Java EE

- Apache Tomcat
  - Servlet, JSP, EL
- Integrazione con DBMS via JDBC
- Esempio
  - https://github.com/egalli64/mdwat
    - Maven
    - Tomcat 9

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 JEE sono eseguite da un reference runtime (application server o microservice)

come tomcat (server virtuale)

fin'ora abbiamo usato tomcat sottostimandolo, in realtà può implementare anche la logica scritta in codice java

scrivere un'applicazione con JEE significa che viene eseguita da un reference runtime

applicativo che supporta lo sviluppo di applicazioni web.

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
- Gestisce il ciclo di vita delle servlet, multithreading, sicurezza, …
- Dalla shell, folder bin, eseguire lo script di startup (set JAVA_HOME)

nella directory bin, ci sono gli eseguibili startup e lo shutdown

esegue lo stesso codice contemporaneamente da più thread di esecuzione (es. smartphone con 8 core può eseguire 8 thread di codice insieme o il sito di amazon che gestisce l'acquisto da parte di più clienti)

Eclipse per Java EE

- Plug-in della Web Tools Platform
  - Eclipse Java EE Developer Tools
  - Eclipse JST Server Adapters
  - Eclipse Web Developer Tools
  - Maven (Java EE) integration for Eclipse WTP
  - HTML Editor
  - Eclipse XML Editors and Tools
- Java EE Perspective

Eclipse Dynamic Web Project

- Approccio nativo Eclipse, alternativo a Maven

rende il progetto portabile su più IDE

- Principali setting nel wizard
  - Target runtime: Apache Tomcat 9
    - 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

parametri sono indicati esplicitamente nell'URL (indirizzo risorsa?parametri) parametri sono solo di tipo string

parametri all'interno del payload e non nell'indirizzo URL

in sostanza fanno la stessa cosa in due modi diversi. si aggiungono in una pagina java o in una HTML. tipicamente li si integrano, ma teoricamente li si potrebbero usare alternativamente

riceve la request e manda la response. ogni servlet è legata a un certo indirizzo. una volta arrivata la request, tomcat va a vedere chi gestisce quell'indirizzo e rigira la richiesta alla servlet relativa. la servlet gestisce la richiesta in base a se sia una get o una post

Servlet vs JSP

java server page

- Servlet: Java puro (HTML visto come testo)

unica classe che lavora su protocollo HTTP

extends HttpServlet

scritto in iava

@WebServlet("/s07/timer")

classe che funziona come syso (sto scrivendo dentro la response), consente di scrivere in un file

doGet()

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

- JSP: HTML con dei frammenti Java al suo interno

scriptlet

scritto in HTML

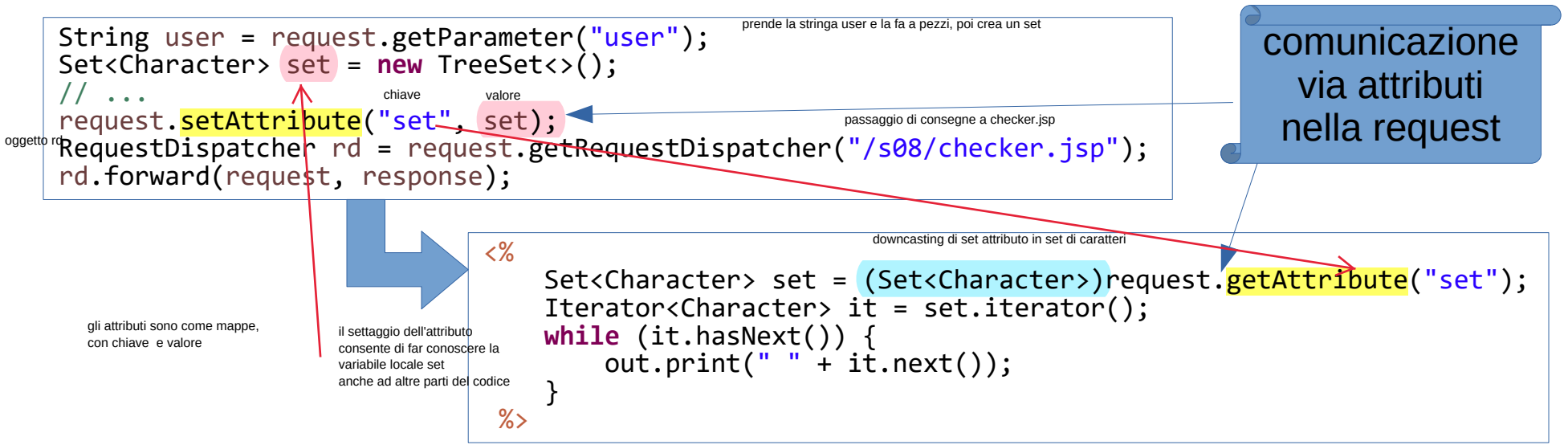
```
<h1>
    <%
        out.print(LocalTime.now());
    %>
</h1>
```

