



UNIVERSITY OF TRENTO

ASSIGNMENT 3

Vinci Nicolò 220229

nicolo.vinci@studenti.unitn.it

Web Architectures 2021/2022

31 October 2021

Contents

1	Introduction	2
2	Description	3
3	App	5
3.1	Deployment	5
3.2	Start web app and Login	5
3.3	User page	8
3.4	Play game	8
3.5	JSP filter	14
3.6	Multiple user	15

1 Introduction

A web app that allows to play memory has been developed. A user can register himself inserting a username and play memory. The memory game consists of turning pairs of cards matching the same number. For example, a card is turned over and it shows the number 5. If the user is able to turn another card and find again the number 5, the score will be incremented. So, a user can perform four attempts to find a match. Every time the user turns two different cards, the number of attempts decreases by one. When the number of attempts is zero, the game ends. The score decreases by one if the user does not find a match. Otherwise, the score increases by $2 * \text{card value}$. For instance, if the user is able to match the number 5, the score will be increased by $2 * 5 = 10$. When the game ends, the user will be redirected to a page where a ranking will be shown. The ranking shows only the top five best scores.

2 Description

The web app has been developed following the Model View Controller (MVC) pattern and the server side should be minimal implementing the memory game client side in HTML+JS+XHR:

- Model: JavaBean
- View: JavaServer Page (JSP)
- Controller: Java Servlet

One JavaBean has been used:

- Games: it is stored in *ServletContext* and contains an *ArrayList* *<Game>*. The object Game has the attribute username and score.

Then, three different JSPs have been developed:

- login.jsp: it allows to insert a new username. It is shown when the user has not inserted a valid username yet.
- userPage.jsp: it shows the ranking with the top five best scores and allows to start a new game.
- playGame.jsp: here the user can play with a grid 4x4 with 16 cards. The number of attempts and the score are shown. A label *Game over* is shown when the user ends the number of attempts.

After that, there are four Java Servlets to serve http requests:

- LoginServlet: it handles the *get* request returning the *login.jsp*. It manages also the *post* request when a user insert a new username from the *login.jsp*.
- UserServlet: it handles only the *get* returning the *userPage.jsp* in order to follow the MVC paradigm.
- GameServlet: it handles the *get* request initializing the card grid 4x4 and returning the *playGame.jsp*. It manages the *post* request from the *playGame.jsp* when the game ends in order to store the score for the actual user.
- CardServlet: it handles the *post* request from the *playGame.jsp*. It returns the card value when the user turns over a card.

Two filters have been developed:

- LoginFilter: it is applied to the root url */*. It checks if the session is non null and if the user has already inserted a username. If it so, the user is redirected to the *userPage.jsp*, otherwise to the *login.jsp*.

- JSPFilter: it is applied to any url `/*`. It checks if the substring `jsp` is in the request url. If it so, an error 401 will be displayed. It is implemented to protect the MVC pattern. A user can not access directly to the JSPs, but he must perform requests to controllers. Then, controllers will return JSPs.

There is also a Listener:

- Listener: when the *ServletContext* is created, it creates a new *ArrayList <String>* in order to store usernames. The *ArrayList* is saved in the *ServletContext*.

A JavaScript file has been developed:

- playGameJS.js: it is imported in the *playGame.jsp*. The function *onClickEv(id)* performs a *get* request to the *CardServlet* every time that a card is turned over in order to discover its value. It also updates the number of attempts and the score. When the number of attempts is zero, the function *endGame()* is called to perform a *post* request to *GameServlet* to store the final score of the user.

A CSS file has been developed:

- playGameCSS.css: it is imported in the *playGame.jsp*. It defines a class *card* to set some style attributes for each *div* of the grid. For instance, it sets the width and the height. Then, it is decided to thicken the card border and make it black when the user points to a card with the mouse. This has been developed with *.card:hover*. Another class called *cardImg* is defined to set the image dimension and it is applied to the *img* tag.

In the folder *webapp/imgs* have been placed images the represent card numbers and the card back.

A summary MVC model schema applied for the web app is reported at figure 1.

