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.

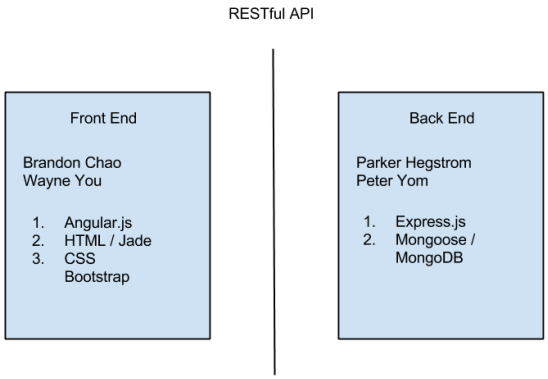

Figure 1: Diagram of our Large Scale Design

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

Add stuff here tomorrow morning

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-existing 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

3.1.1 PUD Events

The extra features to PUD events including an expiry time and priority escalation were handled on the front-end by simply adding fields to the PUD creation form and sending this data to the back-end. Both of these features took front-end input but the functionality was handled separately in the back-end.

3.1.2 Preference-based Sign Up

3.1.3 En Masse Event Creation

We implemented en masse event creation using simple text parsing. Since we have the user input the event with a Name, Description, Start Time, and End Time on separate lines, we simply split the input string based on the new line tokens and create separate events for each series of 4 lines of text.

3.1.4 Schedule by Email

Our front-end implementation of this feature was fairly easy. For text schedule, we simply parse through all of the events within the current display frame and send them to the back-end. For image schedule, we take a screenshot of the calendar in the current display using Javascript's html2canvas() function and send it to the back-end to email.

3.2 Benefits of Our Previous Design

The implementation of the features in this evolution were fairly straightforward due to our design coming into it. Many of the features seemed to be more back-end oriented which also made our lives a lot easier in this evolution. The only completely new feature was getting the schedule by email, which was a matter of parsing through our list of events we already store for text-based, and used a Javascript function to accomplish the image-based one. Even mass event creation was not hard to create due to the way our events were being created in the past. We merely had a for each loop to create an object with the data for each event we parsed through, and then called our previous event creation route on each event. We really saw the benefits of a flexible design coming into this evolution based on the requested features which allowed us to complete this evolution quickly.

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**

- Wayne initially implemented getting the schedule in image format by using the html2canvas() function to take screenshots of the page based on the DOM. However, the limitation to this is that we can only get the image of the time range that is currently being displayed. As such, I experimented with taking and storing images of multiple weeks/months and sending them all at once to the user. I did this by changing the view of the calendar (and the DOM) and taking a screenshot at each requested month using the same function. However, this didn't seem to work because I was not able to change the view of the calendar without it rerendering on the front-end which caused it to keep changing the display. Also, I was having trouble sending an array of image data URLs to the back-end to represent each week or month that was requested. Because of these issues, we ultimately decided to limit the functionality of getting the schedule in image format to only send the current time frame to the user.

b) **Analyzing and Interpreting Data**

- When I was implementing reordering of Slot Sign Ups to allow the owner to decide slots based on their preference, I ran into the issue of the slots not updating their reorder in real-time. However, when I printed them out after grabbing them from the back-end, they were in the expected order. This led me to believe that whichever scope variable we were using to display them on the front-end wasn't being updated correctly. I confirmed this

by printing out the array of objects for the ordered slots from the Angular code and saw that they were indeed incorrect. This was solved by updating the scope variable upon successful execution of the HTTP PUT. We have run into numerous similar issues in the past which made debugging this fairly easy. However, if I had time to continue working on the calendar, one thing I would refactor is cleaning up our controllers and which scope the variables were placed in by creating a better hierarchy of controllers.

c) **Designing System Components**

· One component we had to design was the detail display for preference-based signups. We decided that we wanted the owner to view the Slot Sign Up details modal unless all requested users have signed up for slots already and it was time for the owner to resolve the conflicts. In that case, we bring the owner to the Slot Sign Up resolution modal. After resolution, subsequent clicks of that particular slot sign up will once again take the user to the details modal. We achieved this by maintaining two flags stored within the scope for the particular slot sign up. We maintain one flag to check if the preferences for slot sign ups are complete, as in all requested users have signed up, and one flag to check if the preferences for slot sign ups are final, as in the owner has resolved any conflicts. Each time the slot sign up is clicked, we check these flags to determine which modal to display. This currently does not allow the owner to edit a slot resolution after they submit it, but adding this would simply require a button that would toggle the final flag for this particular slot sign up which would allow the owner to make changes after submission.

d) **Dealing with Realistic Constraints**

· One realistic constraint that we had to consider was when we were implementing the graphical schedule by email. We did this by taking a screenshot of the current time frame of the calendar. While this wasn't very difficult, the nature of the html2canvas() Javascript function we used to do this made it difficult to get a graphical display of calendars that you were not currently on. For example, to get a graphical schedule of the next month, you would need to manually click next month and then send the image file. We experimented with using various other functions to achieve this, but the html2canvas() function provided the easiest implementation.

e) **Teamwork and Team Member Interaction**

· We made an effort to start implementing the features of this evolution a lot earlier this time. As a result, by the night before it was due, we had a lot fewer pressing errors to resolve last minute. I felt that teamwork and team member interaction went a lot more smoothly because of this and because we met up in person earlier to work out bugs.

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**

.