

**Web Platform Development 2**

**Coursework**

Submitted by:

Christopher Campbell – S1424416

Heather Reid – S1430421

James Hall – S1428914

“I declare that all work submitted for this coursework is the work of Christopher Campbell, Heather Reid and James Hall alone unless stated otherwise.”

**Contents**

[**Functionality** 1](#_Toc512805975)

[**Persistence** 3](#_Toc512805976)

[Application Methods 3](#_Toc512805977)

[Database Queries 4](#_Toc512805978)

[Database Schema 6](#_Toc512805979)

[Application Use Case 6](#_Toc512805980)

[Application Flow Chart 7](#_Toc512805981)

[**Design of Share Milestones Functionality** 8](#_Toc512805982)

[**Test Data** 10](#_Toc512805983)

[Integration Testing 10](#_Toc512805984)

[**Security Appraisal and Future Developments** 13](#_Toc512805985)

# **Functionality**

When implementing the web application, our main focus was including all the necessary requirements which were specified in the documentation. The first requirement that was implemented was the registration and login feature. This was to allow the users to be able to have their own username and password to gain entry to the system. We then proceeded to implement the adding and the viewing of the milestones. Once this stage was completed, we worked on editing and updating the milestones and setting their status. The final stage was to look at creating a link to allow users to share their milestones with their friends.

When implementing the web application, we ensured that all of the page names were short and simple for the user to read and understand. We also ensured that all of our URLs were directed from one domain (localhost on port 8080) to avoid any confusion. All of the URLs are consistent throughout the full web application with the only difference being the actual page name. We made sure that none of the URLs or pages displayed any confidential information regarding a user and ensured that the session IDs were stored as cookies. We also made sure that the URLs were free from non-ascii characters and symbols.

You can find all of our URLs below with a brief description of what each page does:

* <http://localhost:8080/register>

The above URL directs the user to the register page where they will be asked to enter a username and a password. Once they have entered these details, they will be logged in and redirected to the home page.

* <http://localhost:8080/login>

The above URL directs the user to the login page and is the first page the user will see. Once they have registered with a valid username and password, they will be able to just login to the web application and be redirected to the home page. If they have not registered and try to login, they will be refused entry to the web application.

* <http://localhost:8080/home>

The above URL directs the user to the home page of the web application. The user will not be able to gain access to this page by changing the URL, unless they have previously registered with a valid username and password and are logged in.

* <http://localhost:8080/messages>

The above URL directs the user to the Milestone page where all the different milestones which are stored in the database are displayed. The user must be logged in to the system in order to gain access to it. The user can then view their own individual milestones (using a URI) on a separate page, where they can view/edit/delete them. Examples of the structure of the URI can be found below:

* <http://localhost:8080/messages/Heather>?
* <http://localhost:8080/messages/Christopher>?
* <http://localhost:8080/messages/James>?
* <http://localhost:8080/addMilestone>

The above URL represents the page which allows users to add new milestones to the database. The user must be logged in to the system to gain access to it or is redirected back to the application login screen.

* [http://localhost:8080/shared\_milestone/](http://localhost:8080/addMilestone)6b9NBsBYt4

The above URL represents how milestones can be shared and viewed. Once a milestone is added, unique random characters are generated and stored with this milestone details. A link is then generated and can be followed to view this milestone by any user, even if they do not have an account in the system.

* [http://localhost:8080/update/](http://localhost:8080/addMilestone)6b9NBsBYt4

The above URL represents how milestones can be edited. Similarly, to the shared milestone functionality, this link gets the milestone related to this link and displays it on screen to update. If the user is assigned to this milestone, they can then edit/update this milestone. It is then saved, overwriting the previous data via ‘UPDATE’ query.

# **Persistence**

As this application includes multiple servlets interacting and passing data, application persistence becomes an issue. Within this application, injected data access objected were utilised to persist the data submitted from each page, along with a user’s JSESSIONID to maintain application state. The data access objects used in this application, provides a persistence layer by abstracting the database from the application. The DAO is injected or implemented to each Servlet class via the singleton design pattern, allowing only one instance of the class to be created.

