Department of Information Engineering and
Computer Science
University of Trento, Italy

Web Architectures - A.Y. 2022/2023

Assignment 2

Authors:
De Menego Luca

Oct 8, 2022

Contents

1 Introduction

The second assignment involves the development of a web-application following the MVC pattern, based on Java Servlets, Java Beans and JSPs. Specifically, the application that has to be developed is a *flags game* in which, given a pre-defined set of flags, the user has to correctly map each one of them with the corresponding capital. For each game, the user is granted three points if all answers are correct, otherwise two. However, after every login each user will lose all the points.

In order to play, a user has to be registered to the application, and all his data must be:

- loaded from a file on server initialization;
- saved into a file on server shutdown.

After login, normal users will access a homepage showing their current score and a button to start playing. A special user with predefined credentials (username `admin`, password `nimda`) will instead access a dashboard showing all currently active users and their respective scores.

Each created session will timeout after a certain amount of seconds, defined as an environment variable in the `web.xml` file.

Any attempt to access pages without the required permissions should be correctly handled; in detail:

- not logged-in users accessing any page different from the *login* and *signup* ones should be redirected to `login`;
- logged-in normal users accessing *admin* pages should receive a `401 Unauthorized` error.

2 Solution

Model View Controller (MVC) is a pattern emphasizing the concept of *separation of concerns*. It is structured in three main parts:

- **model**: it handles data and business logic;

  Implemented using JavaBeans, Java classes following a particular convention that can be easily integrated with JSPs thanks to specially crafted APIs.

- **view**: it concerns everything regarding the user interface and it should only contain frontend code;

  Designed with JSPs, a technology allowing to write frontend code while supplying dynamic content fetched with Java code using special tags.

- **controller**: it handles the commands given by the users, while calling business logic code and dispatching views.

  Organized with Java Servlets, Java programs exposed by a web server that can accept requests and handle them via a very large set of useful APIs.

While the developed servlets are publicly exposed by the web server, all the JSP pages are private, and put inside the `WEB-INF` folder. So, the only way to access the JSPs are the servlets themselves: they act like a checker, verifying whether the received request is properly shaped or malformed. In this way, any attempt to access pages in a wrong or unexpected way will be automatically stopped.

2.1 Project's Architecture

Now that the main idea behind the internal structure of the project has been defined, the complete architecture can be exhibited:

model: POJO and JavaBeans used in the project

`UserBean`: class representing a base user of the web-application, mainly containing a username, a password, a score and a field allowing to understand whether the user is an admin or not.

`UsersBean`: class containing all information about the users, exposing both all the registered users and the currently active ones.

`FlagsBean`: class containing all the information needed to handle the flags game. In detail it exposes a set of predefined flags and the flags to guess, which are randomly generated each time the user starts a new game.

`UserService`: simple class exposing static methods to interact with users' information, namely `login` and `signup`.

`Users`: simple class representing a list of users; the most relevant difference with respect to a normal `ArrayList` is given by the fact that this class also exposes a `toHTML` function, that will be better explained in Subsection 2.5.1. The class contains a `List` instead of *implementing* it to follow the design principle of *"preferring composition over inheritance"*, which seemed a better alternative in this specific use case.

controller: all Java Servlets

`WWFServletContextListener`: more information can be found at Subsection 2.4

`WWFSessionListener`: more information can be found at Subsection 2.4

`/login`: *GET* the login page or *POST* a request for a login;

`/signup`: *GET* the signup page or *POST* a request for a registration;

`/logout`: log out from the web application;

`/user/home`: *GET* the home page for a given user, showcasing his points;

`/user/play`: *GET* the page in which the user can play or *POST* a request containing the answers to the last played game;

`/admin/dashboard`: *GET* the page listing all the currently active users and their points.

filter: filters used to check for the user's authentication state. More information can be found at Subsection 2.3.

view: developed JSP pages

`common`, a JSP included in every other JSP, containing a script for integrating *TailwindCSS* and custom global styles;

`login` and `signup` for users' authentication handling;

`header`, a JSP included in every other JSP excluding `login` and `signup`, where the user's username and a *logout* button are shown;

`user-home` and `admin-dashboard`, the main pages accessed respectively by normal users and administrators after login;

`play`, the page accessed when a new game is started;

`error`, a custom error page.