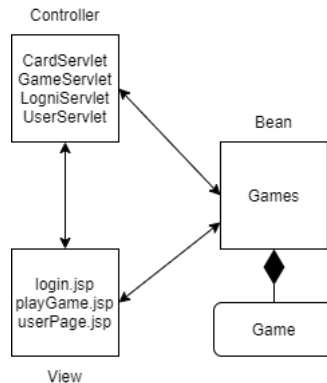


Figure 1: MVC schema

3 App

The web app functioning is described in this section.

3.1 Deployment

First of all, the web app needs to be deployed in the Tomcat server. The folder generated in IntelliJ has been copied in the Tomcat *webapps* folder.

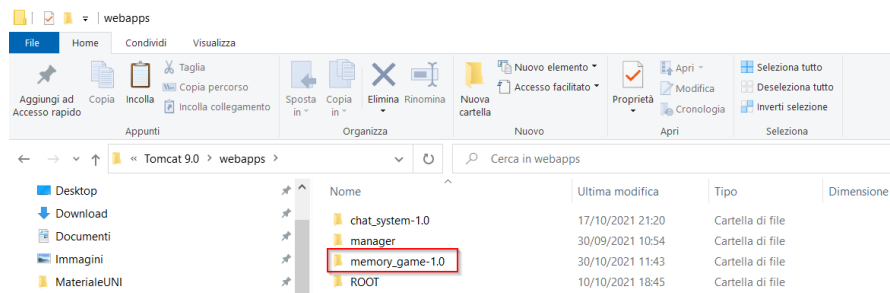


Figure 2: Webapps Tomcat folder

Now, the web app is visible in the Tomcat manager page. It can be stopped, reloaded and undeployed.

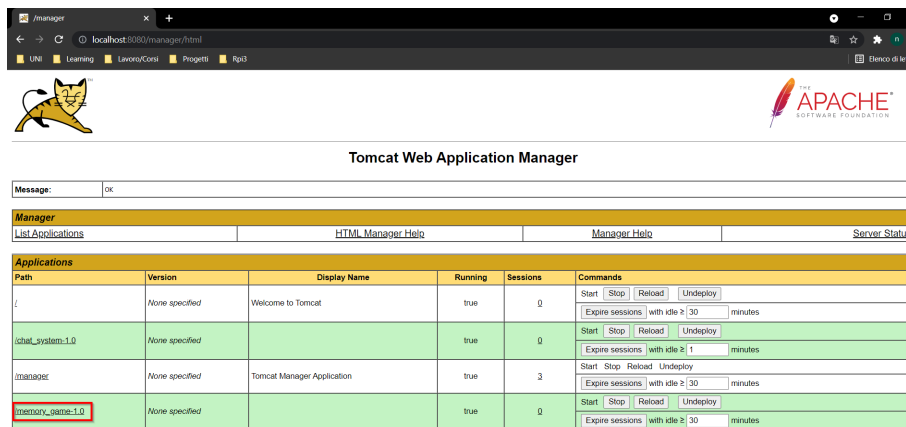


Figure 3: Tomcat manager page

The web app can be started clicking on the link */memory_game-1.0* shown in the figure 3.

3.2 Start web app and Login

When the web app starts, the method *contextInitialized* in the *Listener* is called. It initializes an empty *ArrayList* *<String>* and stores it in the *ServletContext* as

attribute **users**. The ArrayList maintains usernames inserted by users. Then, the *LoginFilter* matches the root path:

```
<filter>
  <filter-name>JSPFilter</filter-name>
  <filter-class>com.example.memory_game.filter.JSPFilter</filter-class>
</filter>

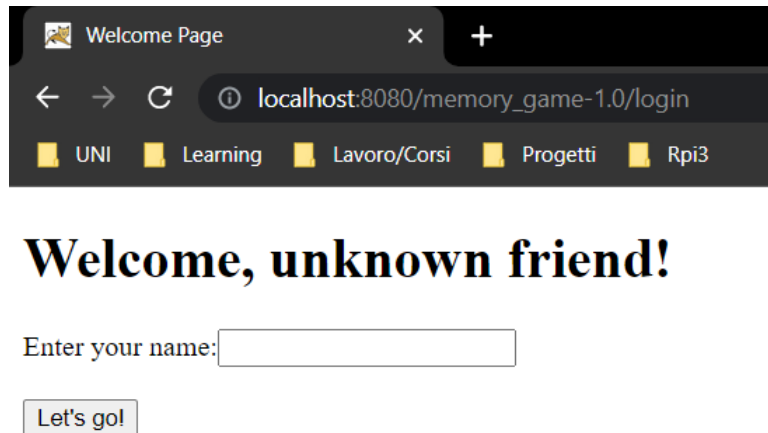
<filter>
  <filter-name>LoginFilter</filter-name>
  <filter-class>com.example.memory_game.filter.LoginFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>JSPFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>LoginFilter</filter-name>
  <url-pattern>/</url-pattern>
</filter-mapping>
```

