CS5003 Practical 03 Group Report

CS5003 Group Report

Introduction

A bike app designed using state-of-the-art ECMAScript 2017. All code are transpiled using Babel.

This application does not attempt to introduce grand functionalities or beautiful aesthetics. It serves as a sandbox for our members that would allow us to test `React.js`.

Language specification

This project strictly enforces the use of ES6 modules and the use of ES6 syntax. However, there are certain files that uses `CommonJS` specifications instead of ES6 `import/exports`. This is due to the limitations in certain libraries we are using (such as `knex.js`). Unfortunately, node v.10 currently still does not natively support ES6 keywords and `import` and `export` syntax. We resolve this problem by importing `babel-cli` to transform our syntax to traditional js syntax.

```
npx nodemon --exec babel-node server.js
```

Meta criticisms

We do not really know what is the functional purpose of building this bike application, since without GPS tracking functionality, it would only be a data recording app at best with a database. However, we nevertheless proceed with the goal of satisfying the minimum requirements while maximising our knowledge gain.

Academic Good Practice: Declaration

Certain parts of the front-end component contains boilerplate styling code from the official [Material-UI] documentation. All other codes for the back-end and front-end algorithm are completely handcoded.

Requirement specifications

This project fulfills all of the following requirements: -

- **User Interface** implemented using React.js
- **RESTful API** implemented using Koa.js
- **Database backend** implemented through MariaDB and Objection.js
- **Algorithm** Most frequent item in Object (handcoded by ajw40)
- **External API** Implemented through Weather API and OpenMaps Tiling
- **Use of JavaScript Library** See technology stack
- **Good Engineering Practices** See code documentation and `jest` testing

Technology Stack

One of the goals of the project is to challenge ourselves. While we can easily just build the project using simple hand-coded JavaScript, we feel that we should try and push ourselves by using the most up-to-date libraries and technologies, which would enable us to understand different JavaScript library patterns as well as the benefits they provide.

Server-side:

- Knex (SQL Query Builder)
- Koa-logger (Logging middleware)
- Koa-router (Routing middleware)
- Koa.js (Web framework)
- mysql (Database driver)
- Nodemon (devtool)
- Objection.js (ORM)
- Passport.js (Validation) [NOT IMPLEMENTED]
- SwaggerUI (Documentation)

Client-side: * React.js * React router * Leaflet.js * Material-UI

Git Workflow

- Adopt mainline development over trunk-based development owing to memeber's unfamiliarity with using git on the terminal
- Added git hooks to ensure that developers do not accidentally push to `master`
- Add GitLab `master` branch protection using local git hooks

Our strategy enable each member to contribute without being present in the same location. Furthermore, we strictly forbid any file exchange via Email or Whatsapp to enforce good directory organisation.

Members are only allowed to push to `develop` branch. Members who accidentally push to `master` will have their commits reverted. Only the team leader is allowed to merge the branches (or delegate specific merge action to a member).

Choosing the appropriate tools

When choosing an npm package, we need to be mindful of the project's maintainers, and its most recent commit history. For our purposes, we also need to assess the quality of documentation, and its community support. These would allow us to find solutions more quickly and reduce development time.

Design stages

We had a very long disussion on how to design the site. You may find proof of the ideation process in `docs/wireframe`. You may also find our user stories in `docs/` directory.

The operation which we envision is as follows:

```
Client (Front-end operation) -> RESTful API -> Server -> Database
```

We first build the server-side to enable the client to perform CRUD operations on our database. We tested this using `curl` and through the browser. We do not to handcode the operation to transform the data into JSON since this operation is executed by default by `Koa.js`.

Server-side

Servers

This project requires us to set up three servers: database, app server, and an http server. We use `src/server/index.js` as our server entry point.

Our development cycle also includes a development database, which we use for off-campus development.

Choosing a Framework

There are 4 main choices we get to choose from: `hapi.js`, `express`, `restify`, and `koa`.

Koa is built and develop by the team from Express. `Express` provides reliability. However, we chose to go with Koa largely due to its integration with async/await.

API

During our API design phase, we laid down the following ground rules:

- Use nouns not verbs
- Use plurals
- Use versioning
- Use query parameters
- Return HTTP code
- Use envelopes
- Server-side field validation

Other things we need to care about is ensuring that all inputs are properly sanitised before operated upon. This is to prevent SQL-injection vulnerabilities.

We have also chosen not to limit the rate for our API. In theory, this could be accomplished through a Redis database and a `koa` middleware. However, we feel that, for the current scope, this is entirely unnecessary and only adds additional development time.

Documentation

For industry-standard documentation, SwaggerUI or OpenAPI comes to mind. For simplicity, we choose to document our API using a plain text file. However, we try to conform to basic principles of clarity where possible.

Database choice

We choose MariaDB. Best fit for our current use-case. NoSQL is not correct of our current use case.

The reason we chose to use a relational database is because it enforces ACID. These properties are crucial as it helps maintain the integrity of our database. As this is a simple application, we do not need to scale our product.

We are also going to be using `knex.js` – a query builder for relational databases. We are also going to use an ORM library to make queries and store user input. We chose `Objection.js` for its clear and concise documentation and relatively low learning curve. Other alternatives include `Loopback`, `Sequelize`, `Bookshelf.js`.

The reason why we choose to use a query builder is to prevent against SQL errors and potential SQL injections. Furthermore, we want to take advantage of `knex` migration feature. This is a feature that allows us to track the structure of our database in version control, making it easier for us to make incremental changes to the database according to the features we add to the application. Every time we update our database schema, we can run `knex migrate:latest` to update our tables.

After setting up our tables, we use `faker.js` combined with the knex seed function to populate our database with fake data.

