

Java EE – Servlet e JSP

- Java Enterprise Edition
 - Servlet, JSP, EL, JSTL
 - DataSource per integrazione con DBMS via JDBC
 - Introduzione a JPA
- Progetto di riferimento
 - <https://github.com/egalli64/mdwa>
 - Maven
 - Tomcat 9 o WildFly 19
 - Hibernate

Java Enterprise Edition (Java EE)

- Prima di JDK 5 nota come J2EE
- In transizione da Oracle verso Eclipse Foundation
 - Jakarta EE <https://jakarta.ee/>
- Estende la Standard Edition (la versione corrente è basata su JDK 8) con specifiche per lo sviluppo enterprise
 - Distributed computing, web services, ...
- Applicazioni Enterprise sono eseguite da un reference runtime (Application Server o microservice)
 - Gestisce il ciclo di vita delle componenti, multithreading, sicurezza, ...

Apache Tomcat

- Web Server che implementa parzialmente le specifiche Java EE
 - <https://tomcat.apache.org/>
- La versione 9 richiede Java SE 8+ e supporta
 - Java Servlet 4.0
 - Java Server Pages 2.3
 - Java Expression Language 3.0
 - Java Web Socket 1.1
- Dalla shell, folder bin, eseguire lo script *startup* (set JAVA_HOME)
 - Porta 8080, configurabile in server.xml, elemento Connector

Red Hat JBoss EAP – WildFly

- **Application Server**, implementa tutte le specifiche Java EE
 - <https://developer.jboss.org/> <https://wildfly.org/>
- Basato sul Web Server Undertow
- WildFly 19 richiede Java SE 8+ (consigliata la più recente LTS) e supporta
 - **Java Servlet** 4.0
 - **Java Server Pages** 2.3
 - **Java Expression Language** 3.0
 - Enterprise Java Bean 3.2
 - ...
- Dalla shell, folder bin, eseguire lo script *standalone* (set JAVA_HOME)
 - Porta **8080**, configurabile in standalone/configuration/standalone.xml
 - socket-binding-group → socket-binding – http

Eclipse per Java EE

- Help → Eclipse Marketplace (WTP – Web Tools Platform)
 - Eclipse Java Enterprise Developer Tools
 - Eclipse JST Server Adapters
 - Eclipse Web Developer Tools
 - Eclipse XML Editors and Tools
 - Maven (Java EE) integration for Eclipse WTP
 - HTML Editor
- Java EE Perspective

JBoss / WildFly in Eclipse

- Nella prospettiva Java EE
 - Servers
 - New Server: Red Hat JBoss Middleware → JBoss AS, WildFly ...
- È necessario un restart, dopodiché i server JBoss sono disponibili
 - Servers
 - New Server: JBoss Community → WildFly 19
- Ora è possibile accedere da Eclipse alla configurazione di WildFly
 - XML Configuration
 - Server Details
 - Filesets
 - JMX

Eclipse Dynamic Web Project

- Approccio nativo Eclipse, alternativo a Maven
- Principali setting nel wizard
 - Target runtime: Tomcat, WildFly, ...
 - Window, Show View, Servers
 - Servers View → New → Server
 - Dynamic Web module version: 4
 - Configuration: Default
 - Generate web.xml DD (tick)
- Project Explorer
 - WebContent: HTML e JSP
 - vs. Deployed Resources → webapp
 - Java Resources: Servlet
- Generazione del WAR
 - Export, WAR file
 - vs. Run as, Maven Install

Request – Response

- Il client manda una request al Web Server per una specifica risorsa
- Il web server genera una response
 - File HTML
 - Immagine, PDF, ...
 - Errore (404 not found, ...)
- Si comunica con il protocollo HTTP, di solito con i metodi GET e POST
 - GET: eventuali parametri sono passati come parte della request URL
 - POST: i parametri sono passati come message body (o “payload”)
- Associazione tra request e un nuovo thread di esecuzione della servlet

Servlet vs JSP

- Servlet: Java puro (HTML visto come testo)

`extends HttpServlet`

`@WebServlet("/s09/timer")`

```
try (PrintWriter writer = response.getWriter()) {  
    // ...  
    writer.println("<h1>" + LocalTime.now() + "</h1>");  
    // ...  
}
```

