ECE 458 – Spring 2015

Evolution Two

Leeviana Gray
Martha Barker
Robert Ansel
Susan Zhang

Contents

1 Retrospective

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

1.1 Good Design Choices

1.1.1 Frontend

For the frontend, the learning curve for the frameworks began to ease up during this iteration; much less time was spent on reading documentation and looking at examples. We were able to solve problems that we had during the last evolution that were due to our inexperience with our chosen technology stack, such as adding an Await call for a Future to prevent the frontend page to render before a DB query returns. Choosing Bootstrap and jQuery as the frontend frameworks was, overall, a good idea. The two frameworks have large user bases and answers to most questions can be found online. Additionally, they work well in conjunction with each other and allowed us to implement the required features. The modularity of the frontend form components came in useful when refactoring or building new forms during this iteration.

1.1.2 Backend

Similarly, on the backend the learning curve was a lot nicer this iteration since Scala/Play weren't completely foreign to us any more. We were able to fairly easily reuse our events model for PUD events and for events allocating time to work on PUD events which saved us a lot of hassle and duplicated code logic. Our AccessType enumeration/field held up well, with the frontend being able to easily change behavior off of it.

The RecurrenceMeta and TimeRange abstractions also held up surprisingly well. Internally both were not the cleanest they could be, but the fact that they were abstractions in and of themselves meant that they could be reused easily whenever recurrence or time was needed and later, when refactoring timerange, the drastic model change didn't break nearly as many things as it could have because of the nice abstraction.

1.2 Bad Design Choices

1.2.1 Frontend

On the frontend side, one of the consequences of using the Play! framework is having to write all frontend code in scala.html files. One recurring issue that we have had with this constraint is that while Eclipse IDE provides feedback for Scala files and html files separately, it provides close to no feedback for scala.html files. Some of the errors that it points out actually are not errors to the compiler, while some of lines that are not flagged as errors cause errors with the compiler. Thus, actual errors would sometimes only be discovered during compilation and imports had to be made manually, which wasted time. Our newness to both Scala and html compounded the issue, but we are gradually improving on this front. Perhaps we could have chosen a different IDE with better scala.html support, but Eclipse was the sole part of the project that we had experience with before we started this project. At the beginning of the next iteration, it would be worth a shot to look for Eclipse plugins that provide better scala.html feedback.

Also, currently regression testing is done manually, making it very time consuming. Whenever new features are added or merge conflicts between branches are resolved, we have to go and test the parts of the application's functionality that the changes may have caused. In the analysis of the previous evolution, we mentioned the possibility of implementing unit tests using the Fluentlenium frontend testing framework. Building up our project with iterative unit tests would have saved time when making sure that new functionality would not interfere with existing functionality. However, due to time constraints, we were not able to implement the unit tests, which caused us to spend significant time manually testing features. One point to keep in mind, though, is that the changing and the refactoring of the models would have made the tests difficult to keep current if we were building up the tests iteratively.

1.2.2 Backend

There were several things that we knew immediately that we wanted to change in the backend and so we subsequently spend the first few weeks simply refactoring and cleaning code. The two main backend overhauls that were implemented were the implementation of database DAOs and the conversion of all model/data

types to JSON. We started to write some DAOs ourselves, but soon discovered ReactiveMongo-extensions, which conveniently created database DAOs that implemented exactly what we were trying to implement but in a much more well-designed way. With the addition of these DAOs to our project, we were able to clean up our code (it saved at least one or two lines of boilerplate code each database call) and make our database calls a lot more consistent. It also gave us a good excuse to rethink and rework our program logic. This also gave us a good, clean, access method for the frontend to make well-defined database calls from the webpages without a lot of unnecessary code and logic.

The switch over to JSON was actually caused by a specific frontend request - they were having difficulty implementing a calendar that needed a JSON object passed to it - that never came to fruition. But we had been seeing lots of documentation that pointed out that JSON version of all the reactivemongo framework objects we were using were specifically written for usage with the Play framework, so it seemed like a good change to make anyways. For literally affecting every single model and piece of code in the project, the switch wasn't actually too bad. Basically everything that was broken needed to be switched out. Using a good IDE (ScalaIDE based on Eclipse) with compile error highlighting, helped immensely.