We divided our setup into development database and staging database. The development database is used for our developer's local development cycle. Once the knexfile script is fully tested, we will push it to the staging database, which is hosted on one of our developer's university server.

Notes of schema

The database design is obviously not optimal. We chose not to normalise the database completely as it allows us to make queries without utilising complex JOIN operations. Furthermore, we do not think it is worth the time to labour on the schema design – our core concern in functionality and ease of access. Furthermore, as all of the queries are done through `Objection.js`, the need for an absolutely perfect database design is not necessary given our limited timeframe.

To satisfy the minimum requirements, we designed to simple models: Users, and Sessions. One user may have many sessions, and sessions may only be associated with one user.

The `knex migrate` was supposed to be self-documenting. In other words, the user should be able to gain an understanding of each tables simply by reading the code.

We also need to write our seeding files. The seeding files populate our database

Data validation

When creating `/register` endpoint, we need to install a `koa-bodyparser` to get the body portion of the POST request and expose it on the `req` object.

When posting to the server, we want to ensure that the data is validated before being stored to the the database. We execute the data validation process on the server-side for security reasons. Here, we use `@hapi/Joi` library to do a server-side validation prior to data insertion.

Login Validation strategy

Unfortunately, we did not manage to get the password validation to work. This is because we did not manage to understand how to integrate it into our technology stack.While Login functionality is not in the requirements, it is reasonable to implement this. We failed to implement this feature because we did not manage to digest `React router` private routing. We do, however, understand how it SHOULD have been implemented without React. The following section provides a proof-of-concept implementation.

It is not preferable to store passwords as plaintext inside our SQL database. We want to ensure that, even in the scenario where our SQL database has been compromised, our passwords are still stored securely. To resolve this issue, we implemented a hashing function prior to insertion. We do not want to be tied down by the intricacies. Hence, we use a third-party plugin for `Objection` ORM to hash the password prior to data insertion. In general, this would only require us to build a form and have a button that submits and make a POST request to our RESTful API, which ideally should return 403 if unauthorised, or 201 if authorised.

The proof of attempt could be found in our `UserRoutes` and `User` model, which includes `Objection.js` password hashing and validation function.

Development

We are going to use Faker.js to generate testing data for development purposes. One of the difficulties we experience is seeding the data that is reasonable. `Faker.js` allows users to generate a random date, but does not allow users to limit the time range. We overcome this by parsing the generated datetime string into a `Date` object and add a random number between 1 - 10 to the Date hour, and then reconvert it back into a ISO string and seed it into the database.

Client-side

React.js

We must first find a way to integrate React.js into our existing project. The issue is that `react-script`, which is used to build our application, uses an earlier version of `babel-jest`, a dependency used for testing, compared to our version of jest. We need to add skip preflight check in our environment variable to stop `react-script` from complaining.

Choosing a CSS Framework

Our selection criteria for the CSS framework focuses on how well the framework integrates with React. For the purposes of this assignment, we do not really care about the size of the actual framework, as we believe it is an unnecessary optimisation (You will be launching it from local server anyway). We also want to make sure that we are not reinvening the wheel when it comes to the design. It is important to cut down on development time when it comes to aesthetics (since this is not an essential requirement of our client (or for marking purposes)), and focus on delivering essential functionalities.

Data-handling

One thing we have to decide upon is whether to handle calculated fields on the server-side or the client-side. We decided to handle calculations on the client side, since we want to get the requested data in its purest form.

Algorithm

We build a simple algorithm that would count the most frequent ride type. This algorithm traverses through an object and returns the most frequent value. We use this to find the most frequent bike type. Note that if there are two bike types of the same frequency, we will choose the first bike value that was recorded.

GPS Data and the Map Component

We scraped a couple of GPX file from HSBC cycle ride routes, and used an online tool to parse those data into GeoJSON format. We originally contemplated adding a file upload button in the SessionsForm and store the parsed data into our database. However, we decided this would take too much engineering time and ultimately abandoned the idea.

We did however, decide to allow users to paste their GeoJSON data into a text area. This is not optimal owing to potential security risks. However, this application is just a proof of concept so we decided not to labour on this point any further.

When submitting the GPS data, we have to make sure we are storing the data in plain text. We call `JSON.stringify` before our `POST` request to transform the geojson data into a string.

We do not particularly care whether we can query the spatial data, since the geojson is parsed when we render it on `Leaflet`, and we do not have extra functionalities that require us to implement a database schema that allows us to query geojson.

The tiling layer uses `OpenMap` API, which fulfils another checkbox on the requirements list.

In the `Map` tab, you should be able to select a session and show the starting point on the map. When you submit a session, you should also be able to submit a geojson file. Note that we did not create a function that validates the GeoJson file, so it accepts all types of json.

Weather Forecasting

The weather forecasting does not actually do anything apart from displaying the weather. We use the Weather API to satisfiy the minimum requirements stipulated in the instructions.

Improvements

The following lists all the things we should have done:

- Adequate unit testing
- Better directory organisation
- More code comments
- Use dotenv properly and not commit `.env` file
- Validate Geojson. This requires us to use a geojson schema.

Challenges

There are several challenges. They may be roughly categorised under the following headers: -

1) Equalising team's knowledge and experience
2) Technical constraints imposed by the lab machines
3) Codebase complexity

One of the major challenges that stalls our development progress is that very complex and deeply nested directory structure of our project. The project became too modular. We think better documentation would have resolve this issue.

Even just building and launching the project would take at least 30 seconds, which for a project of this size should not be the case. This is largely due to the high modularity of our software and the very intricate design.

We also feel that better tests could be written. The coverage of the tests does not extend to client-side code. We think using Storybook.js would have help us achieve this.

We are very much constrained by our inexperience in JavaScript, as well as our ability to adapt to new libraries and technologies. We are by no means satisfied with this Practical,