# Evolution Two

ECE 458

Parker Hegstrom (eph4)
Peter Yom (eph4)
Wayne You (eph4)
Brandon Chao (eph4)

## Contents

# 1 Overall Design

# 2 Back End Design and Analysis

## 2.1 New Features

The back-end introduced sharing on an event granularity through event invitation, request objects, and PUDs. Also, we introduced a powerful new tool called deepPopulate.

DeepPopulate is a mongoose plugin that allows us to do deep population of nested models. During the development of evolution 1, we were unable to deep populate and instead resorted to levels of callbacks on callbacks in a ham fisted attempt to populate down more than 1 level of a mongoose model. For example, when returning a Calendar to the front end, we would first query for a user, populate its calendars, and for each Calendar we would query for the events, owner, and rules within callbacks and add them to the JSON object to return. We could continue deep populating this way, but the levels of callbacks made the code incredibly challenging. DeepPopulate allowed us to populate multiple fields from a model at whatever depth we desired in 1 line. This sped up production and gave us incredible flexibility in how we package information to send to the front end.

While designing requests and redesigning events, we realized that they are very much tied together. When a new event is created, a matching request object is also created alongside it. This request object holds all of the information about which users have been invited to edit the event and their statuses: pending, accepted, denied, rejected. When a user is invited to an event, the _id for the request is added to the user's 'eventRequest' field. The requests in a user's ?eventRequest' field are all of the pending requests that the user must respond to. Upon accepting a request, a copy of the original event is created and placed in the calendar of the accepting user's choosing. This copy is a stand alone event with similar information to the original event. That way, when an edit is made to the original event, the copy is not altered until the user accepts the edits.

PUD stuff

## 2.2 Benefits of Our Previous Design

Our previous design worked well for what it was intended to do. Everything was functional and robust, albeit a bit messy and cluttered in the code at times. However, the modular way in which we organized our routes allowed us to very easily add new functionality to our old code. Unless we were explicitly re-designing a section of the code, altering one part of the code did not affect the rest of the code.

## 2.3 Drawbacks of Our Previous Design

Our previous design had event sharing on a calendar granularity. For example, one event was fixed to one calendar, but the entire calendar could be shared with another user. Unfortunately, the new evolution required us to be able to share and invite single individual events. In order to fulfill this requirement, we had to redesign our event schema and how calendars store events. It was not a painful fix, but it was definitely took some redesigning and rethinking of our calendar's functionality.

Our previous design also worked around our lack of understanding of deepPopulate and promises. There were several areas with multiple levels of nested callbacks and re-routing in order to work around our problems. However, in this evolution, we learned how to properly deepPopulate and to utilize promises. These new developments helped clean up our code, decrease development time, and gave us incredible flexibility and control over our mongoose queries, how we package and populate data, and the overall flow of our code.

# 3 Front End Design and Analysis

The web client required a number of additional functions in order to handle requests and PUD events. Thanks to the modularity of the previous design, most of these additional functions did not require many changes to existing code. This allowed more time to refactor existing code and provide many quality of life upgrades.

PUD events were added in as an additional list retrieved from the server. We displayed them using an Angular repeating list in the sidebar by replacing the previous location for local calendar events. The sidebar was almost out of space for an additional section, and the current design will require a significant UI overhaul if we must add any more sections for the sidebar. In order to display PUD events inside of calendar events, we simply added a section for display in the event detail display that becomes populated via a GET when the event is selected. We could potentially use this method to get more data in the future, but the more data we need to retrieve from the server, the slower the detail modal will load.

Requests were added as a paired object with events. Each event creates an empty request during creation which can be populated by a new form. These requests are shown in a sidebar region that shows sent requests and received requests as well as any updates to sent requests. Requests do not update immediately on other clients and require a refresh to update when sent out.

The rest of the code changes involved refactoring large functions, organizing the code more logically, implementing modals for UI experience, providing more auto-filled inputs, and caching data locally. The lingering problems in the design after these changes are the overuse of $rootScope (a global scope for Angular) and the increasing size of our modal controller. The

first problem is due in part to the scoping of Angular. If we wish to pass a value from one scope to another, it is impossible without passing it through the root scope. However, the root scope does not actually have knowledge of any of its child scopes, so the data passed must be stored in a global scope in order to transfer. The other problem of the scope of the modal controller can be fixed by decomposing the controller since the constituent parts are very loosely connected.

The choice of caching and updating the data locally in conjunction with database updates has come with some benefits and some detriments. For user experience and future use, keeping and updating the data locally drastically reduces the amount of GET requests sent and parsed to maintain data consistency. However, this comes at the cost of increasing the code considerably for the client. Instead of requesting and re-displaying all of the data, the data on local copies must be manually updated to ensure that the user sees values consistent with the database. This will add to development time in the future.