Another much needed refactor that we implemented was the redesign of the timerange object. This had been a long time in coming (and actually timerange had gone through several refactors previously). Better familiarity with the language helped a lot here. We had discovered the concept of a "Duration" object, that we were trying to unsuccessfully locate before, and we had a more clear understanding of how to map form fields to model fields, where previously we were stuck with mostly one-to-one mapping with no trickery. The main problem with timerange was that when users enter dates, they don't want to enter a long string, they probably want to enter time and dates separately. But when stored in the database this in unnecessary. Previously we just had a DateTime designated for a date and a DateTime designated for a time for start and end. Now, we combine them which allow us to do logic on our models much more easily. The addition of a Duration object to timerange (also not mapped directly to a frontend form, we're really proud of this novel idea!), also simplifies logic immensely. Because models are separated from the logic so well, this refactor (that touched concepts that were used throughout pretty much every aspect of the calendar), broke surprisingly little, just the obvious things like direct model parameter setting.

Lastly, the one refactor that didn't get done was the recurrence hierarchy. The design of the RecurrenceMeta object leads to some unwieldy if trees to deal with recurrence whenever recurrencemeta is used. However, it actually seemed to be working in a manner, and it does allow for a lot of flexibility in how we store recurrence data so we decided to hold off and see what further evolutions would bring in terms of recurrence to decide if we needed that much flexibility. If evolution three shows no signs of recurrence trickiness we will revisit the idea of refactoring RecurrenceMeta to be a simpler version. Or just create a better designed version that allows for code re-usability, which might happen regardless of evolution 3 requirements.

2 Evaluation of Current Design

This section is an evaluation of the design choices we made to implement the requirements of evolution 2.

2.1 Design Choices

Main design choices: We chose to extend our current events model for PUDs instead of creating a new model which we firmly believe saved us much more time and headache than if we had tried to define our own model.

2.2 Strengths

2.2.1 Frontend

No major new design decisions were made to the frontend during this evolution. Our selected frameworks from the previous evolution, Twitter Bootstrap v3.3.2 and jQuery, provided the necessary functionality required for this evolution. If any new requirements come up that require features outside the supported functionality of Bootstrap and jQuery, it should be relatively simple to integrate a new frontend framework. The modularity and flexibility allowed by scala.html files allowed frontend components to be reusable. Any new forms or changes to forms were built up much more rapidly than the previous iteration due to the

pre-existing components, which were refactored and moved from the views package to the viewComponents package.

The addition of DBDAOs (CalendarDAO, CreationRequestDAO, EventDAO, GroupDAO, ReminderDAO, and UserDAO) to the backend was extremely helpful for the frontend and can be considered a strength. Instead of needing to pass in parameters through controllers, a call to the respective DBDAO after import will provide the necessary information. A common issue we ran into during the last iteration was not being able to request a DB query from the frontend which caused scala.html classes to have long constructors, but the DBDAOs helped solve this problem.

2.2.2 Backend

We are actually really pleased with how the backend is shaping up, especially with all the new DAO usage. It is very easy to develop new features - writing queries is more consistent and uses less boilerplate and methods for the frontend can be well-defined and packaged. Additionally, with reactivemongo, new models are fairly easy to define and write too/from the database. The exception was Duration, which had no inbuilt read/write method and was not as simple as effecting using a string read from the database (like enumerations). The fact that it was returning some type of JsNumber that maps a BigDecimal, that can't be implictly used as either, and sort of sits there as an arbitrary "JsValue" with number-like behaviors, caused us all kinds of headache. Especially when we weren't yet aware of all of this.

All other models behaved like a charm. The event model was able to be extended to PUDs and used for invitations much more easily than we were anticipating. In fact, we had started work on a new "PUD event" model, before realizing how consistent our code was to our old event model. Our event model gained a few incredibly useful enumeration fields (so glad that we know how to use those effectively now...): The AccessType field was joined by an "EventType" field for Fixed or PUD events, and "ViewType" field for displaying the status of invited events. These abstractions have held up fairly well over the course of developing our new features and we are pleased with the separation of concerns held within them. (i.e. a shared event can be of EventType 'Fixed', with ViewType 'Accepted' and AccessType 'Modify', which modifies frontend behavior accordingly).