Figure 4: Filters definition in *web.xml*

The filter retrieves the ArrayList of usernames from the attribute **users** in the *ServletContext*. If the session object is null or the username stored in session is not present in the ArrayList, the filter redirects the user to the *LoginServlet* which returns the *login.jsp*. Otherwise, if the session object is not null and the username stored in session is present in the ArrayList, the filter redirects the user to the *UserServlet* which returns the *userPage.jsp*.



Welcome Page

localhost:8080/memory_game-1.0/login

UNI Learning Lavoro/Corsi Progetti Rpi3

Welcome, unknown friend!

Enter your name:

Let's go!

Figure 5: *login.jsp* because session object is null

So, at the beginning the user has to insert a username and click on *Let's go!*. When the user click on it, it performs a *post* request to the *LoginServlet*. It handles the request in the method **doPost**. Firstly, it checks if the username inserted is empty. If it so, it sets a request attribute called **error** to true and returns again the *login.jsp*. Now, the *login.jsp* shows a message error *Error in Input Field*.



Figure 6: Error due to empty username

Otherwise, the *Games* bean is initialized in the *ServletContext* as attribute **gamesBean**. Then, the username inserted by the user is stored in the **session** as attribute **usernameInSession**. The response is redirected to the *UserServlet* which returns the *userPage.jsp*



Figure 7: *userPage.jsp*

3.3 User page

The *userPage.jsp* retrieves the attribute **usernameInSession** and shows the actual username. Then, it also retrieves from the **gamesBean** and retrieves the *ArrayList* `<Game>`. If it is empty, it shows *Classifica vuota - Nessuna partita giocata*. Otherwise, it shows the ranking with the top five best scores. The user can start a game clicking on the button *Play Game*. Hence, a *get* request is performed to the *GameServlet* which handles it in the *doGet* method. A list of integer is initialized to represent the value of any card in the 4x4 grid. The *GameServlet* retrieves a context parameter called **mode** set in the *web.xml*.

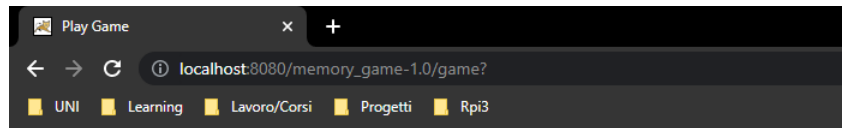
```
<context-param>
  <param-name>mode</param-name>
  <param-value>PRODUCTION</param-value>
</context-param>
```

Figure 8: context-param in *web.xml*

If the parameter is equal to *PRODUCTION*, the list will be shuffled. Otherwise, the list remains plain for test purposes. The list is stored in the *ServletContext* as attribute **grid**. At the end, the *doGet* method returns the *playGame.jsp*.

