ECE 458 – Spring 2015

Evolution One

Leeviana Gray
Martha Barker
Robert Ansel
Susan Zhang

Contents

1 Evaluation of Current Design

1.1 Design Choices

1.1.1 Web interface

We decided that in the interests of portability and accessibility that we would develop a web app. Any mobile/PC application would have been limiting our ability to develop and share our developments. Two of our teammates own windows phones, which would not have been a very portable application to most individuals in the class.

1.1.2 Play

There were several choices to be made when choosing what to use to build our frontend, such as whether or not to use a framework or not (the choice between flexibility and inbuilt functionalities), and if so, which framework to use. We decided to use Play, a Java/Scala based web framework, with the Scala language, which compiles on a JVM regardless. None of us had any experience with Play or Scala, however we thought that it would be interesting to become familiar with new frameworks and languages. In retrospect, we have learned a lot, but many tasks which should have been trivial have taken us hours instead, due to our inexperience.

1.1.3 MongoDB

When looking at databases we had the option of relational, non-relational, or a text file. We decided that the text file option may have a few limitations... like concurrent accesses for one. We decided that the non-relational philosophy seemed to fit our "constantly evolving development" theme much better, although only one group member had used a non-relational database (MongoDB) before. This choice actually became a factor in our framework decision, as Django, a web framework that one of our group members had used before (the same one that had previously used Mongo, in fact), was primarily designed for relational databases.

To interface MongoDB with Play, we used a helper plugin called "reactivemongo", which provided controller and mapping (serialization) tools, to access and interface with Mongo. However, it is in version 0.10 at the moment, and while working admirably, has a very small community.

1.1.4 Bootstrap and jQuery

We decided to use the latest version of Twitter Bootstrap (v3.3.2) for the project's frontend framework due to Bootstrap's large community of users, thorough documentation, and ease of usage. Bootstrap is currently the most popular frontend framework used by developers today, which results in more online resources, such as tutorials and code examples for us to learn from. Its main documentation page provides examples of its components alongside the code used to generate them. The Bootstrap frontend framework consists of three main parts: a CSS stylesheet that formats HTML elements, a CSS stylesheet that defines reusable interface elements, and a JavaScript file that defined jQuery plugin based components. Integration of Bootstrap with a Play project was very simple; each of the files was placed within folders in the public directory and imported into the project using Play's helper methods. Since Bootstrap requires jQuery, we also imported the jQuery's JavaScript library into our project the same way.

Another option that we considered was the Foundation framework, which provided similar features to Bootstrap, such as an extensive library of prebuilt components and a grid-based layout system. Foundation provides better built-in support for mobile browsers and more options for customization. However, mobile support is not required for this evolution and the additional options for customization make Foundation more difficult to pick up than Bootstrap. Bootstrap's greater popularity and larger userbase also make finding resources online easier.

For Evolution 1, selecting Bootstrap as the frontend framework was a good design choice overall. The vast quantity of resources available made picking Bootstrap up a relatively smooth process, even though we each had minimal prior frontend experience. All of the views in a Play project are "scala.html" files, which allow a Scala class to conditionally generate HTML. This allowed for more modular frontend components

that allowed resuse of code, instead of writing the code for an entire page into a single class. JQuery was able to provide any dynamic scripting features that we needed for this evolution, but if future evolutions require features outside the scope of jQuery, additional JavaScript libraries libraries can be imported into the project.

1.1.5 Server Setup

In preparing for this project, we utilized that Duke OIT/CoLab's VM service to acquire our machine. After considering the merits of a ready-made bitnami stack relative to the drawbacks such a stack might incur when it came to extensibility, we determined that working from clean slate would be best. We selected a basic Ubuntu 12 VM and installed all of our packages from scratch and were able to more carefuly control the configuration of our web stack (MongoDB, Play-Scala, Bootstrap/jQuery). The process of configuring the server happened in two primary stages. The initial setup was the completion of the minimum required work to ensure that the git repository containing our application could be fluidly imported into the server and function without any additional configuration. The second stage of this process occurred towards the end of the project, and involved the 'productionalization' of the system. This stage included daemonizing the Play application to run without user interaction, integrating and forcing SSL communications, configuring the application to operate on standard ports and minimizing the amount of information released by any stray error messages.

1.1.6 General Security

The overall security of our server installation is best described as resistent to trolling vulnerability scans and bots, but in the case of a targeted attack, it would not likely hold up against close inspection. The primary points considered in this assessment include the use of self-signed SSL certificates, which could be vulnerable to a man-in-the-middle attack, single-session, cookie-based authentication in which there is a small chance of collisions in the AuthToken generation, the use of salted and hashed passwords stored in the database, and a local-access only database implementation which precludes connections by external hosts. The primary disadvantages of the current authentication system include vulnerability to a cookie-replay style attack, possible AuthToken collisions, the presence of a UserID within the cookie, and the current limitation of any user attempting to log on in two locations will be immediated logged out from his/her first location.

