JavaServer Pages tutorial

Rui Couto José C. Campos

Sistemas Interactivos
Mestrado Integrado em Engenharia Informática
DI/UMinho
April 27, 2021

Contents

1 Introduction

So far, the tutorials have been focussed on client-side programming. Changing the focus from client-side to server-side Web development, this tutorial presents the implementation of a JavaServer Pages (JSP)[1] page resorting to JSTL[2], the JSP Standard Tag Library. The page will be integrated as part of a Java-based Web application.

JSP technology enables the dynamic creation of Web pages on the server. It is similar to PHP, ASP and React's JSX, but builds on the Java programming language and ecosystem. JSP is not a development framework. Frameworks that support JSP include Spring[3], Apache Struts[4] or JavaServer Faces (JSF)[5]. Here we will focus on the JSP language and how it can be used to develop the Web pages of the user interface layer.

JSTL is a collection of tag libraries that implement general-purpose functionalities common to many Web applications. To deploy and run JSP-based Web applications, a compatible Web server with a Java Servlet container, such as Apache Tomcat[6], is required.

2 Base project

A base project is provided to support the tutorial. It contains what is required to start the development of a functionality of the *Game Management System* used as a running example in the previous tutorials: the list of games. The business layer implementation needed for listing the games is provided; a Servlet, implemented according to the MVC pattern, is also provided; as is a JSP page where the list of games will be displayed. Changes should be performed in both the Servlet and the JSP page.

Two version of the project are available. The relevant files are the same, but they were created with different IDEs. One with NetBeans 8.2, the other with

[1] `http://www.oracle.com/technetwork/java/javaee/jsp/`
[2] `http://www.oracle.com/technetwork/java/jstl-137486.html`
[3] `https://spring.io`
[4] `https://struts.apache.org`
[5] `http://www.javaserverfaces.org/`
[6] `http://tomcat.apache.org`

IntelliJ IDEA ULTIMATE 2019.1. Other IDEs can be used, although additional configurations will be required. The code is organised as follows in IntelliJ:

- `web/WEB-INF/ListGames.jsp` - The JSP page that will display the list of games.
- `src/business` - The facade and business classes.
- `src/data` - The persistency classes (note that no actual persistency is implemented).
- `src/web/ListGames.java` - The Java Servlet that serves the JSP page above.

In NetBeans the Java source code is located in the folder `src/java`. NetBeans names "`Web Pages`" the contents of "`web`" and "`Source Packages`" the contents of "`src/java`".

3 Setup

As stated above, the base project is provided as NetBeans and IntelliJ IDEA projects. It can be directly opened in the selected IDE, and run. You might need to configure the application server. Both GlassFish[7] and JBoss[8] have been tested (in NetBeans and IntelliJ IDEA, respectively).

Once the application server is configured, running the project will launch the server and a Web browser should open automatically with URL `http://localhost:8080/GMSBase/ListGames`[9]. For now, no information about the games is displayed on the page, just a message (see Figure 1).

Tasks

1. Open the project in your IDE of choice.
2. Run the application and solve any configuration issues that might be reported.

[7] `https://javaee.github.io/glassfish/`.
[8] `https://www.jboss.org/`.
[9] Depending on configuration the URL can also be: `http://localhost:8080/GMSBase_war_exploded/ListGames`.