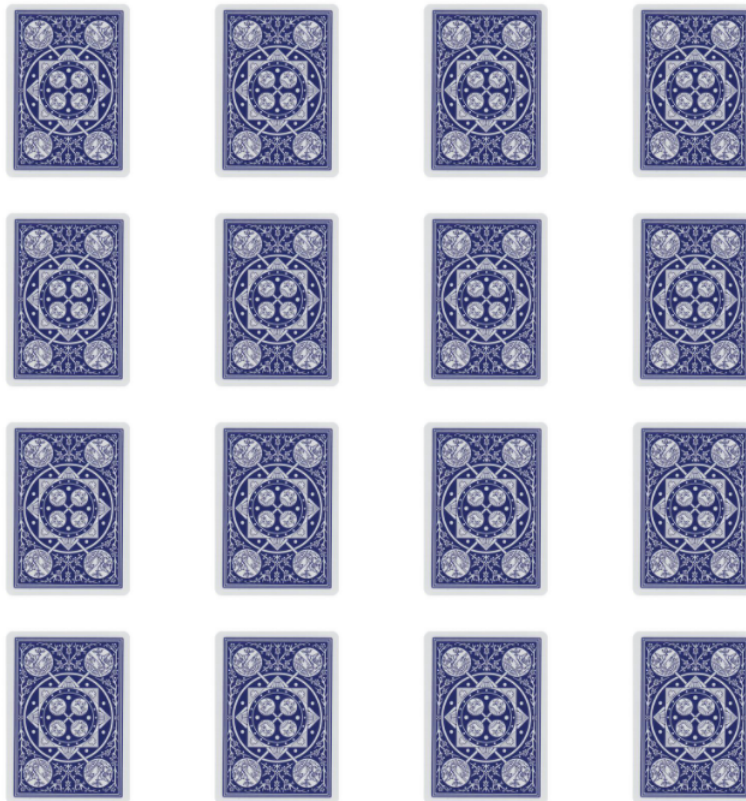
3.4 Play game

The *playGame.jsp* includes the stylesheet *playGameCSS.css* and it shows the number of attempts left identified by an id called *tentativi*. Then, it generates a 4x4 grid exploiting two for cycles. Each row is composed by four *div* tag with attribute class set to *card* and each *div* has an *img* tag. It has an incremental id from 0 to 15 and an attribute class set to *cardImg*. Moreover, the attribute *src* is set to *./imgs/cardBack.jpg* in order to have the card back image at the beginning. It has also the attribute *onclick* set to *onClickEv(this.id)*. This JavaScript function is written in the file *./playGameJS.js*. At the bottom of the page, the score is shown and it is identified by an id called *score*. At the end, a JavaScript variable **requestContextPath** is defined to save the context path and the JavaScript file *./playGameJS.js* is imported.



Welcome to Memory

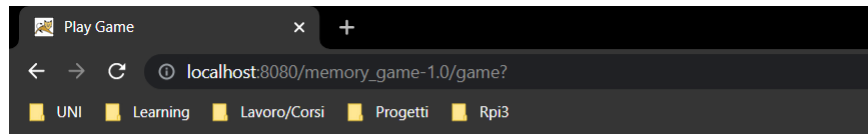
Tentativi rimasti: 4



punti:0

Figure 9: *playGame.jsp*

Now, if the goes on a card, the pointer changes and the border becomes thicker and black.



Tentativi rimasti: 4



Figure 10: Card border and new pointer

If the user click on the first card, the function *onClickEv(this.id)* is called passing to it the id of the card. A variable **clickedCard** is used to distinguish if the user is turning over the first card or the second. if the variable is set to false, the user is turning over the first card. Otherwise, he is turning over the second. It sets to true the variable **clickedCard** in order to remember that one card is already clicked. Then, it stored the id of the clicked card in the variable **previousCard**. The clicked card becomes also unclickable because the attribute *onclick* will be removed. A *XMLHttpRequest* is created in order to perform a *get* request to *CardServlet* to discover the card value and uncover the card. The card id is passed as parameter in the request. The *CardServlet* handles the request in the *doGet* method. It takes the id parameter from the request and the card value list from the attribute **grid** in the *ServletContext*. Then, the id is used as index for the list and the card values is retrieved. At the end, it crafts a JSON object to send as response. The *XMLHttpRequest* object in the *onreadystatechange* function, receives the response, retrieves the card value from the JSON object and uncover the card setting the *src* attribute to the corresponding image number. The value of the first card is stored in the variable **card1**.

