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.

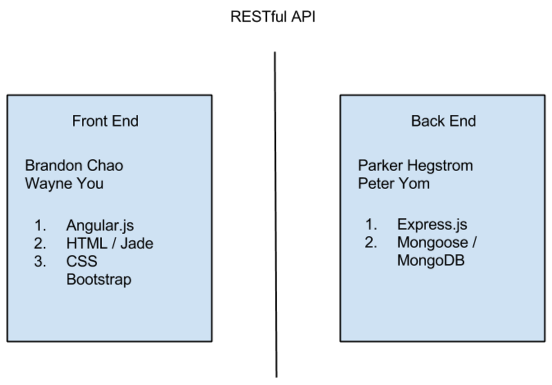

Figure 1: Diagram of our Large Scale Design

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

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**

.

b) **Analyzing and Interpreting Data**

.

c) **Designing System Components**

.

d) **Dealing with Realistic Contraints**

.

e) **Teamwork and Team Member Interaction**

·

Peter

a) **Designing and Conducting Experiments**

- While developing the algorithms to find conflicting times or slot sign up free times, I would write out simple test code with the simple javascript test tools on `w3schools.com`. Javascript Date objects can be compared and modified like simple numbers, as they are essentially large millisecond values, so I tested my algorithms with simple numbers to save time. After I was satisfied with my tests, I created simple test routes and tested them with PostMan with more realistic values.

b) **Analyzing and Interpreting Data**

-

c) **Designing System Components**

- I designed the creation of the conflictSummary, the merge and compare algorithm to find conflicting events and free times, the slot sign up creation, signing up, and canceling of slots. I designed the structure of the conflictSummary object to return to the front end. It is essentially a map with time slots as keys and conflicting events and free times as values. This makes it very easy for the front end to display conflict data per time slot. Compared to previous evolutions, this evolution was less about creating new structures and more about developing algorithms to process information. For example, I wrote the algorithm to compare events with time slots to find conflicting events. I sorted the events by start date and the time slots by end date. Then I had a pointer for each array that would increment accordingly.

d) **Dealing with Realistic Contraints**

- Javascript's asynchronous nature made development incredibly frustrating at times. At times, we needed functions to be called in a specific order but certain functions, namely anything dealing with calls to the database through Mongoose, would take too long and would complete when it was too late. Parker and I found the answer to our problems with the async library and the waterfall function. Waterfall takes in an array of functions and completes the functions in order one after another. This allowed us to have much more control over how our code works.

e) **Teamwork and Team Member Interaction**

- For the back-end, Parker and I met up and worked together regularly. We had group meetings with Wayne and Brandon on occasion as well in order to catch everyone up to speed with all of the new developments. Towards the due date of the evolution, Parker, Brandon, and I met up to finish up. Wayne chose to work remotely despite our suggestions to work together in person. This led to difficulties in communication and quite a few frustrating bugs. For the final evolution, we will need to collaborate and meet up as much as possible especially right before the evolution is due.

Brandon

a) **Designing and Conducting Experiments**

·

b) **Analyzing and Interpreting Data**

·

c) **Designing System Components**

- I designed the finding free times feature for this evolution. While the initial modal to find free times is defined in the modalController, functions to display the conflict summary and interact with events are defined in a separate conflictSummaryModalController. We were originally not able to do this since we weren't able to define multiple controllers that needed the same data; however, the Angular services that we discussed above allowed us to share variables between scopes of different controllers which allowed us to split up the controllers into smaller, more specific files. When a free time is found in the conflict summary, I use it to populate the times in the event creation since those variables are stored in the rootScope. From there,

d) **Dealing with Realistic Contraints**

·

e) **Teamwork and Team Member Interaction**

- For this front-end of this evolution, I worked on the finding free times while Wayne worked on the slot sign ups feature. Since these were very separate features, we did not interact very much throughout the evolution. We met at the beginning of the evolution to plan how we wanted to refactor our code and how we wanted to implement each new feature. Towards the end of the evolution, Peter, Parker, and I met to finish up the evolution. However, Wayne worked remotely and we interacted through messaging. We feel like debugging errors in our evolution may have been easier if we were all able to meet together.

Wayne

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**

.