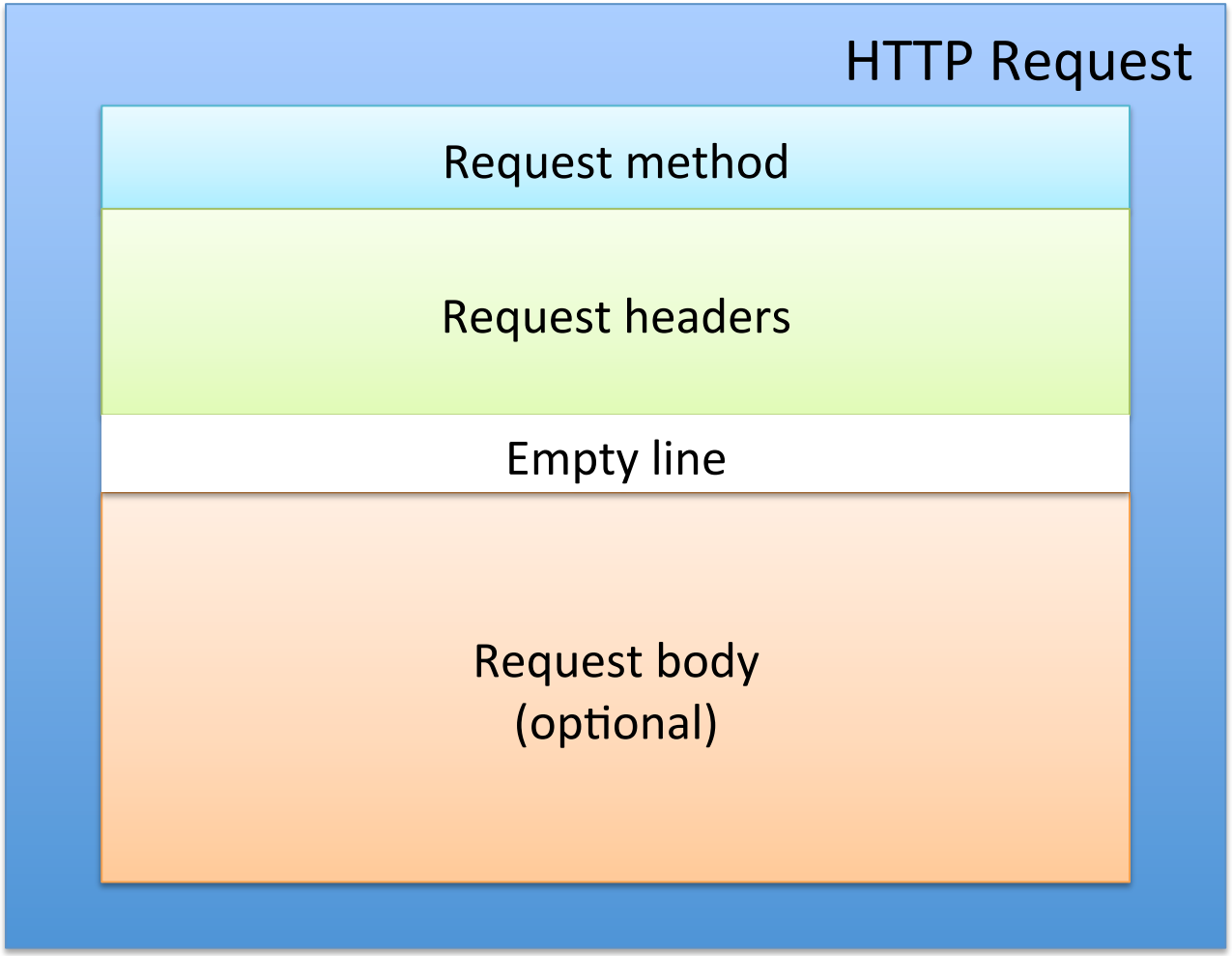
URL: Uniform Resource Locator

- Ogni URL è composto da:
  - protocollo (ad es.: HTTP, FTP, telnet, etc.)
  - user/password (opzionali per accedere al server)
  - indirizzo del server Web sottoforma di
    - IP (ad es.: 157.138.7.88)
    - Nome (ad es.: www.unive.it)
  - porta del servizio TCP a cui connettersi (opzionale)
    - i server Web accettano richieste di connessione sulla porta 80 (default)
  - path della risorsa richiesta (ad es.: /index.html)
  - eventuali parametri della richiesta (opzionali)
    - utili nelle applicazioni Web per lo scambio di dati tra il browser ed il server (ad es., form HTML)

HTTP: HyperText Transfer Protocol

- Stabilisce le “regole” con cui client e server comunicano
- Definisce un formato “standard” (IETF RFC 2616) dei messaggi di richiesta/risposta
  - Client: apre connessione con il server Web specificato nell’URL ed invia ad esso una richiesta HTTP
  - Server: evade la richiesta ed invia la risposta HTTP sulla connessione aperta dal Cliente e la chiude

HTTP: Richiesta Client



HTTP: Metodi di Richiesta

- HTTP definisce i seguenti metodi di richiesta:
  - GET, POST, PUT, DELETE, HEAD
- GET e POST sono le più utilizzate
- Eventuali parametri della richiesta possono essere specificati:
  - nella query string (URL) se si usa il metodo GET

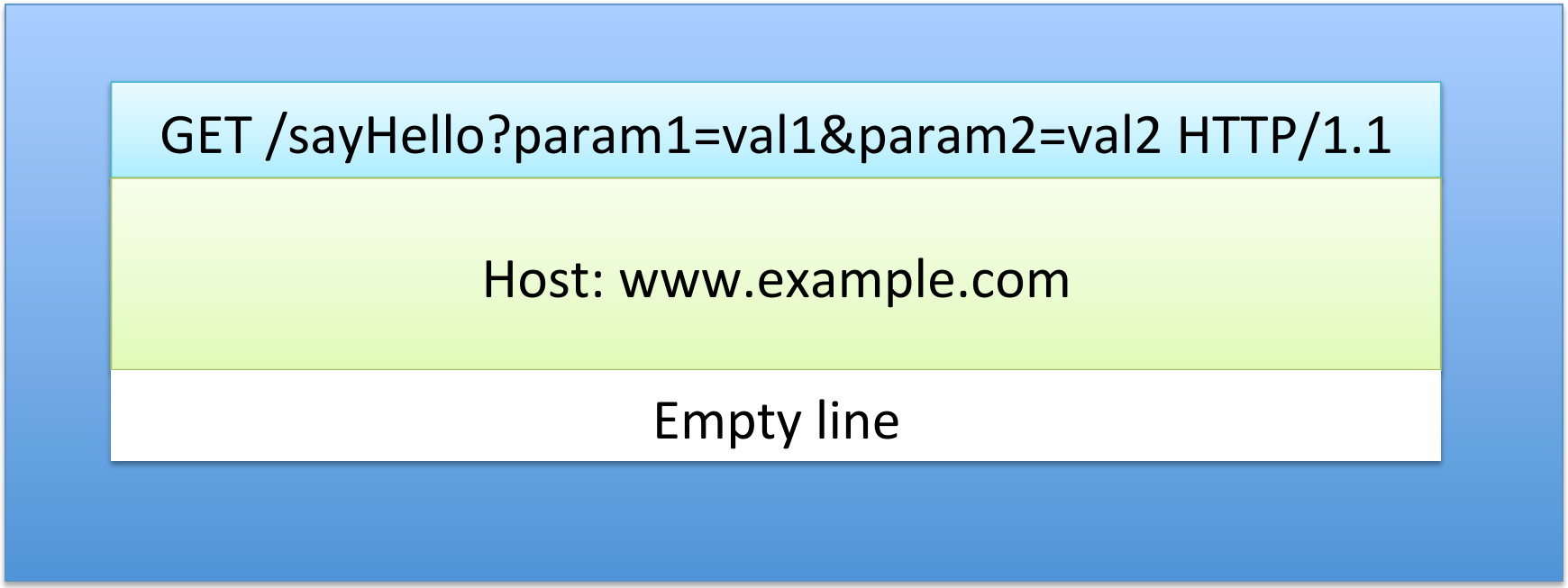
    http://example.com/sayHello?param1=val1&param2=val2
  - nel corpo della richiesta se si usa il metodo POST in combinazione con form HTML

HTTP GET (senza parametri)

http://www.example.com/index.html

HTTP GET (con parametri)

http://www.example.com/sayHello?param1=val1&param2=val2



HTTP POST

http://www.example.com/sayHello

```
POST /sayHello HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded

param1=val1&param2=val2
```

HTTP: URL-Encoding

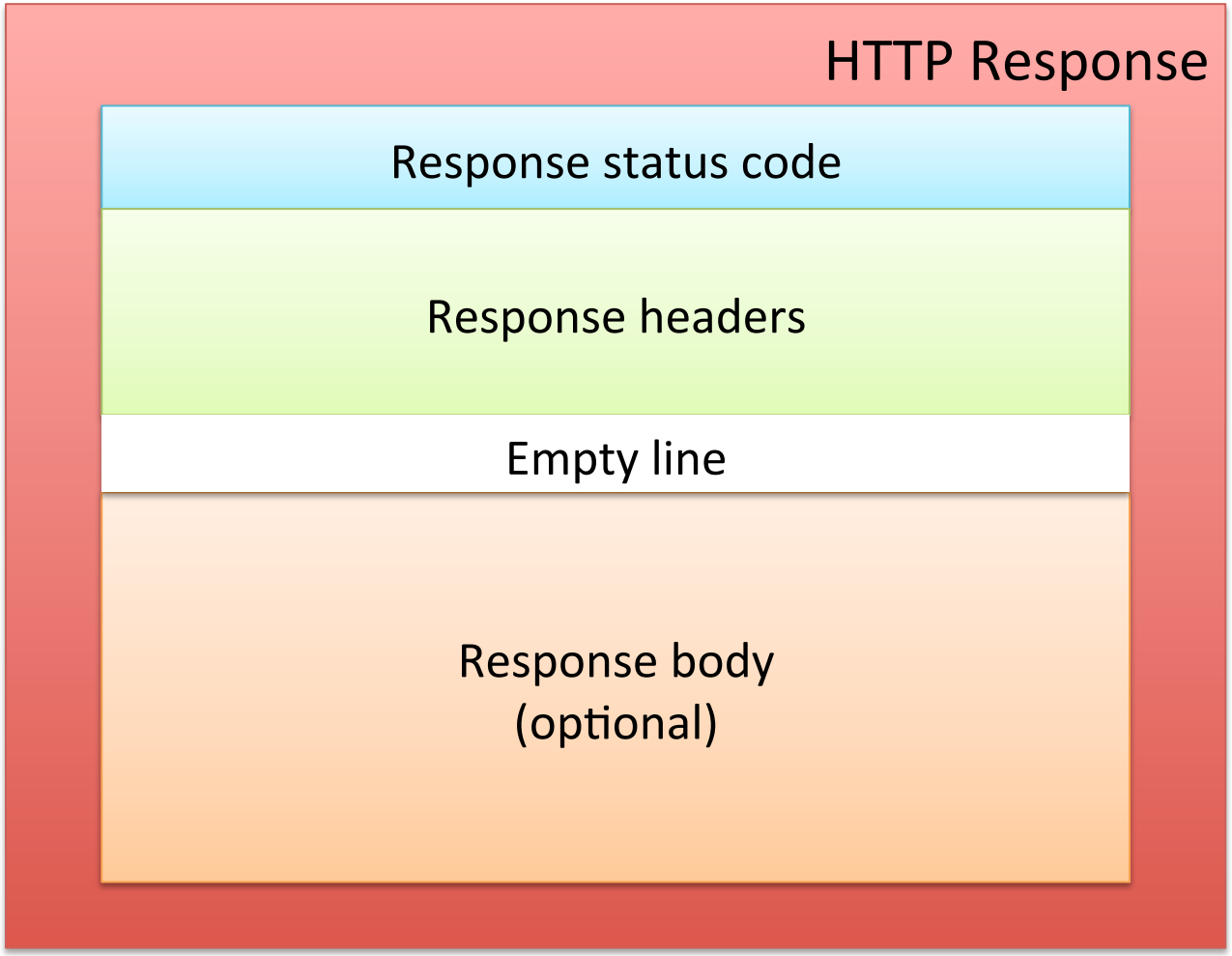
- I parametri inviati tramite form HTML nel corpo della richiesta HTTP POST devono essere opportunamente codificati
- Lista di coppie (chiave, valore)
- Esempio:

  name: Jonathan Doe, age: 23, func: a + b == 10%!

  sono codificati come

  name=Jonathan+Doe&age=23&func=a+%2B+b+%3D%3D+10%25%21

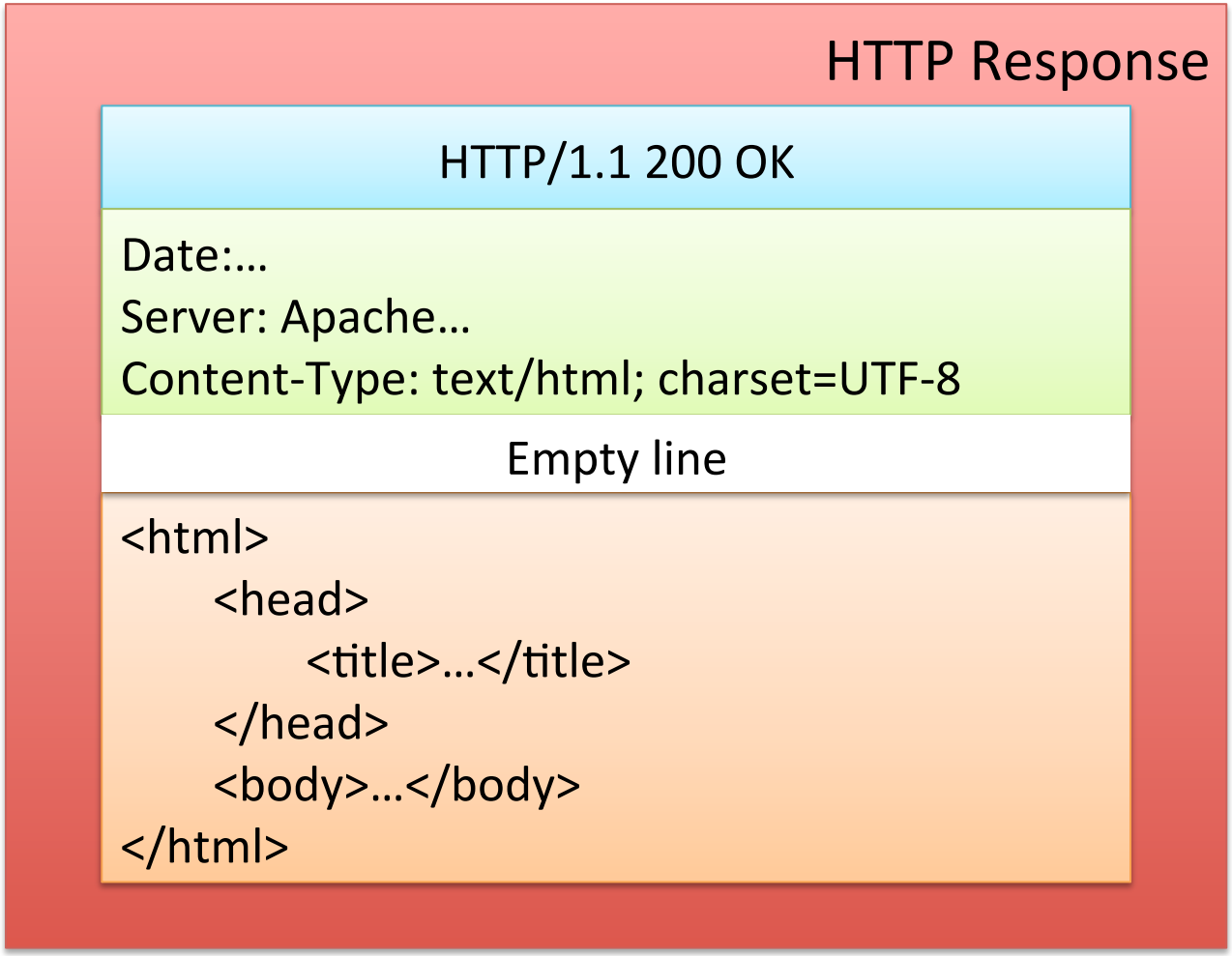
HTTP: Risposta Server



HTTP: Codici di Risposta

- 2xx = Successo
  - 200 OK → la richiesta ha avuto successo
    - GET: l'entità corrispondente alla richiesta viene inviata nella risposta
    - POST: l'entità corrispondente al risultato dell'azione richiesta viene inviata nella risposta
- 3xx = Redirezione
- 4xx = Errore Client
  - 401 Bad Request, 404 Not Found, etc.
- 5xx = Errore Server
  - 500 Internal Server Error, 503 Service Unavailable, etc.

HTTP 200 OK


HTTP Response
HTTP/1.1 200 OK
Date:…
Server: Apache…
Content-Type: text/html; charset=UTF-8
Empty line
<html>
<head>
<title>…</title>
</head>
<body>…</body>
</html>

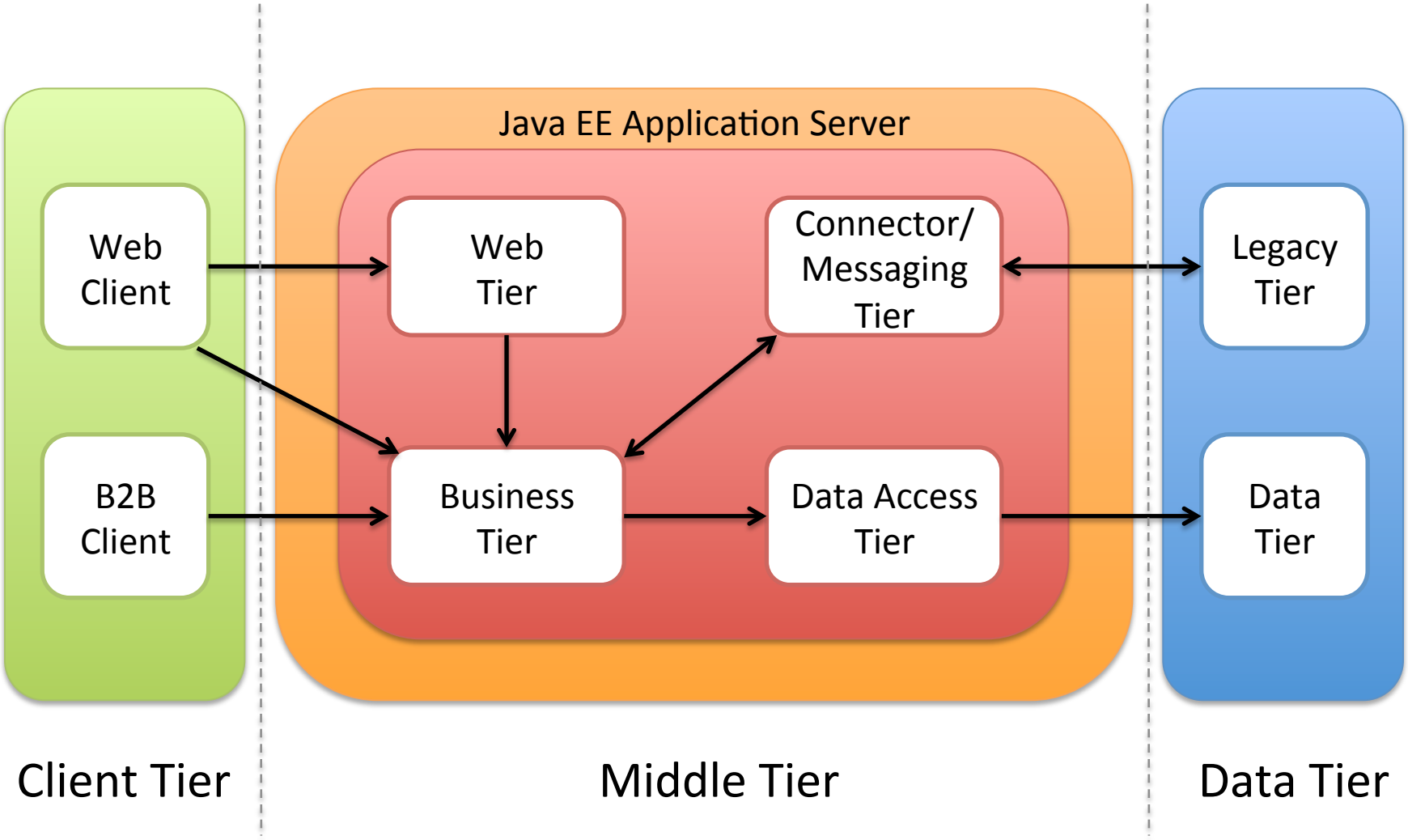
HTML: HyperText Markup Language

- Linguaggio di markup (tag) con cui vengono scritti i documenti Web
- Viene interpretato e renderizzato dai browsers
- Definisce alcuni tag standard
  - <html> → inizio del file HTML
  - <body> → inizio del contenuto da renderizzare
  - <a href="URL"> → hyperlink ad un'altra risorsa
  - <form action="URL"> → form invio dati al server

