Java EE – Servlet e JSP

- Java Enterprise Edition
  - Servlet
  - JSP, EL, JSTL
  - DataSource per integrazione con DBMS via JDBC
- Progetto di riferimento
  - https://github.com/egalli64/jees
    - Tomcat 9 / JBoss EAP 7
      - In Eclipse va indicato come “Runtime” nelle Properties del progetto – Project Facets
    - MySQL / Oracle DB

Servlet vs JSP

- Servlet: classe Java (HTML visto come testo)
  - Estende **HttpServlet**
  - Annotata **WebServlet**
    - Value: URL associato, es: "**/**s02/timer"
  - Metodi doXxx() gestiscono i metodi HTTP
    - I due *comunemente* usati: GET e POST

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

Timer.doGet()

- JSP: HTML (con frammenti Java)
  - URL determinato dal folder in cui si trova
  - Il codice Java può essere messo in tag detti **scriptlet**
    - Modalità considerata oggi obsoleta, vedi EL, JSTL
    - L'oggetto implicito JSP "**out**" è il PrintWriter per la response

```
<h1>
    <!-- HTML comment -->
    <% out.print(LocalTime.now()); %>
    <%-- JSP comment --%>
</h1>
```

/s02/timer.jsp

- È possibile fare tutto solo con servlet o JSP ma conviene combinarle

Servlet e JSP!

- Servlet: gestione della request, esecuzione della logica relativa, generazione dei dati di output
- JSP: generazione della response come pagina HTML, sulla base dei dati elaborati dalla servlet

```
String user = request.getParameter("user");
Set<Character> set = new TreeSet<>();
// ...
request.setAttribute("set", set);
RequestDispatcher rd = request.getRequestDispatcher("/s03/checker.jsp");
rd.forward(request, response);
```

comunicazione via attributi nella request

scriptlet?

```
<%
    Set<Character> set = (Set<Character>)request.getAttribute("set");
    for(Character c: set) {
        out.print(" " + c);
    }
 %>
```

Direttiva include

```
<%@include file="backHome.html"%>
```

Servlet e parametri

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

Forward e redirect

- Per passare il controllo da servlet/JSP corrente ad altra risorsa
  - forward()
    - Mantiene la request originale
    - Nuova URL non notificata al client
      - relativa alla web app corrente
  - sendRedirect()
    - Crea una nuova request
    - Nuova URL notificata al client
      - non relativa alla web app corrente

```
request.getRequestDispatcher(destination).forward(request, response);
```

```
<jsp:forward page="/"></jsp:forward>
```

JSP action element

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

JSTL

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

JSP Expression Language

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

EL su mappe o JavaBean
${ (oggetto implicito EL | attributo)*[.attributo | proprietà]* … **}**
Uso alternativo via sintassi ["…"] più flessibile

Alcuni oggetti impliciti EL

```
<p>Doc title: ${doc.title} <%-- ${doc["title"]} –%></p>
<p>Doc user: ${doc.user.name}</p>
<p>Doc title again: ${requestScope.doc.title}</p>
```

```
pageScope
requestScope
sessionScope
applicationScope
```

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

Parametri della request

```
param
paramValues
```

```
<p>${pageContext.request.method}</p>
```

```
pageContext
```

JSTL: JSP Standard Tag Library

- Librerie di funzionalità implementate come elementi custom utilizzabili in pagine JSP
- 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)
  - …

JSTL condizioni

- JSTL core **if**, verifica dell'EL nell'attributo **test**
  - Vero: il contenuto dell'elemento è inserito nell'HTML

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

- JSTL core **choose**, **when**, **otherwise**
  - Simulano uno switch-case
  - Il choose fa da contenitore per i when e un otherwise (opzionale)
  - Si verificano le condizioni nell'attributo test di ogni when
    - Il contenuto del primo when "vero" viene inserito nell'HTML
    - Altrimenti, si usa il contenuto dell'otherwise

```
<c:choose>
   <c:when test="${param.x != null}">
      <p>Parameter x is ${param.x}</p>
   </c:when>
   <!-- -->
   <c:otherwise>
      <p>No x nor y among parameters!</p>
   </c:otherwise>
</c:choose>
```

JSTL loop

- JSTL core **forEach**, loop su iterabile (array, collection, …)
  - Attributo **items**, via EL accede all'iterabile su cui si vuole operare
  - Attributo **var**, definisce la variabile di loop che assume di volta in volta ogni valore dell'iterabile
- JSTL core **forTokens**, loop su token all'interno di una stringa (ad es., utile per CSV)

```
User[] users = new User[] { /* ... */ };
request.setAttribute("users", users);

// ...

… forward(request, response);
```

servlet

Crea un iterabile
Lo associa a un attributo nella request
Passa il controllo alla JSP

