

Web Platform Development 2:

Group Report

Group H

“I declare that all work submitted for this coursework is the work of alone unless stated otherwise.”

Abdul Dar, Lyle Simpson, David Sumpster

2018

Table of Contents

[Introduction to the Project 3](#_gjdgxs)

[Data Layer & Models 4](#_30j0zll)

[Controllers and URL Mapping 6](#_1fob9te)

[View Layer & Mustache templates 7](#_3znysh7)

[Test Reports 8](#_2et92p0)

[Unit Tests 8](#_tyjcwt)

[Integration Tests 8](#_3dy6vkm)

[Test Scope 8](#_1t3h5sf)

[Test Cases 8](#_4d34og8)

[Security measures 15](#_2s8eyo1)

[Using Sessions 15](#_17dp8vu)

[Using Databases 15](#_3rdcrjn)

[Using Passwords 15](#_lnxbz9)

# Introduction to the Project

The Milestone application was built using the Model-View-Controller design pattern, for separating the data and the presentation layers of the application. It also leverages the Repository Pattern in the data layer, in order to abstract the data-handling logic into dedicated classes and away from the Controllers and Models. The application’s use of each layer of the MVC pattern is described in detail in the following section.

The application is a Spring MVC application, which uses an embedded Tomcat servlet in order to process requests and return responses. The application’s metadata and dependencies are defined in a Maven pom.xml file, and the project adheres to the required Maven folder structure. Development of the application occurred using the IntelliJ IDE, and the Git version control system was used to allow collaborative development.

# Data Layer & Models

The application uses the H2 in-memory database as its data store, giving the application persistence only within its lifecycle. After the application is shut down, the data entered into the H2 database is lost. For production, it would be essential to migrate this to a persistent relational database such as MySQL, PostgreSQL or Oracle, but to mitigate setup issues, it was decided to stick with H2 for this project. This database was seeded with data when the application started, with data entered representing the three main domain entities in the application. These domain entities are:

* User – to represent the person using the application
* Role – to represent the User’s authorization level within the application. There are two roles: USER and ADMIN.
* Milestone – to represent the milestone data that forms the core of the application.

These entities are represented in the codebase in the form of models, which are Java objects with appropriate fields mapping to the database columns. These fields are annotated with JPA annotations, which allow the Hibernate object-relational-mapper (ORM) to take care of mapping object relationships to database level relations. Data for each field in the model can be retrieved using typical Java getter and setter methods. These models have no knowledge of the controllers or views in the application – they simply represent the store of data, and can thus be used anywhere in the codebase.

The relationships between the domain entities can be summarized below:

* A user can have many roles, and a role can belong to many users (Many To Many relationship)
* A User can have many milestones. A single milestone can only belong to a single user (One To Many relationship).

Given that there exists a Many-to-Many relationship between a User and a Role, the application requires an additional table in the database to handle this relationship. For this, a junction table called ‘user\_role’ is created, with foreign keys referencing back to the User and the Role tables. This can be seen in the database schema which is in the Appendix. In addition, to handle the One-To-Many relationship between a user and the milestones they create, a foreign key is added to the Milestone table, with a link back to the user ID of the user that created the milestone.

The repository pattern is used to abstract operations on the database, such as retrieving, saving and updating data, away from the models and the controllers. Repository classes are created for each domain entity, and the application’s use of Spring Data conventions result in simple updating and retrieving of the data via methods such as ‘save’, ‘delete’ and ‘findAll’. These methods encapsulate the JDBC functionality required to access and change data in the database. The repository acts as the negotiator between the data-store and the controllers, with controllers calling the appropriate repository methods in order to get or change the data they need from the data-store. By using this pattern, the controllers never require knowledge of the persistence layer, and there is no dependency between the controller and the database dialect. This promotes the application’s ability to switch to a different database, as all that would need to be swapped out is the repository classes. As noted before, this is an important feature, as the application would require to use a persistent database if it was ever used in a production environment.