Based on which kind of data the beans are exposing, different scopes are needed. Everything related to a particular user must be kept in memory as long as the user is logged in, so the `UserBean` is stored within the user's session. Since every instance of a game is inherently linked with a user, also the `FlagsBean` is stored in the session. The `UsersBean`, instead, is stored directly inside the *ServletContext*, because it contains global information that must be kept as long as the application is running, so its scope will be *application*.

As an additional focal property of the developed beans, we have the synchronization aspect: Java Servlet Containers are typically multithreaded, hence multiple requests to the same Servlet might be executed at the same time. This may lead to thread safety issues if not handled properly, so it is vital to observe that:

- variables accessed from the *ServletContext* are global and can be easily accessed concurrently by different threads, so they need to be read and written making use of the `synchronized` keyword;
- variables accessed from the `HttpSession` belong to a specific user, so in theory there should be no problems deriving from concurrent accesses. However, the user may open two windows on the same browser for instance, so even in this case we need to use `synchronized`.

As a last comment on the project's architecture, any request done without the required permissions will lead to either the login page or an error page showing status code and error message (Figure 6). In detail, the redirection happens in case of not logged-in users, as requested by the project's description. Authentication is mainly managed through filters, presented in Subsection 2.3.

2.2 User's session

In order to keep the information of a logged in user an instance of *HttpSession* is used, by setting

```
request.getSession().setAttribute("user", userBean);
```

Since one of the requirements of the project is to set up a customized session lifespan, when the user logs in or signs up, the session is created and customized with:

```
request.getSession().setMaxInactiveInterval(sessionTimeout);
```

The session timeout is a variable defined as a `<env-entry>` in the `web.xml` file, and saved during `ServletContextInitialization` in the `UsersBean`.

2.3 Filters

A filter dynamically intercepts requests and responses, allowing the developer to examine and forward them. This enables a better modularization of software. In our case filters have been essential for logging requests and checking for the user's authorization when accessing specific pages.

The first filter applied to any request is the `LogFilter`, which simply prints to standard output IP address and timestamp of the request.

The second filter, applied to any request matching `/user/*`, checks if the user has an active authenticated session. If not, the user is instead redirected to the login page.

The third filter, applied to any request matching `/admin/*`, checks if the user has an active authenticated session and if he is an administrator. If the authentication is completely missing the user is again redirected to the login page; otherwise if the user is logged in but not as an administrator, he is redirected to an error page with status code 401 (Figure 6).

The filters have been activated by adding the `<filter>` and `<filter-mapping>` keywords in the `web.xml` file, in the order presented above.

2.4 Listeners

One of the main requirements of the project is to maintain the users' information across server restarts. In detail, data must be loaded during server's initialization and saved on server's shutdown. A class `WWFServletContextListener` implementing `ServletContextListener` has been created, that overrides the `contextInitialized` and the `contextDestroyed` methods to perform the mentioned operations:

- during initialization different environment variables are read, defined in the `web.xml` file as `<env-entry>`:
  - `AdminUsername`: the admin's username;
  - `AdminPassword`: the admin's password;
  - `UsersPath`: path to the file that will contain all information about the users;
  - `SessionTimeout`: seconds representing the lifespan of a session.

  Moreover the users are loaded from the data source defined in the *UsersPath* variable. If the file does not exists, the `UsersBean` will be initialized with one only user: the administrator.
- during shutdown the serialized `UsersBean` is stored in the file defined by the variable *UsersPath*.

Another requirement of the project is given by the fact that the administrator should be able to visualize the list of currently active users. Keeping a variable in the `UsersBean` containing a list of active users in which a user is added on login and removed on logout is quite straightforward. However, the user's session may also expire automatically. To deal with this case, a `WWFSessionListener` implementing `HttpSessionListener` has been developed, which overrides the `sessionDestroyed` function by removing the corresponding user from the list of active users.

2.5 JSPs and UI

The user interface of the web application has been first conceived using *Figma*, an interface design tool. The initial template is publicly exposed, and can be accessed from Figma - WorldWideFlags. To recreate this user interface *TailwindCSS* has been used, a CSS framework making quicker the process of writing and maintaining the code of a web application. As opposed to frameworks like *Bootstrap*, *TailwindCSS* does not provide ready-to-use components: it changes the way of writing CSS by asking the developer to use pre-defined CSS classes, instead of directly writing verbose CSS.