Each servlet handles the given HTTP requests via doGet() and doPost() methods, sending the required data to the DAO to be created, updated, retrieved or deleted from the database. One a page loads with a requested URI (/update/dA4k08L) the servlet then processes the request using the doGet() method, returning the data in a list, then mapping the mustache tags to the given resource(s) requested. The doPost() methods on each servlet handle the submission of form POST requests, sending the data to the required DAO to process the information.

## Application Methods

The methods below are the data access objects (DAO) for the system, providing an interface for accessing the applications database layer.

@Override  
**public synchronized void** add(@NonNull String message, String description, String user, String expectedComplete, **int** actual, String link) {  
 Message m = **new** Message(**index**++, message, description, user, **d**.getTime(), expectedComplete, actual, link);  
 **messages**.add(m);  
}

@Override  
**public synchronized** List<Message> user(@NonNull String user) {  
 List<Message> out = **messages**.stream().filter(m -> user.equals(m.getUser())).collect(Collectors.*toList*());  
 **return** Collections.*unmodifiableList*(out);  
}

@Override  
**public synchronized** List<Message> link(@NonNull String link) {  
 List<Message> out = **messages**.stream().filter(m -> link.equals(m.getLink())).collect(Collectors.*toList*());  
 **return** Collections.*unmodifiableList*(out);  
}

@Override **public synchronized void** delete(**long** id) {  
 **for** (Message m : **messages**) {  
 **if** (id == m.getId()) {  
 **messages**.remove(m);  
 **return**;  
 }  
 }  
}

## Database Queries

The following SQL statement was used to create the users table:

**private void** initTable(Connection conn) **throws** SQLException {  
 *execute*(conn, **"CREATE TABLE IF NOT EXISTS users (name VARCHAR(255) PRIMARY KEY, hash VARCHAR(255))"**);  
}

The following code is used to insert a new users details into the table when they register:

String query = **"INSERT into users (name, hash) VALUES(?,?)"**;  
**try** (PreparedStatement ps = getConnection().prepareStatement(query)) {  
 ps.setString(1, userName);  
 ps.setString(2, hash);  
 **int** count = ps.executeUpdate();  
 ***LOG***.debug(**"insert count = "** + count);  
 **return** count == 1;  
}

The following code is used to check the users details when they try to log in to check if they are registered:

**try** (PreparedStatement ps = getConnection().prepareStatement(**"SELECT hash FROM users WHERE name = ?"**)) {  
 ps.setString(1, userName);  
 ResultSet rs = ps.executeQuery();  
 **if** (rs.next()) {  
 String hash = rs.getString(**"hash"**);  
 **return** hash != **null** && validate(password, hash);  
 }  
}

The following SQL statement was used to create the milestones table:

**private void** initTable(Connection conn) **throws** SQLException {  
 *execute*(conn, **"CREATE TABLE IF NOT EXISTS "** +  
 **"messages (id BIGINT AUTO\_INCREMENT, message VARCHAR(255), description VARCHAR(255), user VARCHAR(255), "** +  
 **"created BIGINT, expectedComplete VARCHAR(255), actual INT(2), link VARCHAR(255) PRIMARY KEY(id))"**);

The following SQL statement was used to insert the milestone details into the table when the user added a new milestone:

String ps = **"INSERT INTO messages (message, description, user, created, expectedComplete, actual, link) VALUES(?,?, ?,?,?,?,?)"**;  
Connection conn = getConnection();  
java.util.Date d = **new** java.util.Date();  
  
**try** (PreparedStatement p = conn.prepareStatement(ps)) {  
 p.setString(1, message);  
 p.setString(2, description);  
 p.setString(3, user);  
 p.setLong(4, d.getTime());  
 p.setString(5, expectedComplete);  
 p.setInt(6, actual);  
 p.setString(7, link);  
 p.execute();  
}

The following SQL statement is used to delete a milestone from the table when a user deletes it:

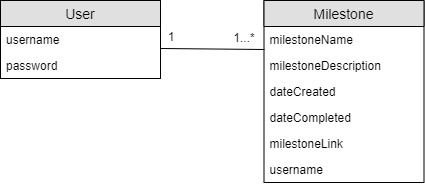
Connection conn = getConnection();  
String ps = **"DELETE FROM messages WHERE id = ?"**;  
**try** (PreparedStatement p = conn.prepareStatement(ps)) {  
 p.setLong(1, id);  
 p.execute();  
}

