CS3099 Deliverable 2 - Group Report

Team 3 - Supergroup C

November 2021

Supervisor	Git Master	Supergroup Rep	Scrum Master	Test Master	Product Owner
Dr Edwin Brady	190003657	200012696	190020048	200013403	190013199

`README.md`	Dependencies and guidance for installation, configuration, testing and running project code.
`Appendix/TestEvidence.pdf`	Test evidences.
Project Wiki	https://gitlab.cs.st-andrews.ac.uk/cs3099group03/project-code/-/wikis/home.
`Scrum-board.xlsx`	Scrum borad.
`weekly_progress_reports.xlsx`	Weekly Progress Report.
Team Responsibilities	See individual report.

1 Overview

The purpose of this deliverable is to provide a progress report, and a "Minimum Viable Product" to demonstrate that we will be able to implement the goals of the project. As of submission we have a cohesive and working implementation that can register a user to our journal, allow them to login (and log out), when logged in allow them to post source code files - which are then displayed on the homepage, and allow users to write comments for posts. Additionally the protocol for migrating content between coding journals has been implemented.

We have followed the scrum development model and created a well documented and structured implementation with strong focus on readability while maintaining full compatibility with supergroup requirements. Lastly, careful thought has been put into testing and, as far as possible, testing was conducted via the mixed means of unit testing, debugging, UI testing, and API testing (via postman).

2 Scrum

2.1 Organisational Practice

The team adhered to the Scrum development model for project organisation. Each sprint lasted for a week. At the beginning of a sprint cycle, the product owner would prioritise the items on the product backlog to define the most important items to be developed in that cycle. Completed tasks are checked off the list. Items would be returned to the product backlog if they could not be completed within the allocated week for the sprint. The Scrum master will then organise a follow-up review to assess the complexity of the task which is not completed in the given time frame. The model below shows our revised sprint cycle. The input to the process is the product backlog and each process iteration produces a shippable software increment.

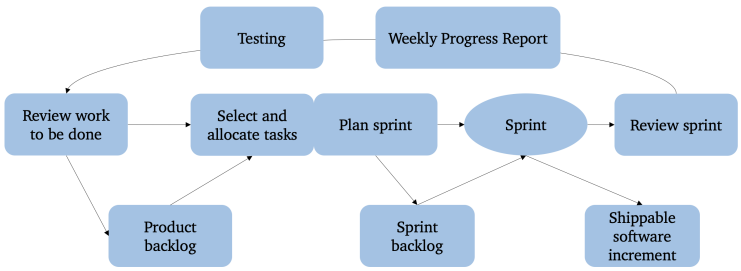

Figure 1: Revised sprint cycle

2.2 Scrum Board

The Scrum board was maintained on Microsoft planner. For example, the implementation of posting new source code files would be assigned to members from the sprint backlog. As the team designed and developed the sprints, tasks would be moved from the product backlog to the sprint backlog. User stories on our board were carefully taken into consideration, as they contain insightful descriptions of features from our business users' perspectives. This allowed us to break the product features down into a set of manageable and understandable blocks that stakeholders can relate to. Tasks were marked as completed after testing and progress reviews.

2.3 Weekly Progress Reports

The "Weekly Progress Bullet Journal" mentioned in Deliverable 1 was used to compile Weekly Progress Reports. Before each Scrum meeting, every team member would fill out the progress form by Monday. The form includes three short questions relating to an individual's progress. The reports serve as reliable documentation for tracking the project progress.

2.4 Scrum Meeting Reflections

Reflecting upon the weekly Scrum meetings, the whole team was involved in selecting the highest priority of tasks we believe could be completed. With the team's collaborative effort, we came up with estimates of how much product backlog our team can cover in a single sprint. Understanding our team's velocity is crucial as it provides a basis for measuring improving performance. Microsoft Teams and Discord were used to conduct real-time communication between members for informal communication, particularly instant messaging, and video calls. All team members would describe their progress in detail since the last meeting and bring up any problems that are raised. Everyone participates in short-term planning and there is no top-down direction from the Scrum master. At the end of the sprint, there would be a follow-up discussion or a review meeting which allows the team to reflect on how things could have been done better and ensure the deliverable will be completed on time.