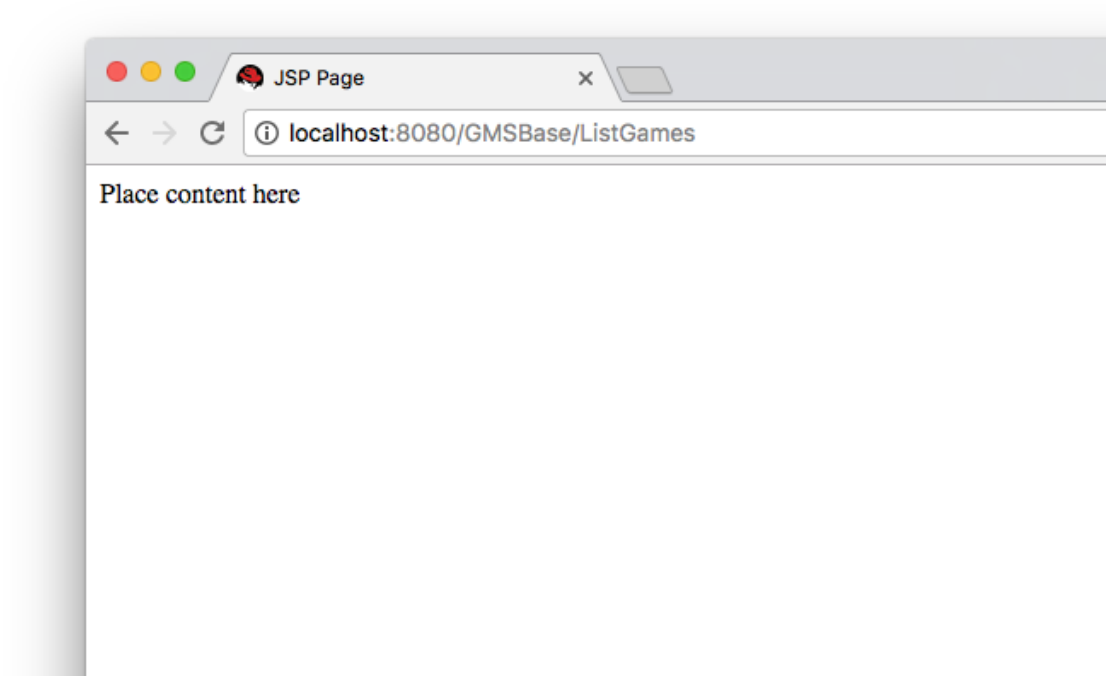

Figure 1: Base template.

3. Check that everything is working as expected by confirming that the page in Figure 1 is displayed on the Web browser.

4 Importing the application

In the previous tutorials, a Bootstrap application has been developed. To save time, we will start from that version. JSP pages are first of all HTML pages, thus the HTML and Javascript code developed for the Bootstrap tutorial can be directly imported into the JSP application. The resources should be placed in the `web` folder.

Tasks

4. Import the resources from the previously developed Bootstrap application.
5. Add to `ListGames.jsp` the content of the `index.html` page previously created to list the games.

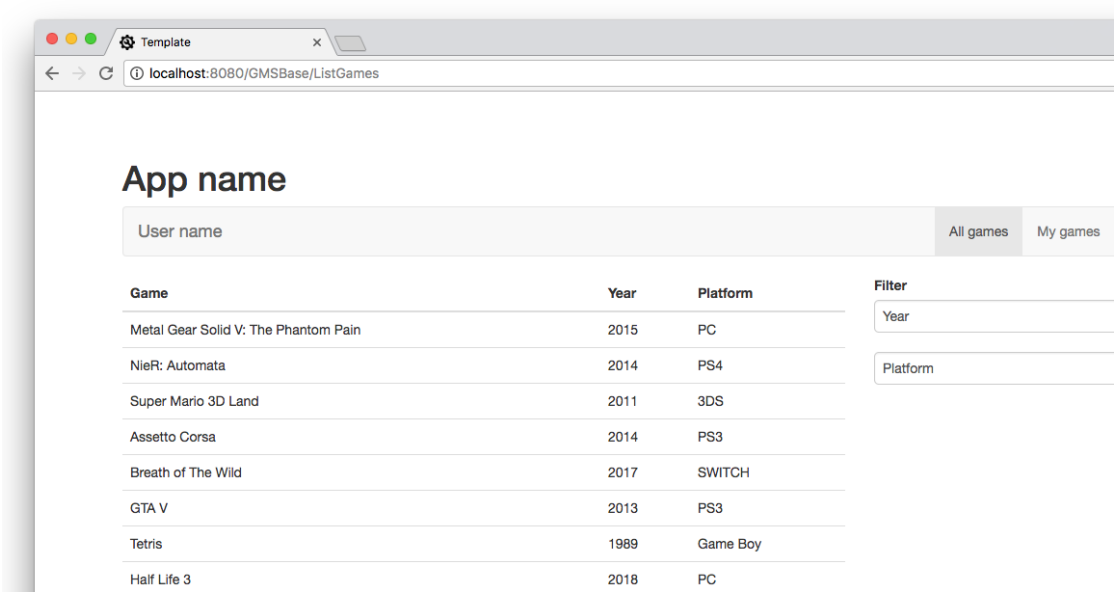

Figure 2: The list of games.