2.5.1 Dynamic content generation

JSPs should always contain as little Java code as possible: they represent the view and layout of the web-application, hence they should not contain business logic. As presented in class, one possible way to keep the code within JSPs minimal is to directly expose from the beans an initial, simple representation of data in HTML format. When needed, our beans expose a `toHTML` function that takes as input the style of the main elements (formatted as TailwindCSS classes) and returns a minimal HTML code with the requested dynamic content.

Whenever the access to a bean is needed, the JSP simply calls:

```
<jsp:useBean id="x" class="y" scope="z"/>
```

and then accesses the properties by using the `<%= %>` tags.

2.6 The game

Both the flags and the number of flags to guess are final variables kept within the `FlagsBean`. Whenever the JSP page asks for the guess list of flags in HTML format, the flags to guess are automatically randomized and the HTML code is consequently created. To make sure the user has answered to all fields, a *required* parameter has been added to each input field, and since we always expect a number its type is defined as `type="number"`. Obviously the check is not only performed client-side, since any malicious user could modify the frontend code. In those cases, if a malformed request arrives server-side, the user is redirected again to the play page, that will show an error message in red.

In order to prevent users from gaining infinite points by performing POST requests with the last given answers, every time the score is calculated after a game has been played, the list of flags to guess is resetted to an empty list. When the POST request is received, if the list of flags to guess is empty it means the user did not pass through our JSP, so an error 400 is returned to the user (Figure 11).

The form is completely dynamic in terms of number of elements, so if we wanted to add new flags or modify the number of flags the user needs to guess we would just need to update these variables in the `FlagsBean`, presuming that the images of the newly added flags are available in the `assets` folder.

3 The application running

A user accessing for the first time any URL within our application domain will be redirected to the login page (Figure 1). If the user does not have an account, he can access the signup page (Figure 2). Checks on the provided input are done both client and server-side, and the current implementation does not allow two users to have the same username. Figures 3 and 4 show some examples of errors, generated respectively client-side and server-side.