HTML: Form

- Sono utilizzati per inviare i dati dal client (utente tramite browser) al server Web
- All'interno contiene tag specifici che definiscono i parametri da inviare
  - <input name=nome type=tipo value=valore>
  - type può avere i seguenti valori:
    - text → campo di testo di 1 riga
    - password → campo password
    - hidden → campo nascosto
    - …
    - submit → pulsante di invio del form

Java EE: Architettura Multi-tier



Contenuti web dinamici

- Necessari per andare oltre le “solite” pagine web HTML statiche
- Una pagina web dinamica varia il proprio contenuto a seconda dei parametri forniti dal client al momento della richiesta
  - Il sorgente HTML della pagina viene generato dal server web in modo dinamico prima di essere restituito al browser e renderizzato
- Client- vs. Server-side scripting

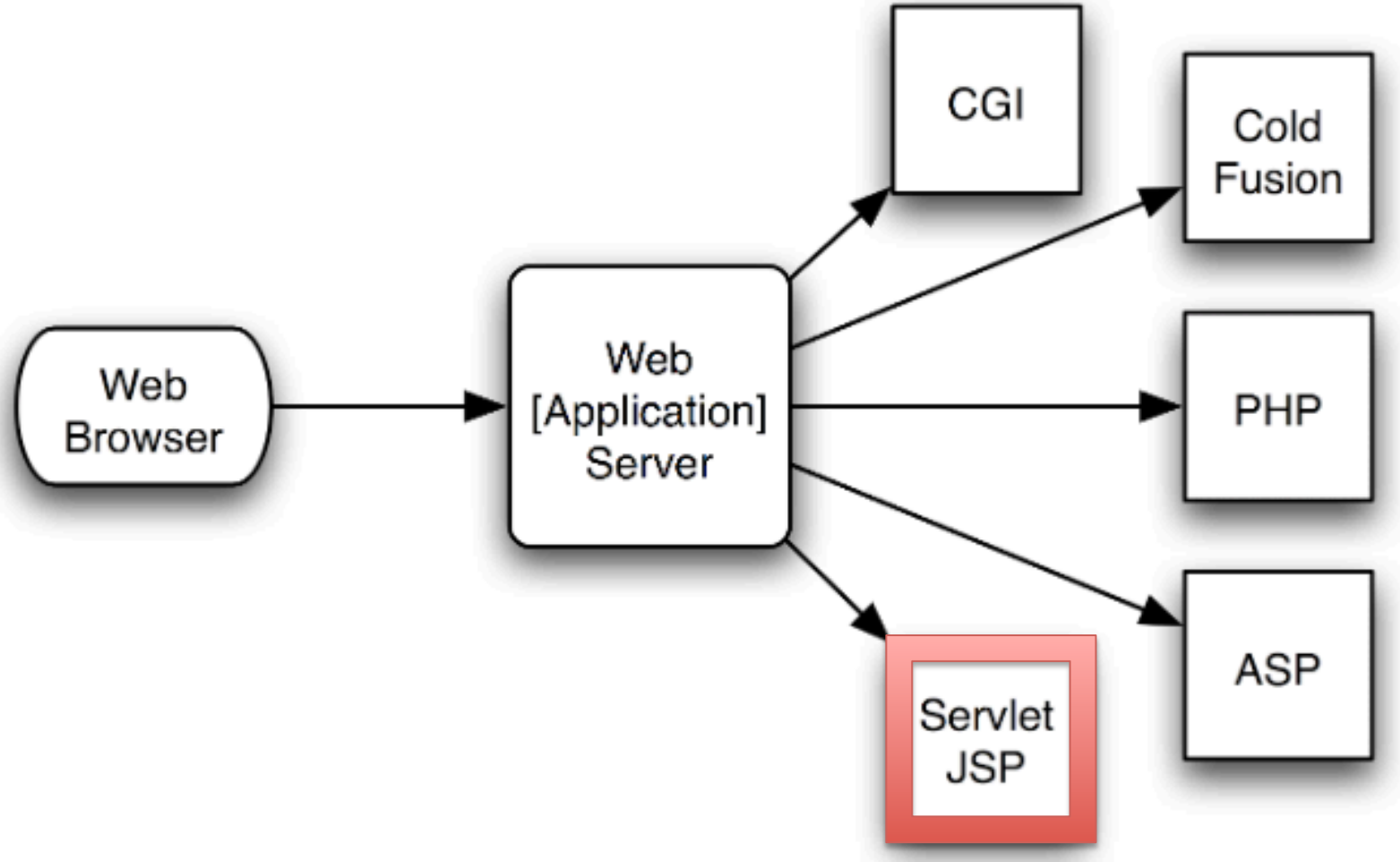
Client-side scripting

- La dinamicità riguarda la singola pagina web
- Cambiamenti in risposta ad azioni specifiche (mouse, tastiera, etc.)
- Il contenuto dinamico è generato da codice in esecuzione sul client
- Principale linguaggio di scripting client-side: JavaScript

Server-side scripting

- La dinamicità riguarda più di una singola pagina web
- Il contenuto dinamico è generato da codice in esecuzione lato (web) server
- Gestisce sessioni utente e controlla il flusso dell'applicazione
- HTML form, parametri nella URL di richiesta, tipo di browser usato, ecc.
- Principali linguaggi server-side: Perl, PHP, Java, ASP
- Estensioni server-side: CGI, JSP, ASP.NET

Tecnologie web server-side



CGI: Common Gateway Interface

- Interfaccia tra il server web e i programmi (**script CGI**) usati per generare i contenuti dinamici
- Ogni richiesta al server provoca l'esecuzione del corrispondente script CGI sottoforma di un nuovo **processo**
- Vantaggi:
  - semplice, diffuso, indipendente dal linguaggio (Perl e Python più utilizzati)
- Svantaggi:
  - altamente inefficiente

ColdFusion

- ColdFusion Markup Language (CFML)
- Sviluppato e controllato da Macromedia
- Potente e di facile utilizzo
- Ancora piuttosto diffuso

PHP

- PHP Hypertext Preprocessor
- Spesso utilizzato all'interno di server web Apache
- Ideale per lo sviluppo di piccole applicazioni web
  - facile e potente
- Molto popolare nella comunità open source
  - **LAMP** (**L**inux **A**pache **M**ySQL **P**HP)

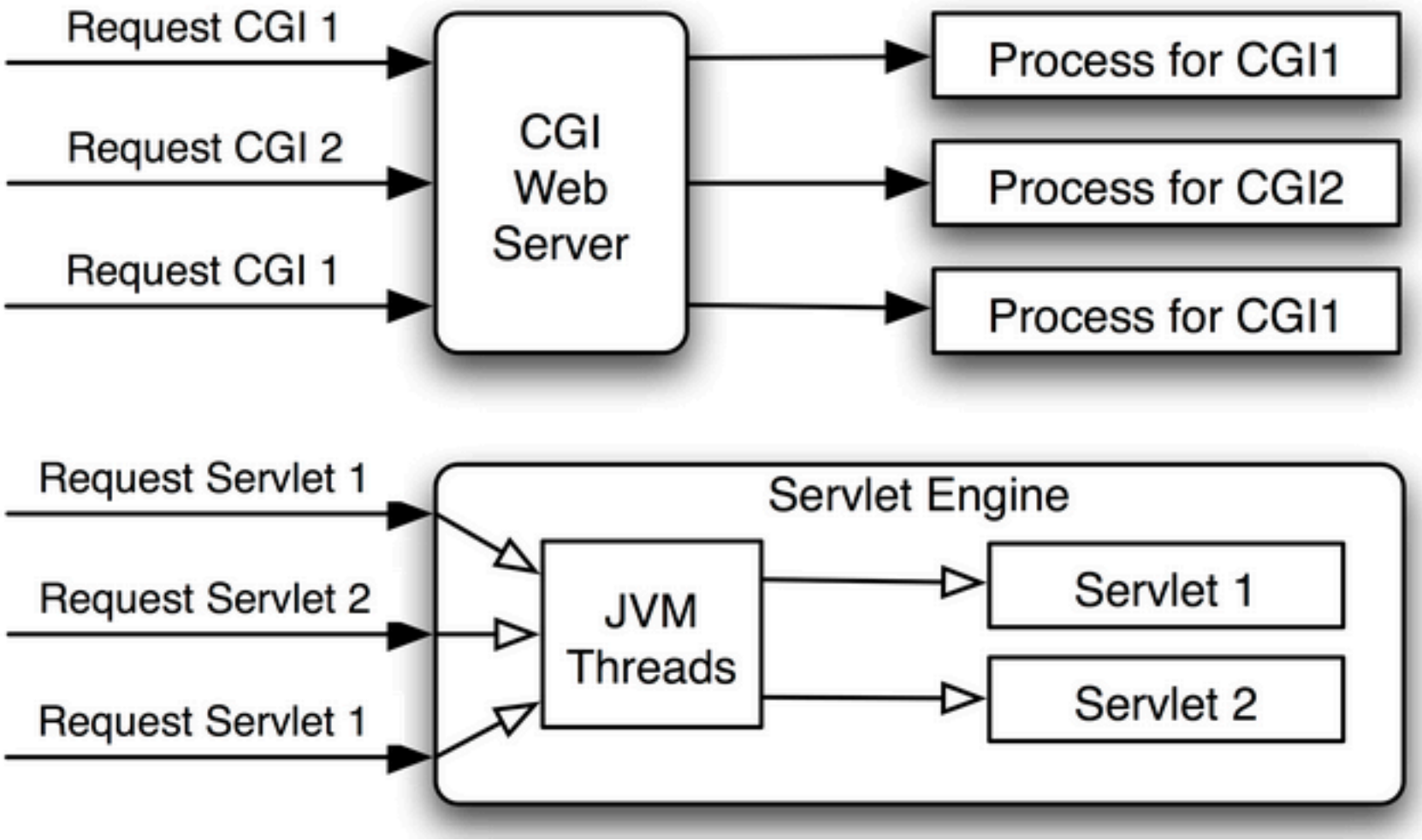
ASP: Active Server Pages

- Sviluppato e controllato da Microsoft
- Estensione per server web Microsoft IIS
- Linguaggi supportati per la creazione di contenuti dinamici:
  - VBScript
  - JavaScript
  - VisualBasic
  - C# (ASP.NET)

