



Assignment 2

16 October 2022

Index

1. The case
2. My solution
3. App running
4. Comments

The case

A little sum-up of the assignment.

1. The web application consists in a base single player guess game. Each player should be registered and logged to play and to accumulate points. The authentication must due a T defined time.
2. All the endpoints must be protected by security guards. No one can access more than is needed. Any attempt must be handled with appropriate HTTP errors and redirects.
3. Some users may be admins. With these permissions they can visit a control panel where user points and active ones are shown.
4. The state of the applications just need to be kept on RAM. But, for persistence, when the web application is shut down, data must be saved to a file and reloaded on launch.
5. The web application must implement the MCV pattern, Java Beans and Java code in JSP pages must be minimal.

My solution

1. The Authentication process stars with the definition of the User base model. Like for every other model it's a Java Bean. Just for the robustness of the code each Bean in this project implements a *create* method that creates a new instances and fill-up all the variables for a base usage. By doing this the class maintain it's flexibility for any use.
All passwords are hashed in *SHA1* strings and stored like that. Just because *JavaScript* Code wasn't the focus of the assignment the passwords are sent raw to the server, if the connection is secured there are no much consequence.
All the other fields are much straightforward.

Listing 1: User.java

```

public class User implements Serializable {
    private String username;
    private String hashedPassword;
    private boolean isAdmin = false;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getHashedPassword() {
        return hashedPassword;
    }

    public void setHashedPassword(String hashedPassword) {
        this.hashedPassword = hashedPassword;
    }

    public boolean isAdmin() {
        return isAdmin;
    }

    public void setAdmin(boolean admin) {
        isAdmin = admin;
    }

    public static User create(String username, String password) {
        User user = new User();
        user.hashedPassword = password;
        user.username = username;
        return user;
    }
}

```

As request my project implements a MCV pattern. In the authentication process takes place the *UserController* that contains all the business logic regarding Users and exposes some public method to provide data to the UI.

Listing 2: UserController.java

```

public class UserController {
    private ArrayList<User> users = new ArrayList<>();

    public UserController() {
        this.deserializeState();
    }

    public ArrayList<User> getUsers() {
        return users;
    }

    public synchronized User registerUser(User user) {
        boolean userExists = users.stream()

```

```

        .anyMatch(item -> item.getUsername()
            .equals(user.getUsername()));
        if (userExists) return null;

        users.add(user);
        return user;
    }

    public User loginUser(String username, String hashedPassword) {
        User foundUser = users.stream()
            .filter(item -> item.getUsername().equals(username))
            .findFirst()
            .orElse(null);

        if (foundUser == null) return null;
        if (!foundUser.getHashedPassword().equals(hashedPassword)) return null;

        return foundUser;
    }

    public User getUser(String username) {
        return users.stream()
            .filter(item -> item.getUsername().equals(username))
            .findFirst()
            .orElse(null);
    }

    private void deserializeState() {
        try
        {
            User adminUser = User.create("admin", HashGen.SHA1("admin"));
            adminUser.setAdmin(true);

            File f = new File("userData");
            if (!f.isFile() || !f.canRead()) {
                this.users.add(adminUser);
                return;
            }

            FileInputStream fis = new FileInputStream("userData");
            ObjectInputStream ois = new ObjectInputStream(fis);

            this.users = (ArrayList<User>) ois.readObject();
            this.users.add(adminUser);

            System.out.println("===");
            System.out.println("Loaded users from file:");
            for (User user : users) {
                System.out.println(user.getUsername());
            }

            ois.close();
            fis.close();
        }
        catch (Exception e)
        {

```

```

        e.printStackTrace();
    }
}

private void serializeState() {
    try
    {
        this.users = (ArrayList<User>) users.stream()
            .filter(item -> !item.getUsername().equals("admin"))
            .collect(Collectors.toList());
        FileOutputStream fos = new FileOutputStream("userData");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(this.users);
        oos.close();
        fos.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

public void destroy() {
    this.serializeState();
}
}

```

Listing 3: RegistrationServlet.java

```

public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException {
    response.setContentType("text/html");
    if (Strings.isNullOrEmpty(request.getParameter("username"))
        || Strings.isNullOrEmpty(request.getParameter("password"))) {
        response.sendError(HttpServletResponse.SC_BAD_REQUEST, "Missing_user_parameter.");
        return;
    }

    User user = User.create(request.getParameter("username"), HashGen.SHA1(request.getParameter("password")));
    user = SharedData.getInstance().getUserController().registerUser(user);
    if (user == null) {
        response.sendError(HttpServletResponse.SC_BAD_REQUEST, "User_already_registered.");
        return;
    }
    response.sendRedirect("login.jsp");
}
}

```

In the listing 3 we can see how the requests are handled with a Controller like *UserController*. The unique identifier of the user is the username, and so it is passed as parameter to the user session to identify them in the future. All the cookie setting is done in the under the hood by Java EE libraries. Since the username is the unique key of the User model it may occur a concurrency problem during the registration. To avoid this the *registerUser()* method is marked as *synchronized* so the executions is always blocked by one thread.

