Programmazione Web

Lez. 14

Servlet Java - Parte 2

Giuseppe Psaila

Università di Bergamo

giuseppe.psaila@unibg.it

Parametri di Configurazione (Contesto)

Parametrizzare le Servlet

- Un tipico errore di programmazione consiste nel mettere valori di configurazione nel codice.
- Negli esempi della volta scorasa, abbiamo messo la stringa di connessione nel codice.
- Un buon design prevede che questi valori vengano **messi al di fuori**.
- Occorre **parametrizzare l'applicazione**.

Contesto dell'Applicazione

- Nel file **`web.xml`**
- si può definire il «**Contesto dell'Applicazione**».
- Il contesto è **l'insieme dei parametri di configurazione** dell'applicazione.
- Le servlet dell'applicazione devono acquisire esplicitamente i parametri dal contesto.

Definire un Parametro

- Per definire un parametro, nel file **`web.xml`**, si può usare l'elemento XML seguente:

```
<context-param>
  <param-name>parametername</param-name>
  <param-value>parametervalue</param-value>
</context-param>
```

Definire un Parametro

- Definiamo la stringa di connessione al DB

```
<context-param>
   <param-name>DBConnString</param-name>
   <param-value>
jdbc:postgresql://localhost:5432/MyDB?user
=MyUser&amp;amp;password=MyPwd</param-value>
</context-param>
```

Attenzione

- In XML, il carattere «**&**» è riservato:
- serve per definire le «entity».
- A noi serve mettere un «**&**» per separare i parametri della connessione, quindi dobbiamo usare l'entità «**&**»

```
?user=MyUser&amp;password=MyPw
```

Acquisizione del Contesto

- Il codice della servlet deve acquisire i parametri di interesse.
- Per prima cosa, si deve acquisire il contesto.

```
ServletContext context=
                    getServletContext();
```

Acquisizione dei Parametri

- A questo punto, si può acquisire il parametro

```
String DBConnString=
      context.getInitParameter(
            "DBConnString");
```

Gestire le Sessioni

HTTP è Stateless

- Il protocollo HTTP nasce «**stateless**», cioè «**senza stato**».
- Infatti, ogni singola chiamata HTTTP dovrebbe essere indipendente dalle altre.
- Ma ci sono applicazioni in cui questa situazione non è adeguata.

Esempi di Inadeguatezza

Quando l’approccio stateless non è adeguato?

- Per gestire funzionalità specifiche per gli utenti (login degli utenti).
- Per gestire le procedure composte da più passi.
- Per gestire il «carrello della spesa» (o «shopping cart»).

Qual è il Problema?

Il problema dato dall’approccio «stateless» è:

- Manca la certezza che le chiamate HTTP vengano dallo stesso utente o che riguardino la stessa procedura.

Soluzione: le Sessioni

- Una «Sessione» può essere vista come un’insieme di variabili.
- Che sopravvivono tra una chiamata HTTP e l’altra,
- in modo che il server-side script possa recuperarle e gestire il processo.

Idea: i Cookie

- I «cookie» sono delle variabili che vengono inviate dal server attraverso il protocollo HTTP.
- Il browser li riceve e li ritrasmette nella successiva richiesta HTTP che viene inviata dalla stessa pagina.

Il SessionId come Cookie

- La sessione viene gestita dal server.
- Quindi non viene inviato al client l’intero stato della sessione, ma solo il suo SessionId.
- In questo modo, il server può gestire anche quanto tempo la sessione può/deve rimanere attiva.

Sessioni nelle Servlet

- Oggetto di tipo **`HttpSession`**.
- L'oggetto sessione è associato alla richiesta

```
HttpSession session =
        request.getSession(true);
```

- Il parametro **`true`** dice di creare un nuovo oggetto di sessione, se questo non viene trovatonella richiesta.

Sessioni nelle Servlet

- `HttpSession session = request.getSession(false);`

- Se la sessione non è stata precedentemente creata, si ottiene **`null`**.

Scoprire se la Sessione è Nuova

- È possibile sapere se una sessione è stata appena creata,
- con il metodo **`session.isNew()`**
- Che restituisce un **`Boolean`**.