The resulting Web page should look like the one developed in the last tutorial (see Figure 2)[10]. However, at this stage, the page contents is being dynamically provided by the application server. That is, the Servlet is serving an "*empty*" page, which then fetches the list of games to display. While this illustrates that it is possible to mix server- and client-side control logic, in this particular case it makes sense to send the page already with the games' information (thus avoiding a further call to the server).

We will get back to this mix of server- and client-side logic in Section 8. For now, the next step is:

Tasks

6. Remove, from the served Web page, the Javascript event handlers that dynamically fetch the list of games, and add, using JSP, the list provided by the facade. Just remove the handlers associations to the events, do not remove the Javascript files.

[10]The exact contents of the games' list might differ.

5 Listing the games

This section shows how to use JSP to generate a Web page with data provided by the application, on the server side (instead of later fetching the data from a Web service, on the client side).

We start by declaring that we are using the JSTL library. This is achieved by adding the following JSP directive to `ListGames.jps`:

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

Use of the JSTL library enables a more declarative style of programming (as opposed to writing actual Java code on the JSP) and is the recommended approach[11].

Now, we can use the tags provided by JSTL[12] to define the contents of the page. An example is the `forEach` loop, which allows iteration over the elements of a list. Knowing that the `ListGames` servlet sets the `games` attribute of the HTTP request, to the list of (Java Beans representing) games to be displayed, we can write:

```
1 <c:forEach var="g" items="${requestScope.games}">
2     Game: ${g.name}<br/>
3 </c:forEach>
```

In this example, we are using EL[13] expressions (the syntax `${expr}`). EL expressions simplify the access to data stored in Java Bean components and other objects (request, session, application, etc.). Hence with `${requestScope.games}` we are referring to the list of games, and with `${g.name}` to the name property of a particular game.

Tasks

7. Modify the page to fill the table with the games provided by the facade.

[11] JSP itself has XML equivalents to the traditional JSP syntax. See `http://www.oracle.com/technetwork/java/syntaxref12-149806.pdf` for a JSP reference covering both styles of code.

[12] See `http://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/` for a JSTL reference.

[13] Expression Language – available with JSP 2.0.

For that you need to use a `foreach` loop to create the rows of the games' table.

8. Create also a variable in the Servlet to display the `username` in the page.

The resulting Web page should list the games provided by the application. Inspecting the HTML code it should be possible to see that the games are directly present in the HTML provided by the Web server.

While this implementation is simpler than the AJAX implementation, the page will now only be able to update the games' list when it refreshes. Hence, if the list is paginated for presentation purposes (as indicated in the mockups), navigating between groups of games implies a page refresh, and managing which group is currently being displayed needs to be managed on the server side.

Deciding which user interface control logic should go on the server side, and which user interface control logic should go on the client side is part of the applications design process. It will be influenced by considerations such as the responsiveness of the user interface, but also about the load that the user interface poses on the application/web servers.

6 Templating

As previously seen, reusing CSS provides several advantages, both during the development and the maintenance of an application. The same is true for Web pages' contents. There is usually no reason for repeating code (consider, for instance, the header of the application's pages which will be the same in all of them). In fact, in the current example, only a part of the application page (its main content) changes from page to page. Thus, it is possible to create templates, which are filled with the corresponding content. This way, elements as the footer or navigation bar, can be specified in the template, while the content is dynamically set (see Figure 6).

6.1 Creating the template

The template will consist in a Web page, in which a fragment can be rendered according to a specific request.

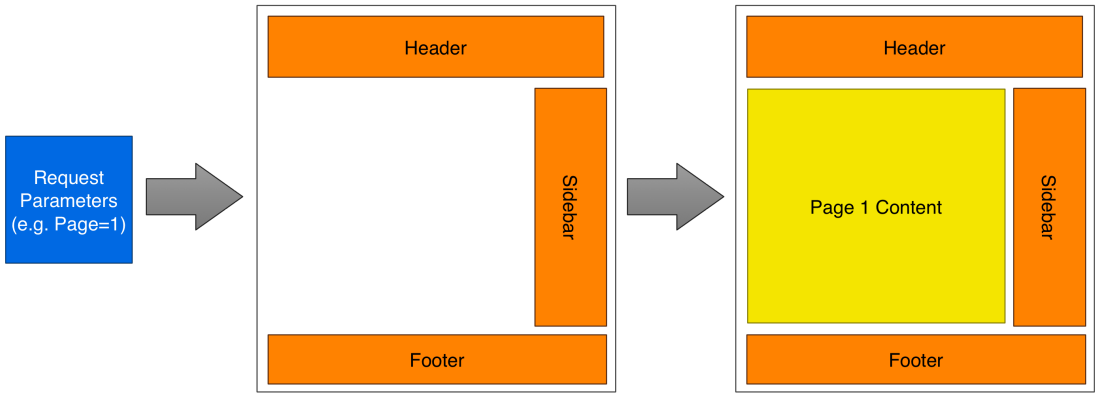

Figure 3: Parametrising a template.