Java Servlet/Java Server Pages (JSP)

- Sviluppato da Java Community Process
- Parte dello standard Java EE
- OO, platform-independent, efficiente, scalabile,…
- Consente la separazione tra il livello di presentazione (interfaccia) e la logica applicativa

CGI vs. Servlet/JSP



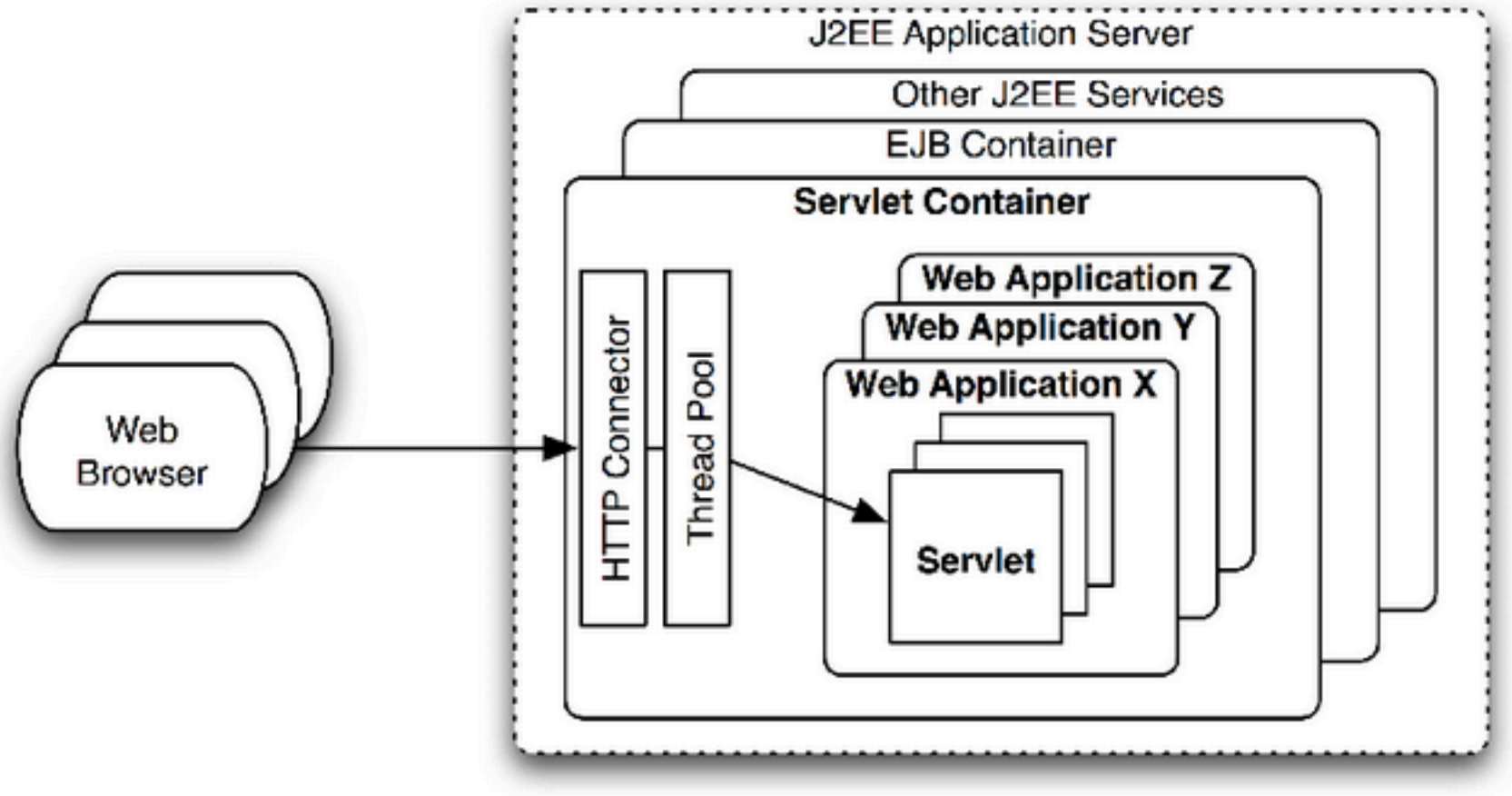
CGI vs. Servlet/JSP

- CGI
  - 1 richiesta client → 1 processo server
- Servlet/JSP
  - 1 richiesta client → 1 thread all'interno dello stesso processo server JVM
- Ottimizzazione delle risorse
  - Processo vs. Thread

Java Servlet: Vantaggi

- Condividono tutti i vantaggi del sw scritto in Java:
  - Portabilità, OO, riuso, supporto di librerie già esistenti, efficienza, sicurezza, etc.
- Si basano su una ben consolidata API specifica per il protocollo HTTP:
  - processing delle richieste
  - generazione delle risposte
  - gestione delle sessioni e dei cookies

Servlet e Applicazioni Web



Servlet e Applicazioni Web (2)

- Sia le Servlets che le JSPs sono eseguite all'interno di archivi Web (WAR)
- I WAR a loro volta sono in esecuizione su un Servlet Container (parte delle specifiche Java EE server)
- Nel caso di JBoss, il Servlet Container coincide con il servizio Apache Tomcat

Servlet e Applicazioni Web (3)

- Le applicazioni web sono isolate l'una dall'altra all'interno dello stesso Servlet Container
- Il Servlet Container fornisce tutti quei servizi di "basso livello" necessari per il ciclo di vita di Servlets e JSPs:
  - gestione delle connessioni HTTP, sessioni, threading, sicurezza, gestione delle risorse, monitoring, deployment, etc.

Ma che cos'è una Java Servlet?

- È una normale classe Java che consente l'interazione richiesta/risposta con un'applicazione secondo il modello client/server
- Le Servlets sono progettate per gestire qualunque tipo di protocollo richiesta/risposta
- Tipicamente vengono usate per l'implementazione di applicazioni che interagiscono secondo il protocollo HTTP (richiesta/risposta via Web)

Java Servlet: Packages

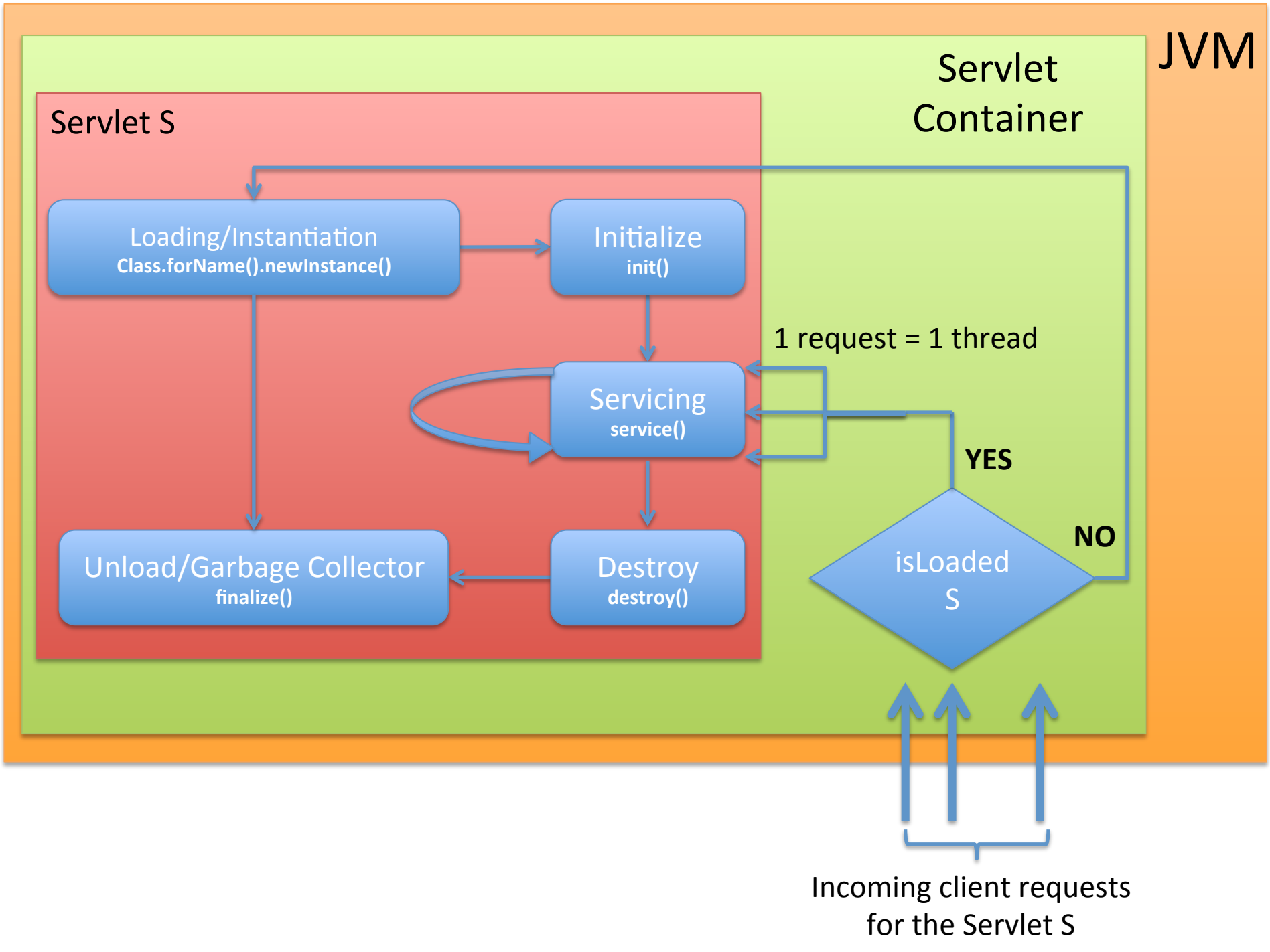
- È definita all'interno del package standard javax.servlet
- Ogni Servlet deve implementare l'interfaccia javax.servlet.Servlet
  - specifica i metodi relativi al ciclo di vita
- Le Servlets che gestiscono protocolli di richiesta/risposta generici devono estendere javax.servlet.GenericServlet
- Le Servlet specifiche per la gestione del protocollo HTTP devono estendere javax.servlet.http.HttpServlet