Attributo di Sessione

- Un attributo di sessione può essere un qualsiasi oggetto Java.
- Per elaborare la richiesta, occorre recuperare il suo valore dalla sessione.
- Il metodo **`getAttribute`** di **`HttpSession`** recupera l'attributo
  `Object session.getAttribute(`name**`)`**

Esempio: Shopping Cart

```
ShoppingCart items =
    (ShoppingCart)
       session.getAttribute("items");
if (items != null) {
        doSomethingWith(items);
} else {
 items = new ShoppingCart(...);
 doSomethingElseWith(items);
}
```

Spiegazione

- Viene usata una classe **`ShoppingCart`**, definita nella applicazione.
- Dalla sessione, si estrae l'oggetto, denominato **`items`**.
- Se la sessione è stata appena creata, quell'oggetto non esiste, quindi va creato;
- se invece esiste, veien manipolato.

Impostare un Attributo

- Il metodo **`setAttribute`** di **`HttpSession`** imposta (o sostituisce) il valore di un attributo

```
session.setAttribute("items", items);
```

La Sessione nella Risposta?

- La sessione viene inviata automaticamente nella risposta
- O meglio, viene inviato un coockie con l’id della sessione.

Chiudere la Sessione

- Per chiuedere una sessione, occorre «**invalidarla**»
- Da quel momento in poi, non potrà essere più usata e la succesiva invocazione della servlet non avrà l'oggetto sessione associato.

```
session.invalidate();
```

Pattern MVC

Model-View-Controller

- È un pattern architetturale, inventato molto tempo fa, prima della nascita del World-Wide Web.
- Tuttavia, ha trovato applicazione naturale in questo contesto.

Model-View-COntroller

- **Model**
  L'insieme dei dati sui quali si deve operare, cioè la base dati dell'applicazione
- **View**
  La vista, per l'utente, dei dati e delle funzionalità; in altre parole, è l'interfaccia utente.
- **Controller**
  Il gestore delle azioni e delle trasformazioni da operare sui dati

Obiettivo Fondamentale

- Disaccoppiare l'elaborazione dalla visualizzazione.
- In modo da rendere indipendente (il piùù possibile) la visualizzazione dall'attività di elaborazione.
- Vi sono moltissime proposte per realizzare il pattern MVC, ciascuna con le sue limitazioni.
- Certamente, per ottenere una buona soluzione MVC per il web, occorre disaccoppiare il codice procedurale dell'elaborazione dalla generazione del codice HTML.

Limiti dei Server-Side Script stile PHP/JSP Semplice

- Il codice procedurale annegato nella pagina HTML è strettamente accoppiato alla pagina stessa.
- È certamente meglio delle servlet che generano il codice HTML direttamente, ma il pattern MVC è lontano.

ThymeLeaf

- È una proposta molto interessante, adatta ad introdurre il pattern MVC nelle servlet in modo efficace.
- Infatti, consente di mettere la pagina HTML al di fuori della servlet,
- ma il controllo della generazione della pagina finale rimane in carico alla servlet, che prepara un model dedicato.

ThymeLeaf

Sito Ufficiale

- Home Page
  `https://www.thymeleaf.org/`
- Distribuito su GittHub
  `https://github.com/thymeleaf/thymeleaf/releases/`

Librerie

Nella cartella **`lib`** della web app, copiare:

- tutti i file **`jar`** contenuti nella cartella **`lib`** del file **`zip`**
- i seguenti file contenuti nella cartella **`dist`**:

`thymeleaf-3.1.2.RELEASE.jar`
`thymeleaf-extras-springsecurity5-3.1.2.RELEASE.jar`
`thymeleaf-extras-springsecurity6-3.1.2.RELEASE.jar`
`thymeleaf-spring5-3.1.2.RELEASE.jar`

TemplateEngine

- Il cuore di ThymeLeaf è chiamato «**Template Engine**»
- Esso fornisce (tra gli altri) un metodo «**`process`**» che elabora un «**template**», usando delle «variabili di contesto», per generare l'output finale.

Variabili di Contesto

- Un «contesto» è un insieme di variabili (oggetti Java) che servono come input per elaborare il template.
- Occorre creare un nuovo contesto prima di passarlo al metodo **`process`** del **`TemplateEngine`**.

Creazione del Contesto

```
Context context = new Context();
context.setVariable("name", "John");
```

- Il contesto è inizialmente vuoto.
- Viene aggiunta la variabile «**`name`**».

