ECE 458 – Spring 2015

Evolution Three

Leeviana Gray
Martha Barker
Robert Ansel
Susan Zhang

Contents

1 Retrospective

This section is a retrospective view of the design choices we made for evolutions 1 and 2 considering the requirements of evolution 3.

1.1 Good Design Choices

1.1.1 Frontend

On the frontend, our increased familiarity with the frameworks improved our efficiency of development. If we forgot how to implement a feature or were unsure of exact syntax, we could usually go back to a previously written class and reference code from there. For cases that we hadnt handled before, we still had to refer to additional online resources, but this was less common. Bootstrap and jQuery were mostly able to handle the requirements of this evolution; adding dynamic fields was the only feature that required a workaround. Once again, the modularity of the frontend forms allowed for rapid creation of any new forms. DBDAOs were also extremely useful for this iteration, allowing for routes in the controllers to be less complex and decreasing dependencies.

1.1.2 Backend

Our backend model design held up very well for slot sign up implementation. For signup events, we were able to reuse our event model again (which allows us to reuse large portions of controller and view code) and only had to add a SignUpSlot abstraction model, plus a couple extra controller methods to deal with the creation of signup slots and the process of signing up and deleting your signup for slots. Our strong model/view/controller definition kept the impact of backend changes minimal, especially since we've learned how to effectively use form mapping to completely separate the frontend views of the models (form fields) from the fields of the models themselves.

1.2 Bad Design Choices

1.2.1 Frontend

The specificity of using the Play! framework with Scala and jQuery made adding dynamic fields difficult. The process of adding simple fields to the frontend using jQuery is well documented and easy to implement. Adding additional modular scala.html fields non-dynamically using Plays Scala repeat helper is also relatively simple. However, adding modular scala.html fields dynamically using jQuery is very poorly documented. We ran into multiple errors involving JavaScript variable scoping, parenthesis matching, and issues with nested scripts before we found a workaround. We are likely to run into similar issues in the final evolution as requirements increasingly get more complicated, but we will handle those as they arise.

1.2.2 Backend

Implementing the scheduling of free times was a bit trickier, because of multiple possible interpretations and implementations of the evolution requirements as well as tricky query logic required to implement the possible time slots and conflict detection query. A small subset of the event creation form and logic ended up being replicated in the Scheduler controller. The replication of the creation form was due to the fact that all the options given to the user while creating a form were not remotely necessary for a simple scheduling query, plus we had to include a field for entities that the new event was to be shared with. This problem could have been mitigated by implementing a list of sharees directly in the event model instead of using our CreationRequest abstraction, which can occasionally be useful but does cause some extra logic

With this evolution, we also found the need to associate multiple timeranges for an event (for signups and scheduling queries), which caused some havoc but not nearly as much as were expecting.

2 Evaluation of Current Design

This section is an evaluation of the design choices we made to implement the requirements of evolution 3.

2.1 Design Choices

Main design choices: SignUpEvent - Sign up are created like regular events, with the ability to add multiple TimeRanges and a slot duration. The controller then iterates through the TimeRanges that the user specifies and uses the "slot duration" to create a list of appropriate SignUpSlots (with no user signed up) which is stored in the event model. When the owner shares this event with another person via rules (modify or seeall) this event shows up on the other's user's "sign up event" list, and they can sign up for slots on the event details page.

When they sign up for a slot, an event is added to their main "fixed events" calendar. If they want to change their slot, they can just delete their event and the corresponding slot will be cleared in the signup event.

Free Times - To find free times, users enter their timeRanges of interest, duration they want to schedule something for, possible recurrenceMeta data, entities they want to schedule this event with (groups and/or users), and the name and an optional description for this event. The query returns the possible timeslots specified by the query (including recurrence dates) along with conflicting events and a prefilled form with the information needed to create an event and create creation requests, which is presented to the user as a handy dandy button.

2.2 Strengths

2.2.1 Frontend

Once again, we did not make any major design decisions on the frontend. We added a Scheduler page for the Find free times requirement and added to the editEvent page for the Slot signups requirement. It made more sense to display potential slot signups on the EventInfo page than to make a new page, so we added a conditional display for slot signups at the top of EventInfo. We did not need to write any additional new components to build any of the new pages or features for this evolution, which reflects well on the reusability of our existing collection of ViewComponents. We did however, refactor out the recurrence fields and their respective jQuery out into a scala.html class, since the code was getting lengthy and the set of components are now used in two frontend classes.