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

servlet e jsp si dividono i compiti: la servlet è orientata più alla logica, il jsp alla view. il modo per farli comunicare sono gli ATTRIBUTI, che sono gestiti come una mappa nella request. tramite un attributo, la servlet mette nella request il risultato della sua "ricerca", ma non è ancora response. ci pensa il jsp a ordinare i dati (messi nella request tramite attributo) in un file html

Servlet e JSP

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



ogni volta che faccio una get è sempre la prima volta.
i cookie (file con una durata) sono un metodo per mantenere delle informazioni.
session è una classe basata sul cookie e serve per tenere traccia della conversazione tra server e l'utente
nella session posso mettere informazioni, ad esempio, sul login e sul carrello (es. amazon)

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
}

// …
```

chiave start, value non esplicitato

oggetto attribute (nella slide prec salvato nella request, qui salvato nella session

qui mi ritorna un null (cioè dentro la variabile start mi mette un null)

Elementi JSP

direttiva — ordini che do a tomcat su come trattare la pagina

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

commento — commento jsp visibile solo al programmatore

dichiarazione — dichiaro variabile proprietà in una servlet (non si dovrebbe fare mai)

scriptlet — modo per scrivere del codice java in html (ora non si fa più)

espressione

Proprietà (**?!**) e metodi

Nel body del metodo che implementa JspPage._jspService()

metodo

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

visto poco

javabean=classe con dei dati dentro, ha proprietà private con un getter e un setter per accedervi e un costruttore di default (es. : public class User {

```
private String name;
  private int id;
   public User(String name,
int id) {
      this.name = name;
      this.id = id;
   }
   public User() {
   }
   public String getName() {
      return name;
   }
   public void setName(String
name) {
      this.name = name;)
```

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()%>
```

variabili vivono in questi scope:

```
page
request
session
application
```

JSP action element

non useremo scrivere così

```
<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

visto poco

JSP useBean /2

come accedere ai parametri dalla jsp

```
http:// … /s12/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

codice su lato servlet che mette nella request un doc con queste info:

oggetto bean composto da un titolo (cheatsheet) e un utente associato (tom)

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

JavaBean aggregato come attributo nella request da servlet a JSP

${ è per scrivere del codice java in foglio jsp

jsp, per stampare, avrà scritto:

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

stamperà il titolo del doc ($ ="vai a prendere")

oggetti impliciti EL per gli scope

tomcat cerca il doc qui: (e alla fine lo trova in requestscope)

```
pageScope
requestScope
sessionScope
applicationScope
```

richiesta generata da un form compilato da un utente

tratta questi due parametri come un array

```
http:// … /s13/direct.jsp?x=42&y=a&y=b
```

Chiamata diretta a un JSP

oggetti impliciti EL per i parametri

get parametro

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

lo usiamo in jsp

visto poco

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 → null

se voglio prendere un solo parametro, es. il primo nome passato dall'utente nel form (es. checklist flaggata, prende il primo parametro flaggato)

lo usiamo in una servlet

mi ritorna tutti i valori passati dall'utente con quel parametro (es. tutti i nomi di cantanti scritti nel form dall'utente)

se metto un nome e l'utente non mi ha passato un parametro con quel nome=null

per passare dalla servlet (logica) alla jsp (attività di visualizzazione)

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

per fare un forward (tienilo a mente perchè servirà tanto per il progetto)