Che Cosa è un Template

- Un «template» è una pagina HTML (o un documento XML)
- che contiene specifici comandi di ThymeLeaf,
- I comandi ThymeLeaf sono attributi con prefisso «**`th:`**», definito su uno specifico namespace.

Esempio di Template

```
<!DOCTYPE HTML>
<html>
<head>
 <meta charset="UTF-8">
 <title>ThymeLeaf Examples</title>
</head>
<body>
<p xmlns:th="http://www.thymeleaf.org"
    th:text="${'Hello ' + name}"></p>
</p></body></html>
```

Esempio di Output

```
<!DOCTYPE HTML>
<html>
<head>
 <meta charset="UTF-8">
 <title>ThymeLeaf Examples</title>
</head>
<body>
<p>Hello John</p>
</body></html>
```

Package da Importare

```
import org.thymeleaf.*;
import org.thymeleaf.context.*;
import org.thymeleaf.templateresolver.*;
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.TemplateSpec;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import
org.thymeleaf.templateresolver.ClassLoaderTemplat
eResolver;
```

Creare il TemplateEngine

- Il processo di creazione del **`TemplateEngine`** non è banalissimo.
- Perché dopo aver creato l'oggetto,
- occorre creare un «**`TemplateResolver`**»
- il cui scopo è quello di cercare il file con il template.

Creare il TemplateEngine

```
TemplateEngine templateEngine =
    new TemplateEngine();
ClassLoaderTemplateResolver templateResolver =
    new ClassLoaderTemplateResolver(Thread
.currentThread().getContextClassLoader());
```

Spiegazione

Il **`TemplateResolver`** ha il compito di:

- recuperare il file con il template;
- gestire il particolare formato.

- Il **`TemplateResolver`** deve essere passato al **`TemplateEngine`**.

Creare il TemplateEngine

```
templateResolver.setTemplateMode(
                              TemplateMode.HTML);
templateResolver.setPrefix("/ThTemplates/");
templateEngine.setTemplateResolver(
                              templateResolver);
```

Spiegazione

- Per prima cosa viene specificato il formato in input (HTML).
- Si imposta il prefisso del perocrso, a partire dalla cartella **`classes`**.
- Si imposta il resolver per il **`TemplateEngine`**.

Elaborazione del Template

Siamo finalmente pronti ad elaborare il tempate.
I parametri del metodo **`process`** sono:

- Il nome del file con il template
- Il contesto
- Il **`PrintWriter`** sul quale inviare l'output.

```
templateEngine.process("Hello.html",
                       context, out);
```

Esempio Complesso: Nominativi

Riprendiamo l’esempio dei Nominativi.

- Usiamo la stessa struttura dati che abbiamo usato per serializzare i risultati come documento JSON.
- Invece di serializzarla, la mettiamo nel contesto di ThymeLeaf
- e la usiamo per elaborare un template.

Struttura Dati

```
public class Nominativo
{
    public String Name;
    public int Age;

    public Nominativo() { }
    public Nominativo(String n, int a) {…}
}
```

Struttura Dati

```
public class Result
{
    public int elementi=0;
    public ArrayList<Nominativo> list;

    public Result() {… }
    public void append(String Name, int Age)
    {…}
}
```

Prepariamo il Contesto

- La nuova servlet è un po’ più complicata, ma non troppo.
- La query e il riempimento della struttura dati li prendiamo dall’esempio precedente.
- La struttura dati viene messa nel contesto
- Generiamo la pagina HTML dal template.

Connessione al DB

```
Result r = new Result();

Try {
 Class.forName("org.postgresql.Driver");
 String url =
"jdbc:postgresql://localhost:5432/MyDB?user=MyU
ser&password=MyPwd";
 Connection conn =
    DriverManager.getConnection(url);
 Statement stmt = conn.createStatement();
```

Query

```
String q =
   "SELECT \"Name\", \"Age\" FROM \"Names\";";
ResultSet rs = stmt.executeQuery( q );

while ( rs.next() )
{
 String  name = rs.getString("Name");
 int age  = rs.getInt("Age");
 r.append(name, age);
}
```

TemplateEngine

```
TemplateEngine templateEngine =
  new TemplateEngine();
ClassLoaderTemplateResolver templateResolver
 = new ClassLoaderTemplateResolver(Thread
   .currentThread().getContextClassLoader());
templateResolver.setTemplateMode(TemplateMode.HTML);
templateResolver.setPrefix("/ThTemplates/");
templateEngine.setTemplateResolver(templateResolver);
```

Contesto e Template

```
Context context = new Context();
context.setVariable("res", r);

templateEngine.process("Nominativi.html",
                       context, out);
```

Il Template

Il template deve:

- creare una tabella con i nominativi, se vi sono dei nominativi;
- scrivere che non ci sono nominativi, se non ve ne sono.

