Evolution Three

ECE 458

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

Abstract

In this evolution, we added two new features to our shared calendar web application. Specifically, a user can now create Slot Sign Up events, which allow other users to "sign up" for an appointment. The second feature is more a service as it allows the user to determine who has event conflicts given certain time frames. In addition to these new features, some major back-end refactoring was done and will be further explained in the following document.

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 members code. Figure 1 below shows a high level diagram of how we decided to design our calendar web application.

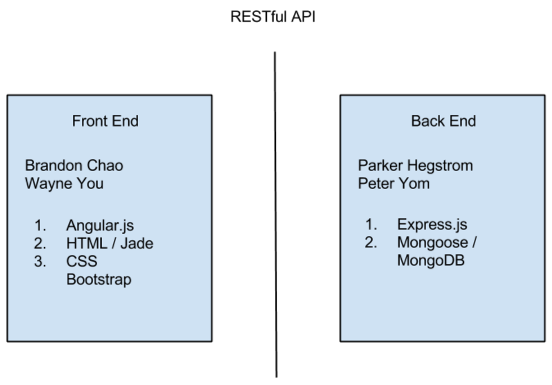

Figure 1: Diagram of our Large Scale Design

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.

At the beginning of evolution three, our group spent four days simply refactoring and making large design change decisions. The back-end did undergo one major change as we now handle repeated events differently. This will be further discussed in the following section.

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

The two new features as well as other design developments are discussed below:

2.1.1 Slot Sign Up Events

To implement the slot sign up feature, we created an entirely new structure in the database. This structure, `SlotSignUp.js`, allowed us to develop without the fear of breaking something already created in the first two evolutions of the project. To handle the dynamic nature of the `SlotSignUp`, we used the composition design pattern. More specifically, a `SlotSignUp` is composed of individual `Slots`, which can be created and associated with any user that signs up for a specific `Slot`. Initially, though, the `SlotSignUp` is composed of free blocks of minimum sign up length. As people sign up for slots, the `Slot` objects are created and tied to that user. This information is stored in the `attendees` property of the `SlotSignUp`.

A user must be able to determine the status of the sign up slot (list attendees, what blocks they've signed up for, what free time is left, etc.), and we wanted to make this information as easy as possible to display and edit for the front end. Hence, we stored an attendee's `email` with his/her slots. This allowed for immediate access to client-ready, displayable information and removed the need for the client to issue additional database queries. Before, the client would have been given a list of user ids and would have had to query the DB with those ids, presenting un-needed latencies to the end user of our application.

Lastly, we wanted the slot sign up to be easily modified or extended in the future, thus was the impetus for using the composition pattern. The `Slot` was extracted into its own model file and is therefore independent of the `SlotSignUp`. Say, for example, we wanted to add Alerts to our individual Slots. This would be as easy as creating a new `Alert` property in `Slot.js`.

2.1.2 Find Free Times

Find free times did not require any database changes, nor did it require any new models to be created. This requirement was solely algorithm driven and the control for this feature can be found in `freeTimeRoutes.js`.

There is one route in `freeTimeRoutes.js`: `PUT /ftr/findConflicts`. The goal of this route is to accept users, usergroups, and a list of time slots, and return a comprehensive 'conflict summary'.

First, an array of available events is created from each passed in user. This array contains only events that the logged in user has view, modify, or view-busy access to from the list of passed in users and user groups. The array of all events is sorted by `start` time. On the other hand, the list of time slots is sorted by `end` time. We then developed an algorithm to compare the array of events with the array of time slots in $O(n + m)$ time with m=number of events and n=number of time slots. In the conflict summary, each time slot has an array of events which conflict (aka have times that interfere with the start and/or end times of the time slot).

In the conflict summary, each time slot also has an array of free times between the start and end times for the time slot. This is determined with the 'slotSize' in mind.

We designed the conflict summary to be as complex and detailed as possible to minimize the number of calculations needed to be done by the front end.

2.1.3 New Implementation for Repeated Events

Before, our repeated events were somewhat fake and display driven. We would have one real event in the data base and simply display the event in the future if there was a repeat. We decided we wanted a more powerful approach, one that allowed for repeated events to behave similarly (deleting all at once, sharing same properties) but also behave independently. This new implementation also allowed our code for Find Free Times to run and handle the repeats properly without any additional modification.