`doGet()`

- JSP: HTML con dei frammenti Java al suo interno

scriptlet

```
<h1>  
    <%  
        out.print(LocalTime.now());  
    %>  
</h1>
```

`/s09/timer.jsp`

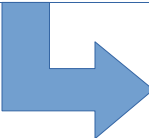
- In entrambi i casi il web client può fare la sua request via:
 - HTML link, form o AJAX via XMLHttpRequest

Servlet e JSP

- Servlet: gestisce l'interazione con il metodo HTTP e la logica del controller
- JSP: generazione di un documento HTML nella response

```
String user = request.getParameter("user");  
Set<Character> set = new TreeSet<>();  
// ...  
request.setAttribute("set", set);  
RequestDispatcher rd = request.getRequestDispatcher("/s10/checker.jsp");  
rd.forward(request, response);
```

comunicazione
via attributi
nella request



```
<%  
    Set<Character> set = (Set<Character>)request.getAttribute("set");  
    Iterator<Character> it = set.iterator();  
    while (it.hasNext()) {  
        out.print(" " + it.next());  
    }  
%>
```

Session

- Le connessioni HTTP sono stateless. HttpSession identifica una conversazione (cookie)

```
HttpSession session = request.getSession();
LocalTime start = (LocalTime) session.getAttribute("start");
// ...

if (start == null) {
    // ...
    session.setAttribute("start", LocalTime.now());
} // ...

if (request.getParameter("done") != null) {
    session.invalidate();
    // ... page generation with goodbye message
}

// ...
```

Elementi JSP

direttiva "page"

direttiva
"include"

commento

dichiarazione

scriptlet

espressione

```
<%@page contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Hello JSP</title>
</head>
<body>
  <%@include file="header.html"%>
  <!-- Just as example --%>
  <%! int unreliableCounter = 0; %>
  <h1>
    <% out.print("Counter was " + unreliableCounter); %>
    now is
    <%= ++unreliableCounter %>
  </h1>
  <!-- ... -->
</body>
</html>
```

Proprietà (!?) e metodi

Nel body del metodo
che implementa
JspPage._jspService()

Le espressioni JSP **non** sono
terminate dal punto e virgola!
(argomento di out.print())

JSP useBean

Servlet

```
request.setAttribute("user", new User(name, id));
```

User è
un JavaBean

JSP script

```
<%  
    User user = (User) request.getAttribute("user");  
%>  
<%=user.getName()%>  
<%=user.getId()%>
```

page
request
session
application

JSP
action
element

```
<jsp:useBean id="user" class="dd.User" scope="request">  
    <jsp:setProperty name="user" property="name" value="Bob" />  
    <jsp:setProperty name="user" property="id" value="42" />  
</jsp:useBean>  
<jsp:getProperty name="user" property="name" />  
<jsp:getProperty name="user" property="id" />
```

Default values

JSP useBean /2

http:// ... /s14/fetch.jsp?name=Tom&id=42

accesso esplicito
ai parametri della
request

```
<jsp:useBean id="user" class="dd.User">  
  <jsp:setProperty name="user" property="name" param="name" />  
  <jsp:setProperty name="user" property="id" param="id" />  
</jsp:useBean>
```

accesso implicito

```
<jsp:setProperty name="user" property="name" />  
<jsp:setProperty name="user" property="id" />
```

deduzione implicita

```
<jsp:setProperty name="user" property="*" />
```

```
<jsp:getProperty name="user" property="name" />  
<jsp:getProperty name="user" property="id" />
```

JSP Expression Language

```
request.setAttribute("doc", new Document("JSP Cheatsheet", new User("Tom", 42)));
```

JavaBean aggregato come attributo nella request da servlet a JSP

```
<p>Doc title: ${doc.title}</p>
<p>Doc user: ${doc.user.name}</p>
<p>Doc title again: ${requestScope.doc.title}</p>
```

oggetti impliciti EL
per gli scope

```
http:// ... /s15/direct.jsp?x=42&y=a&y=b
```

Chiamata diretta a un JSP

oggetti impliciti EL
per i parametri

```
<p>${param.x}</p>
<p>${paramValues.y[1]}</p>
```

pageScope
requestScope
sessionScope
applicationScope

Servlet e parametri

- Dalla request
 - `getParameter()`
 - Ritorna il valore del parametro come `String`
 - Chiamato su un array, ritorna il primo valore
 - `getParameterValues()`
 - Ritorna i valori associati al parametro come array di `String`
 - Se la request non ha quel parametro → `null`