quando faccio un forward sto lavorando sulla mia app

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

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

JSP action element

quando faccio redirect sto lavorando

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

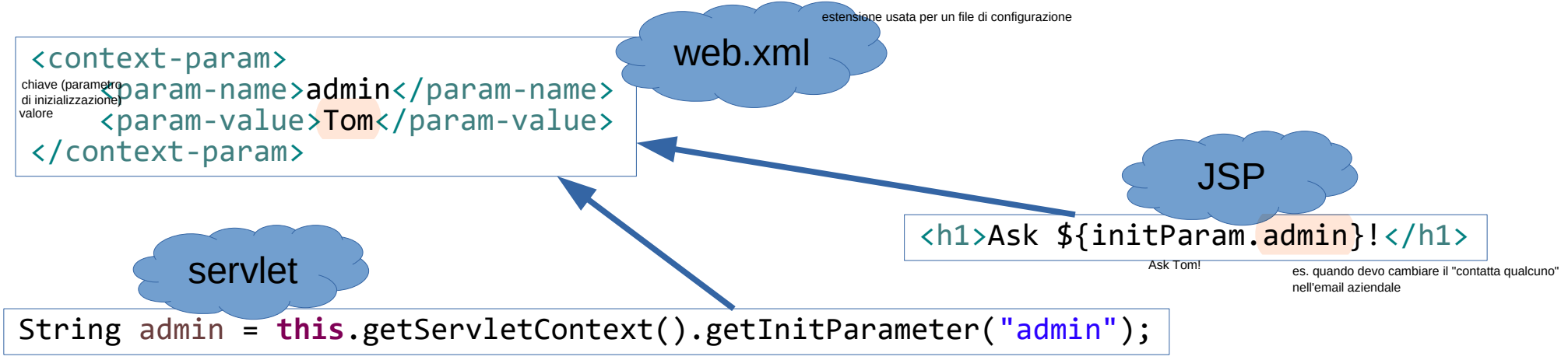
JSTL

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

nel file di configurazione posso mettere delle variabili visibili in tutta la webapp

context-param

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



in un sito web posso metterci una pagina di errore? sì, la error page devo defininirla nel file di configurazione .xml per dire a tomcat di far vedere la pagina di errore creata da me, programmatore e non quella di default del browser

Pagine di errore

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

```
<error-page>
    <error-code>404</error-code>
    <location>/s17/404.jsp</location>
</error-page>
<error-page>
    <error-code>500</error-code>
    <location>/s17/500.jsp</location>
</error-page>
```

Page not found

Internal error

quando c'è un 404 usa quetsa pagina e non lo standard del browser (la semplice scritta 404 not found)

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

Oggetto implicito EL

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

per fare i cicli logici devo usare java (cioè scrivere una scriptlet), ma sarebbe una porcheria, per cui non si usa java ma JSTL

<artifactId>jstl</artifactId> nel POM, quindi posso usare in jsp la libreria jstl

JSTL: JSP Standard Tag Library

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

in una pagina jsp posso accedere a un database, anche se tipicamente uso il DAO per accedere a un database

questopezzo si cerca su internet

tag name

è come se facessi un importing

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

qui siamo in html, quindi l'attributo va messo tra doppi apici, ma qui dentro c'è del codice java perchè comincia con ${

if fa parte del blocco core

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

se questo è vero (se cioè c'è un parametro x)

allora questo x viene inserito nell'html (se non ci fosse non scriverei questa riga, è come quando cerchiamo tutta la roba inerente a barry white ma non ci sono film, per cui il box "filmografia" non lo faccio comparire proprio, non è che lo faccio comparire-ma vuoto

con questo posso decidere, quando scrivo delle tabelle in html con il risultato di una ricerca, di non far comparire per niente il box se è vuoto (no risultati) o di farlo comparire, ma con l'avviso "non ho trovato nulla del genere"

se aggiungo una tabella, qui avrò una tabella con user associati ad id. questo è più comodo se devo fare una tabella con tanti elementi, facendo il for each e mettendolo dentro una tabella (cioè con <tr><td> al posto di <p>). la stessa cosa si può fare con le immagini.

JSTL core loop

- Su un iterable c:forEach, su stringa tokenizzata c:forTokens

```
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

variabile di loop=user voglio looppare su questo

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

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

```
<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

abbiamo visto come applicativo da J SE, ora anche su EE

libreria per gestire scambio di dati

JDBC

- Nel POM, dipendenza dal JDBC scelto (oppure: drive condiviso in tomcat lib)
  - Es: mysql (groupId), mysql-connector-java (artifactId)
- Il caricamento del driver JDBC può essere fatto allo startup della webapp
  - Es: Class.forName("com.mysql.cj.jdbc.Driver");
- Definizione di una risorsa in context.xml nelle META-INF della webapp, o nel folder Servers, Tomcat
- In web.xml, nel WEB-INF del progetto, aggiungere una reference alla resource
- Ora il data source è utilizzabile da servlet e JSP

```
<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>
```

metto codice sql nell'html

anche in jee è possibile gestire un flusso di eventi, come ad esmepio il ciclo di vita delle servlet. la creazione di una servlet è un evento in sè.

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

inizializzo servlet

cancello servlet

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

un pò come il discorso sulla sicurezza all'inizio delle slide.
usiamo i filter come servlet ma non per gestire la request, ma per fare altre azioni, come il log, e poi passano
simile al servlet, perchè riceve una request, ma riceve anche informazioni sul filtro a cui deve passare il controllo dopo.
una volta arrivato al filtro, l'utente riceve come response una diversa da quella che aveva chiesto (es. voglio accedere
alla mia posta elettronica, ma ancora non mi sono loggato, quindi mi ridirige verso la pagina di log in)

Filter

- In ingresso: audit, log, security check
- 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]