3 Implementation

The Implementation of our Minimum Viable Product is divisible to 'back-end' and 'front-end' and both sections are designed to work without directly accessing any files on the others' file space. The back-end consists of an `express.js` server and an SQL database, while the front end is our react app and it's corresponding components. The project has been broken down into smaller sections (as far as reasonable) and thought has been put into reusability of code with reusable, or potentially reusable, code put into a `utils` (utility) directory allowing ease of access. The idea was that throughout development, a substantial library of reusable code emerges helping us reduce duplicate code on a project wide scale and thus saving valuable time.

The core reasoning behind designing our project the way we have was to allow for as much modularity as possible, hence reducing the potential for merge conflicts and other similar issues. We were able to work together towards the same objectives separately, with one persons work able to be plugged in to another just like many external dependencies. This came with a risk of increased difficulty when combining complex functionality, however, we were able to largely avoid such issues via good internal comments and a wiki providing additional detailed insight. The largest problems arose when code was not properly tested or turned out to be broken/incomplete and we experienced some hiccups with this. We concluded that a suite of unit tests for client side functionality in particular is a suitable and much needed solution to this problem.

Currently our code is designed to run on `localhost` (5000 for server and 3000 for react - though ports can be changed manually if needed) while the database is hosted on the school server by a member of our team. We have not yet made use of school systems such as Nginx for either server or client due to the mixed status of on/off campus working we are left to deal with. Having said that however, our minimum viable product can be deployed on school Nginx with minimum configuration if localhost is not a suitable medium for demonstration. For now we have concluded that since this deliverable consists of an incomplete larger product that is still early in development, a development build should suffice for demonstration purposes.

3.1 Registration

The ability to register users was the first major part of the MVP we worked on and it was when we made the bulk of design decisions. Additionally, Register was what we used to establish client-server-database communication and our implementation of this requirement became a framework and guide for other parts of the MVP. An in-house input validation system was created utilising `TypeErrors`, and our custom `ValidationErrors`. These errors are used to indicate the correctness of necessary inputs in the request while any additional (unwanted) data in the request is discarded. This was done to allow for appropriate server responses with invalid or malformed input resulting in a status 400, while non input related errors could invoke the status 500 - allowing for better consistency with industry standards.

Registration additionally required the generation of a unique user ID and much thought has been put towards insuring UUIDs are unique. The functionality we landed on utilises `Date.now()` for milliseconds precision meaning that two users cannot be assigned the same UUID unless they register at the exact same milliseconds. To further guarantee uniqueness, a string of 6 random characters is scrambled into the ID further decreasing the odds of the same UUID being Assigned. Lastly, the database will not register a user with a UUID if it already exists in the database, so worst case scenario, a very unlucky user would just have to try again. This code can now be found in `backend/Server/utils/generator.js` and has been found to be quite useful in other parts of the implementation as well.

For Insertion into the database, Registration utilises our in-house database API functions which has been designed to provide a large degree of functionality in regard to accessing and manipulating the database. Using these allows us to have more confidence in our implementation and reduces the basis for error as they have been thoroughly tested. A huge benefit of this was the confidence that something would work even if you could not currently access the database, allowing for quicker development and helping us focus on our individual tasks.

3.2 Login

We implemented a minimal viable login validation to increase account security.

`src\front-end\client\src\utils\validate-client.js` provides input validation functions:

- `validateLogin()` is responsible for server-side input validation. The function performs a number of validation checks using `userid`, `password` and `state` to determine whether the user input is valid. It returns `json` object showing result of validation in format.
- `validateInput()` was used for input validation for both the length and syntax of given String data. `checkLength()` checks if input data is compatible with the length bounds.
- `validatePassword()` was designed for passwords input validation. Password must include 1 lowercase char, 1 uppercase char, 1 digit and cannot contain `Pass`.

`src\front-end\client\src\utils\Cookies.js` is used to get, set and delete cookie.

`src\back-end\Server\routes\api\login.js` handles the login registration form and validates user inputs and logins users. The backend check if input is valid,

- If input validation is passed, then the request is processed.
- If input validation is not passed or wrong data format is received, then send bad request.
- If there is an error that the result is not client input, then send internal server error.