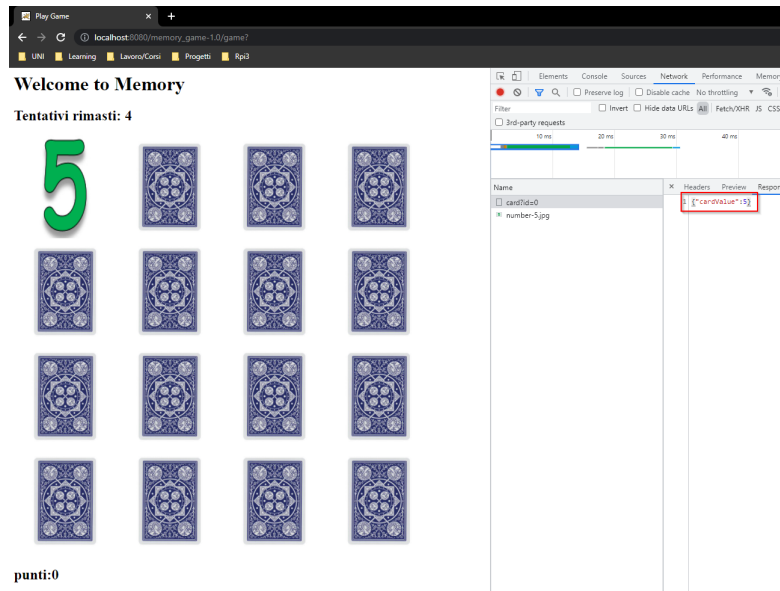
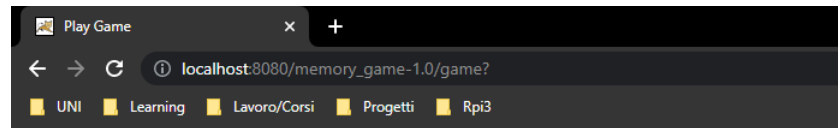


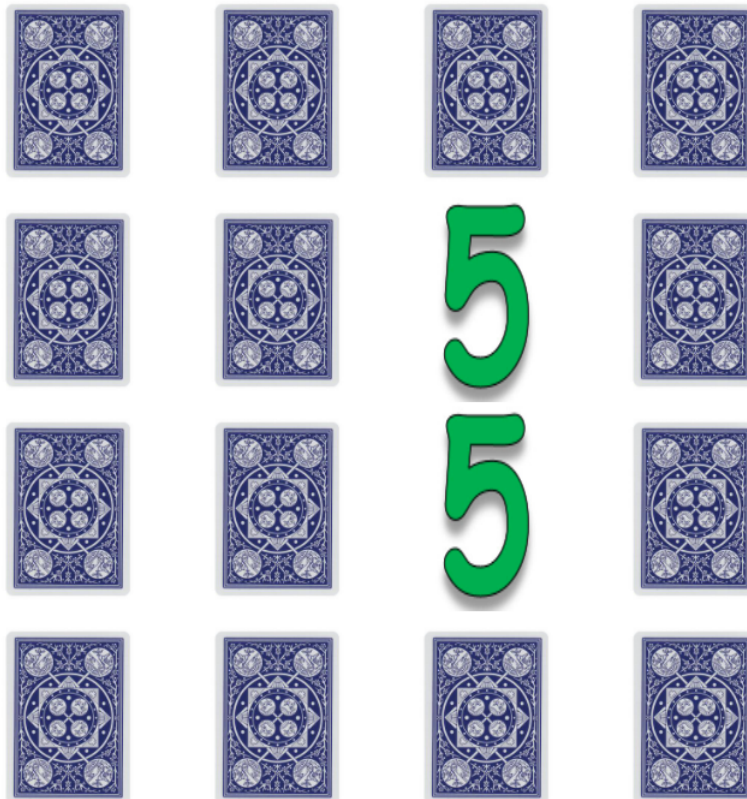
Figure 11: First card turned

Now, the user can turn over another card and again the *onClickEv(this.id)* is called passing to it the id of the card. However, the variable **clickedCard** is set to true. The clicked card becomes unclickable and the number of attempts decreases immediately calling the function *updateTentativi()*. A new XMLHttpRequest is created in order to perform a *get* request to *CardServlet* to discover the card value and uncover the second card. When the *CardServlet* returns the card value, it is stored in the variable **card2** and the card is uncovered setting the src attribute to the corresponding image number. Then, if the function *checkCard(id)* is called to compare the two variables **card1** and **card2**. If they match, the two cards remains uncovered and unclickable and the score is updated.