2.3 Weaknesses

2.3.1 Frontend

A weakness of the frontend is that some of the logic is not as clean as it could be. For example, events.scala.html is cluttered with if and else statements, which is not conducive to extensible design. Due to our unfamiliarity with Scala and html we are currently not sure how we could refactor this in scala.html files, but we will definitely look into this in the next evolution. Also, the some of the frontend classes have become large and more difficult to read. Once again, events.scala.html is a prime example. The events.scala.html page handles both Fixed and PUD events, which could definitely be separated out into two individual classes. We did not establish guidelines for where DBDAOs can be called, so they are currently called in both controller and frontend classes. Ideally, in the next iteration we will set a more rigid standard for this.

2.3.2 Backend

Recurrence is our big weakness right now. The way the recurrence model is designed is not conducive to actually using it with logically clean code. Because of this we have put off cleanly implementing several logical features like mass recurrence deletion/editing (which we actually have all the pointer logic for) until we figure out what to do with the model. We are able to implement recurrence functionality just fine (recurring events/PUDs/reminders), because our model has all the necessary flexibility and data, it just takes some repeated code on both the frontend and backend for each type of recurrence which we dislike. Because of this, recurrence behavior is sometimes rather unintuitive and not well designed. We plan on reevaluating our recurrence model early on in next iteration and doing a complete intuitive recurrence implementation at that time.

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: Every debugging session is a series of conducted experiments... one specific example would be when attempting to figure out what exactly the JsValue trying to be read by the Duration reader was. By trying to turn it into a string and parse it different ways I got surprisingly useful error message that told me things about my format. Like "can't parse 64.8E8", which let me know it was not just reading a string and was definitely treating something as a number. Albeit an unparsable sort of longish number (which I eventually discovered was a JsNumber that was actually a BigDecimal that needed to be handled very delicately)

Analyzing Data: So timezones. When I originally refactored the timerange object, I was surprised that the program was still standing. However, there was a rather irritating side effect that wasn't there before. Whenever I would edit or save an event, the displayed time was 5 hours after the time I had set. Since our timezone happens to be UTC-5 at the moment, I took this to mean that somehow timezones were messing with it. After some more consideration, I realized that meant that the data was probably being passed back to me as a timezone aware object and I was storing it as a UTC object, which added +5 to it. And then when display/reading it back out, it was displayed as a UTC timezone unaware object. Since I wasn't currently controlling reading out from the database I had to alter my read in to account for this difference. However, I wasn't actually able to figure out how to deal with timezones effectively with the joda DateTime object so it's still an active area of investigation.

Design for needs/constraints: One of the things that adding in PUDs caused was a change in how we viewed events. Suddenly events could have different behavior depending on the type of event they were and had to be shown differently depending on its state even when its the same type of event etc. This was the main factor behind adding the viewType, accessType, and eventType fields to the event model in order to cause different behaviors on the frontend.

Teamwork: I mostly did model/backend work, like the creation/editing of models and methods for using those models in the context of our calendar's behaviors. Most of the heavily lifting was in reasoning out the logical flow for things like PUDs and sharing events. We met as a team as regularly as we could (our schedules don't line up very often) to discuss next steps and what we were working on, and communicated a lot through Facebook messenger.

3.2 Martha

One of the requirements I worked on this evolution was the calendar events to reserve time for PUD events. We decided to use the same model for PUDs and for events scheduled to reserve time for them (which we have been calling PUDEvents) as we used for regular events. Leevi added an eventType field to the model to keep track of the different types. One of the decisions we made was whether to update the database with the PUD details or just update the list of events sent to the front end. We originally wanted to just update the list so that we had fewer modifications to the database. That meant that when we went to the information page for an event it accessed the database and did not get the PUD details that fit in the PUDEvent. We then decided to just update the database when populating PUDEvents with PUDs so that the new details were always available.