Java Servlet: Esempio

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

public class HelloWorldServlet extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    PrintWriter out = resp.getWriter();
    out.println("<html>");
    out.println("\t<head><title>Hello World</title></head>");
    out.println("\t<body>");
    out.println("\t\t<p>Hello " + req.getRemoteAddr() + "!</p>");
    out.println("\t\t<p>Server time is " + new Date() + "</p>");
    out.println("\t</body>");
    out.println("</html>");
    out.close();
  }
}
```

Java Servlet: Ciclo di Vita

- A fronte di una richiesta da parte del client il container:
  1. Verifica che la Servlet sia già stata caricata
     a. Se non lo è, provvede a caricare la classe corrispondente e ne genera un'istanza
     b. Inizializza l'istanza appena creata invocando su di essa il metodo init
  2. Invoca il metodo service corrispondente all'istanza della Servlet passando come argomenti gli oggetti che rappresentano la richiesta e la risposta
- La rimozione della Servlet dal container si ottiene tramite una chiamata al metodo destroy


JVM
Servlet Container
Servlet S
Loading/Instantiation
Class.forName().newInstance()
Initialize
init()
1 request = 1 thread
Servicing
service()
YES
NO
isLoaded S
Unload/Garbage Collector
finalize()
Destroy
destroy()
Incoming client requests for the Servlet S

Java Servlet: Ciclo di Vita (2)

- I metodi init e destroy vengono chiamati solo una volta, rispettivamente alla creazione e rimozione della Servlet
- Il metodo service viene chiamato una volta per ciascuna richiesta (spesso in modo concorrente da più threads)
- Nel caso di HttpServlet al posto del metodo service (che pure è presente) vengono invocati metodi più specifici che corrispondono al protocollo HTTP:
  - HTTP GET → doGet
  - HTTP POST → doPost

Java Servlet: Inizializzazione

- Per customizzare l'inizializzazione una Servlet può implementare o fare overriding del metodo init
- Il metodo init prende come argomento un'istanza di javax.servlet.ServletConfig
  - contiene i parametri di inizializzazione
  - utilizza il file descrittore in **WEB-INF/web.xml**
- Anche GenericServlet implementa l'interfaccia ServletConfig
  - metodo init con nessun argomento

Java Servlet: Inizializzazione

```
public class HelloServlet extends HttpServlet {
  private String name = null;

  public void init(ServletConfig config) { //like a constructor
    super.init(config);
    this.name = config.getInitParameter("name");
  }

  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException {
    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter();
    out.print("<html><head><title>Hello</title></head>");
    out.print("<body><p>Hello ");
    if (this.name == null) {
      out.print(req.getRemoteAddr());
    } else {
      out.print(this.name);
    }
    out.println("!</p></body></html>");
    out.flush();
  }
}
```

Richiesta/Risposta: **service**

- La gestione della richiesta/risposta è affidata al metodo service che ha i seguenti argomenti:
  - ServletRequest: richiesta del cliente (lettura)
  - ServletResponse: risposta al cliente (scrittura)
- Nel caso specifico di HttpServlet il metodo service è un "dispatcher" verso altri metodi specifici del protocollo HTTP (doGet, doPost, …)
  - HttpServletRequest: HTTP request
  - HttpServletResponse: HTTP response

Lettura:**(Http)ServletRequest**

- La richiesta del client consente di accedere alle seguenti informazioni:
  - Client: IP/hostname, porta
  - Browser: locale, headers, security
  - Request: headers, cookies, path, parameters, content-type, -length, -encoding, -body
  - User: authorization/authentication (role)
  - Session: attributi della sessione
  - Request shared storage: attributi della richiesta

(Http)ServletRequest

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {
  String clientIp = req.getRemoteAddr();
  int clientPort = req.getRemotePort();
  Locale browserLocale = req.getLocale();
  boolean isBrowserSecure = req.isSecure();
  String userAgent = req.getHeader("User-Agent");
  Cookie[] clientCookies = req.getCookies();
  String requestUri = req.getRequestURI();
  StringBuffer requestUrl = req.getRequestURL();
  String contextPath = req.getContextPath();
  String servletPath = req.getServletPath();
  String requestParamValue = req.getParameter("myParam");
  String contentType = req.getContentType();
  String user = req.getRemoteUser();
  boolean isAdmin = req.isUserInRole("admin");
  List shoppingCart = (List)
    req.getSession().getAttribute("shopping-cart");
  Object myTempData = req.getAttribute("my temp data");
  //Other operations
}
```

Scrittura:**(Http)ServletResponse**

- La risposta al client consente di accedere alle seguenti informazioni:
  - Codici di stato
  - Headers
  - Cookies
  - Contenuto: length, type, body
  - URL Encoding: session tracking
- È importante specificare il codice di stato (default HTTP 200 OK) e gli headers della risposta HTTP prima che questa sia inviata al client

(Http)ServletResponse

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse resp)
 throws ServletException, IOException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<html>");
  out.println("\t<head><title>Hello World</title></head>");
  out.println("\t<body>");
  out.println("\t\t<p>Hello " + req.getRemoteAddr() + "!</p>");
  out.println("\t\t<p>Server time is " + new Date() + "</p>");
  out.println("\t</body>");
  out.println("</html>");
  out.flush(); // sends data back to the client
  out.close(); // not required (called automatically)
}
```

Java Servlet: Distruzione

- Il metodo destroy viene invocato ogni volta che la Servlet deve essere deallocata
  - ad es.: stop del server, reloading dell'applicazione
- Possibile chiamata del metodo service su una Servlet su cui è stato invocato il metodo destroy in ambiente multi-threading
  - Thread-safe
- Tutte le risorse allocate al momento della init sono rilasciate
- Il metodo destroy viene chiamato una sola volta durante il ciclo di vita della Servlet

Java Servlet: Distruzione

```
public class IPLoggerServlet extends HttpServlet {
  private Writer ipLog = null;
  public void init(ServletConfig config) throws
   ServletException {
    super.init(config);
    try {
      ipLog = new FileWriter(config.getInitParameter("file"));
    } catch (IOException e) {
      throw new ServletException("Failed to open log file");
    }
  }
  protected void doGet(HttpServletRequest req,
                       HttpServletResponse resp) throws
   ServletException,IOException {
    this.ipLog.write(req.getRemoteAddr() + "\n");
    resp.setContentType("text/plain");
    resp.getWriter().println("Logged " + req.getRemoteAddr());
  }
  public void destroy() { //like finalize with a guarantee
    try {
      this.ipLog.flush();
      this.ipLog.close();
    } catch (IOException e) {
      super.log("Failed to flush/close log file", e);
    }
  }
}
```

Java Servlet: Deployment

- Prima di poter essere usata una Servlet deve essere:
  - Compilata tramite le classi fornite dalla Servlet API
    - Java EE SDK
    - Servlet container: ${jboss.common.lib.url}/servlet-api.jar
  - specificata nel descrittore dell'applicazione web (WEB-INF/web.xml)
  - impacchettata in un archivio WAR
  - distribuita su un Servlet container (Java EE AS)
  - acceduta tramite URL(s)

Il Descrittore Web: WEB-INF/web.xml

- File di configurazione dell'applicazione web
- Specifica:
  - Nome, descrizione
  - Filters e Servlets (mapping + init parameters)
  - Listeners
  - Session timeout
  - Welcome files: index.html, index.jsp, etc.
  - Risorse e riferimenti a EJB
  - …

Il Descrittore Web: WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>My Application</display-name>
  <description>
   Description of my application
  </description>
  ...
</web-app>
```

Java Servlet: Definizione e Mapping

```
<servlet>
 <servlet-name>IPLoggerServlet</servlet-name>
 <servlet-class>example.servlet.IPLoggerServlet</servlet-class>
  <init-param>
   <param-name>file</param-name>
   <param-value>/WEB-INF/ip.log</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
 <servlet-name>IPLoggerServlet</servlet-name>
 <url-pattern>/ip</url-pattern>
</servlet-mapping>

<servlet-mapping>
 <servlet-name>IPLoggerServlet</servlet-name>
 <url-pattern>/logmyip</url-pattern>
</servlet-mapping>
```

Definizione

Nome

Fully-qualified Class name

Parametri di inizializzazione (opzionali)

URL Mappings

Struttura Applicazione Web

- Supponiamo di aver sviluppato l'applicazione myServletApp
- Packaging in una gerarchia di directories specifica
  - myServletApp
    - Contenuti statici (.html, .gif, .jpg, .js, .css, etc.)
    - File JSP (.jsp)
    - WEB-INF/web.xml (descrittore obbligatorio)
    - WEB-INF/classes/ risultato della compilazione dei sorgenti (.class)
    - WEB-INF/lib/ librerie aggiuntive (.jar)

Struttura Applicazione Web (2)

- Supponiamo di:
  - aver sviluppato l'applicazione myServletApp
  - aver implementato 3 Servlets
    - HelloServlet, HelloWorldServlet, IPLoggerServlet
    - usando il package it.sipe.javaee.servlet
- La struttura delle directories sarà la seguente:
  - myServletApp/
    - WEB-INF/
      - web.xml
      - classes/
        - it/sipe/javaee/servlet/
          - HelloServlet.class
          - HelloWorldServlet.class
          - IPLoggerServlet.class

Struttura Applicazione Web (3)

- L'applicazione web può essere compressa in un archivio WAR (file .war)
- L'archivio non deve contenere la root dell'applicazione (myServletApp/)
  - il nome dell'archivio diventa quello dell'applicazione
- L'archivio WAR può essere generato con i tools messi a disposizione dal JDK o, meglio, da un tool chiamato Ant

Deployment dell'applicazione web

- Operazione specifica che dipende dal particolare Java EE AS
- Solitamente è sufficiente copiare la directory root dell'applicazione (o il corrispondente archivio WAR) nella directory di deployment del server Java EE
- Su JBoss questo si traduce in copiare all'interno di ${jboss.server.home.url}/deploy/
  - la directory root dell'applicazione myServletApp/

  oppure
  - L'archivio WAR dell'applicazione myServletApp.war

Accesso alle Servlet

- Garantito tramite URL mapping
- URLs multipli possono puntare alla stessa Servlet
- I mapping possono basarsi anche su “wildcards”
  - *.ip, /ip/*, etc.
- Gli URLs sono relativi al contesto dell’applicazione (/myServletApp/)