To implement this new design, we created a new structure, the `RepeatChain`. The purpose of this structure was to keep track of all events that were created as a repeat chain, allowing us to edit individual events in the chain as well as delete all repeated events the same time.

2.1.4 Async.js

One difficult we've mentioned many times in our discussions and presentations has been the asynchronous nature of javascript. In this evolution, we further implemented a new node module: `asyn.js`. Under the hood this module works with promises, but this api allowed us control asynchronous dependencies. After witnessing the power of this module, we were able to clean up the call back hell situations we got ourselves into in the past.

2.1.5 Schema Methods

Before this evolution, we never used Mongoose Schema methods which leaving our route files cluttered and at times difficult to read. Schema methods allowed us to extract Schema specific behavior and treat our Schemas like classes with an api. For example, we have a `User` schema. Adding a schema method allowed us to call functions on a `User` like `User.convertToEmail()`. Extracting schema methods severely cut back on our repeated code throughout the project. For a working example, see `./models/User.js`.

2.2 Benefits of Our Previous Design

The major benefit of our previous design was its modularity and composition-driven nature. The modularity allowed for easy addition of new features and simultaneous development by all of our team members. It also allowed for easy and testable code as well, because we could test single files or single methods that composed the entire module.

The composition-driven nature of our design proved to be the most powerful thing we've done thus far. That is, we were able to add new features or extend the functionality of previously working code in extremely easy manner, all the while having these changes to database model files persisted throughout the entire database. For example, this evolution required a User to be able to keep track of slots he/she had signed up for or sign up events he/she had created. This was easily done by adding a new property to the Schema Model that "pointed" to the actual `Slot` or `SlotSignUp` objects in the database.

2.3 Drawbacks of Our Previous Design

The way we initially design and implemented repeated events proved to be extremely limiting with the new requirements of evolution three. As mentioned before, repeated events used to be simply display-driven, meaning that there was really only one event stored in the data base.

This shortfall was changed such that all repeated events are now actual events in the database.

3 Front End Design and Analysis

4 Individual Portion

Parker

a) **Designing and Conducting Experiments**

- A problem we were having dealt with the data type of a documents `_id` value. This `_id` value is was is assigned to each document in mongo db. Sometimes we were getting a `string` and other times the type was `object`. The experiment was simple. Create a document in the data base and access the `_id` property by using various methods. Specifically, I would access it using either `user.id` or `user._id` and printing out the `typeof`. Viewing the results yielded the conclusion that `.id` returns type `string`.

b) **Analyzing and Interpreting Data**

- I think my previous answer also works for this bullet as well. How did I discover that we had a typing problem with the document `_.id`? I had noticed weird error messages in the console, analyzing the console to pinpoint where the problem was coming from exactly. Then, after inserting print out statements, I began to study the `typeof` return value to determine what the end problem was.

c) **Designing System Components**

- As mentioned earlier in this document, the way we previously implemented repeated events was not ideal. So, I designed the new repeated events system. I wanted to make the repeated events act as one in some ways, but in other ways be able to act independently. To do this, I made it such that all repeated events are created in the database. They are of the same type as normal events. However, I did create a new data type in the database, the `RepeatChain`. When an event was created with repeats, a `RepeatChain` will be created such that the system always knows what events were created by a repeat. This allows for easy deletion or editing of all of the events at a time.

d) **Dealing with Realistic Constraints**

- One constraint we found was that it was very difficult to create a mongoose schema with JSON object as property value. For example, property `attendees` was composed of an array of JSON objects. For some reason, mongodb / mongoose could not handle the editing of such JSON objects and save them properly. Hence, we had to devise our own way to edit the JSON objects to get around this constraint. The solution: set the property to null, and recreate the JSON object every time rather than try and edit the JSON object individually while still a part of the mongoose document.

e) **Teamwork and Team Member Interaction**

- For the most part, everything went very well. All member contributed equally. Specifically, Peter and I met often to discuss progress on our individual work and to discuss what the next design decisions would be. There were some times, though, when there was a lack in communication between the front end and back end which led to some difficult problems down the road.

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 Constraints**

·

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**

.