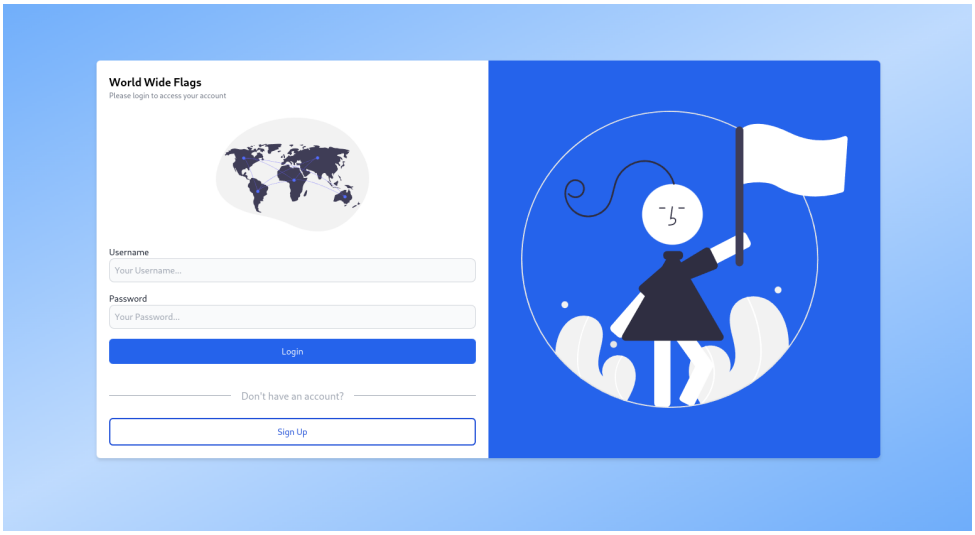

Figure 1: The application's login page

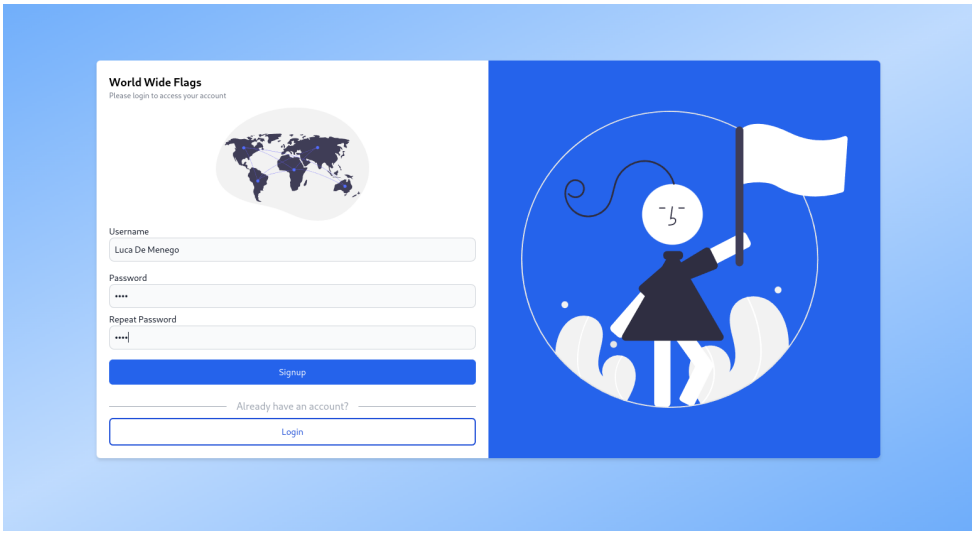

Figure 2: The application's signup page

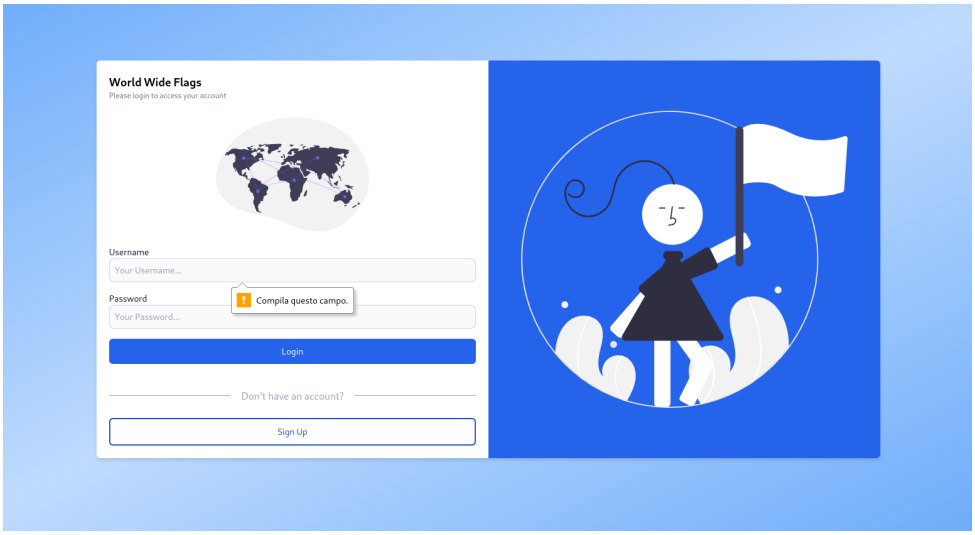

Figure 3: Client error example on the login page

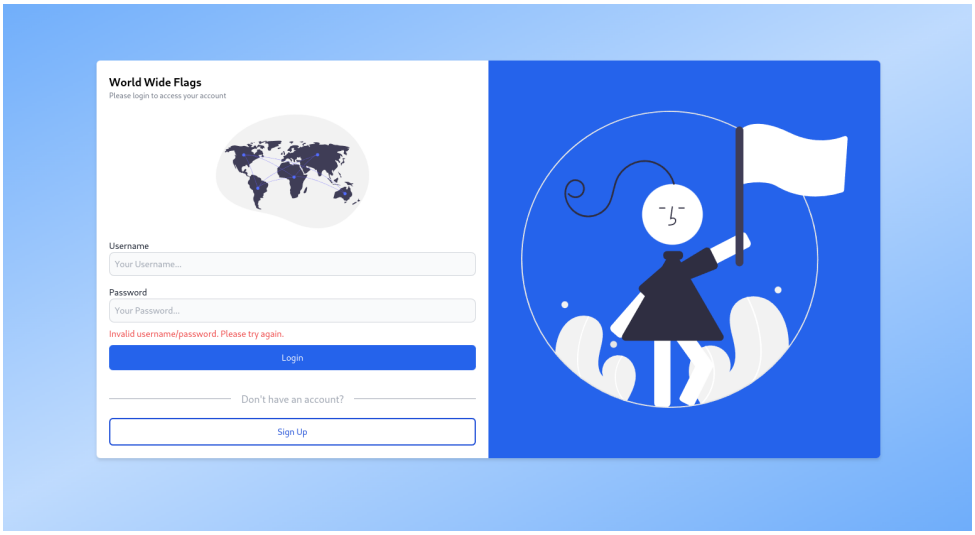

Figure 4: Server error example on the login page

After login or signup, the user will access his homepage (Figure 5), where he can see his points and start a new game.

