ECE 458 – Spring 2015

Evolution Four

Leeviana Gray
Martha Barker
Robert Ansel
Susan Zhang

Contents

1 Retrospective

In retrospect, for this evolution we had to add new controllers and models to our design as well as add methods and fields to existing controllers and models, but we did not have to make significant changes to existing code to implement the new features so our previous design worked well for maintainability.

1.1 Good Design Choices

1.1.1 Frontend

On the frontend, we were able to modify a few of our existing pages to incorporate new features. We modified and reused the events page, the event creation page, and the event info page. The events page needed only minor modifications to show unresolved sign up events differently and show the resolution option at the right time. The event creation page had more major modifications to include preference based sign ups which we displayed a different set of form fields for than other event types. We were able to implement these dynamic fields the same way that we did in previous iterations using jQuery. For the event info page we modified the sign up slot options so that they could be preference based or not. For all of these modifications, we mostly just added logic and did not have to change existing code. Once again, reusable components made building frontend fields relatively easy. The recurrenceHelper component that we created in the last iteration to handle the fields and jQuery associated with filling a recurrenceMeta object was able to be reused for the new PUD escalation feature on the create event page.

1.1.2 Backend

PUD expiration was simply adding a textbox and lines of logic, due to the fact our all-purpose timerange object that we were using for our PUD's "duration" already had "start" and "end" date objects built in, so that was nice. Escalation involved just putting in a filter query based on parameters. In fact, even scheduling was mostly data structures and an intense algorithm on the backend. The clearly defined boundaries between the frontend, controllers and models, which have only gotten better over time, allow us to maintain this ease of adding new features even as our project becomes bigger.

1.2 Bad Design Choices

1.2.1 Frontend

As the number of different types of events increases, we have more different combinations of information in the create event form, the event page, and the specific event information pages. Our design uses the same html page for those functions for every type of event so the pages are getting cluttered with conditional logic for all the possible combinations of user options, which is not conducive for extensibility. This makes the pages more difficult to read and modify because we are trying to do so many different things with them and Eclipse's IDE is not helpful when editing Scala.html files.

1.2.2 Backend

The part that always takes the most significant consideration and adjustment is the model changes and this iteration wasn't any different. Our events model was getting quite cluttered - we had a bunch of random fields in our Event object for different types of specific objects, (i.e. specifically PUD related fields like priority), so in order to clean up our design and make logic a little more clear, we factored these fields out into extra "meta" objects that were optional (PUDMeta, SignUpMeta). This cleaned up the event model a bit and allowed us to specify things like priority as required for all PUDs, even if it's not a required field for all events in general.

Our sign up determination and scheduling algorithms are the most complicated methods in the controllers, though that was probably was unavoidable due to nature of the algorithms. They could probably be broken down into smaller helper method to improve readability.

Lastly, on a front-end submission/form-related note, we also finally concluded that it was just easier to have all forms, no matter how trivial, be represented as a form, for the ease of reusability and code clarity. Hence new models like our "preferenceForm".

2 Evaluation of Current Design

2.1 Design Choices

Since PUDs are a type of event, to add PUD expiry we were able to reuse the end date fields that we were already using for fixed events. This means that all we needed to do to implement this requirement was modify some query logic in the controller and what fields the user sees when making and looking at PUDs in their calendar.

To implement PUD escalation, we separated out the PUD fields into their own model that is part of the event model. The PUD model has fields for escalation start, frequency, and amount. Since PUD escalation is similar to recurrence, we reused the recurrence model to store the start date and frequency. We then added a filter query to the events controller.

For preference based sign up we reused the slot sign up controller we used in the last iteration and added new methods to it for the new requirements. We also reused the sign up slot model and added a field for a list of possible users. We also added two new models, SignUpMeta and SignUpPreferences. When users indicate their sign up preferences, the new sign up information is appended to the signUpSlots list in the model and the events are added to the user's calendar with a different viewType from fixed events or resolved sign up events. If the preferences are resubmitted, the old preferences are deleted and the new preferences are appended to the list.

At the specified resolution time, the sign up event creator sees the option to go to a new resolution page. There is a backend query which assigns suggested users to the sign up slots in the list. These are displayed with a dropdown containing the users available for each timeslot. On form submission, a new sign up event that is not preference based is created with the assignments and the preference based sign up is deleted.

2.2 Strengths

2.2.1 Frontend

For this evolution, we were able to reuse existing components for the majority of the new form features, allowing for rapid development. There were two cases where existing components could not be used to implement functionality, so we created two new component types: number2.scala.html, a number component that sets a field but takes in a possible value from the event object; and signUpDropdown.scala.html, which is similar to the original dropdown, but takes in a list of UserSignUpOptions instead of Strings. The fact that we only created three new form components since the first evolution reflects well on the reusability of the original components that we created.

2.2.2 Backend

Overall, the backend separation between the model, view, controller worked out very well for being able to add things effectively and for being able to parallelize work between the backends and frontends, especially for this particular evolution. Our eventual solution for passing back and forth slot scheduling information actually worked out pretty nicely with our new "SignUpMeta" Event field that we introduced to contain all relevant sign up related fields. We ended up being able to just pass the forms for the meta field (as opposed to the entire Event object) back and forth.