# Controllers and URL Mapping

The controllers are the mediator between the model layer and the view layer of the application. Incoming HTTP requests to the application are mapped to the appropriate controller based on the URL from which the request originates, and then in the controller class itself, the appropriate method processes the request. Each of these methods is responsible for returning the appropriate Mustache template after processing the request, as well as injecting any Mustache variables into the template. Each controller will also communicate with the data layer in order to process POST, PUT and DELETE requests. The controllers respond to submitted user input in the application’s templates, deciding whether or not the input is valid and how to process it.

There are 4 controllers defined within the application, and each defines methods for the HTTP verbs mentioned in parentheses:

* HomeController – responsible for handling the request for the home-page (GET)
* UserController – Responsible for handling login and registration (GET, POST)
* AdminController – Responsible for handling requests to Admin page (GET)
* MilestoneController – Responsible for handling requests to view, create, edit or delete milestones in the application. (GET, POST, DELETE)

Each controller class is marked with an annotation that defines the root URL for the requests to which it responds. For example, the MilestoneController will handle all requests for URLs beginning with ‘/milestones’, and inside this controller class, will define methods for handling the different requests that may be made to URLs extending this endpoint. As a demonstration of this, the URL to create a milestone is ‘/milestones/create’. The MilestoneController defines two methods associated with this endpoint. Firstly, there is a method to handle GET requests – this method returns the Mustache template with the form that is used to create the milestone. The second method for this endpoint handles POST requests, which occur when the user submits the form to create the new milestone. This method processes the inputted data, checking whether or not it is valid. If it is valid, the data is passed to the data layer, allowing the new milestone to be inserted into the database. Finally, the controller returns a HTTP redirect response, redirecting the user to the page that allows them to view their milestones.

# View Layer & Mustache templates

The view layer of the application is simply the Mustache templates which render HTML onto the page. These templates are comprised of normal HTML tags, combined with Mustache variables that allow injection of dynamic data, processing of lists of data, and conditional rendering of data based on attributes injected by Controllers.

Another feature of Mustache templates is the ability for a template to insert code from other templates, in order to promote reuse of code and cut down on duplicated view logic. This feature was leveraged in the application to define ‘header’ and ‘footer’ partial templates, which were responsible for dealing with features specific to every page in the application. The ‘header’ defined the HTML head and title attributes, as well as inserting all CSS stylesheets into the application. The footer was responsible for closing off the HTML body and html tags, as well as inserting the JavaScript files that were used within the application. By putting all this logic into partial templates, these templates could be included within every other Mustache template in the application, which helped to prevent mistakes, as well as cut down on code duplication.

# Test Reports

## Unit Tests

The application did not have a comprehensive suite of unit tests, due to time constraints towards the end of the development process. A few basic controller unit tests do exist in order to determine that the content of each page matches expectations, using basic assertions. However, it would be imperative to do more testing in the next iteration of the development phase, in order to provide safety when refactoring code and making changes. There are also additional tests for authorization within the application. These tests ensure that, when an anonymous, unauthenticated user hits the application, they are only allowed to view pages that are freely accessible, such as login and registration pages. Tests are in place to ensure that any pages that require authentication are not shown, and instead perform HTTP redirects to relevant pages such as the home page or the login page.

## Integration Tests

### Test Scope

The following Functional Tests were carried out:

1. Load application in browser
2. Register an account
3. Login to existing account
4. Add milestone
5. Remove milestone
6. Edit milestone
7. Share milestone link

