PGCertIT Final Project Reflection

Matthew Ding
Auckland ICT Graduate School
University of Auckland
Auckland, New Zealand
mdin874@aucklanduni.ac.nz

I. System Architecture

The backbone of our system is formed by two back-end technologies – Node.js using the Express framework, and DB Browser for SQLite. Node.js is responsible for running the whole web app, while DB Browser is used to create the web app's database using SQLite. The front-end of our web app is created using HTML, CSS, and JavaScript.

In our Node.js web app, we are using Handlebars to produce HTML views for each page, which are then sent to and rendered by the client. These views are inserted into a layout file containing the header and navbar, which is constant across every single web page. We have three main route handler files – app-routes.js, auth-routes.js, and api-routes.js. The route handlers within these files handle a range of GET, POST, and DELETE requests, and carry out a variety of tasks, such as rendering a Handlebars view, calling one or more DAO methods, sending JSON data back to the client, or all of the above. Notably, the route handlers within api-routes.js only ever receive requests from the Java Swing app from the individual project, and never from the website. In our public folder we store the user avatar images, CSS, and client-side JavaScript files which enable certain local functions such as checking existing usernames when creating an account. The file which contains the middleware functions is stored in its own folder. All of our main configuration (e.g., setting up Express, Handlebars and routes, and starting the server) can be found in the app.js file, and all of the dependencies (e.g., bcrypt for hashing and salting of passwords and jimp for image processing) are specified in the package.json file.

The web app also contains an SQL file which is used to create the database. In the database, we have four main tables – these are called users, articles, comments, and votes. The web app interacts with the database using DAO (data access object) methods, which are called by various route handlers and are contained within two files – app-dao.js for DAOs that the main application uses, and auth-dao.js for DAOs that are used for authentication. These DAO methods utilise the sqlite npm package to run SQL statements and retrieve relevant data. The DAO files are stored within the modules folder, which also contains any image uploads, the database.js file (which requires the sqlite package and creates the dbPromise which can then be used by the DAO methods to access the database), and the config file for Multer (which allows for image file uploads to the web app when creating a new article or editing an existing article).

II. Individual Contributions

In terms of my contributions to the project, on the front-end I was responsible for the overall CSS and styling of the website. This encompassed the colour scheme, font choice, header and navbar, headings, page layouts, and implementation and design of article cards. Within the navbar, I implemented anchors to allow for navigation around the site and added functionality so that when the user is logged in, they will see different links in the navbar (create article, my articles, my account, log out) to when they are logged out (log in and create account). This was achieved using helpers in the overall Handlebars layout for the website. I also created the log in and change password Handlebars views. For the create account, edit account, and change password views, I implemented local checking so that if the user enters a username that's already taken (i.e., already in the database), they will be prompted to choose another username. The user also has to enter two matching passwords, otherwise they will be prompted to enter two that match. These two functions operate without having to submit the form first, and the submit button is disabled until they have entered both a valid username and matching passwords. This code was written in three client-side JavaScript files which run in the above views using the <script> tag.

With regards to the back end, my main contribution was implementing authentication into the website. This included adding the middleware functions (and implementing them across various route handlers), authentication routes, authentication DAO methods, and authentication token and cookie generation. I was also responsible for writing and testing all of the API route handlers, creating various DAO methods, and implementing the hashing and salting of passwords into the authentication routes using bcrypt methods that my teammate Hollie had rewritten.

Some of my other general responsibilities included acting as the maintainer for the shared master repository, which involved reviewing and approving all merge requests and resolving merge conflicts, tidying up the project code before submission, adding comments, and organizing the file structure of the project so that all files were appropriately named and in appropriate folders.

III. Individual Project

The individual project was a very daunting prospect initially – while I am quite comfortable with web programming, Java has been something that I've struggled with for a significant portion of the course, and I've constantly felt like I've been playing catch-up. Due to this, I had to spend some time

relearning Swing and SwingWorker, which were the main tools we would be using to create our Java app.

