MVP Implementation

Samuel Edson
Grant Gunnison
Sabrina Drammis
Danny Sanchez

employ.me

Programming

Basic Coding

In terms of readability, we follow normal Javascript and coding conventions. Some to note are camelCase variables, keeping lines no more than 80 characters wide (within reason), and double space tabs because Javascript often has many nested callbacks. We also include comments where needed, including things like, “//TODO after MVP: Authentication check goes here“ for when we need to extend something or fix it. We talk more about commenting in the Modularity section.

Our ‘app.js’ is very short, and organized into imports, configuration, routes, and launch. The first thing I do when I see a node project is look into ‘app.js’ to try and figure out how it works. Out ‘app.js’ elucidates the structure of the app, and what functionality it uses, especially through the routes section. From that section alone, a naive reader can see we have a main page, students, classes, and skills associated with their own functionality within the app.

For client side code, we define no global and everything is defined in a function, the Controller. We have two such Controllers, StudentProfileCreationController and IndexController, that act like classes with private and public variables explicitly declared inside objects (called ‘private’ and ‘public’ for simplicity). More on the Controllers in Modularity.

For database calls, we follow the convention on the Mongoose website for multiple .method’s in a row. It shows us to line up the dots. This does help the reader quickly discern what the query is asking for because each line defines an action or filter. Here is an example:

```
Student.findById(req.params.studentId)
        .populate('skills')
        .populate('classes')
        .exec( function (err, student) {
```

We also use the same pattern for function calls with multiple inputs that would span more than 80 characters across, or otherwise be hard to read.

We have found that sending success and errors is a pattern that is very common with web apps. We created a ‘utils.js’ file with functions to do just that. We have sendSuccessResponse that sends 200 Status and a json input. We also have sendErrResponse which sends a specified error code and an error value. These two functions get used by basically every route, so it prevents a lot of code duplication and errors.

On a similar note, we do actually make sure to send error responses when there is an error with the query, or when there is a server error too.

In terms of language usage and algorithms, at the current MVP state, we barely make use of any complicated data structures. We use arrays and dictionaries a lot, but our app is in a state where other data structures are not really useful. When thinking about language usage, we opt to use the highest level pattern for whatever we are trying to accomplish. This contributes to shorter functions and code that is easier to read.

Modularity

With a larger app, it is very important that the directory structure make sense. This could also be an aspect of our app that would help with Modularity, because if the directory structure makes sense from the start, then extending the app becomes a simpler process. The app also stays more organized for longer because authors will put new files in the right place. Our app is organized to separate ideas in a cohesive manner. We place all client side code and styling in the ‘public’ directory. Jade templates and all views go inside the ‘views’ directory. All server side code goes in ‘app’ and configuration goes in ‘config’. Inside of ‘app’, we organize code based on functionality, so any requests for urls are handles in ‘routes’, data and models are separated into their respective folders, with Mongoose schemas separated out from their models.

For routes, we used comments to make the functionality of our API and app structure very obvious. We used a standardized way of commenting in the route for ‘student.js’, ‘classes.js’, and ‘skills.js’. Here is an example of one of these comments:

```
/* Set the classes for a student
 *
 * POST /students/:studentId/classes
 * Body:
 *    - classes: list of class mongo _ids
 *               set as the student's classes
 * Response:
 *    - success: 200:
 *        if the classes were sucessfully set
 *    - error 404:
 *        if the studentId is not valid
 */
```

It contains a description, the action and url, and the body and response of the API. This makes the API obvious, so that when we implement the rest of our app for the final, it will be easy to look back and figure out what each route is for. Also, it prevents us from duplicating functionality because we will know exactly what features we have and don’t have. In addition, the body specifies the *type* of input as well. This is extremely useful later on because Javascript is not strongly typed, so it allows a front end programmer to discern what needs to be sent to the server to get a correct response without having to read through the code like a detective. That aspect will not only save errors, it will save time, so we will be able to write a more complete web app for the final.

As mentioned in Basic Coding, we define Controllers that control the studentProfileCreation and index views. These Controllers are organized in a way that allows them to be extended very easily, because they are defined like classes. None of the internal code is exposed unless explicitly returned in the return object of the Controller definition, so there is no accidental leakage. The code is organized, so if you want to find where the window events are, you need only look at the function ‘eventListeners’. Also, any code you want run from the start just needs to be called in ‘init’. This is a good design pattern because it is readable and removes a lot of possibility for error.