1.1.7 Database Design

Our Mongo database - caldb, is divided into 6 collections:

Calendar - Events are associated with this entity. Owned by a user AuthUser - Associated with authentication User - Associated with other user metadata, like calendars owned Rule - Defines an entity (group/user) and access (BusyOnly, SeeAll, Modify) Group - Defines a group of users, owned by one of them Reminder - Defines an event reminder/notification

1.2 Analysis

1.2.1 Weaknesses

The TimeRange model needs some work. It currently stores four DateTimes, one each for startDate, startTime, endDate, endTime. Practically, only two of those fields are necessary but that's not how people think when intuitively filling out calendar forms online with date/time pickers and interfacing with their events.

The dynamic nature of the TimeRange object, and its varying usages can cause some programmatic issues. Since TimeRange is used in defining events, reminders, and recurrence metadata, all the TimeRange parameters ended up being set as optional, since they are not always needed in all applications. However, a side effect of this is that now you can define events that have no start dates or times defined that will pass all the form checks, because technically all the TimeRange object's fields are optional.

With all of this in mind, we may be looking at splitting up the TimeRange object and/or not using it in certain model definitions in future iterations.

There is also a great need for a "Recurrence Metadata" hierarchy. The original idea was to have a parent "recurrenceMetadata" object that all the smaller (DailyMeta etc.) recurrence meta objects inherit from. They share common behaviors and parameters, and only differ in the specifics of their implementation of recurrence generation. However, our limited understanding of Scala hierarchies and class relationships prevented this from being as easy as we hoped and we decided to address this later. This will hopefully be addressed early on in the next evolution of our project.

We did not have sufficient time to write unit tests during this evolution, making the refactoring of our code trickier. Unit tests will become crucial as the evolutions become more complex, so it will be important to write unit tests during the next iteration. Potential frameworks we could use for writing frontend tests are Selenium or Fluentlenium, both which use CSS selectors to manipulate frontend objects.

Most of the frontend code started out as a long single class and ended up being refactored into more reusable classes as we learned more about Scala and HTML. Some sections of the code can still be extracted into scala.html classes to improve overall readability of the frontend pages. Additionally, the app/views directory currently contains scala.html files for both pages and components. For organizational purposes, it would be a good idea to separate the classes into different folders depending on whether the specific class renders a page or a custom component.

For some pages, the new page is being generated before the database executes the transaction. Some of the Mongo methods we use in our design such as update() return futures which we still do not fully understand. The futures are placeholders for results that have not executed to do operations in parallel. We think we might be able to use them to wait to refresh the page until after the database has executed. In the meantime, some pages (EventInformation) need to be refreshed right after making a change.

1.3 Individual experiments

1.3.1 Leeviana

Play has this very interesting framework concept of "Futures". Play uses a fully asynchronous model, which is pretty neat actually, but tricky to wrap our heads around. When working with database queries that return "Future" instances of objects, the expected results are often not where you expect them to be. Coming from C/Python/Java backgrounds we expect instructions to be executed in the past, and not the future.

This most often comes up while trying to create helper "get" methods from the database, like "getUser", where we would ideally want to pass in a BSONObjectID and get out an User object. In fact, the Scala language design expects for such calls to "intensive operations" to return a Future and for the calling methods to devolve into chaos soon afterwards (the last part may have been based more on experience than canonical knowledge...). Often the symptoms were the same, future objects floating around, strange methods to extract types from those future objects and lots and lots of values that just aren't there (well, in the present time at least).

I personally spent hours on one such getUser method and gained much knowledge and understanding about futures while trying to fit it into my programming model and constantly failing. I got very close and was able to accurately return a Future object of the correct type to the methods I wanted. However, the actual usage of those Future objects in the calling methods still eluded me, though I did learn about constructs like "flatMap"s, and will be an ongoing task to tackle throughout further evolutions.

As far as teamwork goes, I mostly worked on the layer between the server and the frontend, from MongoDB documents to their subsequent Play model mappings to behaviors in Play controllers. Our team showed great solidarity while pulling an all-nighter together.

1.3.2 Martha

Susan and I worked primarily on the front end, but also did some work with the controllers. The initial learning curve was using Scala and HTML to make forms and pages to connect to the back end later. Neither of us had very much experience with HTML and no experience with Scala, so we had to start learning them before we could make progress. Because we had defined a detailed programming in the large design, we were

able to start the front end before we had models and controllers to connect with. We also made time to work as a group, particularly when the front end and back end started coming together. Later in the project, I started working on the Events.scala controller because I realized that some of the functions I wanted to implement such as sorting the events list were much better off done in the controller than in a view page.