Finally, the Jade template for our single-page site has been growing considerably. Almost all of the relevant HTML for the calendar page is within a single file. If possible, the created modals may need to be moved into their own HTML files, and the sidebars should be converted into UI routes for Angular to better organize the HTML. As it is now, future development will become harder the more we are required to sift through the file for details.

## 4   Individual Portion

**Parker**

a) **Designing and Conducting Experiments**

· A special feature of javascript in general is that it is asynchronous. To deal with this asynchronous nature of the code, callback functions are used. However, multiple callback functions with callback functions (also known as callback hell) makes it harder to find bugs and/or simply read the code. I researched the use of promises alongside mongoDB and Mongoose. After researching, I then tested the promise implementation–accessing the database with a promise and utilizing the exposed promise API. Minor kinks were worked out, but Peter and I were able to clean up some of our code with promises from then on.

b) **Analyzing and Interpreting Data**

· At times, router methods we wrote did not behave properly, and by that I mean that certain errors were thrown to the console. Different HTTP error codes made up the majority of these errors, so I had to interpret what the server was telling me in order to both find the problem and determine what the best solution would be. One example of this that comes to mind was a server 500 response we were getting from issuing a PUT to the `/pud/reorder` endpoint. After analyzing more console outputs, I was able to find that the Express router

treats `/pud/reorder` and `/pud/:pudID` (which is a query parameter) as the same, so both routes were being run.

c) **Designing System Components**

· One new component I designed in the back-end was the PUD. We, as a group, wanted to integrate the PUD into the overall project and specific events without much change to existing code. With that in mind, I made the PUD a separate document type in our database, BUT it shared some of the pre-existing functionality we had developed for alerts and repeats. I was able to reuse a lot of code by using the populate as well as the Alert schema to handle such things. An interesting requirement for the PUD was that you could set a time interval (i.e. weekly for the repeat), yet, once the PUD was marked complete, you wouldnt want it to immediately show up in the list of outstanding PUDs. This was solved by editing the GET /pud method to check the `createdDate` for each PUD.

d) **Dealing with Realistic Contraints**

· As I stated in the previous design document we submitted, javascripts asynchronous nature continued to prove difficult at times. Specifically, problems arose when we began writing more functions both in our route files and in our schema files rather than rewriting code all over the place. Each time we made a call to the database, the value would have to be returned to a callback function, so as we segregated our code more and more, we continued to have to abide by the callback rules of javascript, slowing our development speed tremendously at times.

e) **Teamwork and Team Member Interaction**

· Peter and I worked very together on the backend, and, as a whole, I thought our group worked much better together than we did through the first evolution. I was able to work on new features concurrently with the rest of the group, meeting occasionally to make sure there were no design misunderstandings or any small typo-like bugs that would cause integration problems down the development path.

**Wayne**

a) **Designing and Conducting Experiments**

· Since the new changes to the calendar were not very intensive, I was able to spend more of my time learning how the server handled the HTTP requests during my debugging. While I was previously able to use the console to check the data sent in each request, I was unable to determine exactly what the server requested in some methods and how it was parsed. This led to some problems with Javascript lack of typing, and with this evolution, I was able to better tailor the HTTP requests that I wrote to send data without needing to confer with Parker and Peter on all of the details.

b) **Analyzing and Interpreting Data**

· One problem that was solved relatively quickly thanks to some discussion and better understanding of Javascript as a whole came about from sending a request accept. The request

body sent an object which contained a calendar ID field, and checking the sent data never seemed to show why the accept constantly failed. After doing some server logging, it became apparent that the problem was the the field sent was CalID, and the field requested was calID, and Javascript went ahead and crashed the server on accessing an undefined value.

c) **Designing System Components**

· I worked on the Event Request/Invites handling for this iteration as well as the PUD sidebar list. Request handling was mostly isolated except for two changes for events. The first change was manually adding an event into the calendar when an event was accepted, and the second was displaying a related request's details inside of an event's details. These changes were very easy, but they shared very similar code to existing functions. It may be to our benefit in the future to create more generalized functions and pass in particular parameters to them in order to reduce redundant functionality.

d) **Dealing with Realistic Contraints**

· We moved away from the previous calendar implementation to an Angular implementation because the previous calendar only supported URL redirects or a single type of modal for event display. By switching to the Angular calendar, we were able to call a Javascript function that could provide more functionality. Also, Javascript has continued to be a frustrating language to deal with due to its loose object definitions and lack of typing. Many times a bug has been a spelling error leading to an undefined error that masks the source of the problem, and all debugging requires printing out the objects in question to determine if certain fields exist or if the object itself even exists.

e) **Teamwork and Team Member Interaction**

· We were able to meet much more often for this evolution and were able to work out details relatively quickly. The back-end team of Peter and Parker worked out most of the design for the objects being stored as well as the API. Brandon and I worked out the changes to the client, and we were able to continue to work mostly independently for this evolution as well.