I started off by creating three separate classes – one for the app itself (AdminApp), one for the GUI JFrame (AdminJFrame), and one for the GUI JPanel (AdminJPanel). Together, these produced a simple GUI which acted as the starting point for the app, containing all of the basic components (e.g., JLabels, JTextFields, JButtons) save for the JTable. As I went through each topic, I would modify the app to incorporate the key ideas and concepts that I had just learnt, and I found that this approach allowed me to construct the program in a clean and systematic way that made sense to me. Eventually, all of the code for constructing the GUI was moved into the AdminApp class.

Sending the HTTP requests to the server presented a big challenge, as we had never done this before. Fortunately, Dr Andrew Meads provided us with an example program that contained the required code for achieving this, which simply needed to be modified to send these requests to our API route handlers and handle the response accordingly. The four methods for making these HTTP requests were contained within the API class. The conversion of the response JSON string into a Java object or list of objects was handled by the Jackson library, which was also provided.

When it came to SwingWorker, I needed to identify which tasks were time-consuming and needed to be carried out by a SwingWorker – these were the POST request to api/login, the GET request to /api/users, the GET request to /api/logout, and the DELETE request to /api/users/:id. As such, I created four different SwingWorkers, one for each of the identified time-consuming calls to the web app. Each SwingWorker called a different API method, which would make an HTTP request to the appropriate API route handler on the server-side. The done() method for each SwingWorker would then make the appropriate changes to the GUI, such as populating the table once logged in, deleting the selected row when the delete button is pressed, or clearing the table once logged out. These SwingWorkers were created and executed in response to an action event, such as the log in button being clicked.

Creating the table which would display all of the website's users and user information proved to be the most challenging aspect of the project. At first, the goal was to only display the table if the user had successfully logged in, however I ran into a lot of trouble when trying to create the table and append it to the JPanel after the log in button was clicked. As such, I decided to instead create an empty table using a table model with just the column names first, which would be visible when starting the program. Once the user was logged in, the table would then populate with all of the users. Upon logging out, this table would then be cleared. I ran out of time to implement the Adapter and Observer patters – when a user was deleted from the database, I simply retrieved the table model and removed the selected row from it.

To arrange the GUI components, I used a combination of BoxLayout and BorderLayout – BoxLayout for individual JPanels (e.g., a JPanel containing the username label and username input field, arranged horizontally in a single line), and BorderLayout for the overall content pane (e.g., log in inputs and buttons at the top, table in the centre, and delete user button at the bottom).

In our Node.js web app, I realized that I had missed a few things while writing our API routes. The first issue was that when the route handler for /api/login received a POST request from the Swing app and tried to retrieve the form values using req.body, it was returning an empty object {}. This was because I had forgotten to add Express's built-in JSON parser to our app.js file, which converts JSON strings into JS objects. Once I added this, I was able to retrieve the inputted username and password from the POST request and authenticate the user. However, I then ran into another issue – when I tried to log in as a non-admin, this caused the Swing app to stop working, instead of just displaying an error message as was intended. While examining my API route handler code, I found that despite adding code to check if the user is an admin in the other route handlers, I had neglected to include this in the /api/login route handler, which is arguably the most important place to have it. After adding an appropriate if statement, this issue was resolved.

IV. Topics Taught In Class

The three main topics taught in class that were utilized extensively throughout the project were Node.js and Express, Handlebars, and SQLite.

Node.js (using the Express framework) was used to run the entire web app, and within Node.js we used npm to install various packages (e.g. bcrypt, uuid), route handlers for different pages and requests, serving of static files (e.g. CSS, client-side JavaScript, avatars), sending of JSON data (e.g. all users in the database to the Swing app), receiving query parameters from GET requests (e.g. sorting articles), receiving form values from POST requests (e.g. log in, create account, edit account, change password, create article, and edit article forms), modules for our different DAO methods, and routers for our different groups of route handlers (e.g. app-routes, api-routes, auth-routes).