2.3 Weaknesses

2.3.1 Frontend

After working on this project for four evolutions, weve come to realize that even though we tried to separate the frontend from the backend during the planning stages, the two sides are not completely independent from each other. Its easy to build new form components with our reusable components, but the more important part is ensuring that the new components map to the backend fields correctly. This cannot be tested until the backend is completed. It is easier to build up parts of a new form and test the fields incrementally than to complete the entire form and then test the entire thing. When working on something like this in the future, it would be useful to find a way to show the intermediary form data before the backend receives it in order to decouple the frontend and backend.

Though we did talk about the possibility of adding automated tests during previous evolutions, we never did get around to adding them. However, we became much better at manual testing throughout the course of the four evolutions. Additionally, we added more conditional logic to pages that handle all types of events, causing them to become harder to read and less extensible.

2.3.2 Backend

The algorithm method (sign up determination) could end up being a bit more computationally intensive than most of the controller, but I don't know if there is actually a better way to do it, besides continuously looping through and assigning slots in a logical order, it could stand to be broken up into smaller methods though, as mentioned above.

One of the most subtle weaknesses of the backend is the fact that all events are deleted through a single "delete event" controller method. For simplicity and code reuse this is great (i.e. one method to delete the event and delete all slave events, instead of doing that logic multiple times). However when trying to figure out what exactly happens when you delete an event and why the delete method actually does, the compilation of uses for the single controller is unhelpful. One possible way of dealing with this is to do something similar to the create[Event] method, where there's a bunch of checks for certain structures and types and separate, tailored, methods are called, potentially allowing for even more logical code coherency and reuse.

3 Individual experiments

- Tell me about an experiment you designed and conducted
  - Tell me about data you analyzed and interpreted
  - Tell me about how you designed a system component to meet desired needs
  - Tell me how you had to deal with realistic constraints
  - Tell me how you contributed to team work and interacted with your team members.

3.1 Leeviana

The algorithm I ended up using for sign up determination (to maximally fill slots using preference), used this number called minNum, which basically counted up until there were slots with less optiona than it and the lowest "preference number" of the options was less than the minNum. When such slots are found, minNum is reset to 1. When no slots are found in an iteration minNum goes up. This makes a "best effort" at maximally filled slots (I believe it isn't perfect, but there are no obvious counterexamples that I can think of). The check for preference number less than minNum is an attempt at causing a user's lower preference choice to be assigned first, all other things (number of people that signed up for the slot) equal. This means that if a single person signs up with preference numbers 4, 3, 2, 1, the first runthrough would not just assign the first slot with the lowest amount of signups (the 4), but would actually only hit the "1" signup slot in the first iteration, which is realistically what the users would expect.

The formulation of this algorithm came from lots of iterations of thought. The first ideas for the algorithm were even more brute force-like with checks throughout the entire slot list for every single assignment to ensure optimality. Then after more thought, this became the minNum concept as applied to the number of userOptions each slot had. The preference weighting came later when I ran my algorithm and realized that when I signed up for the 4, 3, 2, 1 pattern as described above, I was assigned to my 4th choice, which was maximally filling but not logically satisfying from the user's standpoint.

As always, I did the model/controller additions and modifications, shaping the data flow in such a way that frontend integration would be possible without too much effort.

3.2 Martha

One problem I ran into during this iteration was getting the list of users available for each possible sign up slot on the schedule resolution page. The sign up model has a list of sign up slots and each sign up slot has a list of users available. A form of type SignUpMeta is passed to the front end to display for the user and submit slot selections, but I had difficulty getting the slot list out of the form in a way that I could iterate through it.

The syntax to access certain fields from front end pages has been a difficult problem many times before this, so I tried every possible line of code that had worked for other form fields to get the information out, but none of them worked because we had never pulled a list out before. I also read a lot of internet documentation on forms, especially the repeat function, but none of them helped. Eventually I just worked around the problem by passing the list to the front end seperately from the form and used it with the form to display the necessary fields for the user to assign sign up slots based on preference.

I worked with Susan on frontend pages with Susan again, occasionally doing a tiny bit of back end when I needed to for what I was doing on the frontend. We also met as a full group several times to work on the design at the beginning of the iteration and towards the end of the iteration as we were pulling the project together.

3.3 Robert

3.4 Susan

An issue I ran into during this iteration was setting users for slots during schedule resolution for SignUp events. We were able to have each slot show up on the schedule resolution page with a dropdown that showed the users that prioritized the slot and the priority number that they assigned it. The form was able to submit and make changes to the backend, but caused a null pointer error for timeRange end on the events page.

To pinpoint the source of the problem, I created a SignUp Event with preference-based signup and a preference resolution date in the past. The owner of the event, Susan, set preferences for the slots on the event information page and also invited the user Bob to the event. Bob set their preferences for the event, and from the Susan account I navigated to the resolve preferences page. At this point, I checked the database and saw that the events the submission created did not have end data. I assigned both Bob and Susan slots and clicked submit, triggering the error. I checked the database again and it turned out that the events created by the submission lacked an end date and time. I went back and checked the form mappings and found out that it was missing end date and end time fields for form submission. Additionally, the userID field was being mapped incorrectly. Once these were fixed the form submission produced the expected backend results.

A recurring constraint that we had to handle was the fact that it was difficult to manipulate integers on the front end in Scala.html files. Therefore, the backend would have to pass in the length of a list if fields are generated for each element that list, like when signing up for slots on the event information field. This forces an additional parameter to be passed to the frontend, but did not cause serious problems.

Once again, I worked on the frontend pages and forms with Martha. Whenever we needed a backend method modified or suspected there was an issue with backend code, we could contact Leevi.