Listing 4: web.xml

```

<session-config>
    <session-timeout>5</session-timeout>

```

```
</session-config>
```

In the listing 4 we can see how the due of the authentication is done. It's not done programmatically but it works.

2. Let's see now how the data is shared across multiple Servlets.

Listing 5: SharedData.java

```
public class SharedData {
    private UserController userController = new UserController();
    private MatchController matchController = new MatchController();
    private SessionsController sessionsController = new SessionsController();

    ...

    public static SharedData instance;

    public static SharedData getInstance() {
        if (instance == null) {
            instance = new SharedData();
        }
        return instance;
    }

    ...
}
```

SharedData is a *singleton* that keeps alive the controller instances. It even handles the destruction process, triggering the serialization in files, of the controllers that manage the data.

Listing 6: SharedData.java

```
...
public void destroy() {
    this.userController.destroy();
    this.matchController.destroy();
}
}
```

So every shared data can be accessed link in listing 2

```
SharedData.getInstance().getSessionsController().getActiveUsers()
```

3. All the matches that users play are store like a *Many to One* relation with the correlated User instance. This allow a base structure for any other possible future features. Since every match is recorded when the User gets to the summary patch a sort of query is triggered and result is delivered to the UI straightforward.

Listing 7: MatchController.java

```
...
public int getUserPoints(String username) {
    return matches.stream()
        .filter(item -> item.getUsername()
            .equals(username) && item.getPoints() > 0)
        .map(Match::getPoints)
        .reduce(0, Integer::sum);
}
```

```

    }
    ...
}

```

4. Security guards are implemented with the *JavaX Filter* interface. If the users has an active valid session with the username attribute the validation process can go on with the next Filter registered in the *chain*. `@WebFilter("/app/*")` set's the guard to all the pages under the *app* directory. All the login and registration servlets are in the root folder.

Listing 8: UserCheckFilter.java

```

@WebFilter("/app/*")
public class UserCheckFilter implements Filter {
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws ServletException, IOException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;
        HttpSession session = request.getSession();

        Object isLogged = session.getAttribute("logged");
        if (isLogged != null && (boolean) isLogged) {
            chain.doFilter(request, response);
            return;
        }
        response.sendRedirect("../login.jsp");
    }
}

```

The admin guard is actually kinda the same, but here it's needed to require the user instance to validate the permission.

Listing 9: AdminCheckFilter.java

```

@WebFilter("/app/admin.jsp")
public class AdminCheckFilter implements Filter {
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws ServletException, IOException {
        ...
        String username = (String) request.getSession().getAttribute("username");
        User user = SharedData.getInstance().getUserController().getUser(username);

        if (user != null && user.isAdmin()) {
            chain.doFilter(request, response);
            return;
        }
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED);
    }
}

```

5. Pages with complex UI are written in JSP. Despite the requirement to use of Java as minimap as possible in JSP, here it's just enough to prepare the raw data to be shown in the UI.

Listing 10: match.jsp

```

<form action="${pageContext.request.contextPath}/app/matchValidation" method="post">
<%
    Match match = SharedData.getInstance()

```

```

        .getMatchController()
        .generateMatch((String) session.getAttribute("username"));
        int i = 0;
        for (Pair<String, String> country : match.getCountries()) {
            %><p><%=i+1 + ".␣" + country.getRight()%></p><%=
            i++;
        }
        i = 0;
        for (Pair<String, String> country : match.getCountriesToGuess()) {%>
            " alt
            <select required name="guess<%=i%>">
                <option disabled selected value> -- select an option -- </option>
            <%=
            int j = 1;
            for (Pair<String, String> item : match.getCountries()) {%>
                <option value="<%=item.getRight()%>"><%=j++ + ".␣" + item.getRight()%></option>
            <%}
            i++;
            %></select><br /><%=
        }
    %>
    <input type="hidden" name="localTime" value="<%=match.getLocalDate()%>">
    <input type="submit" >
</form>