The user login function `loginUser()` attempts to check validate user id and password to complete the login process. This is the core server-side user login function. Login account data including userid and password received from client. Depending on the result of the validation, the backend server sends response to client:

- If the user does not exist, send error response
- If the user exists, send successful login response
- Otherwise, send database error response

3.3 Supergroup Login Protocol

We have decided to adopt the `Single Sign-On`(SSO) Protocol to implement Supergroup Login.

When `Home journal` is mentioned, it is referring to the journal at which the user created their account and when `Secondary journal` is mentioned it is referring to the journal which the user wants to login using their home journal login.

3.3.1 Username Format and Discovery

When the user is registered, they are given a unique username. This will take the format `xxxx:t<team number>`, where `xxxx` is a random string of any length. for example, `kfdj3w:t03` is a possible username. The number preceding `:t` helps determine the users home journal. The secondary journal can find the URL of the home journal using a `lookup table` at the location `/cs/home/gbs3/cs3099-journals.json`. The secondary journal can use the team number as the key.

3.4 Comments

When a post is clicked on, it will first send a get request for it. The server then gets the postID from the request, validates it, and then queries the database for the comments for that specific post. Results are returned to the server, which then returns the request's success plus the comments back to the client.

The comment section then selects the `AP_commentBOX` div. If the selected div doesn't exist, it returns null. Otherwise, if the div is not empty, it is emptied - because when a comment is written it needs to update it. Finally, for each comment received from the request, it adds a new child to the div containing the user's first name, the comment's creation date, and the contents of the comment itself.

For posting a comment a textbox is used, which one can then write to and submit using a button. On submission, the contents of the textbox are retrieved, along with the user's id and the post's id which are stored in an object called `data`. `data` is then validated. Upon failure, an error message is alerted and logged for the user. Otherwise, it sends `data` to the server which validates it again, then creating and storing a comment on success with a unique id and other relevant information. If server-side validation fails, it returns an error back to the client. Finally, comments are then refreshed, displaying the new comment beneath the post.

3.5 User Interface (UI)

The UI was implemented via react. Initially, when the client first runs `App.js` it will create a `BrowserRouter`. Showing the navigation div - which contains a link to the home, upload, and account pages. When clicked, they will load their respective components, with the home page being the component loaded on launch.

The home page loads the first 10 posts, detailing information for each one. When the title for one is clicked on, it loads it and displays it's title and description, the contents of the associated file, and finally, it's comments. A user can then post a comment if logged in.

The upload page requires you to login before you can make an upload. Once logged in, you can write the title and description for a post and select a file to upload. If validation passes, the post data will be stored in the database and the file itself in a directory on the server. Its success is then alerted for the user. Otherwise, an error will be alerted instead.

Finally, the account page shows the user two forms. One is for signing up, and the other for logging in. Successfully filling in either form will create a cookie containing your user details and showing that you're logged in. Difference being that the sign up will also create a new user in the database.

3.6 Supergroup Content Migration Protocol

The migration protocol is the last thing we worked on and is the one part of our MVP that can be considered 'unfinished'. The core implementation has been completed and to the best of our ability we have attempted to follow the supergroup protocol outline and API specification. The implementation for exporting or 'migrating' posts to other journals has been completed and progress has been made on allowing the importation of content

into ours. We used axios for sending requests to other journals as to remain consistent with our react front end and can say with confidence that we will be able to complete this requirement in the future.

There were some problems with the supergroup protocol and API specifications however, and outside from a lack of clarity, the two documents we had to work with were either not detailed enough or not fully compatible. This issue becomes clear when the supergroup is looked at as a whole. in the last days before the deadline, it became clear that every group had a slightly different implementation or API regarding the protocol, and it became clear that an unforeseen problem had emerged and it was too late to fix it. A last minute super group meeting was called 1 day before the deadline to address this, though by then it was too late.

As a result, we had no opportunity for proper testing regarding the protocol and decided to refrain from spending the last few days scrambling over something we would have to tear out later and redesign. Instead, we progressed as far as we could and decided it would be better in the long run to focus on areas we had more certainty and/or control over as a team. We hope you can agree with our handling of this situation, I have included screenshots of the supergroup scrambling in the appendix (with people's identities protected as we do not aim to put anyone on the spotlight).

3.7 Database and Object Design

We determined what data must be stored and how the data elements interrelate. With this information, we can begin to fit the data to the database model.