### Test Cases

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1: Load application in browser** | | | | |
| **Pre-conditions**   * The user has imported the project into IntelliJ IDEA as a Maven project * The user has internet access, and a browser installed | | | | |
| **Step** | **Action** | **Expected Response** | **Pass/Fail** | **Evidence** |
| 1 | Run project from IntelliJ IDEA | Console shows log, application runs on port 8000 | **P** | https://i.gyazo.com/e7b8e22997e8bd3f919d9ec6afbba01f.png |
| 2 | Open browser, and navigate to localhost:8000 | Browser presents application homepage | **P** | https://i.gyazo.com/7c77bd93845ebeee869eec4dc39137a1.png |
| 3 | Open browser, and navigate to 127.0.0.1:8000 | Browser presents application homepage | **P** | https://i.gyazo.com/8165cdd694d864ee91136cf1557978a1.png |
| 4 | **Check post-condition 1** |  | **P** |  |
| 5 | **Check post-condition 2** |  | **P** |  |
| **Post-conditions**   1. User has application homepage loaded in browser 2. User is not logged in | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2: Register an account** | | | | |
| **Pre-conditions**   * The user is on the application homepage | | | | |
| **Step** | **Action** | **Expected Response** | **Pass/Fail** | **Evidence** |
| 1 | Click “Register” on homepage, or from menu | System displays registration form | **P** | https://i.gyazo.com/9a629e46eaaa47c4793ed32ccb31b9af.png |
| 2 | Complete form | Application checks entered data passes field validation, and username is available | **P** | https://i.gyazo.com/3b41b4f5f0e63798bb43d8885018a59e.png |
| 3 | Submit form | Application stores new user details, and displays homepage, ready to login | **P** | https://i.gyazo.com/7c77bd93845ebeee869eec4dc39137a1.png |
| 5 | **Check post-condition 1** |  | **P** |  |
| **Post-conditions**   1. New user details are saved in the database | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **3: Login to existing account** | | | | |
| **Pre-conditions**   * The user is on the application homepage * The user has an already existing account | | | | |
| **Step** | **Action** | **Expected Response** | **Pass/Fail** | **Evidence** |
| 1 | Enter login details | Application displays username and hides password | **P** | https://i.gyazo.com/c59545c4e4f987722d315c884e55eee4.png |
| 2 | Click “Login” | Application checks user exists, welcomes user, removes login/register options, adds logout option, adds admin page link if admin | **P** |  |
| 3 | **Check post-condition 1** |  | **P** |  |
| **Post-conditions**   1. The user is logged in | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **4: Add Milestone** | | | | |
| **Pre-conditions**   * The user is logged in | | | | |
| **Step** | **Action** | **Expected Response** | **Pass/Fail** | **Evidence** |
| 1 | Click “Create Milestone” on homepage, or from menu | Application displays create milestone form | **P** | https://i.gyazo.com/c0f1f4be1c4259bd4c5c3b198b29bb2e.png |
| 2 | Complete form | Application checks field validations | **P** |  |
| 3 | Submit form | Application stores new milestone details, and displays milestone list | **P** | https://i.gyazo.com/77ebe6e4bdce4c6c2bf80001f5b96231.png |
| 4 | **Check post-condition 1** |  | **P** |  |
| **Post-conditions**   1. New Milestone is created | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **5: Remove Milestone** | | | | |
| **Pre-conditions**   * The user is logged in * The user has already created milestone(s) | | | | |
| **Step** | **Action** | **Expected Response** | **Pass/Fail** | **Evidence** |
| 1 | Select milestone to be removed | Application displays milestone details | **P** | https://i.gyazo.com/d433d23206ac747801aefc3c2e39d64d.png |
| 2 | Click “Delete milestone” | Application asks user to confirm deletion | **P** | https://i.gyazo.com/48b45854630a11ee0de7ec834072f11f.png |
| 3 | Confirm deletion | Application removes milestone from current project | **P** | https://i.gyazo.com/416eaa59810cd5ad77eec00541bf31a3.png |
| 4 | **Check post-condition 1** |  | **P** |  |
| **Post-conditions**   1. Selected milestone is removed | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **6: Edit Milestone** | | | | |
| **Pre-conditions**   * The user is logged in * The user has already created milestone(s) | | | | |
| **Step** | **Action** | **Expected Response** | **Pass/Fail** | **Evidence** |
| 1 | Select milestone to be edited | Application displays milestone details | **P** | https://i.gyazo.com/53718d87855fa8ad19c9dc6d40d734ab.png |
| 2 | Click “Edit milestone” | Application displays edit milestone form | **P** | https://i.gyazo.com/907614021df74b33cce90c54390667bd.png |
| 3 | Edit form | Application checks field validations | **P** |  |
| 4 | Submit form | Application updates milestone details | **P** |  |
| 5 | **Check post-condition 1** |  | **P** |  |
| **Post-conditions**   1. Selected milestone is updated | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **7: Share milestone(s) link** | | | | |
| **Pre-conditions**   * The user is logged in * The user has already created milestone(s) | | | | |
| **Step** | **Action** | **Expected Response** | **Pass/Fail** | **Evidence** |
| 1 | Milestone can be marked as public | Milestone is public | **P** |  |
| 2 | Copy sharing link | System adds link to clipboard | **P** |  |
| 3 | **Check post-condition 1** |  | **P** |  |
| **Post-conditions**   1. Sharing link stored to clipboard | | | | |