```

6. To monitor the active session it has been implemented a *HttpSessionListener* that implements *HttpSessionListener* and *HttpSessionAttributeListener* as it follows:

Listing 11: match.jsp

```

<form action="{pageContext.request.contextPath}/app/matchValidation" method="post">
@WebListener
public class SessionListener implements HttpSessionListener, HttpSessionAttributeListener {
    ...
    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        String username = (String) se.getSession().getAttribute("username");
        if (!Strings.isNullOrEmpty(username))
            SharedData.getInstance().getSessionsController().getActiveUsers().remove("username");
    }

    @Override
    public void attributeAdded(HttpSessionBindingEvent sbe) {
        String username = (String) sbe.getSession().getAttribute("username");
        if (!Strings.isNullOrEmpty(username))
            SharedData.getInstance().getSessionsController().getActiveUsers().put(username, true);
        /* This method is called when an attribute is added to a session. */
    }
}

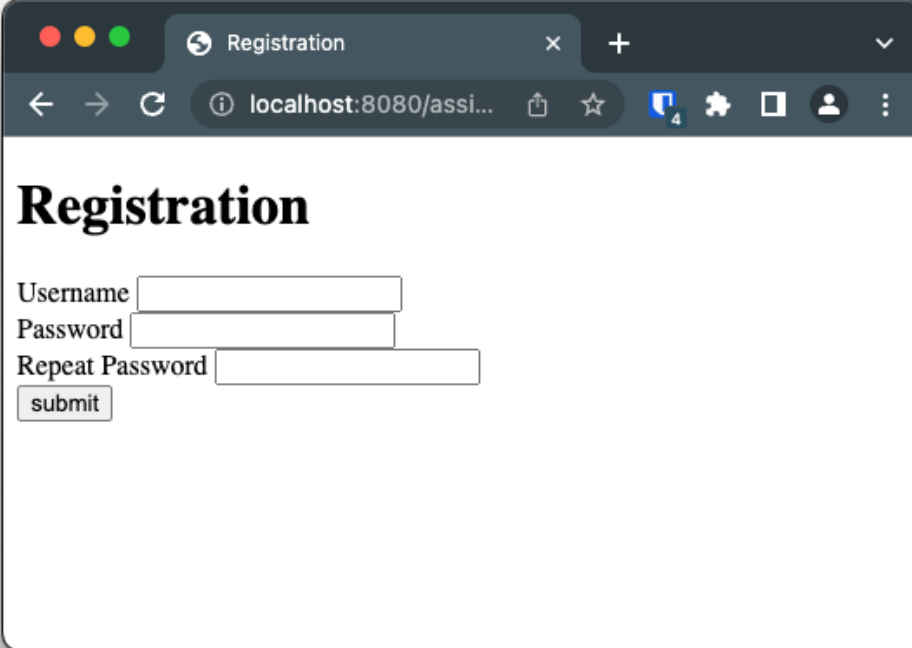
```

No more registration calls are required since the decorator *@WebListener* does everything.

7. As it can be seen in listing 1 the persistence of the data is kinda straightforward. When the program shuts all of the controllers serialize the data through the invocation of *serializeState()* and they load it up at launch invoking *deserializeState()*

App running

Here some screenshots of the app running. The UI is the same as shown in the assignment example.



