Evolution Four

ECE 458

Parker Hegstrom (eph4)
Peter Yom (pky3)
Wayne You (wxy)
Brandon Chao (bc105)

Abstract

New features were added to our shared calendar web application in accordance with evolution four's requirements. Among these functionality additions is the support for PUD escalation, calendar export through email, preference based sign up events, and mass event creation.

Contents

1 Overall Design

The overarching design principle we wanted to achieve was modularity. In doing so, we believed we would be able to work separately (with occasional meetings to work through minor problems with the API) and refactor without the worry of breaking another member's code. Figure 1 below shows a high level diagram of how we decided to design our calendar web application.

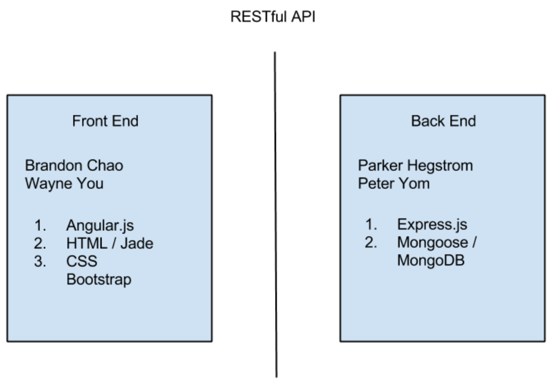

Figure 1: Diagram of our Large Scale Design

Essentially, our back end team provides an exhaustive RESTful API service to our front end. As we received the new requirements for evolution two, the benefits of our modular design came to light as we met to discuss both the refactorings from evolution one that needed to be done and the edits to each modules system design in order to account for the added calendar functionality–event requests and persistent until done events.

The following sections further discuss design choices and implications of those design choices for both our front end and back end teams.

2 Back End Design and Analysis

2.1 New Features and Developments

Email Calendars

This feature was implemented using the `nodemailer` module that we imported. The back end simply handles the emailing here as the front end actually bundles the proper information (event objects or png) before issuing the request to the server. See front end discussion for more specifics on how the PNG file was created.

Text Input for Event Creation

This feature was pretty simple to execute as we already have a dedicated route to event creation. Hence, given text input, the front end parses the input and issues the needed amount of POST requests to the `/event` endpoint.

Priority Escalation for PUDs

This feature was a little difficult to implement as previously we had handled priorities by the PUDs index in the users array of PUDs. We thought about scrapping this idea and implementing a priority number, but in the end, we didn't want to change previously written functionality. Hence, we continued to use the array index method for priority. To allow for certain PUDs to escalate, we set a `willEscalate` property in the database. Using the `setInterval()` function provided by node, the server will check every day which PUDs need to be escalated.

Preference Sign Ups

2.2 Benefits of Our Previous Design

The major and most noticeable benefit from our previous design was how re-usable it was. From the beginning, we treated the back end of our application as an exposed services API, providing a set of methods that virtually any front end could use. This re-usability came about because we continued to write short and single purpose methods for every service. And on a broader sense, all of our "routes" were written to complete a single task—create an event, delete a user, send an email, etc. So, for example, when faced with text parsing, all that needed to be done was the actual parsing of the text, because once the data was gathered, it was just a matter of looping through and initiating a POST request to the `/event` end point for each event.

The re-usability came about from our file organization as well. In the third evolution, we began to increase our use of Mongoose.js "schema" methods. This allowed us to treat our database models effectively as java classes. Many of our route files might need to convert user ids to user emails, and because of schema methods, we were easily able to make that functionality visible to any file that needed it by using javascripts `require()` method.

2.3 Drawbacks of Our Previous Design

The most time consuming feature was the preference based sign up. Why? Well we discussed that this type of feature—one that modifies pre-existing functionality—was the hardest, but the main reason for its difficulty came about because we made too many assumptions when designing the `SlotSignUp` model.

Before, we had pretty rigid process for handling the slot sign up creation that used async.js `waterfall()` method. Now, though, certain parts of that async waterfall did not need to be called or additional asynchronous functions needed to be added to the function array. These additions or omittances changed the dependencies between each individual function in the waterfall function array. The required changes were great in number and therefore so were the errors.

In addition to specific design pitfalls, another more general problem arose: our file sizes were becoming very large and difficult to follow. As we added more features to pre-exisiting code, it became apparent that in some files we had just continued to populate the files without giving thought to potentially extracting the code out into a new file—the `SlotSignUpRoute.js` is almost 500 lines long. I think the solution here is to recognize this earlier and spend more time on discussion of new feature integration rather than just attempting to cram in the new features all into preexisting architecture.

3 Front End Design and Analysis

3.1 New Features and Developments

3.2 Benefits of Our Previous Design

3.3 Drawbacks of Our Previous Design

4 Individual Portion

Parker

a) **Designing System Components**

- I designed the system component that handled the auto PUD creation for all invitees. This required a variable number of asynchronous functions to be executed back to back. More specifically, it was dependent on the number of invitees to the sign up event. To accomplish this in an efficient manner, I created a function that created an array of functions. (see functions `createWaterfallArray()` and `createFunctionInArray()` in the file `SlotSignUpRoutes.js`). In short, I bundled all of the PUD creation code for one user into a variable and created one for each invitee. Async.js executes the array of functions. The benefit from designing the component in this way was to simplify the code and make it more readable. Without this array of functions, there would have been very deep callback hells. The design also allowed us to continue to make use of pre-written libraries that helped with asynchronous things, minimizing the amount of code we had to write in the end.

b) **Designing and Conducting Experiments**

- For the auto pud creation feature in preference based sign ups, we had to perform a number of asynchronous operations based on an input size (number of invitees). To do this, I had to create an array of functions which was only made possible by the fact that javascript uses first-class functions. To test this function array creation, I started writing simple examples. I would then loop through the array of functions, executing each and making sure they executed correctly. The next step was to gradually turn my simple examples into ones that performed the functionality required by the feature.

c) **Analyzing and Interpreting Data**

- One use of first class functions is the callback, a way that javascript handles asynchronous functions. I wrote my own asynchronous function that utilized a callback exit. Unfortunately this function produced an error I had never seen before, one that dealt with http protocols. Specifically the response header was being set after the response had been sent. Confusing... . However, I was able to read the stack trace in the console, pinpoint the location of the error, and deduce the type of error. With the error type and location, I was able to eventually fix the bug by reading internet forums and Express.js API pages.

d) **Dealing with Realistic Constraints**

- A realistic constraint I encountered was my lack of knowledge for how javascript treated functions. To get around this, I simply had to do a lot of side reading on my own. I watched video tutorials and read online forums like stack exchange.

e) **Teamwork and Team Member Interaction**

- For this evolution, I would say we worked much more independently than in previous evolutions. Our schedules did not permit too many in person meetings. This turned out to be completely fine as, apart from some small changes, our group was able to function very efficiently in this manner. It was a learning experience for the better!

Peter

a) **Designing and Conducting Experiments**

·

b) **Analyzing and Interpreting Data**

·

c) **Designing System Components**

·

d) **Dealing with Realistic Constraints**

·

e) **Teamwork and Team Member Interaction**

·

Brandon

a) **Designing and Conducting Experiments**

·

b) **Analyzing and Interpreting Data**

·

c) **Designing System Components**

·

d) **Dealing with Realistic Contraints**

.

e) **Teamwork and Team Member Interaction**

.

Wayne

a) **Designing and Conducting Experiments**

.

b) **Analyzing and Interpreting Data**

.

c) **Designing System Components**

.

d) **Dealing with Realistic Constraints**

.

e) **Teamwork and Team Member Interaction**

.