Approccio di ThymeLeaf

- Si usano degli attributi XML aggiuntivi definiti su un namespace con prefisso **`th:`**
- Non vengono definiti elementi, solo attributi.

Approccio di ThymeLeaf

- L'attributo **`th:if`**
  consente di condizionare la generazione dell'elemento al quale appartiene
- L'attributo **`th:each`**
  consente di generare occorrenze multiple dell'elemento cui appartiene.

Cuore del Template

```
<div th:if="${res.elementi == 0}">
<p>Nessun nominativo trovato</p>
</div>

<table th:if="${res.elementi > 0}">
<tr th:each="item: ${res.list}">
<td th:text="${item.Name}"></td>
<td th:text="${item.Age}"></td>
</tr>
</table>
```

Espressioni di ThymeLeaf

- Il valore di un attributo con prefisso **`th:`** deve essere un'espressione che il template engine deve valutare.
- Per essere valutate, le espressioni vanno all'interno di **`${…}`**
- Nel caso di **`th:if`**, il valore deve essere un Booleano.

```
th:if="${res.elementi == 0}"
```

Variabile dal contesto

```
th:if="${res.elementi == 0}"
```

- **`res`** è la variabile nel contesto, con la struttura dati preparata dalla servlet.
- Il confronto produce un valore Booleano.

Iteratori

- L'attributo **`th:each`** serve per far generare copie multiple dell'elemento cui appartiene.
- Si definisce un iteratore, che scandisce una lista.

```
<tr th:each="item: ${res.list}">
```

- **`item`** è un iteratore, che scorre la lista prodotta dall'espressione.
- Vengono generate tante copie di **`tr`** quanti soo gli elementi nella lista.

Contenuto di un Elemento

- L’attributo **`th:text`** genera il contenuto dell’elemento cui appartiene.

```
<td th:text="${item.Name}"></td>
```

Output

```
<table>
<tr>
<td>Pippo</td>
<td>30</td>
</tr>
<tr>
<td>Pluto</td>
<td>20</td>
</tr>
…
</table>
```

Output

Nominativi

Pippo	30
Pluto	20
Paperino	50
Topolino	50

Riassumendo su ThymeLeaf

- Rispetto ad altri approcci di tipo MVC
- ThymeLeaf è una soluzione molto interessante,
- perché è estremamente snella
- e disaccoppia moltissimo il view dal controller.
- Lo stesso template può essere utilizzato da diverse servlet.
- Inoltre, nel template si vede chiaramente la struttura dell'HTML da generare.

Considerazioni Finali

Che Cosa non Vediamo

- Non vediamo JSP, ma nella prossima slide diremo due parole.
- Non vediamo MongoDB, ma se serve connettersi da Java, occorre ceare un oggetto **`MongoClient`** e da lì si procede come abbiamo già visto.

JSP

Java Server Pages

- Nasce come server-side script in senso stretto, cioè codice HTML con annegate parti procedurali (alla PHP).
- Quando viene invocato la prima volta, un file JSP viene tradotto in una servlet, subito compilata ed eseguita.
- Alla succesiva invocazione, viene eseguita direttamente la servlet.

JSP

JSP Avanzato

- Sono state create «**librerie di tag**», dove la definizione dei tag è basata su codice Jva.
- L’obiettivo è di non usare mail il codice Java, lavorando direttamente sul view tramite i tag.
- Il problema è che view e controller non sono disaccoppiati, ma fortemente accoppiati.

Servlet.zip

- In questo file trovate tutti gli esempi che abbiao discusso nelle slide.
- Non riusciremo a fare una esercitazione dedicata, quindi vi conviene provare da soli, perché le servlet potrebbero essere oggetto del secondo progetto.