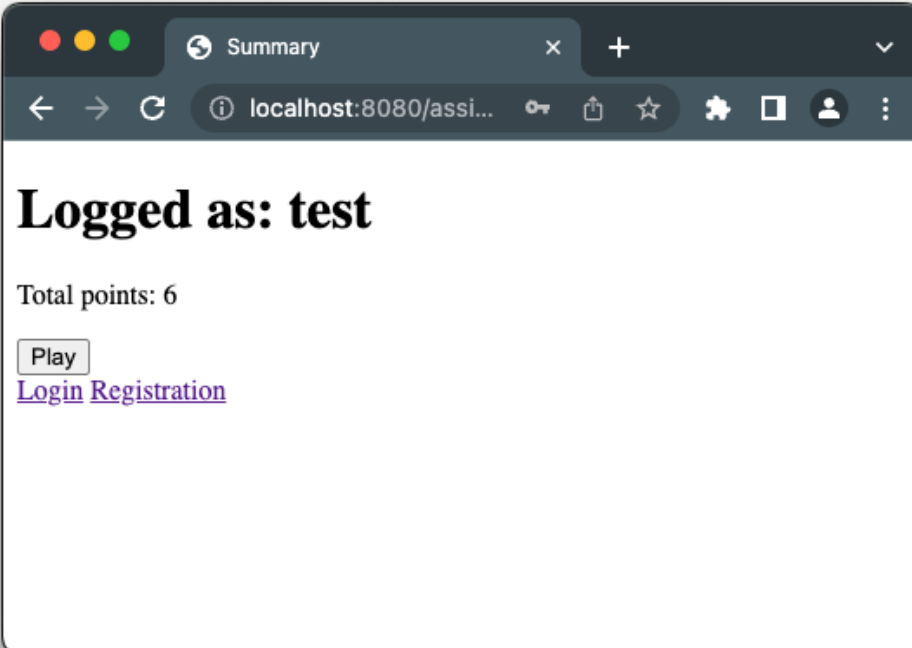
A screenshot of a web browser window showing a registration form. The browser's address bar displays 'localhost:8080/assi...'. The page title is 'Registration'. The form contains three input fields: 'Username', 'Password', and 'Repeat Password', each followed by a text input box. Below these fields is a 'submit' button.

Registration

Username

Password

Repeat Password



A screenshot of a web browser window showing a summary page. The browser's address bar displays 'localhost:8080/assi...'. The page title is 'Summary'. The page content shows 'Logged as: test' in a large font, followed by 'Total points: 6'. Below this is a 'Play' button and two links: 'Login' and 'Registration'.

Summary

Logged as: test

Total points: 6

[Login](#) [Registration](#)

Match

localhost:8080/assi...

Logged as: test

1. Algiers

2. Yerevan

3. N'Djamena

4. Prague

5. Djibouti


6. Libreville

7. Jakarta


8. Vilnius

9. La Valletta


10. Kiev



-- select an option --



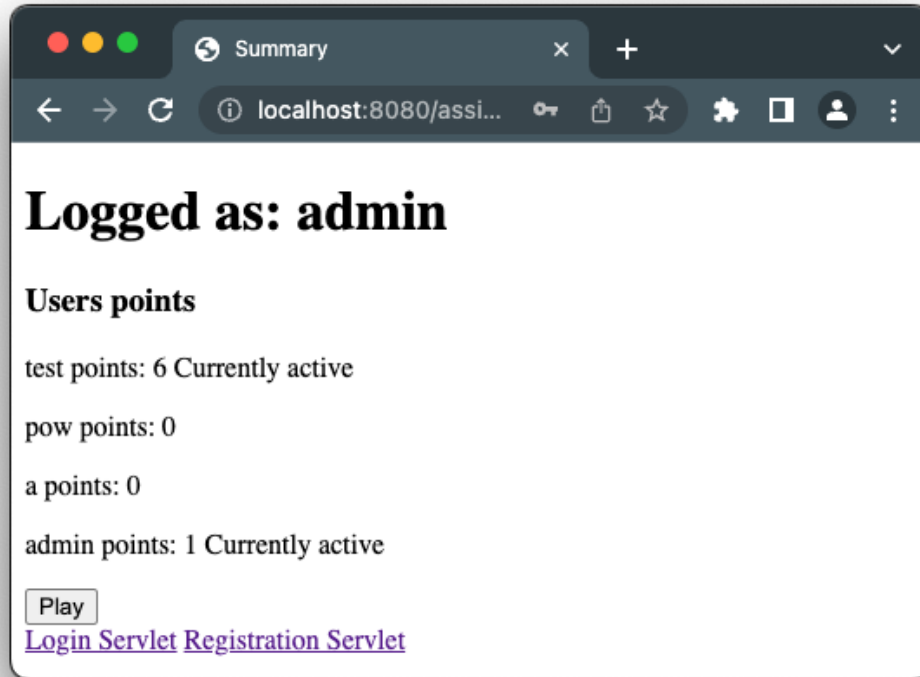
-- select an option --



-- select an option --

Submit

9



Comments

1. I'm happy about the structure of the code. It seems that it would scale up quite easy if new more feature may occur in a hypothetical future. The MVC pattern and the OOP gives me no doubts on where the instructions needs to be and they enforce a more rigid way of implementing solutions. Servlets seem a consolidate framework to build web digital product. I had some issue with the implementation of the Front-End code, JSP seems a bit clunky to use, even more when JavaScript is required. But in the other hand printing HTML in Java Servlet classes is unsustainable. I would love to see in the future new ways to write HTML with a Java Back-End codebase.