Accesso alle Servlet

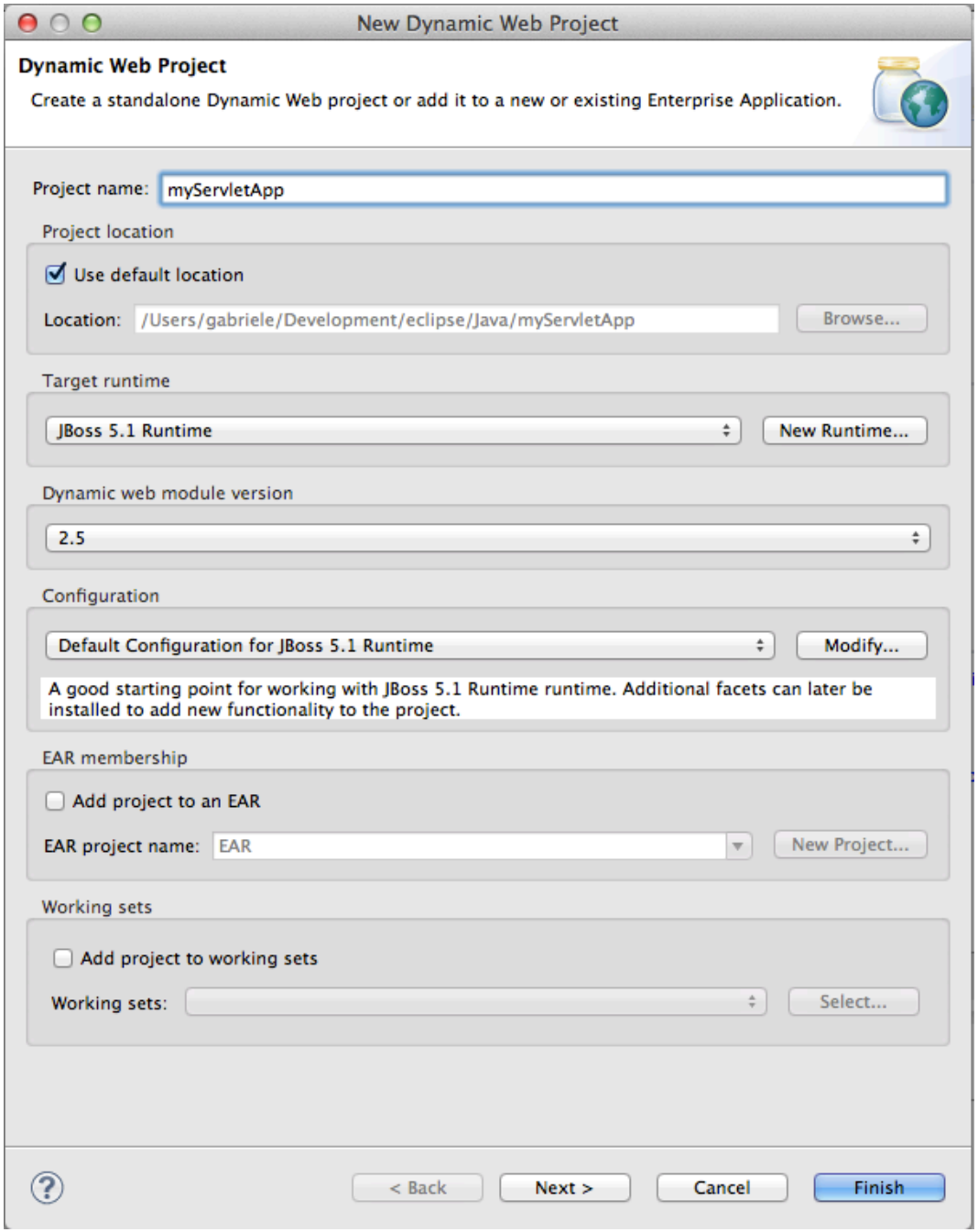
```
...
<servlet>
  <servlet-name>IPLoggerServlet</servlet-name>
  <servlet-class>
    example.servlet.IPLoggerServlet
  </servlet-class>
  <init-param>
    <param-name>file</param-name>
    <param-value>/WEB-INF/ip.log</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>IPLoggerServlet</servlet-name>
  <url-pattern>*.ip</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>IPLoggerServlet</servlet-name>
  <url-pattern>/ip/*</url-pattern>
</servlet-mapping>
...
```

Java Servlet + Eclipse

- Eclipse semplifica notevolmente lo sviluppo di un'applicazione web
- Usare il wizard per creare un Progetto Web Dinamico
  - questa operazione aggiunge tutte le librerie Java EE necessarie al progetto
- Usare il wizard per la creazione di Servlet
  - Facilita l'implementazione dei metodi della Servlet
  - Aggiorna automaticamente il descrittore (web.xml)
- Packaging WAR dell'applicazione
- Un-/Re-/Deploy del WAR

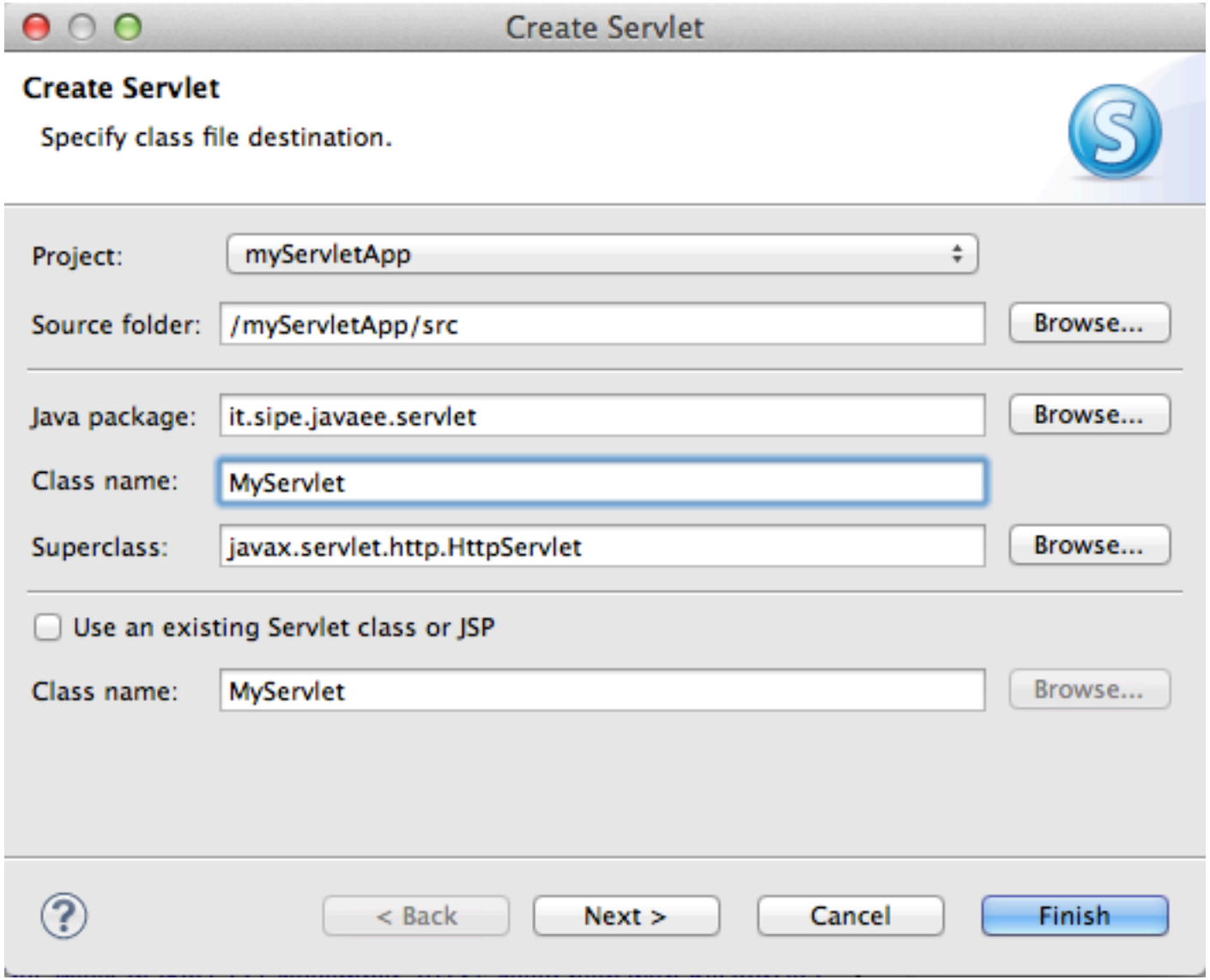
Eclipse: Creazione Progetto Web Dinamico

- File → New → Dynamic Web Project
- Specificare il nome del progetto (ad es. myServletApp)
- Selzionare JBoss 5.1 come target runtime
- Click su Next → Next → Finish


New Dynamic Web Project
Dynamic Web Project
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.
Project name: myServletApp
Project location
Use default location
Location: /Users/gabriele/Development/eclipse/Java/myServletApp
Browse...
Target runtime
JBoss 5.1 Runtime
New Runtime...
Dynamic web module version
2.5
Configuration
Default Configuration for JBoss 5.1 Runtime
Modify...
A good starting point for working with JBoss 5.1 Runtime runtime. Additional facets can later be installed to add new functionality to the project.
EAR membership
Add project to an EAR
EAR project name: EAR
New Project...
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Cancel
Finish

Eclipse: Creazione Servlet (1)

- Click destro sul progetto creato New → Servlet
- Specificare il package
  - ad es.: it.sipe.javaee.servlet
- Specificare il nome della classe Servlet
  - ad es.: MyServlet
- Click su Next → Finish
- La Servlet verrà automaticamente mappata sull'URL /MyServlet


Create Servlet
Create Servlet
Specify class file destination.
Project:
myServletApp
Source folder:
/myServletApp/src
Browse...
Java package:
it.sipe.javaee.servlet
Browse...
Class name:
MyServlet
Superclass:
javax.servlet.http.HttpServlet
Browse...
Use an existing Servlet class or JSP
Class name:
MyServlet
Browse...
< Back
Next >
Cancel
Finish

Eclipse: **web.xml**

- Aggiungere un parametro di inizializzazione nel descrittore web.xml
- Utilizzare il tag <init-param>

```
<servlet>
  <description></description>
  <display-name>MyServlet</display-name>
  <servlet-name>MyServlet</servlet-name>
  <servlet-class>it.sipe.javaee.servlet.MyServlet</servlet-class>
  <init-param>
      <param-name>name</param-name>
      <param-value>World</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>/MyServlet</url-pattern>
</servlet-mapping>
```

I metodi: **doGet** e **doPost**

- Implementare il metodo **doGet**
  - per rispondere a richieste HTTP GET
- Implementare il metodo **doPost**
  - per rispondere a richieste HTTP POST
- Spesso si implementa solo uno dei due metodi (ad es. doGet) e l'altro (ad es. doPost) richiama semplicemente il metodo già implementato