The following SQL statement is used to view a list of milestones:

String ps = **"SELECT id, message, description, user, created, expectedComplete, actual, link FROM messages"**;  
Connection conn = getConnection();  
**try** {  
 List<Message> out = **new** ArrayList<>();  
 PreparedStatement p = conn.prepareStatement(ps);  
 ResultSet rs = p.executeQuery();  
 **while** (rs.next()) {  
 Message m = *rs2message*(rs);  
 out.add(m);  
 }

## Database Schema

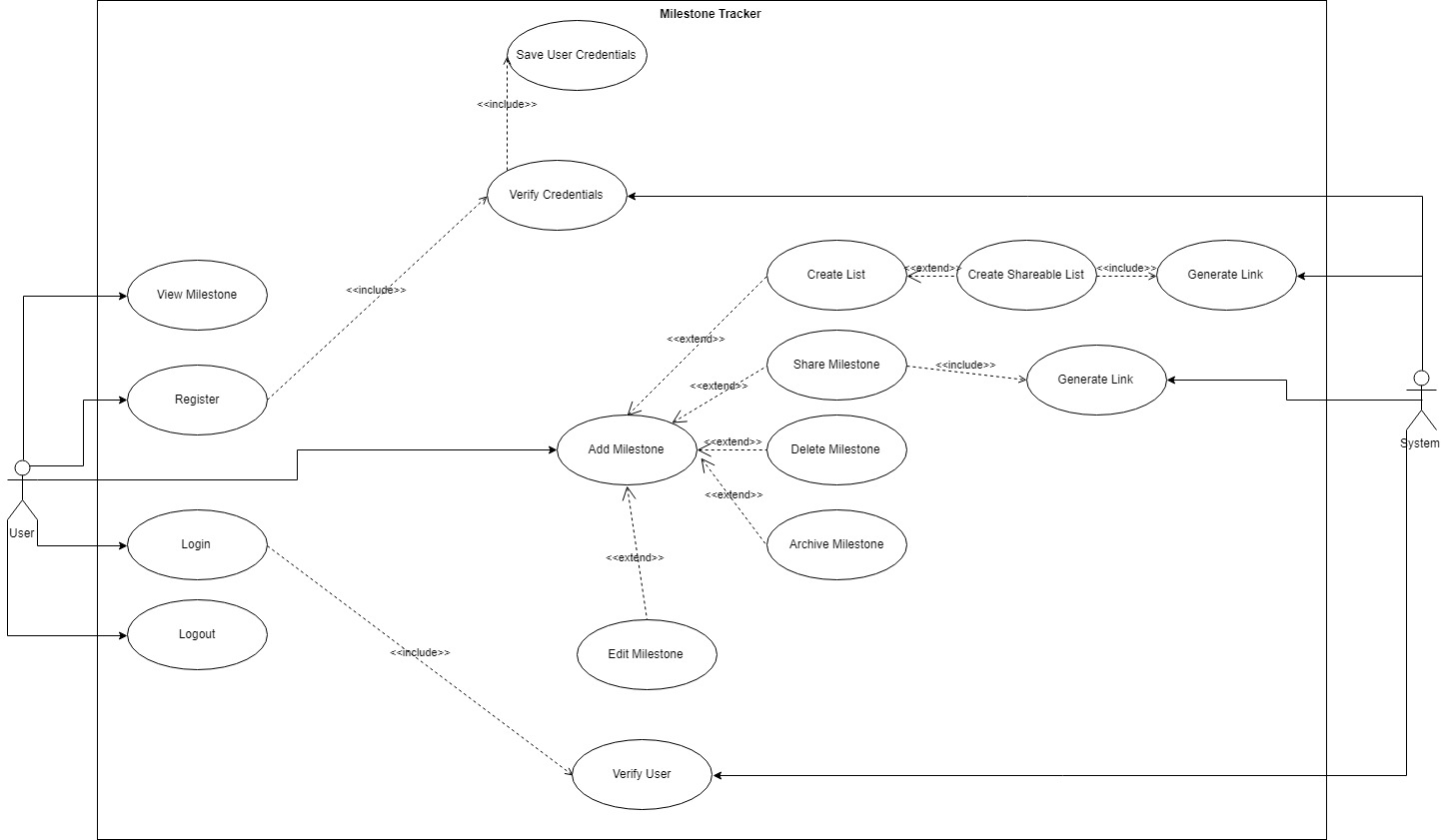
The database schema for data storage within this application is as follows:



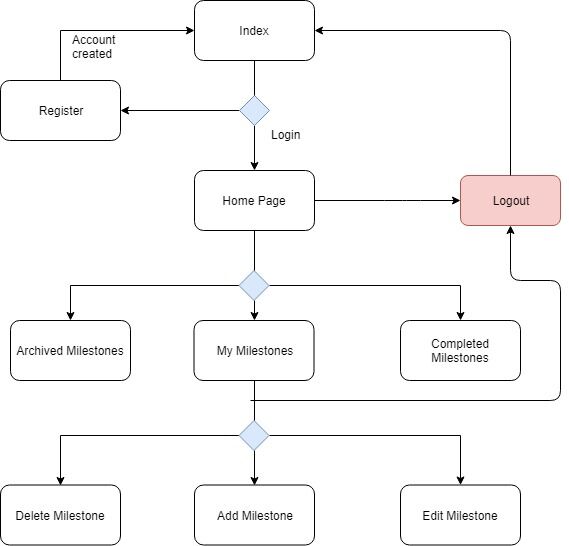
The users table consists of the username and password which is required to both login and register to use the application. The millstone table holds the milestone name, description, date created, date completed, unique link, and user who created the milestone.

From the above diagram, the database tables have been implemented as a one to many relationship; one user can have many milestones, but one milestone can only have one user. This provides additional application security as only one user can only ever perform actions, such as delete or update, on their personal milestones.

## Application Use Case



## Application Flow Chart



# **Design of Share Milestones Functionality**

The share links/milestone feature of the application allows you to share your milestone with friends. When the milestone is added, a random string is generated and stored with the entered milestone information. This random string then acts as a universal resource identifier for the milestone and allows the milestone information to be retrieved/viewed.

The random string is generated at client-side using a JavaScript function when the page loads. The function places the random string into an input box with a hidden property so it does not display on screen.

<script>  
  
 var aLink = Math.random().toString(36).substring(2, 15) + Math.random().toString(36).substring(2, 15);  
  
 const values = {  
  
 link: aLink  
}  
  
const keys = Object.keys(values)  
const length = keys.length  
  
for(let i = 0; i < length; i++){  
 const key = keys[i]  
 document.getElementsByName(key)[0].value = values[key]  
}  
  
  
</script>

Once the milestone information is entered and the form is submitted, the doPost() request, in the servlet, saves all the information into the milestone table, along with the random string.

A page was the created and routed allowing for the retrieval on the link URI e.g. localhost:8080/sharedmilestone/\*. Once this page is loaded the doGet() request looks for the milestone with this link as the identifier. The following function searches the database for the milestone where the link is equal to the given URI and returns it if found:

@Override  
public List<Message> link(String link) {  
 String ps = "SELECT id, message, user, created, link FROM messages WHERE link = ?";  
 Connection conn = getConnection();  
 try {  
 List<Message> out = new ArrayList<>();  
 PreparedStatement p = conn.prepareStatement(ps);  
 p.setString(1, link);  
 ResultSet rs = p.executeQuery();  
 while (rs.next()) {  
 Message m = *rs2message*(rs);  
 out.add(m);  
 }  
 return out;  
 } catch (SQLException e) {  
 throw new H2MessagesException(e);  
 }  
}

To allow the sharing of these milestones it was important to allow anyone to be able to view it, even if they do not have an account. So, the decision was made to leave the page public and not put any functions on the page which could potential cause hard.

The doGet() request on the servlet processes the link and if nothing is returned, it simple outputs a message alerting the user that not milestone exists.