# Security measures

The application at hand allows users to create an account with a username and password and requires a login to access all functionality. This in result will require security measures to ensure client information is kept safe. Although this is a milestone application, one may assume information stored will not be sensitive but ethically you must assume the client’s information is private and confidential and as a developer must keep it safe.

## Using Sessions

The server uses sessions to allow multiple people to login and utilise the application simultaneously. It does reveal the java session ID through cookies in application console in the browser. However, if the project requirement stated security then HTTPS would be implemented or a 2-cookie system to validate no data was leaked when using public networks to access the website. If the cookie is deleted from the console and the user is not logged in nothing will happen but, if they are logged in they would be logged out. Although that doesn’t seem like a security threat in that situation it really is, as on mac OS you can copy the cookie and Google Chrome has cookie related applications in which you paste the cookie, but it updates to refresh the site to the previous state in which it was therefore someone who clones the cookie could potentially login under that session. To prevent this, IP logging would be implemented to make sure the users IP matches that of what he logged in with, and if the IP changes he would be logged out and blocked ideally from the website. This could be made more secure as by using https to ensure a secure link for the login page, but it would cost a lot of time and money and this is only coursework which doesn’t require these features therefore they haven’t been included.

## Using Databases

The H2 database is protected against SQL injection and the user is also protected against Cross-Site Request Forgery. Furthermore, pages are protected against non-logged in users, this is to ensure no user can access the application without an account and to make sure that each specific user can only view their own millstones and guest can be linked in by a shared link.

JavaScript client-side validation has been used to ensure that the client’s password is complex enough and is repeated to make sure it matches. Along with that, the client username is forced between 6-35 characters containing only letters, number and underscores while the password must contain a lower and upper-case letter with at least one number included.  Next it will check to determine whether or not the username is different from the password to be able to register an account all checks/requirements must be met. **We use regular expression.**

As the database is volatile in nature meaning if you stop running it in IntelliJ it will not have saved any milestones created by the user unless hard coded. Thus meaning data is erased upon closing IntelliJ, however in a real world application this would not occur as information will need to be retrievable.

## Using Passwords

The admin passwords are stored as plain text hardcoded into the application due to time constraints and main reason being it’s easier to implement it this way. Although this is not the most secure way to implement this, if there was more time, it wouldn’t be hard coded in and would be stored offline in a secure room.

Currently the client passwords are stored using bCrypt, it hashes the passwords slowly making them one way only ensuring they can’t be read while also requiring a vast amount of processing power to be able to guess/hack them. In order to improve this, they would need to be salted as well in order to enhance security making it more difficult for anyone to guess the password. As no password even if they are the same will appear the same once hashed and salted whereas if its only hashed the has code will be the same for that password.

What can be done to make an application more secure?

* Use a https protocol to make user login more secure as it over the internet
* Using web security tools for vulnerability checks and testing sql injection etc. Netsparker and OpenVas
* Have security plugins in website