- Determining data to be stored: part of requirements analysis.
- Determining data relationships: sometimes when data is changed, other data that is not visible can also be changing.
- Logically structuring data: Once we determined the relationships and dependencies amongst the various pieces of information, it is possible to arrange the data into a logical structure which can then be mapped into the storage objects supported by the database management system.

Database Operations are implemented by Agent Design Pattern. `src\back-end\Database\dbAgent.js` provides a surrogate or placeholder for database to control access to it. Because all operations relating to database in JavaScript is asynchronous, the agent creates `Promise` connections and queries to handle requests. Other team members can easily get/put/query objects without a fully understand the data transmission between database and server.

There are mainly three data classes in our implementation: `User`, `Post`, `Comment` (defined in `src\back-end\Database\util`), with each one represents a corresponding object stored in database. Their member functions help developers do a certain action easily.

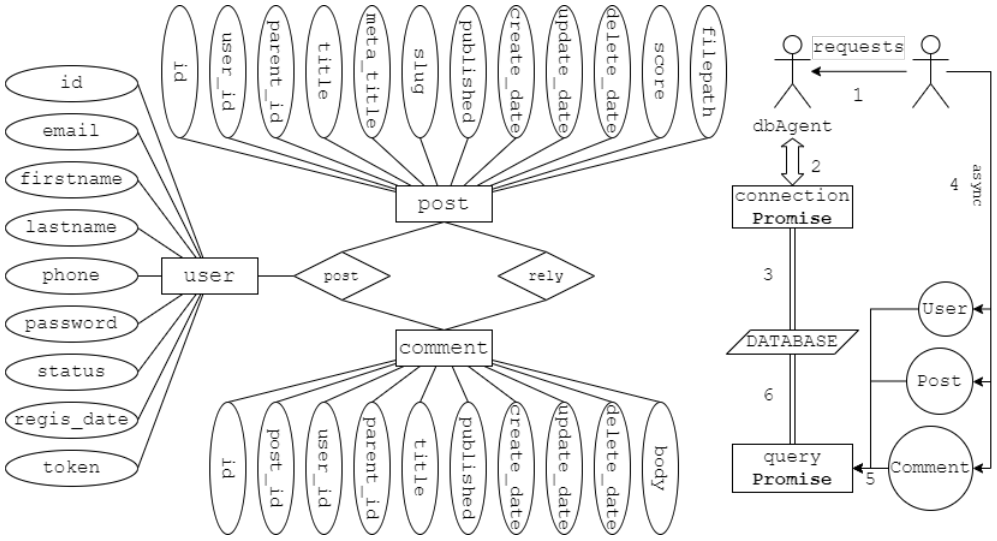

Figure 2: Entity–relationship model & Database Model

4 Changes to Our Plan

We have not felt the need to deviate from our plan - any technologies we choose in deliverable 1 have remained with our team and project. We believe that thanks to our research and comparisons of relevant technologies as part of Deliverable 1 we were able to narrow down a suitable suite of tools that worked well together and with our team. This has managed to turn out in practice as well as on paper and as such nothing has been changed.

5 Supergroup Interaction

The Supergroup representative has been attending the Supergroup meeting and discuss how we can ensure our journals can be compatible with each other. They have made contributions to how the supergroup protocol must look like and relayed the decisions made from the meetings to the whole group. A presentation was made to the group, by the supergroup representative, to inform them of the Supergroup login protocol.

To aide with the understanding of the protocols a documentation was created, this enable the group members to look up the functionality if they were unsure. An Endpoint documentation was also created for the group members to understand the exact data format required. The Supergroup meetings were at least once a week and much more often when it was necessary. The meetings were conducted on Microsoft teams.

6 Test

- **Automatic Test**
  Automatic testing (`src\back-end\Test`) mainly tests the communication between server and database. It automatically generates different random objects, that is `User`, `Post` and `Comment`, and then test every member function of them. It contains 4 test suits and 29 dependent tests.
- **Manual Test**
  Test cases includes edge cases, normal cases, abnormal cases. See evidences in `./TestEvidences.md`.

7 Conclusion

In conclusion, following the scrum methodology, our team created an MVP for deliverable 2 that achieves the required functionality. Although there were a few difficulties, through communication in meetings on teams and on discord, we could follow the plan set out in deliverable 1. Leaving us in a position capable of producing the next deliverable when it is released.