@Override  
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 if (!authOK(request, response)) {  
 return;  
 }  
 String link = *linkFromRequest*(request);  
 String loggedInUser = UserFuncs.*getCurrentUser*(request);  
 if (db.link(link) == null) {  
 String err = "No such milestone: " + link;  
 issue("text/plain", HttpServletResponse.*SC\_NOT\_FOUND*, err.getBytes(Charsets.*UTF\_8*), response);  
 return;  
 }  
 Map<String, Object> map = new HashMap<>();  
 map.put("user", link);  
 map.put("matches", loggedInUser.equals(loggedInUser));  
  
 List<Message> messages = db.link(link);  
 if (messages.size() > 0) {  
 map.put("messages", messages);  
 }  
 showView(response, *LINK\_MESSAGES\_TEMPLATE*, map);  
  
}

In the table where the milestone information is displayed, it appends the link to the end of the applications URL mustache tags. This link can then be copied with one click and shared with whoever they wish to share it with.

# **Test Data**

## Integration Testing

|  |  |  |  |
| --- | --- | --- | --- |
| Login/Register | | | |
| **Action** | **Expected Result** | **Status** | **Evidence** |
| Localhost:8080 | Redirects the user to the login “Localhost:8080/login”. | OK - working as expected |  |
| Login/Register without input | Error message will prompt “Neither the username or password can be empty” | OK -working as expected |  |
| Password is wrong | Error message will prompt “The password is incorrect” | OK -working as expected |  |
| Logging in (User exists) | User is relocated to the home page “Localhost:8080/home” | OK - working as expected |  |
| Successfully Registered | User is relocated to the home page “Localhost:8080/home” | OK - working as expected |
| Navigational Bar | | | |
| Home | User is relocated to the home page “Localhost:8080/home” | OK -working as expected |  |
| View my milestones | User is relocated to the Milestones page “Localhost:8080/Messages” | OK - working as expected |
| Add a new milestone | User is relocated to the Add Milestone page “Localhost:8080/addMilestones” | OK -working as expected |
| Logout | User is relocated to the Login “Localhost:8080/Login” | OK -working as expected |
| Home Page | | | |
| No milestones | Message prompts on homepage “There are no milestones” Link to “Localhost:8080/addMilestone” | OK -working as expected |  |
| Milestones available | Lists the milestones, with details on display down the homepage. | OK -working as expected |  |
| Add a new milestone | | | |
| Add milestone | Milestone will be successfully added to the database | OK -working as expected |  |
| View/Alter Milestones | | | |
| View all milestones | “Localhost:8080/messages” displays all milestones from every user | OK -working as expected |  |
| Milestones for specific users | “http://localhost:8080  /messages/\*” displays all the results from the logged in user.  \* = Username | OK -working as expected |  |
| Delete | Delete button will remove specific milestone | OK -working as expected |  |
| Select Milestone to Update | “http://localhost:8080  //\*” Update button will move user to a separate page.  \* = Selected Milestone | OK – Working as expected |  |
| Update selected Milestone | Inputting information into the forms should allow the user to update the selected milestone. | NO – Not working as expected |  |
| Share Milestone | “http://localhost:8080  /sharedmilestone/\*” the URL generated with the milestone to view the specific record.  \* = Generated Link | OK -working as expected |  |
| Private Pages | | | |
| /messages | Accessing these pages whilst not logged in should return the user to “http://localhost:8080  /login” | OK -working as expected |  |
| /home | OK -working as expected |
| /update/\* | OK -working as expected |
| /addmilestone | OK -working as expected |
| /viewmilestone | OK -working as expected |

# **Security Appraisal and Future Developments**

To ensure the application is secure, detailed below is an appraisal of security measures currently implemented into the application and what could be added to increase security.

**Authentication**

The main security feature which the web application has is the login feature. This ensures that only registered users can access the system and each user has a secure password only known to them. Each user must enter their own username and password to ensure that they are who they say they are and to minimise the chance of someone gaining unauthorised access.

Requests are handled on each page from communication with the servlet. Each do and post method will only run if the ‘authOK’ method returns true. This helps prevent against unauthorised requests being entered into the applications database.

The JAAS SDK could be used to provide further authentication and authorization protocols to the system. The JAAS SDK provides authentication for users, determining who is currently running the system and authorization to ensure the users have access rights required to use the features of the application. This service allows users to be managed independently whilst extending system security.

**Password Hash and Salt**

Each username and password are stored in the user’s database table which then determines who can login. The password was hashed before being stored in the database to prevent it being stored in plain text and readable to anyone that has access to the database.

To ensure the password is completely secure and increase prevention against common attacks, such as dictionary attacks, the PBKDF2 hashing algorithm has been used along with a randomly generated password salt.