I metodi: **doGet** e **doPost**

doGet

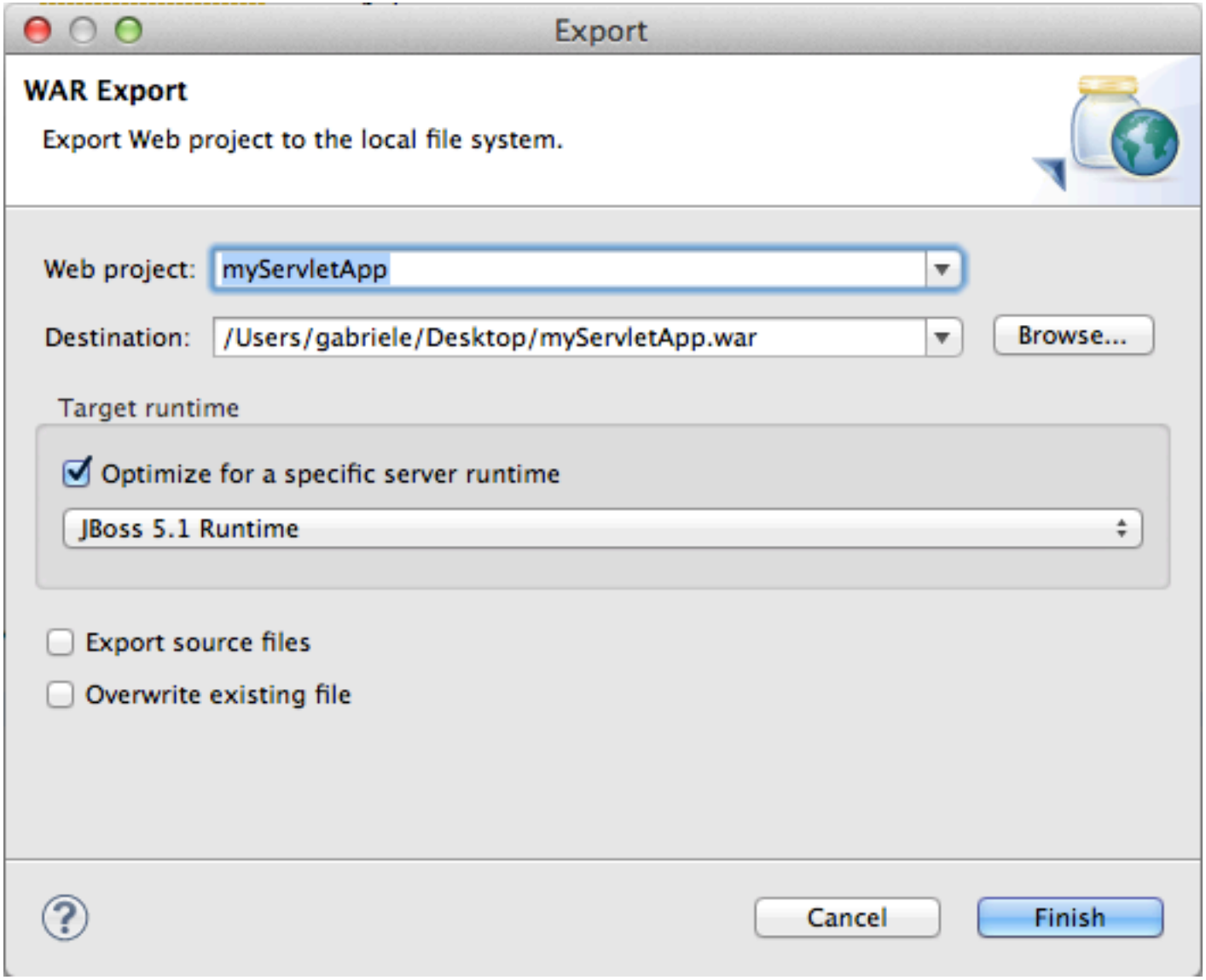
```
/**
 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    // TODO Auto-generated method stub
    response.setContentType("text/plain");
    PrintWriter out = response.getWriter();
    out.print("Hello ");
    out.println(super.getInitParameter("name"));
}
```

doPost

```
/**
 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    // TODO Auto-generated method stub
    doGet(request, response);
}
```

Eclipse: Packaging dell'applicazione

- Click destro sul progetto Export → WAR file
- Specificare il nome, la destinazione e ottimizzazione per il target runtime (JBoss 5.1)


Export
WAR Export
Export Web project to the local file system.
Web project:
myServletApp
Destination:
/Users/gabriele/Desktop/myServletApp.war
Browse...
Target runtime
Optimize for a specific server runtime
JBoss 5.1 Runtime
Export source files
Overwrite existing file
Cancel
Finish

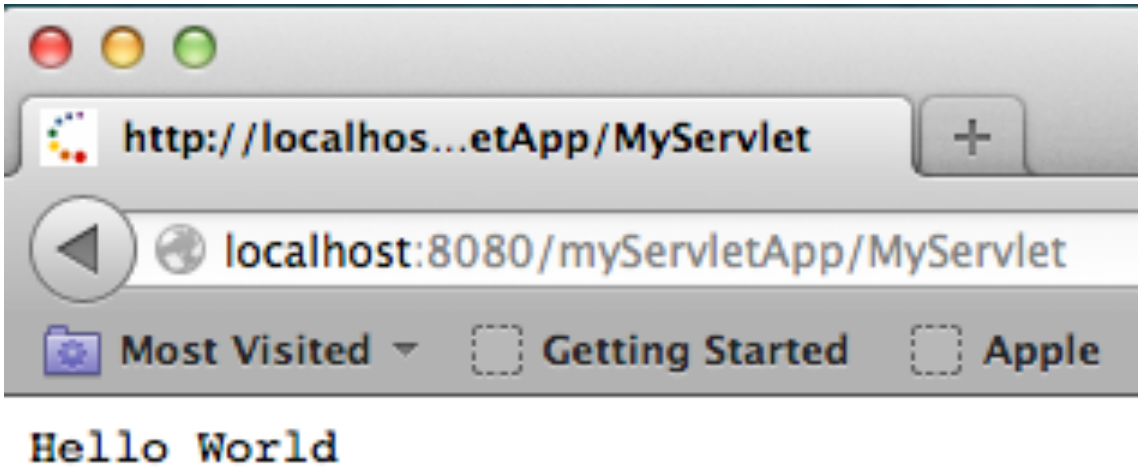
Deployment dell'applicazione

- Copiare l'archivio WAR generato all'interno della directory ${jboss.server.home.url}/deploy
- Avviare il server Jboss via script o tramite Eclipse
- Osservare il deployment sul log della console

```
14:36:11,527 INFO  [TomcatDeployment] deploy, ctxPath=/myServletApp
```

Testing dell'applicazione

Accedere al seguente URL:
http://localhost:8080/myServletApp/MyServlet



Lab: esercitazione 1

- Creare un progetto Web su Eclipse chiamato **myServletApp**
- All'interno di esso creare una Servlet **SayHelloServlet** che è mappata sull'URL /sayHello (relativo all'applicazione Web)
- Implementare il metodo **doGet** in modo che:
  - la Servlet accetti un parametro **nome** dalla richiesta HTTP GET (query string)
  - a fronte della richiesta

  http://localhost:8080/myServletApp/sayHello?**nome=X**

  restituisca al client una pagina HTML che contiene la stringa "**Hello X**"
- **NOTA:**
  - il **valore** del generico parametro **paramName** contenuto nella richiesta HTTP si ottiene invocando il metodo

    getParameter(String paramName)

    sull'oggetto **HttpServletRequest** argomento del metodo **doGet**

Lab: esercitazione 2

- Creare una Servlet **ConvertiTempServlet** che è mappata sull'URL /convertiTemp
- Implementare il metodo **doGet** in modo che:
  - la Servlet accetti **3 parametri** dalla richiesta HTTP GET: **temp**, **da**, **a**
  - a fronte della richiesta restituisca al client una pagina HTML che contiene:
    - la conversione del valore della temperatura specificata nel parametro **temp** dalla scala espressa nel parametro **da** a quella indicata dal parametro **a**
    - oppure un errore, nel caso di problemi sui parametri di input
- Le scale consentite sono indicate con "**C**" (Celsius), "**F**" (Fahrenheit) e "**K**" (Kelvin) e i valori di temperatura possono rappresentare **int** o **double**

Gestione della Richiesta

- Le Servlet possono gestire le richieste
  - **Direttamente**
    - risposta diretta al client
  - **Indirettamente**
    - Lato Server:
      - tramite l'inclusione lato server del contenuto di un'altra risorsa (**include**)
      - tramite l'inoltro lato server della richiesta ad un'altra risorsa (**forward**)
    - Lato Client:
      - tramite l'inoltro della richiesta ad un'altra risorsa previa comunicazione con il client (**redirect**)

Gestione Indiretta della Richiesta: Lato Server

- Lo **stesso oggetto** che rappresenta la richiesta viene trasferito ad un'altra risorsa server (ad es. una Servlet)
- Il forwarding della richiesta è interamente eseguito **lato server** dal container ed è **trasparente e nascosto al client** (ecco perché l'URL nel browser non cambia!)
- Per invocare un'altra risorsa la Servlet deve ottenere un oggetto **RequestDispatcher** con il metodo della **richiesta**

getRequestDispatcher(String url)

RequestDispatcher

- L'istanza dell'oggetto **RequestDispatcher** può includere o inoltrare la coppia **(req, resp)**
- L'URL è relativo al Servlet Context (intera app)
  - se inizia con il carattere "/"
- L'URL è relativo alla Servlet corrente
  - se **non** inizia con il carattere "/"
- Può specificare un qualsiasi tipo di risorsa:
  - Es., Servlet, JSP, HTML

RequestDispatcher: **include** vs. **forward**

- Con il metodo **include**, RequestDispatcher aggiunge il contenuto della risorsa alla Servlet "chiamante"
- Con il metodo **forward**, RequestDispatcher chiude il buffer di output della Servlet chiamante e passa totalmente la gestione della richiesta alla risorsa chiamata