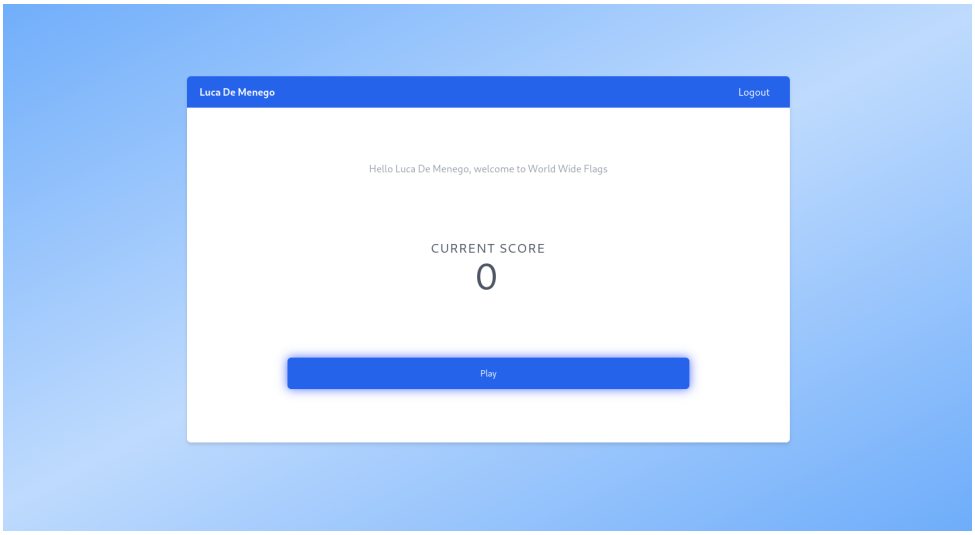

Figure 5: User's homepage

If at this point the user tries, for instance, to access `/admin/dashboard`, he will receive an error 401 (Figure 6).

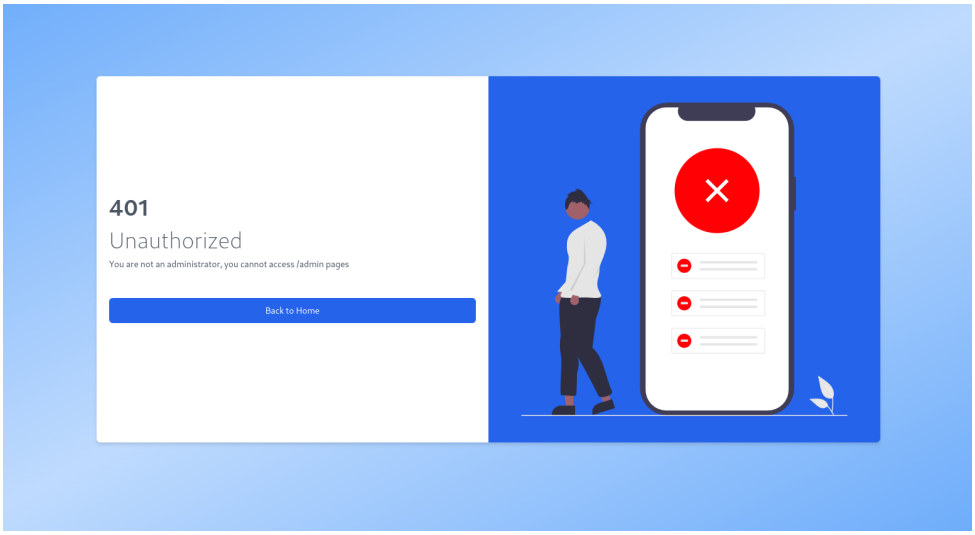

Figure 6: Error when a normal user tries to access `/admin/dashboard`

If the user decides to play, a randomized set of flags to guess will be initialized, and the user will be presented with a form containing the actual game (Figure 7) and a button that, when clicked, shows a dialog with details on how the game should be played (Figure 8).