Welcome to Memory

Tentativi rimasti:1

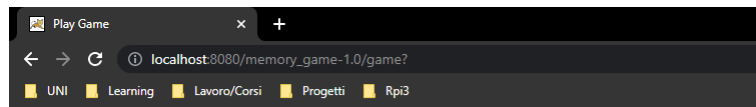


punti:8

Figure 12: Cards match

Otherwise, the score decreases by one and after one second the two cards become clickable and covered again. After that, the function *endGame()* is called in order to check if the number of attempts is zero. If it so, all cards become unclickable and a new label shows *Game over* below the attempts number. After one second, a XMLHttpRequest is crafted to perform a *post* request to the *GameServlet* passing as parameter in the body the final score of the game. The *GameServlet* handles the request in the *doPost* method. It retrieves the score

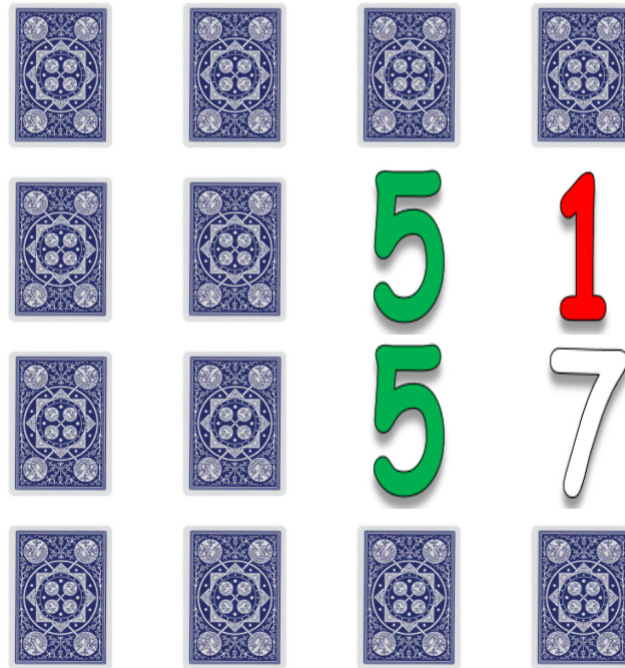
parameter in the body request, the username from the session attribute **usernameInSession** and instantiates a new *Game* object with them. Moreover, the games bean from the *ServletContext* attribute **gamesBean**. It gets the *ArrayList* *<Game>* from the games bean and it adds to it the new *Game* object. Then, the *XMLHttpRequest* object in the *onreadystatechange* function waits. When, the ready state is 4 and it receives the status code 200, it changes the *window.location* to a path built with the variable **requestContextPath** and */user*. So, a *get* request is performed to the *UserServlet* which returns again the *userPage.jsp*.