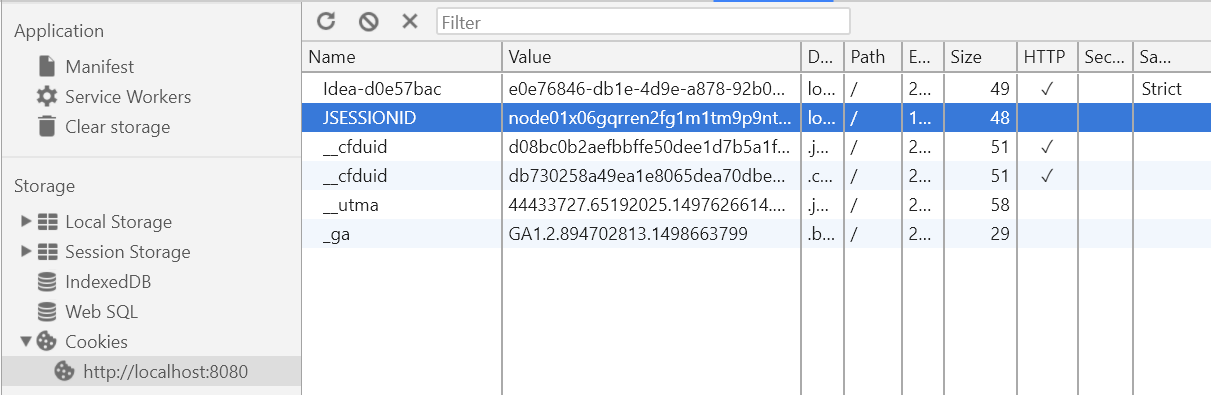
The PBKDF2 implementation uses the SHA-1 hashing algorithm to hash the passwords. Although this is a fast hashing algorithm, passwords can often be cracked quicker using brute-force attacks due to the speed of this algorithm.

However, as the password stored in this application utilises a random salt and iterates over the PBKDF2 algorithm a random number of times, this makes the application passwords extremely difficult to crack.

To further improve the security of the application, the PBKDF2 algorithm could make use of the SHA-256, which is a much slower hashing algorithm but is more protected against attacks like brute force password attacks.

**Sessions**

The web application also makes use of sessions which can be seen in the picture below. Sessions can be used to maintain the applications state between page requests and responces. This web applications stores sessions under JSESSIONID using cookies. If the JSESSIONID is deleted, the user is logged out of the system and is required to login again with their credentials. Once the user is logged out of the system the JSESSIONID is destroyed for that given user.



The web application also makes use of HTTP Requests. The hypertext transfer protocol (HTTP) is the core communication protocol used to access content over the internet. It is a message-based protocol that works by sending request messages to the server and receives response messages in return. Although the use of HTTP Requests is secure, the implementation of HTTPOnly would minimise the risks of unauthorised session hijacking which would increase the security by maintain data integrity and confidentiality.

To further increase the security of the application, session timeouts could be implemented to ensure that if the browser window is closed, the session is destroyed. The same could be applied for a given time of inactivity; the system would automatically destroy that session. This would help enforce session integrity, helping prevent against session hijacking and prevent any unauthorised users gaining physical access and viewing confidential information.

**Database**

The web application makes use of a database. The database is used to store the user login details and the milestone information. By using a database, the web application becomes more susceptible to database injection attacks (using SQL injection and NoSQL injection) which could lead to an unauthorised person gaining access to the entire database. To minimize the chances of this happening, we implemented user input validation to ensure that no malicious data could be entered into the system.

When querying on the database, parametrised queries were used. Parameterised queries provide a placeholder for values which are supplied when the request is executed; binding the values to the parameters before executing the code. This helps prevent against common SQL injection attacks. The application developed in this coursework, makes use of parameterise queries and has been tested against this type of attack.

To further secure the data being entered into the database, the data should be escaped and validated at client side, preventing any unwanted data reaching the database in the first place. Privileged users should also be created ensuring each user has least privileges to needed to perform each action.

**Future Security**

There are many more security features which we could have included in our implementation of the web application. The current applications security is fit for the purpose of this coursework, however, if fully deployed on the web, the considerations highlighted above would be essential in maintaining a high level of security for users of the application.