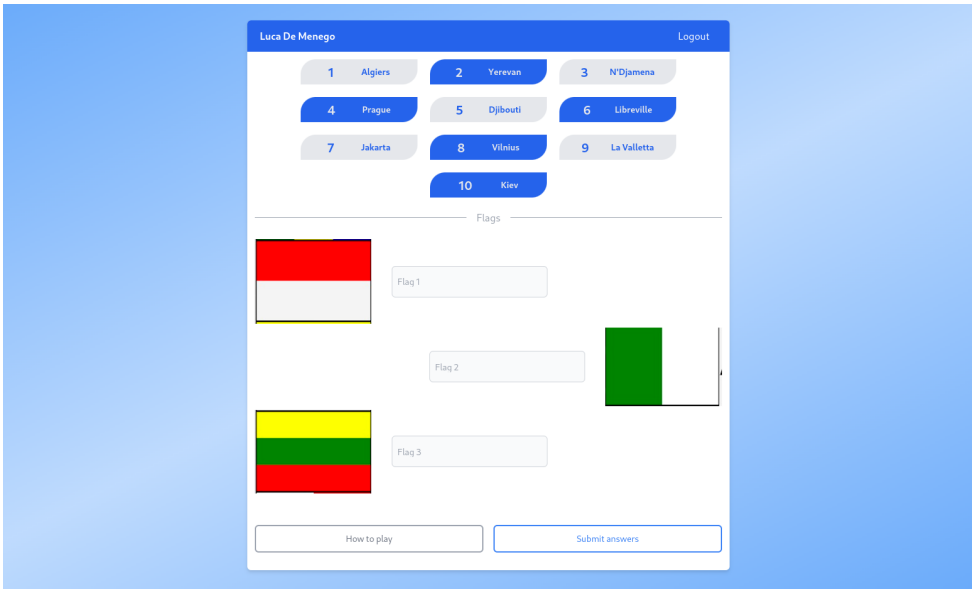

Figure 7: Game main page

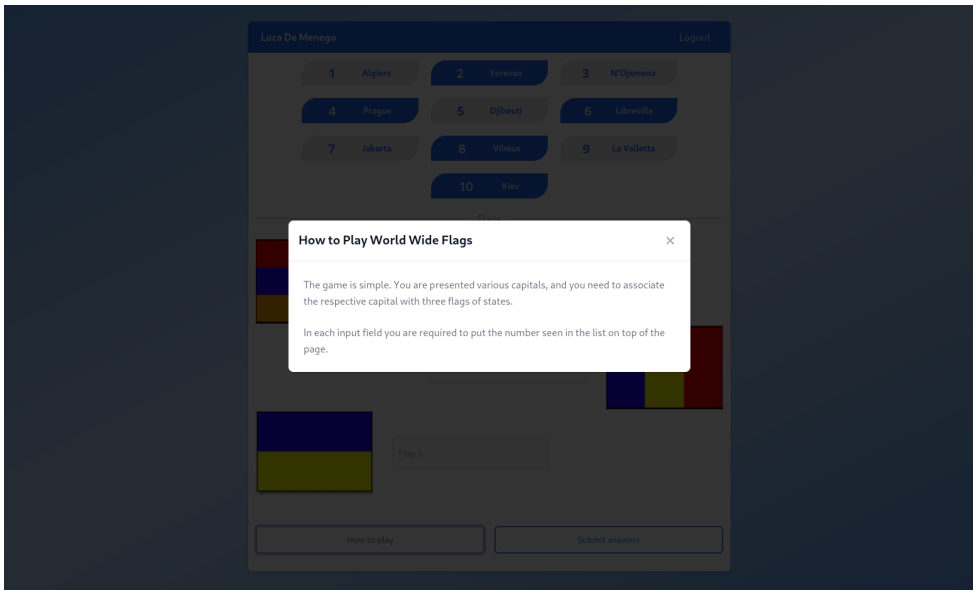

Figure 8: Game modal describing how to play

If the user does not fill all fields, a client-side check will immediately inform him (Figure 9). If the user removes the client-side check and tries to submit an answer with some fields not filled, he will still be presented with an error (Figure 10). As a last possibility on the game page, if the user manages to perform a POST request without having passed through our JSP, he will be presented with an error page (Figure 11).