I worked on populating the PUDEvents with the highest priority PUD that fit in the duration of each PUDEvent. When I first wrote the updatePUD method, it did not have an error, but it also did not work. Sometimes this is the most frustrating because the method is not behaving the way it should but there is no help from the compiler. I started debugging by checking the database. We can access the events in our database in the console, so I was able to check the properties of events to make sure that the PUDs and PUDEvents that I thought were in the database were actually in the database and that the duration of the PUD would fit in the duration of the PUDEvent. Once I checked this, I confirmed that the PUDEvent should

be populating with a PUD and could work on why that was not happening. I then added print statements in the code to determine where it was failing. The first issue was that I was accessing the duration for the PUDEvent in a different format than I was accessing the duration for PUD which meant that the durations would never match and the PUD could never populate the PUDEvent. This was relatively easy to fix and easy to see when it was fixed using print statements. The other issue was that the database query to find PUDs that fit in the PUDEvent was not finding any events despite them being in the database. While I have learned a lot about scala syntax and database queries, I still struggle sometimes with writing database queries that do what I intend for them to do. To debug the query syntax I simplified the query. I first queried for just the eventType and the PUDEvent populated with details from the highest priority PUD. Then I queried for just the duration and the query did not find any events so I knew the problem with the query was in checking duration. I messed with the syntax until the query checking duration found events. Once I had each part of the query working separately, I combined them into one query that searched for both components and each PUDEvent was populated with the highest priority PUD that fit within its duration.

We met as a team to discuss design and strategies, but we wrote most code individually or in groups of two. When we were working individually, however, we communicated using a Facebook thread and also using Trello. We asked and answered questions on the Facebook thread and kept track of the requirements on Trello. This allowed us to work well as a team even when we could not meet with the whole group.

3.3 Robert

Experiment: Data Analysis: Meeting Desired Needs: Realistic Constraints: Contribution:

3.4 Susan

I ran into an issue when trying to render a row of the Events page differently when a user declines an event invitation. When an event invitation is declined by the user, the viewType of the event should be changed to Declined in the backend, so a conditional statement on the frontend should be able to check for this and use a different html row type to render the frontend differently. Originally, I was trying to do compare the enumerated value of models.enums.ViewType.Declined and the viewType field of an event, but even after declining an event on the frontend, there was no visible change in the events row. This was strange since the value of ViewType can only be set to the enumerated values since the input to the field was a dropdown.

One way I tried to solve this was to check to see if I used the correct Bootstrap row class by removing the conditional comparison statement. Once removed, the row did turn red, meaning that it was the correct class. Another concern was whether the viewType has been updated correctly in the database. Querying the database showed that the viewType of the event had indeed been changed to Declined. The Eclipse editor for scala.html files does not provide feedback, so its difficult to check variable types. At first, I tried to call toString on both entities, but this did not appear to change the result on the frontend. To check the actual results of the toString methods, I had them appear within a td tag in the row. The ViewType enum returned the expected value of Declined, but the events viewType field returned Some(Declined). A quick search revealed that a get call must be made to access the enclosed value. After adding the get, the two entities are equal when an event is declined, so a declined event can be rendered appropriately.

Originally, any fields that required the user to input another user had textboxes associated with them. This resulted in the possibility that the user might try to add an unregistered user, which would cause a system error. To combat this, I designed a dropdown that displayed the list of all users. Each option in the dropdown is a users email, which is supposed to be unique for each user. When first implemented, there was inconsistency in loading the options of the dropdown, but this was solved using Await statements in the backend for the query. Our unfamiliarity with the frameworks that we chose to use for our project imposed constraints on what we are able to do in exchange for ease of implementation in other ways. For example, we were not able to figure out how to implement sorted queries from the database in the backend, so some of the sorted tables were actually sorted in the frontend, which worked but was not the best practice. For the frontend UI components, Bootstrap limits its columns to 12 units wide and forces a distinct look and feel in exchange for rapid startup in learning and ease of usage.

Once again, for this evolution I worked on the frontend with Martha. Whenever we needed information that was not currently provided by the DBDAOs, we asked Leevi to write the respective query. At the beginning of this evolution, we created a Doodle poll for every day between the start of the evolution and the due date, which made it easier to schedule meeting times. We also made more use of Trello this evolution,

which made tracking bugs and tasks that still needed to be completed easier when we were not working in the same place or at the same time.