Handlebars was used to display all pages of the website – each page had its own view, which would be inserted into a consistent overall layout. Within each view, we used a range of HTML concepts, such as tables, anchors, images and forms. Helpers also came in handy for only displaying certain parts of the web page if the user was logged in, such as the navbar, which would display more options if the user was logged in. They were also used to iterate through arrays and display each element on the page, which could be seen in the home page where all articles in the database were displayed.

SQLite formed the foundation of our project, which was started by creating an ER (entity-relationship) model based on the brief, and then mapped into a relational model which was subsequently converted into SQL. Within the SQL,

DDL (data definition language) was used to create tables and establish primary keys and foreign key relationships, and DML (data modification language) was used to insert data into these tables. We also ran SQL from Node.js using the sqlite npm package in order to retrieve data from the database. The methods that interacted with the database were grouped into DAO modules, and used the sql-template-strings package to allow the safe use of template literals in the actual SQL statements. DB Browser was the program of choice to run the SQL and create the actual database.

Other miscellaneous topics covered in our project were CSS (colouring, borders, web fonts, selectors (ID, class, type, DOM, pseudo), box model, transitions (used with the hover pseudo-class), responsive design, grid, and flexbox), JavaScript (modifying HTML elements, button clicks (event listeners), functions, async functions, await, and fetch), Multer for image uploads for articles, fs for renaming and moving these images, JIMP for processing these images, and cookies for login authentication.

V. Topics Not Taught In Class

In order to fulfil the requirements of the project brief, the team had to research and implement a number of technologies which we had not been taught in class. The first of these was authentication, which encompassed three key areas – hashing and salting of passwords, authentication tokens, and middleware.

Hashing of passwords is a popular method of protecting passwords in storage and involves using a one-way algorithm to convert a plaintext password into an indecipherable string, which is then stored in the database. The problem with hashing on its own is that two passwords that are the same (e.g., "abc123") will produce the exact same hash if put through the same algorithm. Hackers are able to exploit this by creating a table of common passwords and their hashes, and then running these against the database – if a hash in the table matches a hash in the database, they can use their table to identify the plaintext password. As such, salting is often used alongside hashing, where a randomly generated value is added to the plaintext password before hashing, producing different hashes for the same plaintext passwords.

Initially, we were storing user passwords as plaintext in the database, however we quickly learned that this was a terrible idea, as anyone who gained access to the database could then log into any user account. Using the npm package bcrypt, we were able to implement hashing and salting into our web app. When a user creates an account, or when they change their password from the edit account page, the plaintext password is fed into a bcrypt method which generates the hashed and salted password and stores it in the users table in the database. When a user attempts to log in, their inputted password is compared against the hash for the inputted username using another bcrypt method, and if they match the user is logged in.

In order to achieve log in functionality, we needed a way of tracking a logged in user over all pages of the website, which is where authentication tokens and cookies came in. An authentication token is simply a unique, randomly generated string that has limited use for us until it is turned into a cookie. Our web app utilizes the UUID npm package to generate an authentication token for a user when they create an account or log in. This authentication token is then stored in the database under that user, and a cookie is created from it, which is the key component that allows us to identify whether a user is logged in or not.

The part of our web app that is able to use this cookie is the middleware. Our middleware file contains two methods – addUserToLocals() and verifyAuthenticated(). addUserToLocals() runs on every route handler before the route handler's function, and uses the method retrieveUserWithAuthToken() to retrieve a user from the database according to the authentication token stored in the cookie, and adds them to res.locals.user before carrying out the route handler function. The route handler and Handlebars can then check if res.locals.user exists to determine whether the user is logged in or not, and whether to display certain content on the page. If there is no cookie, then no user is retrieved and the user is treated as if they are logged out. verifyAuthenticated() prevents access to a certain route if there is no user in res.locals.user, and redirects them to the login page.