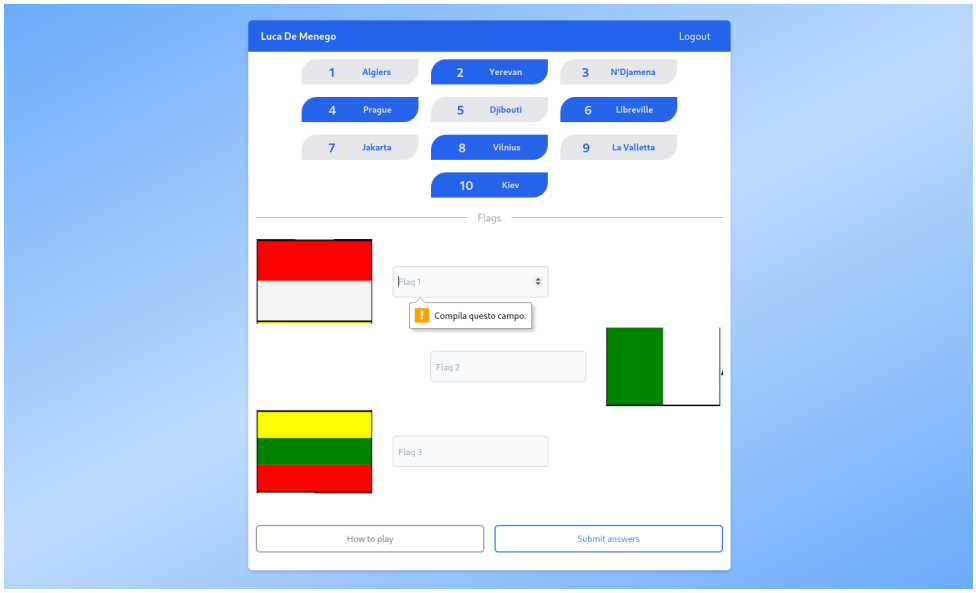

Figure 9: Game page, client error when the user submits without having filled all fields

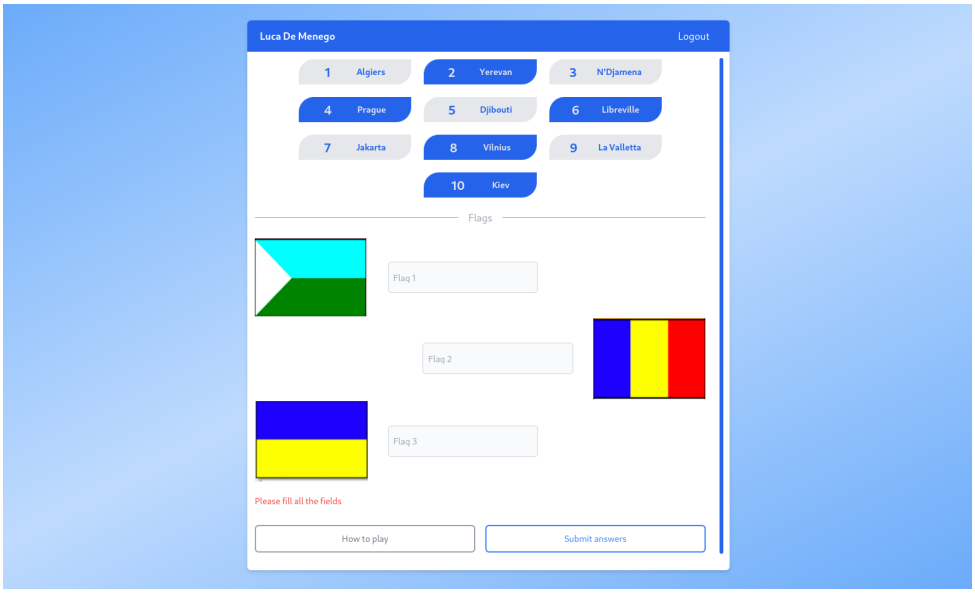

Figure 10: Game page, server error when the user submits without having filled all fields, and client-side checks do not work