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

JSP

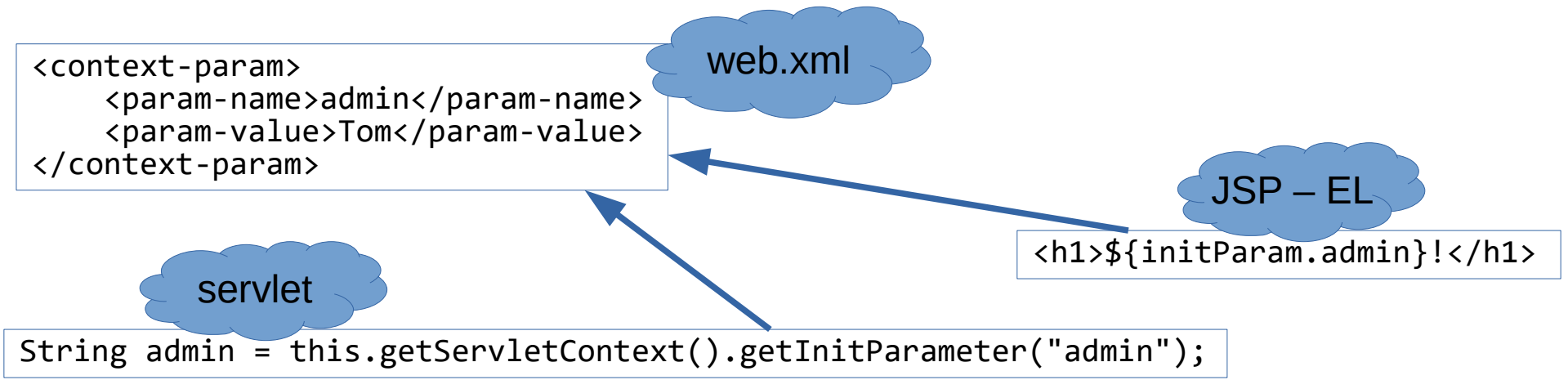
Via forEach + EL accede all'iterabile
Una componente alla volta

Session

- Le connessioni HTTP sono stateless
  - Con **HttpSession** possiamo definire una conversazione
  - *(il meccanismo, per noi completamente trasparente, è gestito via cookie JSESSIONID)*
- La sessione è gestita dal container
  - La accediamo via HttpServletRequest.**getSession**()
- Lo stato di una conversazione è determinato dagli attributi della sessione
  - HttpSession.**setAttribute**(name, value): il nome è la chiave usata per identificare l'oggetto
  - HttpSession.**getAttribute**(name): ritorna l'oggetto nella sessione con quel nome (o null)
- Per terminare una sessione usiamo HttpSession.**invalidate**()

context-param

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



Pagine di errore

- Mapping tra tipo di errore e pagina associata
  - Definito in web.xml

```
<error-page>
    <error-code>404</error-code>
    <location>/s12/404.jsp</location>
</error-page>
<error-page>
    <error-code>500</error-code>
    <location>/s12/500.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"]}
```

"class" non è utilizzabile in EL
Come proprietà (keyword java)
Uso della sintassi da array associativo

DataSource per Tomcat

- JAR del driver JDBC nella directory lib di Tomcat
- Si definisce
  - Una risorsa nel conf/context.xml di Tomcat
    - Eclipse: Project Explorer – Servers
  - Un riferimento alla risorsa nel WEB-INF/web.xml
    - Tra le “deployed resources” della web app
- Il data source è utilizzabile nella web app via
  - Servlet, Resource Injection su javax.sql.DataSource
  - JSP, JSTL taglib sql, attributo dataSource

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

```
@Resource(name = "jdbc/me")
private DataSource ds;

… = ds.getConnection();
```

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

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

MySQL serverTimezone=…

```
<Resource name="jdbc/me" type="javax.sql.DataSource" driverClassName="oracle.jdbc.OracleDriver"
auth="Container" url="jdbc:oracle:thin:@127.0.0.1:1521/xe" username="me" password="password" />
```

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>
```

```
<?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>
```

```
<datasource jndi-name="java:jboss/datasources/meDS" pool-name="meDS"
 enabled="true" use-java-context="true">
   <connection-url>jdbc:mysql://localhost:3306/me</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>
```

MySQL serverTimezone=…

oracle.jdbc.xa.client.OracleXADataSource

```
<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
- Accesso agli attributi nel servlet scope
  - JSP EL: via application scope

```
${applicationScope.start}
```

  - Servlet: accesso via servlet context

```
getServletContext().getAttribute("start");
```

Filter

- Applicato alla request prima e dopo l'effettiva gestione della risorsa indicata
  - In ingresso: log, controlli di sicurezza, ridirezione, cambiamenti in header e payload
  - In uscita: modifica della response generata – più complesso, non entriamo nei dettagli
  - Più filtri possono essere applicati alla stessa risorsa
- **Filter** interface, dobbiamo definire solo il metodo **doFilter**()
  - ServletRequest, ServletResponse (nota: non HttpServlet…)
  - **FilterChain**, reference al prossimo filtro, o alla risorsa indicata per la request
- Annotazione @**WebFilter**
  - **urlPatters**: Indicano mapping o match esatti
    - Path matching se finisce con "**/***", extension matching se inizia con "***.**"
  - **dispatcherTypes**, default è request, tra le altre possibilità forward e error