As the frontend logic gets more complex, the scala.html files are able to handle the logic relatively well. It is easy to manipulate the existence of frontend html elements through logical checks performed by Scala, and even though we are limited by the use of only if and else statements, they are able provide the desired end functionality.

2.2.2 Backend

Much of the new event creation logic (like recursion) was refactored out of the original event creation request to improve code readability and allow us to reuse that logic in other controllers and event creation methods. Particularly in the implementation of the "auto create event and create creation requests from conflict query results" button. In addition to being factored out, recursion logic was greatly simplified this iteration, by the implementation of a "recurDuration" field that is set to a Period of time that can be used to generically increment any DateTime in our calendar. Additionally, we also finished the process of refactoring out the "event filter" query responsible for determining which events a user has access to. This query is returned in its raw form so that it can be easily combined with other query parameters.

Besides refactoring, most of the major design decisions are described above. The backend gained a new, incredibly useful SignUpSlot model, as well as some new controller code for dealing with Scheduling and SignUp logic, in new controller classes. Most of this can be seen in the new code and is fairly self-explanatory.

2.3 Weaknesses

2.3.1 Frontend

We were not able to find additional time to look into automated testing during this evolution, so everything is still tested manually. However, we have become more adept at accessing the backend database, interpreting error messages, and identifying problem areas, so the process is definitely more efficient that before. An issue we mentioned in the last evolutions writeup was that the Eclipse IDE does not provide useful feedback for

scala.html classes, causing errors to be found at compile time. An alternative text editor, Sublime Text 2, is able to identify matching bracket and html tag pairs, which is useful for debugging purely frontend issues. Sublime was not useful for providing feedback on the Scala aspects though. However, we were not able to find an Eclipse scala.html plugin that corresponded with the version of the Play! framework (v2.3) that we are using.

The design of the editEvent page to account for adding additional sign up blocks limits the sign up blocks to 10, since we are hiding and showing divs instead of actually dynamically adding divs. This is not as extensible as it should be, but please see Susans individual section for more details on the problems encountered and rationale behind this decision.

2.3.2 Backend

As pointed out before, it would be nice if the forms used for the scheduling query and event generation was exactly the same as the form for the query, but it isn't and so there are duplicated fields in that form serving the same function as in the events form.

The recurrence code is much better now, but as we've been using the old version for a while and have been ignoring a few nice features that recurrence could offer, recurrence functionality/behavior may be a bit undefined or strange.

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

Experiment: Here's a non-debugging one. Implementing any new features that need to use the RecurrenceMeta information would take on average 20 lines of extra code due to the specific recurrenceMeta objects for each type of recurrence, DayMeta, WeekMeta, MonthMeta etc. The type of recurrence would always have to determined in order to know which object to utilize. I wanted to see if we could reduce the amount of code.

In order to conduct this experiment, I replaced the dependency on the specific Meta objects with some initial logic when first reading in the form data. This initial logic defined a "recurDuration" field, which stored Period object that I could generically use in my other logic.

When I went back to my initial recurrence case statements I found that I was indeed able to reduce the amount of code significantly in each place in my code where I dealt with recurrence issues. This was a success.

Analyzing Data: What happens when you run your program and the database closes all connections and crashes? Probably an infinite loop in the new code written. What could cause that? Probably a duration of 0 in the scheduling slots. There were lots more such instances while debugging.

Design for needs/constraints: For scheduling queries, the "query" form was designed to only have the individual fields of the event model that SignUp events care about, plus entities needed for creation request generation.

Teamwork: Again, I did the model/forms/controllers code, while my teammates worked on frontend integration.

3.2 Martha

For the scheduler page, submitting the form to find free time returns the user to the same page but with the free time options and conflicts showing at the bottom and the form fields populated with the users requests. When I first submitted the form it was returning to the form page but without the populated fields or the free time options shown. Because of the form that was being shown, I realized that on clicking submit the wrong method was being called. I confirmed this by putting print statements in the method that I thought

was being called and the method I wanted to be called and seeing that the wrong method was indeed being called. At first I thought the problem was in the routing. I tried to change the POST route using other forms routes as examples, but that did not work. Susan has more experience with forms than I do, so I then asked her for help and she quickly identified the issue. The form method in helper has an action parameter which is the route when the submit button is clicked. I had been confused and had the wrong route there rather in the routes page. Fixing this exposed an error that had been hidden by the incorrect route. When I clicked the submit button on the form I got a Bad request: missing boundary header error. Other users had the same error so there was help on the internet. I needed to add the enctype parameter to the form specifying how the data should be encoded when sent to the server.