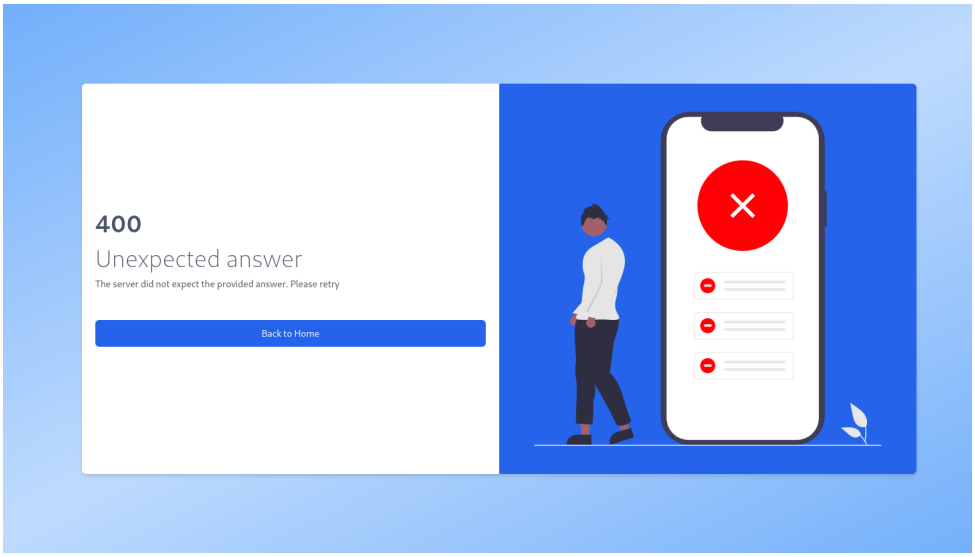

Figure 11: Error page received after having performed a POST request without passing through the game's JSP

After having submitted the form correctly, the user's score will be updated and the user will be redirected to its homepage.

If at this point an administrator logs in, he will be redirected to the admin's dashboard and he will see the active user who has just played the game, with his updated score (Figure 12).

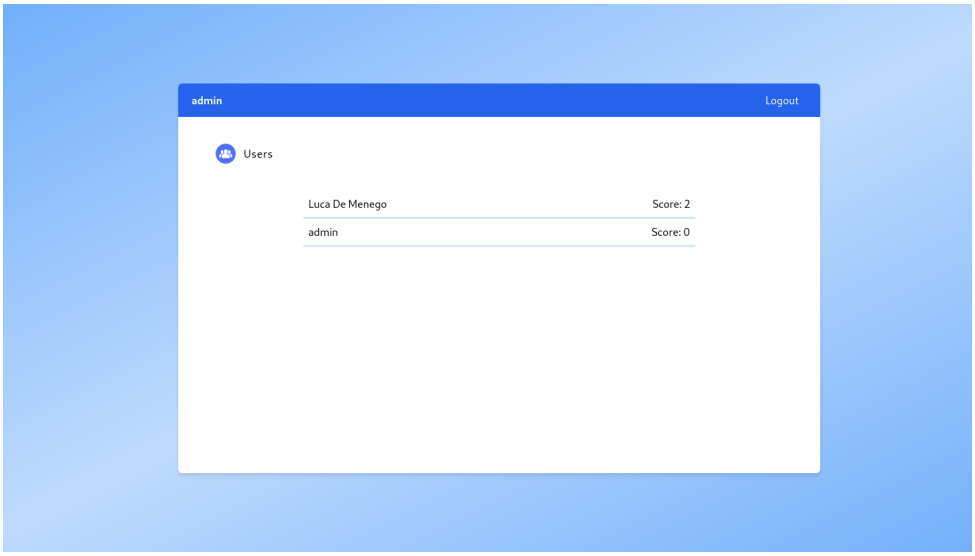

Figure 12: Admin dashboard showing all currently active users and their score

When the user logs out or the session expires, the administrator will be able to see it by updating the page.

4 Comments

The developed web-application is vulnerable to a Persistent Cross-Site Scripting attack: an attacker could sign up with a name containing malicious code, and when the name is presented to the administrator, the latter could potentially be downloading malwares or sending private information to the attacker. However, since the maximum number of characters allowed for the username is 15, it is quite hard to do actual damage. In order to completely solve the problem, the input should be filtered, and the output on the admin page encoded.

The only way in which I think an attacker could exploit the vulnerability with only 15 characters available is using the following username:

```
<base href=//0>
```

However, this would simply redirect the administrator to `0.0.0.0` after pressing logout, which just integrates a problem with the flow of the application, but does not include real security issues.

Special thanks to https://undraw.co/ for the illustrations.