RequestDispatcher: Esempio

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse resp)
 throws ServletException, IOException {
  if (!this.isAuthenticated(req)) {
    req.getRequestDispatcher("Login.html").forward(req, resp);
  } else {
    PrintWriter out = resp.getWriter();
    resp.setContentType("text/html");
    out.println("<html><head><title>Secure</title></head>");
    out.println("<body>");
    req.getRequestDispatcher("NavBar").include(req, resp);
    ...
    out.println("</body></html>");
  }
}
```

Gestione Indiretta della Richiesta: Lato Client

- Alternativa al forwarding gestito lato server
- Il forwarding della richiesta ad un'altra risorsa passa prima nuovamente al client
- Il container istruisce il browser sulla **nuova richiesta** da eseguire tramite header HTTP
- Il client inizia una **nuova** comunicazione con il server all'URL specificato (l'URL stavolta cambia nel browser!)
- Per invocare un'altra risorsa la Servlet deve chiamare il seguente metodo sull'oggetto **risposta**

sendRedirect(String url)

Forward vs. **Redirect**

- Forward (lato server):
  - la richiesta originale del client non viene modificata e passa da una risorsa all'altra
  - Possibile inoltrare la richiesta solo a componenti server interni allo stesso dominio dell'applicazione
- Redirect (lato client)
  - la richiesta originale viene "persa" e il server indica al browser di effettuare una nuova richiesta verso la risorsa alla quale vogliamo inoltrare il controllo
  - Possibile inoltrare la richiesta a risorse esterne al dominio dell'applicazione

Forward vs. **Redirect**

- Forward (lato server):
  - Preferibile perché più “leggero” se la richiesta può essere interamente gestita dallo stesso dominio
- Redirect (lato client)
  - Inevitabile quando è necessario trasferire il controllo a componenti che risiedono su altri domini (ad es. broker per il pagamento online)

Passaggio di Dati: **Request Scope**

- Necessario quando la gestione della singola richiesta è affidata a più Servlets
- Le Servlets condividono i dati della richiesta nella cosiddetta **request scope storage**
- **ServletRequest** definisce i seguenti metodi per i dati condivisi nel request scope:

void setAttribute(String k, Object v)

Object getAttribute(String k)

void removeAttribute(String k)

Request Scope

- I dati sono disponibili per l'intera durata della richiesta e vengono eliminati una volta che questa è stata servita

SecServlet

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse resp)
 throws ServletException, IOException {
  if (!this.isAuthenticated(req)) {
    req.setAttribute("error", "Must be authenticated!");
    req.getRequestDispatcher("Login").forward(req, resp);
  } else {
    ...
  }
}
```

LoginServlet

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse resp)
 throws ServletException, IOException {
  PrintWriter out = resp.getWriter();
  ...
  String error = req.getAttribute("error");
  if (error != null) {
    out.println("<p style='color:red'>" + errror + "</p>");
  }
  ...
}
```

Passaggio di Dati: **Session Scope**

- HTTP è un protocollo **stateless**
- Spesso necessario mantenere informazioni di stato del cliente attraverso più richieste HTTP (**session**)
- Java EE servers offrono questa possibilità in modo semplice per lo sviluppatore dell'applicazione

Session Scope

1. Alla prima richiesta, il server crea un oggetto “session” e gli assegna un ID univoco (**sessionID**) che invia al client
2. Ad ogni richiesta successiva, il client invia **sessionID** sottoforma di cookie o di parametro della richiesta
3. Il server recupera l’oggetto corrispondente al **sessionID** ricevuto e lo rende disponibile all’applicazione
4. L’applicazione usa l’oggetto sessione per memorizzare (alcuni) dati dell’utente

Session Scope: Cookie

- Il server capisce automaticamente se il browser (client) supporta i cookies
- Se i cookie sono supportati, il server crea un session cookie (**JSESSIONID**) e lo invia al client come risposta della prima richiesta
- Il client invierà al server il cookie **JSESSIONID** ad ogni successiva richiesta

Session Scope: Response

- Se i cookie **non** sono supportati (o sono disabilitati), il server include l'ID della sessione in ogni URL che restituisce al client (response)
- Gli URL devono essere opportunamente codificati con il metodo **encodeURL(String url)** della classe **HttpServletResponse**
- Un URL codificato contiene la stringa "**;jsessionid=xxx**" in coda al path della richiesta ma prima di eventuali parametri specificati nella query string (HTTP GET)

HttpSession

- Per accedere all'oggetto che identifica la sessione utente, invocare il metodo **getSession()** della classe **HttpServletRequest**
- Per salvare dati in sessione invocare il metodo **session.setAttribute(key, value)**
- Per recuperare/rimuovere un dato dalla sessione invocare

session.getAttribute(key)

session.removeAttribute(key)

HttpSession (2)

- Le chiavi sono di tipo **String** mentre i valori sono **Object** (serializzabili)
- Gli oggetti **HttpSession** sono distrutti al momento dell'invocazione di **invalidate()**
- Le sessioni che non sono utilizzate oltre un certo timeout vengono automaticamente invalidate

HttpSession: Esempio

```
public class RequestCounterServlet extends HttpServlet {
  private static final String COUNTER = "request-counter";
  protected void doGet(HttpServletRequest req,
                       HttpServletResponse resp)
   throws ServletException, IOException {
    HttpSession session = req.getSession(); // auto-created
    Integer c = (Integer) session.getAttribute(COUNTER);
    c = c==null? new Integer(1) : new Integer(c.intValue()+1);
    session.setAttribute(COUNTER, c);
    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter();
    out.print("<html><head><title> Counter</title></head>");
    out.print("<body>You visited this page " +c+  " times. ");
    String url = resp.encodeURL(req.getRequestURI());
    out.print("<a href=\"" + url + "\">Reload?</a>");
    out.print("</body></html>");
    out.flush();
  }
}
```

Configurazione della Sessione

- Specificare il valore per il tag **<session-timeout>** figlio di **<session-config>** nel file **web.xml**
- Il valore (espresso in minuti) indica per quanto tempo il Servlet Container mantiene una sessione inattiva (in memoria o su disco)
- Valori <=0 indicano che la sessione non verrà mai distrutta a meno di azioni esplicite (ad es. logout dell'utente)
- Il timeout ha conseguenze significative sulle performance (uso delle risorse) del server

Timeout della Sessione

```
<!ELEMENT session-config (session-timeout?)>
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

Passaggio di Dati: **Application Scope**

- I dati contenuti nello scope request e session sono validi rispettivamente all'interno di una richiesta e di una sessione
- Se è necessario condividere dati per l'intera durata dell'applicazione occorre un altro scope di memorizzazione: application scope
- Ottenere un riferimento a **ServletContext** tramite il metodo **super.getServletContext()**

ServletContext

- Ottenere un riferimento a **ServletContext** tramite il metodo **super.getServletContext()**
- Usare l'istanza recuperata per invocare i metodi get/set/remove per ottenere, valorizzare, rimuovere attributi dell'applicazione
- Utilizzato per caching e dati globali all'applicazione

Application Scope: Esempio

```
public class UniqueVisitorsServlet extends HttpServlet {
  public static final String VISITORS = "VISITORS";

  public void init() throws ServletException {
    super.getServletContext().setAttribute(VISITORS,
      new HashSet());
  }

  protected void doGet(HttpServletRequest req,
                       HttpServletResponse resp)
   throws ServletException, IOException {
    ServletContext ctx = super.getServletContext();
    Set visitors = (Set) ctx.getAttribute(VISITORS);
    synchronized (visitors) {
      visitors.add(req.getRemoteAddr());
    }
    PrintWriter out = resp.getWriter();
    out.print("<p>" + visitors.size()+ " Unique Visitors</p>");
  }
}
```

Note

The output of this servlet can be included in others (for tracking purposes):
`req.getRequestDispatcher("/unique").include(req, resp);`

Inizializzazione Applicazione

- Come le Servlets, anche le applicazioni possono avere parametri di inizializzazione
- Definiti nel **web.xml** come tag **<context-param>** prima di qualsiasi dicharazione di Servlet
- Acceduti tramite il metodo **getInitParameter** della classe **ServletContext**

Servlet.getInitParameter

vs.

ServletContext.getInitParameter

Inizializzazione Applicazione: Esempio

```
...
<web-app ...>
  <display-name>...</display-name>
  <description>...</description>

  <context-param>
    <param-name>globalVariableOne</param-name>
    <param-value>some global value</param-value>
  </context-param>

  <context-param>
    <param-name>globalVariableTwo</param-name>
    <param-value>another global value</param-value>
  </context-param>

  <servlet>...</servlet>
  ...
</web-app>
```

Thread-Safety

- Un Java EE server è un'applicazione Java **multi-threaded**
- I dati che sono condivisi possono venir acceduti in modo concorrente
- Necessario proteggere:
  - Oggetti che riferiscono attributi negli scope **session** e **application** (context)
  - Dati in memoria principale: classi e variabili di istanza
  - Risorse "esterne": file, connessioni a DB e/o di rete

Thread-Safety (2)

- I dati sono acceduti in modo concorrente da:
  - Diversi componenti web del server
  - Richieste concorrenti alla stessa componente server

Thread-Safety: Esempio

```
public class GlobalRequestCounterServlet extends HttpServlet {
  private static final String COUNTER = "global-counter";

  protected void doGet(HttpServletRequest req,
                       HttpServletResponse resp)
   throws ServletException, IOException {
    ServletContext ctx = super.getServletContext();
    Integer c = null;
    synchronized (ctx) { // also possible to sync on 'this'
      c = (Integer) ctx.getAttribute(COUNTER);
      c = c == null ? new Integer(1) :
                      new Integer(c.intValue() + 1);
      ctx.setAttribute(COUNTER, c);
    }
    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter();
    out.print("<html><head><title> Counter</title></head>");
    out.print("<body>You visited this page " + c + " times. ");
    String url = resp.encodeURL(req.getRequestURI());
    out.print("<a href=\"" + url + "\">Reload?</a>");
    out.print("</body></html>");
    out.flush();
  }
}
```

Response: Binary Data

- A volte è utile restituire una risposta al client che contenga dati binari (ad es. immagine)
- Occorre utilizzare **getOutputStream** anziché il solito **getWriter**

```
setContentType("mime/type")
setContentLength(sizeOfData)
setHeader("pragma", "no-cache")
```

Response: Binary Data

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse resp)
 throws ServletException, IOException {
  File imageFile = new File("some-image.gif");
  FileInputStream in = new FileInputStream(imageFile);
  try {
    resp.setContentLength((int) imageFile.length());
    resp.setContentType("image/gif");
    resp.setHeader("pragma", "no-cache");
    ServletOutputStream out = resp.getOutputStream();
    byte[] buf = new byte[1024];
    int nread;
    while ((nread = in.read(buf)) > 0) {
      out.write(buf, 0, nread);
    }
    out.flush();
  } finally {
    in.close();
  }
}
```

Welcome File List

- Configurabili in **<welcome-file-list>** nel **web.xml**
- Definisce i file da restituire a fronte di una richiesta di una directory sul server

http://www.example.com/dir/

```
<!ELEMENT welcome-file-list (welcome-file+)>
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```