Forward e redirect

- forward()
 - Metodo di RequestDispatcher
 - getRequestDispatcher() sulla request
 - La risorsa target può essere servlet, JSP, HTML
- sendRedirect()
 - Metodo della response
 - URL tipicamente esterno al sito corrente

```
RequestDispatcher rd = request.getRequestDispatcher(destination);  
rd.forward(request, response);
```

```
<jsp:forward page="../index.html"></jsp:forward>
```

JSP action element

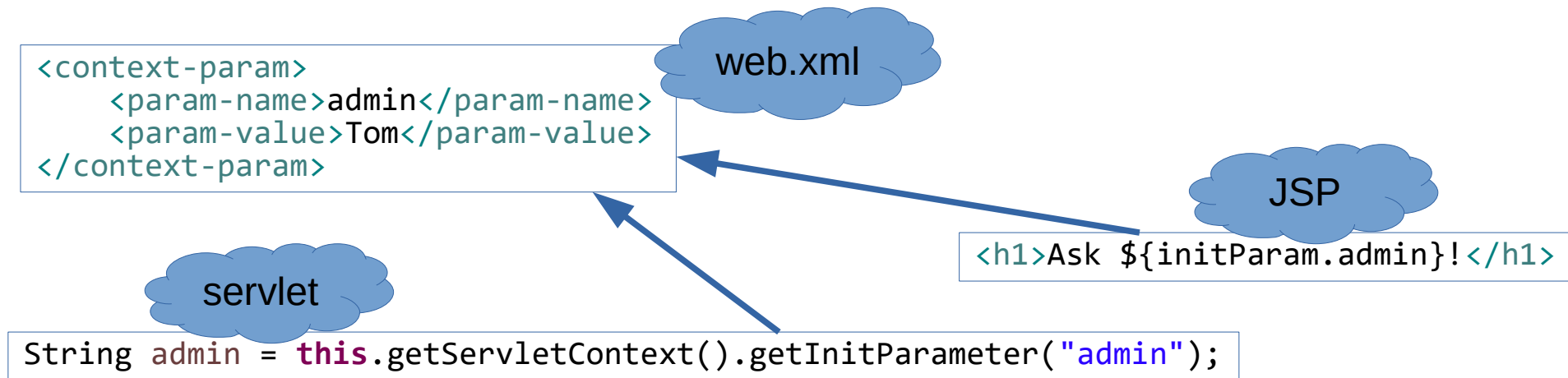
```
response.sendRedirect("https://tomcat.apache.org/");
```

JSTL

```
<c:redirect url="https://tomcat.apache.org/" />
```

context-param

- Parametri visibili in tutta la webapp
- Definiti in WEB-INF/web.xml



Pagine di errore

- In web.xml si specifica il mapping tra tipo di errore e pagina associata

```
<error-page>
  <error-code>404</error-code>
  <location>/s19/404.jsp</location>
</error-page>
<error-page>
  <error-code>500</error-code>
  <location>/s19/500.jsp</location>
</error-page>
```

Page not found

Internal error

Da JSP si può accedere
all'eccezione che ha causato
l'internal error

Oggetto implicito EL

```
${pageContext.exception["class"]}
${pageContext.exception["message"]}
```

JSTL: JSP Standard Tag Library

- Nel POM va indicata la dipendenza per `javax.servlet` (groupId) `jstl` (artifactId)
 - Versione corrente: 1.2
- Nel JSP direttiva taglib per la libreria da usare
 - core (c), formatting (fmt), SQL (sql), functions (fn), ...

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

```
<c:if test="${param.x != null}">  
    <p>Parameter x is ${param.x}</p>  
</c:if>
```

JSTL core loop

- Su un iterable `c:forEach`, su stringa tokenizzata `c:forEachTokens`

```
User[] users = new User[] { /* ... */ };  
Double[] values = new Double[12]; // ...  
String names = "bob,tom,bill";
```