Once again, Susan and I worked primarily on the front end. We met a few times with the whole group to design the iteration and work as a group. Susan and I also met just with each other to work on the front end.

3.3 Robert

During the course of this evolution, I focused on improving our form submission, validation and error workflow. For most forms, when a field error is detected (bad formatting, etc), the server sends the original form data back and includes errors showing which fields were flawed. For forms which have dynamically generated content (such as a list of calendars associated with the current user), this content must be re-queried from the database, and regenerated and formatted before responding to the request. I developed an alternative solution in which any of the data contained within the Form can be stored as a cookie by the client and then repopulated directly from the cookie-cache in the case of a failed form validation.

First I experimented with full Form object to JSON conversion, but after spending a significant amount of time studying the complexity of Scala Play Form objects, I established that such an advanced conversion process would be infeasible. Instead I moved on to the concept of converting JSON to/from generalized model and mapping objects which can be operated upon in scala directly, only converting the simpler objects which directly needed to be maintained by the form.

I also began work on the concept for a CookieDAO (data access object) which, should further cookie-based interaction be required, will greatly simplify the process of reading and altering the content of the client-side stored data values.

The second major area I worked on was on the process of high level validation of submitted forms as a whole, rather than as a set of individual fields. Previously, forms fields were only able to be validated on an individual basis and, even then, only based on the formatting of the string they contained. I set up the code necessary to validate the forms as a whole (for example ensuring startDate ¡ endDate, etc.), as well as factor out the code needed for validation into external methods where more extensive database-based validation can occur, checking for references to non-existent objects and other more complex validation steps.

3.4 Susan

For the multiple Slot sign up requirement, having the editEvent page dynamically create fields when clicking on a button was a bit tricky. To populate a field in a list in Scala, one can index into the field using square brackets. For example to populate the second timeRanges startDate, the field corresponding field would be timeRangeList[1].startDate. This functionality was tested by hardcoding additional fields into the frontend, filling them, and checking that the values matched in the backend database.

Initially, I tried to append the set of start date, start time, end date, and end time fields to a div using jQuery and a JavaScript counter variable to index. First I made sure that the plus button was clickable by setting off an alert when the button was clicked. However, after adding in the code for the necessary fields, the compiler threw an error stating that it did not know the value of the counter variable. It turns out that Scala is compiled on the backend, while JavaScript and jQuery are compiled on the frontend. This means that when the Scala components for each field are compiled, they are not able to access the counter variable. Another alternative that I tried was to make a single div that contained the four fields, and to clone and append the div when the button is clicked. The plan was then to change the attributes of the input fields so they would map to a difference index. After implementing this, the jQuery to allow date and time selection did not work for the cloned dibs. After inspecting the source code, I found that cloning causes duplicates of each name and id, which would be difficult to select to change individually and confused the associated jQuery scripts that allowed for date and time picking. The third option I tried involved the Play Scala

repeat helper; this helper takes in a number n and a set of fields, generates the fields n times and maps them correspondingly to the list. However calling the helper more than once would restart the indexing, making it not feasible for dynamic field creation. Additionally, the compiler threw errors about nested scripts and matching parenthesis when the code generated by the helper was appended to an existing div.

To finally solve the problem, I decided to hide and show existing divs holding sets of fields based on incrementing and decrementing JavaScript counter. Since fields will map to the form regardless of whether the divs are displayed, we also included a new field in the Event model, timeRangeCount, so the backend knows how many signup blocks are actually valid. When the plus icon is clicked, an additional set of fields will show up, and when the minus icon is clicked, the most recently added field will disappear. Edges cases are also handled accordingly so the counter cannot go out of reasonable bounds. Though hiding and showing divs is not the most ideal for extensible design, the constraints of using Scala with the Play framework forced us into using this design for the time being. If there is free time during the fourth evolution, I will look into more extensible ways of accomplishing this.

For the third evolution I once again worked on the frontend with Martha. Martha and I would meet to work on the frontend, and if we found errors or needed additional functionality from the backend we could contact Leevi. It was very useful to have multiple class workdays during this iteration, since it guaranteed that all four members were available to meet during the time.