## Brandon

a) **Designing and Conducting Experiments**

· As we were refactoring our calendar, one change we made was to allow the user to select User Groups by name and Users by email instead of using their ID. This required changes to the back-end routes as well as the front-end to connect and display it on the calendar. When the back-end first implemented it, we found that the elements were not updating in live time as we added them through UI interaction. This was strange because once we refreshed the page, the elements in question would then display. I used console logging to examine the data objects before and after refreshing and found that we weren't updating the front-end object stored in the $scope in Angular. As such, it wouldn't update correctly at first until we refreshed and the data was correctly populated using a GET from the back-end. To fix this, we had the back-end return the necessary object on the POST which

we used to update the $scope variable immediately. This caused Angular to re-render and display the data as expected.

b) **Analyzing and Interpreting Data**

· answer

c) **Designing System Components**

· One new component I worked on for this iteration was PUD. Wayne implemented a solid design for it using Angular and I worked mainly on integrating the UI with the existing parts of the project. To do this, I redesigned select parts of the dashboard.jade file to make the code more modular and readable and to display more cleanly in our application. However, one other thing I would like to do soon in the future is to pull out the modal html files into separate files to make the dashboard even more concise.

d) **Dealing with Realistic Contraints**

· One constraint I had to deal with is our use of the Angular Bootstrap Calendar. Although it is definitely more flexible than the previous calendar implementation we were using, since we did not design it from scratch, we are still somewhat dependent on how the creator of the Angular Calendar implemented its API. For example, since the whole calendar is added in our page as one element, it is difficult to make specific changes to it that are not supported by the Calendar's API. I wanted to add more user interaction functionality such as allow doubling clicking to create events, or adding in various elements to the imported calendar display, but in order to do these, it would require us to either modify the source code of the calendar or implement the changes in a hacky way using the available API calls to the calendar. In the future, I hope to find a way around this so we can add more features and alter the calendar to our liking more specifically.

e) **Teamwork and Team Member Interaction**

· We met up more frequently and earlier in the evolution and thus were able to set team goals and priorities much earlier. As a result, we were able to wrap up the functionality portion of evolution 2 on time and had extra time to update the UI and refactor our code. Wayne and I worked well together on the client-side.

**Peter**

a) **Designing and Conducting Experiments**

· The asynchronous nature of javascript has been both a very powerful feature and an absolute nuisance for our group. During the development of evolution 1, Parker and I tried several work arounds of javascript's asynchronous callback functions. With our workarounds with nested callbacks and whatnot, our code was functional but becoming increasingly messier and harder to read. While refactoring the code base for evolution 2, we decided to go more in depth with ?promises'. I did research on promises, tested the functionality of promises, and ultimately added promises to the code in order to clean up what we called 'call-back hell'

b) **Analyzing and Interpreting Data**

· Javascript's asynchronous nature gave us a few headaches here and there. At one point, the front end would call a route, but the route was running multiple times. Every time this route would be called, the server would return the necessary data back to the front end and then crash. The reason for the crash was because the route was running multiple times, the header would be set multiple times as well. After carefully playing with the code, trying a few work arounds with counters, and searching, I eventually found a level in a schema where it was calling callbacks in a loop. Upon fixing this issue, the route worked perfectly.

c) **Designing System Components**

· I designed a large portion of how event request invites function. I created a new schema for a `Request` object. Each `Request` object is associated with an `Event` object and vice versa. When a new `Event` is created, a matching `Request` is also initiated, because I realized that each event can have multiple event requests stemming out from it. Key elements of the `Request` schema are a reference to the original `Event` it stems from, an array of all of the user ids that the original creator has invited to the request, an array of those each invited user and their status (pending, accepted, rejected, etc), and an array of edits to the `Event`. When an invited user accepts a pending request, he creates a copy of the original event and adds it to one of his calendars. That way, if the original creator of the event edits the event, the copied event is not altered until the user re-accepts the changes.

d) **Dealing with Realistic Contraints**

· Javascript's asynchronous nature made it so that some routes would send a response back before parts of the route code would complete. This was a nightmare to deal with, but we fixed this problem by looking into promises. Promises allowed us to dictate that certain parts of the code could not be entered until specific parts of the code had been completed. With promises, we became much more in control of how our code works.

e) **Teamwork and Team Member Interaction**

· Our group was much more cohesive and organized than we were for evolution 1. We were able to plan out meetings ahead of time, hash out design details earlier, and finished all of our evolution 2 requirements with time to spare. As usual, Parker and I met often as the back-end team very often.