La servlet crea alcuni attributi nella request e passa il controllo a un JSP per la visualizzazione

```
<c:forEach var="user" items="${users}">  
  <p>${user.name},${user.id}</p>  
</c:forEach>
```

```
<c:forEachTokens var="token" items="${names}" delims=", ">  
  <p>${token}</p>  
</c:forEachTokens>
```

```
<c:forEach var="value" items="${values}" begin="0" end="11" step="3" varStatus="status">  
  <p>  
    ${status.count}: ${value}  
    <c:if test="${status.first}">(first element)</c:if>  
    <c:if test="${status.last}">(last element)</c:if>  
    <c:if test="${not(status.first or status.last)}">(index is ${status.index})</c:if>  
  </p>  
</c:forEach>
```

Altri JSTL core tag

- choose-when: switch (e if-else)
- out: trasforma HTML in testo semplice
- redirect: ridirezione ad un'altra pagina
- remove: elimina un attributo
- set: set di un attributo nello scope specificato
- url: generazione di URL basato sulla root

DataSource per Tomcat

- Si mette una copia del JAR del driver **JDBC** nella directory **lib** di Tomcat
- Si definisce
 - Una risorsa nel **conf/context.xml** di Tomcat (in Eclipse: Project Explorer – Servers)
 - Un riferimento alla risorsa nel WEB-INF/**web.xml** della web app
- Il **data source** è utilizzabile da servlet e JSP nella web app

```
<Resource name="jdbc/me" type="javax.sql.DataSource" driverClassName="com.mysql.cj.jdbc.Driver"
auth="Container" url="jdbc:mysql://localhost:3306/me?serverTimezone=Europe/Rome" username="me"
password="password" />
```

```
<resource-ref>
  <res-ref-name>jdbc/me</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

JSTL sql taglib

```
<sql:query dataSource="jdbc/me" var="regions">
  select * from regions
</sql:query>
```

DataSource per JBoss / WildFly

- (per MySQL) in modules/system/layers/base/com/mysql/main
 - Una copia del driver JDBC mysql-connector-java-8.0.xx.jar
 - Un file di configurazione **module.xml**
- In standalone/configuration/**standalone.xml**
 - Aggiungere ai datasources un **datasource** e un **driver**
- Nella web app, WEB-INF/**jboss-web.xml**
 - Si definisce il nome del data source utilizzabile nel codice

```
<module xmlns="urn:jboss:module:1.5" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-8.0.xx.jar" />
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```

```
<datasource jndi-name="java:jboss/datasources/meDS" pool-name="meDS"
  enabled="true" use-java-context="true">
  <connection-url>
    jdbc:mysql://localhost:3306/me?serverTimezone=Europe/Rome
  </connection-url>
  <driver>mysql</driver>
  <security>
    <user-name>me</user-name>
    <password>password</password>
  </security>
</datasource>
<drivers>
  <!-- ... -->
  <driver name="mysql" module="com.mysql">
    <driver-class>com.mysql.cj.jdbc.Driver</driver-class>
    <xa-datasource-class>
      com.mysql.cj.jdbc.MysqlXADataSource
    </xa-datasource-class>
  </driver>
</drivers>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <resource-ref>
    <res-ref-name>jdbc/me</res-ref-name>
    <jndi-name>java:jboss/datasources/meDS</jndi-name>
  </resource-ref>
</jboss-web>
```

```
<sql:query dataSource="jdbc/me" var="regions">
  select * from regions
</sql:query>
```


Context lifecycle servlet listener

- Servlet chiamata all'inizializzazione e distruzione della web app
- `@WebListener` implements `ServletContextListener`
 - `void contextInitialized(ServletContextEvent sce)`
 - `sce.getServletContext().setAttribute("start", LocalTime.now());`
 - `void contextDestroyed(ServletContextEvent sce)`
 - Eventuale cleanup delle risorse allocate all'inizializzazione

```
<h1>The web app started at ${applicationScope.start}</h1>
```

Filter

- In ingresso: audit, log, security check, ridirezione
- In uscita: modifica della response generata

filtro su tutte le request

O magari "*.jsp"

```
@WebFilter(dispatcherTypes = { DispatcherType.REQUEST }, urlPatterns = { "/" })  
public class FilterAllReq implements Filter {  
    // ...  
  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  
        throws IOException, ServletException {  
        // ...  
        chain.doFilter(request, response);  
        // ...  
    }  
}
```

La logica può andare prima o dopo il doFilter() [IN o OUT]

JPA – Esempio

- DAO: Data Access Object
- ORM: Object/Relational Mapping
- Hibernate ORM
 - <https://hibernate.org/orm/>
- JPA: Java Persistence API