An example would be as follows. The Servlet defines the page to render:

```
request.setAttribute("page","page1");
```

The main JSP page reads the page to render, and "*loads*" its contents as required[14]:

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <header>
            <h1>Header</h1>
        </header>
        <main>
            <!-- Loading the page according to the parameter -->
            <c:choose>
                <c:when test="${requestScope.page=='page1'}">
                    <jsp:include page="page1.jsp" />
                </c:when>
                <c:when test="${requestScope.page=='page2'}">
                    <jsp:include page="page2.jsp" />
```

[14] `<jsp:include page="page1.jsp"/>` compiles `page1.jsp` and includes the result in this page.

```
                    </c:when>
                </c:choose>
            </main>
            <footer>
                Footer
            </footer>
        </body>
</html>
```

Auxiliary JSP pages define the contents to display in each case. For example, for testing purposes, we could initially have in `page1.jsp`:

```
This is the content of page1.
```

Tasks

9. Create a template that renders the header, footer and navbar.
   (a) Start by creating a new JSP page where the template will be placed (e.g. `Template.jsp`).
   (b) Use the content of the `ListGames.jsp` as a basis for the template, and replace the list of games with code to dynamically load `ListGames.jsp`, based on the value of a `page` attribute in the request.
   (c) In the `ListGames` Servlet, change the redirect to `Template.jsp` instead of `ListGames.jsp`, **and** add the `page` parameter to the request, defining the page content to render.
   (d) Modify `ListGames.jsp` to have only the list of games.
10. Check that the list of games is still being displayed.

7 Data handling

Exchanging data between the browser and the application can be achieved in several ways. The most common, in the current scenario, are GET and POST

HTTP request methods[15].

7.1 GET

The GET method should be used to request resources (a Web page or, more generally, some data) from the server. GET requests should not be used for operations that cause side-effects.

With a GET request, data to send to the server, usually known as *query string*, is sent as part of the URL, which means that it becomes visible. The query string corresponds to *key-value* pairs, placed after a `?` in front of the URL, with each key-value pair separated by `&`. A pair is represented as `key=value`:

```
http://url/page?key1=value1&key2=value2&...
```

For instance, performing a search for the keywords "`Mestrado Integrado`" in `http://www.uminho.pt`, generates the following URL[16]:

```
https://www.uminho.pt/PT/pesquisa/Paginas/results.aspx?k=Mestrado%20integrado
```

7.2 POST

The POST method should be used to submit data to be processed (e.g., from an HTML form) and/or to request the execution of operations with side-effects (e.g., to update or create resources).

A POST request encapsulates the data in the HTTP request, so the data is not visible in the URL. Consider the form:

```
<form method="POST">
    Username:<input type="input" name="username"/><br/>
    Password:<input type="password" name="password"/><br/>
    <input type="submit" value="Login"/>
</form>
```

[15] For more on HTTP request methods see `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`.

[16] Note: "`%20`" is the space character.

When the user clicks in `Login`, the browser generates a POST request[17], with two variables (`username` and `password`) corresponding to the two input fields in the form. The generated HTTP request will be something like:

```
POST /Login HTTP/1.0
Accept: text/html
If-modified-since: Tue, 18 Apr 2017 14:00:00 GMT
Content-Type: application/x-www-form-urlencoded
Content-Length: 39
username=...&password=...
```

It is worth mentioning that large volumes of data (e.g. images) need to be send via POST requests, since GET requests usually have a limit of 8KB.

7.3 Accessing the data

Data from both kinds of requests is accessed the same way in the Servlet: through the `request` variable. For the first example, data can be access as:

```
String k1 = request.getParameter("key1");
```

Similarly, for the POST data:

```
String username = request.getParameter("username");
```

The `request` variable is also available at the JSP level. In fact, it is one of a set of implicit objects defined by EL which are useful in this context:

- `pageContext` — The context for the JSP page. Provides access to various objects including:
  - `servletContext` — The context for the JSP page's Servlet and any Web components contained in the same application.
  - `session` — The session object for the client.
  - `request` — The request triggering the execution of the JSP page.

[17] Note the `method` attribute in the `form` tag, which controls which type of request is generated. It can be POST or GET.

  - `response` — The response to be returned by the JSP page.

- `param` — which maps a request parameter name to a single value
- `paramValues` — which maps a request parameter name to an array of values
- `cookie` — which maps a cookie name to a single cookie
- `pageScope` — which maps page-scoped variable names to their values
- `requestScope` — which maps request-scoped variable names to their values
- `sessionScope` — which maps session-scoped variable names to their values
- `applicationScope` — which maps application-scoped variable names to their values

Hence, in JSP the above code becomes:

```
<c:set var="k1" value="${param.key1}"/>
```

and

```
<c:set var="username" value="${param.username}"/>
```

Note that the Servlet can set attributes on the `request` object using method `setAttribute()` (see `ListGames.java`). These can then be accessed in the JSP with `getAttribute()`. Hence you would write:

```
request.getAttribute("games");
```

to get the list of games on the JSP.

Using the `requestScope` implicit object of EL, the above can be simplified to `${requestScope.games}` or, further, to simply `${games}` as in:

```
<c:forEach var="g" items="${games}">
    Game: ${g.name}<br/>
</c:forEach>
```

The above works as variables are searched for from `pageScope` (the default scope) to `applicationScope`.

It is good practice to define variables with the narrowest scope possible and you should be careful not to unintentionally overlap a variable in a narrower scope.

Tasks

11. Implement the pagination feature (e.g. 5 games by page).

    (a) Start by adding some more games to `GamesDAO.java`;

    (b) Update the `listGames` facade method to support asking for a specific *page* (in the pagination sense). The method will now have a parameter identifying the page to display, and will thus produce a partial result (see the method `subList()` in the Java `List` interface).

    (c) Update the JSP file so that the pagination links (at the bottom of the table) are generated according to the number of pages needed to display all games (add an attribute to the request to indicate how many pages there are). The pagination links should also now send the number of the page to display (e.g., using a GET request).

    (d) Update the pagination links so that "Next" and "Previous" links are displayed/active if appropriate (depending on the number of pages and the current page being displayed, respectively).

8 Completing the application

Having checked that everything is working as expected, the application can now be improved and completed. A set of additional tasks is proposed in this section.

Tasks

12. The above solution implies a refresh every time a pagination link is pressed. This does not provide the best possible user experience. To avoid these page refreshes, try the following:

(a) Add a further parameter to the Servlet used above (or create a new Servlet) to ask for the serialisation of the list of games into JSON[18].

(b) update the JSP file by adding event handlers to the pagination links that make an AJAX request for the JSON list of games and update the HTML with the resulting data.

(c) One problem with this solution is that it is no longer possible to bookmark a specific pagination *page*. This is inconsistent with what users will expect and thus goes against the *Consistency* usability principle. With HTML5 you can solve this by manipulating the browser's `window.history` object inside the event handlers of the pagination links[19].

13. Implement the `My Games` page.

(a) Create a new method in the DAO to return a different list of games (a statically defined list will be enough to test the interface).

(b) Add a parameter to the Servlet(s) used above (you could also create a new Servlet but be careful with code duplication) to return a JSP page (or JSON list, is you implemented the serialisation feature) with the games belonging to the user. Consider using the same JSP page as above.

14. Implement the `Information on a Game` page (see mockups in Tutorial 1).

(a) As before, create a new method to return a game's information (it can be static information for now), and update or create a new Servlet and create a JSP page to show the game's information (make use of the template).

15. Create the `Login` page.

(a) Start by creating the business methods to support a login (predefine a set of users).

(b) Create a modal window[20] to display a form, in which the user can log

[18] You will have to set the content type of the response to `"application/json"`, instead of `"text/html"`, and output the JSON text, instead of forwarding to a JSP.

[19] Modern browsers support this. See, for example, `https://developer.mozilla.org/en-US/docs/Web/API/History_API`.

[20] Check Bootstrap modal windows `http://getbootstrap.com/javascript/#modals`

in.

(c) Make the login window as part of the template, so that it can be accessed in any page.

(d) Make use of the session variable to keep the information regarding the login (e.g. user name).