Welcome to Memory

Tentativi rimasti:0

Game over



punti:7

Figure 13: End game

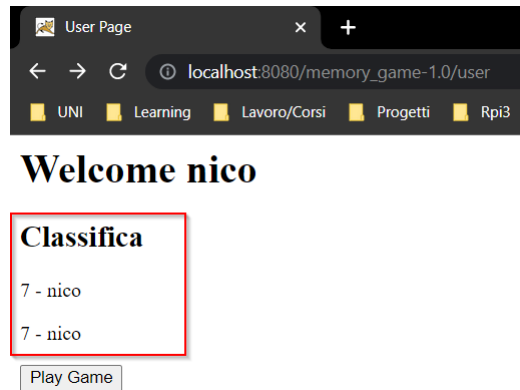


Figure 14: *userPage.jsp* with ranking game

3.5 JSP filter

The *JSPFilter* defined in the *web.xml* guarantees the MVC paradigm, because the user can not access directly to JSPs.

```
<filter>
  <filter-name>JSPFilter</filter-name>
  <filter-class>com.example.memory_game.filter.JSPFilter</filter-class>
</filter>

<filter>
  <filter-name>LoginFilter</filter-name>
  <filter-class>com.example.memory_game.filter.LoginFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>JSPFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>LoginFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

Figure 15: JSPFilter definition in *web.xml*

It matches any path and it checks if the substring *.jsp* is present in the request URL. If it so, the user is trying to access directly to a JSP and an 401 unauthorized error will be returned.

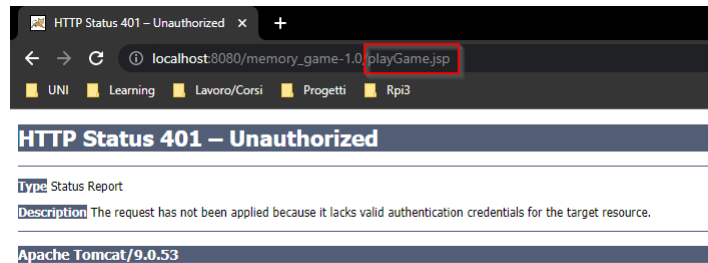


Figure 16: Unauthorized request

3.6 Multiple user

The web app can be tested with different browsers in order to see that multiple users are able to play and they will be shown in the ranking. The actual situation from google chrome is reported in the figure 17. The logged user is *nico*.

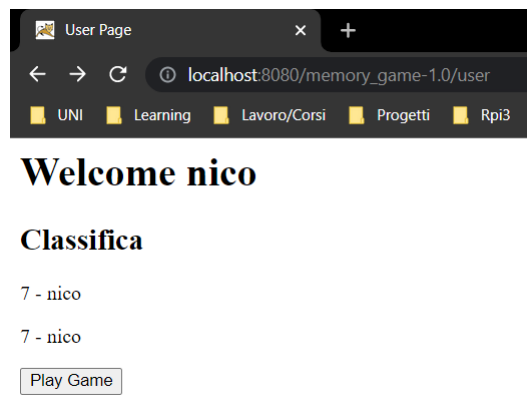


Figure 17: Chrome *userPage.jsp* from *nico* user

Now, from firefox we can access to the web app with a different username *madda*.

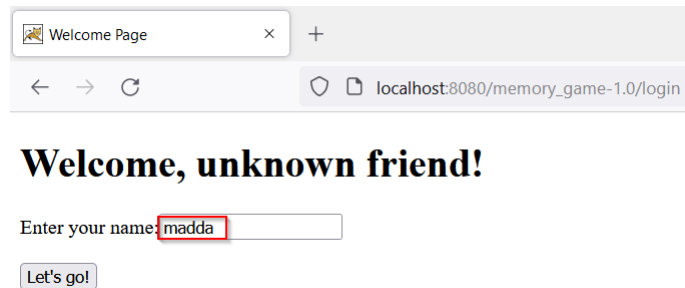


Figure 18: Firefox *userPage.jsp* from *madda* user

The user *madda* is able to see the same ranking of the user *nico*.



Figure 19: Firefox *userPage.jsp* from *madda* user

The user *madda* can play a game and return back to the *userPage.jsp* with the new ranking.

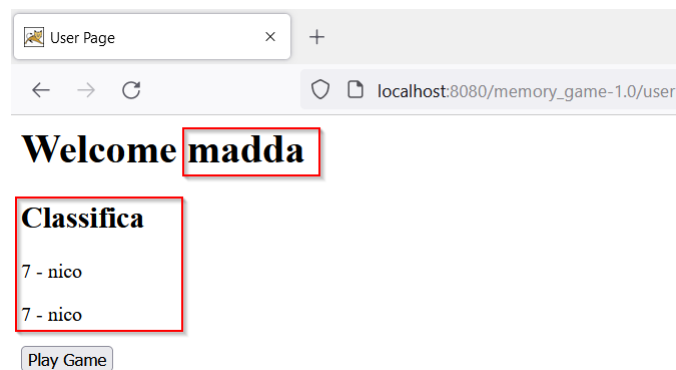


Figure 20: Firefox *userPage.jsp* from *madda* user with the new ranking

Now, if the user *nico* refreshes the page or plays another game, he will be able to see also the *madda* score.

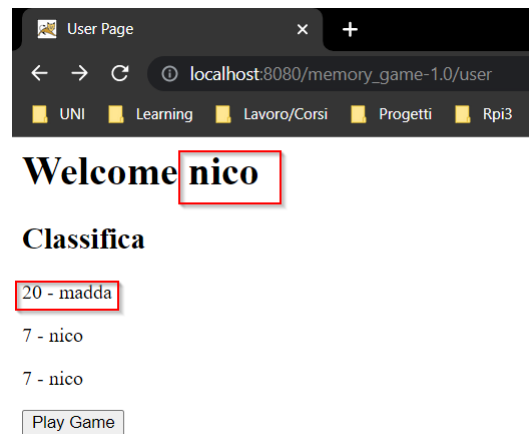


Figure 21: Chrome *userPage.jsp* from *nico* user with the new ranking

However, if there are more than five scores, the ranking will report only the five best scores.