While writing the API route handlers, which would interact with our individual Java Swing apps, I learnt how to create a DELETE route handler and how to receive HTTP DELETE requests. Response codes were also a topic that I was unfamiliar with, however after doing some research I was able to successfully send back response codes from the web app to the Swing app.

The create article and edit article pages of our website necessitated the use of a WYSIWYG (what you see is what you get) editor, which would allow the user to edit the format of the article without having to directly edit the HTML markup. We decided on TinyMCE as the editor of choice and were able to implement it through the use of internal JavaScript in the appropriate Handlebars views.

Comments in articles represented the most significant challenge to the team, as they utilised several concepts that we were unfamiliar with. Because our website had to support up to two levels of nesting, each nested comment had to store a reference to its parent comment's ID in the database. Upon retrieving all of the comments for an article in a one-dimensional array, we then had to convert this into a two-dimensional array to reflect the proper nesting of the comments and all parent-child comment relationships. Within the individual article view, we could then iterate through this 2D array to display nested comments.

VI. Working In A Team

Working in a team for this project has been a very pleasant experience. Despite never working together before, all of the

team members got on really well, and we never had any major disagreements or arguments, either personally or with regards to the project. Every team member carried their weight, and displayed a positive and proactive attitude, which contributed to a productive working environment.

To me, the main advantage of working in a team has been collaboration. The opportunity to work together every day in person meant that it was easy to communicate with the other members if there were any major bugs with the code, which subsequently made it easier to resolve, as every member was always willing to jump in and lend a helping hand. Furthermore, having daily stand-up meetings each morning helped keep us on track, and ensured that we were on the same page at every step of the project. The ability to divide up tasks between multiple people and trust that they would get them done also took away a lot of the stress that comes with having to do everything on your own, and meant that we completed tasks a lot faster. Furthermore, each member offered a unique skillset and perspective that benefitted the entire team, allowing them to approach problems in different ways and offer new potential solutions.

However, we did encounter a few difficulties, the most notable of which was managing our Git workflow. Having never programmed with other people on the same project before, using Git collaboratively was something we hadn't encountered before, and it took a bit of time and effort to set things up and ensure that everyone was familiar with the sequence of commands. Common issues that arose were making changes to the local master branch, forgetting to create a new branch for each feature, and forgetting to pull new changes from the upstream master and merging them into the completed feature branch before pushing and submitting a merge request to the shared repository. Merge conflicts were frequent in the early stages of the project and caused quite a few headaches, as we were unsure if these were normal or not, however we eventually became for more adept at resolving them. By the end of the project, the whole team was very comfortable with using Git in a collaborative setting, and this is a skill that I'm sure will be hugely useful if or when we decide to enter the industry.

One drawback of dividing up tasks between the members that became more and more apparent as our web app grew more complex was a lack of understanding of what other members had worked on. At the start of the project this wasn't an issue, as we were simply creating the core Handlebars views and route handlers. However, as we began to implement more complex features into the web app (such as comments and sorting articles), it became harder and harder to comprehend what the code was doing, especially if only one person had worked on it and had used programming techniques that were unfamiliar to the rest of the group.

It also would've been helpful to have had a defined team leader or project manager, as for the most part we were working without a clear road map and didn't have that one person who was looking at the overall picture of the project and making the tough decisions. Despite this, the relatively small size of the team meant that we were able to manage without too many issues.

For a lot of people, the mere mention of a "group project" can cause significant stress and fear – however, I only have positive things to say about my group and the work we were able to accomplish. Looking at the final result, I am proud to say that we were able to accomplish most of the requirements of the project to a high standard. The past three weeks, and the PGCertIT course overall, have been a truly valuable learning experience, and I hope to be able to keep developing the skills that I've learnt to enter the industry and become a better programmer going forward.