Working on the Events.scala controller was challenging because I had never worked with Scala or databases. The first function that I worked on was Events.index to sort the events before passing them to the HTML page. It seemed like it should be very simple, but I had difficulty with the syntax and quiery. We were basing our code off of examples online when we were still learning the language, and the code to quiery the database for events had an unneccesary operator in the find method. It worked without the sorting, but when I tried to use the sort method with the find method I got an error: "Can't canonicalize query." It took a long time to figure out what this error meant and to fix it. It ended up being one line of code to sort the events, but because of my inexperience what seemed like an easy task ended up being very time consuming. In the process of reading documentation and looking at example code I learned a lot about Scala and Reactivemongo which made future methods much easier to write.

1.3.3 Robert

During the course of the development of our web application, I was tasked with working on the backend of the server, setting up and managing the server-operations, and focusing on the authentication system. Leeviana and I initially focused our time on developing a roadmap for the backend database storage platform. We basically specified the entirety of our database and the relationships between different collections within it on a whiteboard and considered the potential drawbacks of various possible data structures. After the initial planning stages, most of my work was completed on my own rather than collaboratively, simply because I had some of the more specific tasks that didn't lend themselves to siginifcant collaboration.

I initially dedicate a significant amount of my time attempting to use the modiva.Silhouette Scala/Play Authentication library, both in it's latest, and stable iterations. Unfortunately, based on the growth of our own codebase over time, and on the increasingly difficult task of matching what existing samples I could find of Silhouette to our own work, I was forced to set an ultimatum on the time I would cease all development in this area and instead build a more rudimentary authentication without initial support for features like OAuth and external sign-on providers. Because of the newly enforced time constraint, I had to abandon upwards of 1000 lines of code in order to provide an effective solution in time for the submission deadline.

At that point in time, I was forced to rapidly re-assess my approach. I considered the fact that my authentication structure would necessarily have to fit into the already extensive structure of our applicatoin. Based on this fact, I tailored the authentication system to our existing datastructure, at the cost of some best practices, such as code duplication and a lack of centralized database access control structures. I opted for a highly modular, importable authentication module which tied into the existing request headers and utilized secure cookie-based session management by matching an AuthToken and UserID in the cookie to an existing list of authenticated user sessions stored in our local database. This approach, in addition to our earlier careful planning of the backend database, allowed me to bring a fully functional authentication structure into the project very late into its development.

In terms of data analysis, the most significant portion of my time went towards the analysis of the Silhouette authentication library. During the process of porting much of our application to this framework, I cross referenced 5 different projects which implemented different versions of this package, each in a slightly different way. I learned a significant amount and even after abandoning the approach, I implemented some portions of my authentication system in a fashion guided by similar principles to those which I had seen in this significantly more advnaced library.

The biggest experiment I conducted was the process of literally copying different package's relevant code segments into our own project's codebase and watching what areas of the code would fail to compile as I added larger and larger portions of code into our application. This process gave me a good feel for what areas of this authentication framework were largely independent, and which areas were more closely coupled. I was able to clearly identify modular and repetitive errors and see places in which the package had dangerously centralized dependencies, or seemed to spread to thin over many different files.

1.3.4 Susan

There are multiple ways to render components with Scala and HTML: directly putting the entirety of the code in a class, calling a Play helper method, and writing a custom, reusable class. Initially we wrote the front end code using the direct and helper method ways. However, directly writing the code makes it less modular and the helper methods provided minimal flexibility over layout, so we decided to refactor the forms into modular custom components. During this process, my unfamiliarity with Scala and HTML caused some working components to be rendered strangely or disconnected from the backend. Testing what was causing errors was a very iterative process, since only some form fields provided error feedback. The solution to this was to keep the original copy of the working form in a branch while iteratively replacing field code with components in a new branch. Changing the components iteratively allowed me to see exactly what field was wired incorrectly by comparing the result of a fixed input set into changed form to the original form in the database.

When the input type of the form does not match what the model is expecting, pressing the submit button results in redirecting back to the new events page. This meant that the div's type or format of input did not match with what the model was looking to bind to the value. This resulted in searching for appropriate Bootstrap field components that had input types/formatting that matched with what the model was expecting. Another common issue was that even though the input was the correct format and type, the value did not bind correctly to the model. Checking the name of the field and making sure that the Scala method was being called correctly often solved this issue.

A major component I worked on for this evolution was coding and wiring the event creation form. The fields of the form had to match the event object's expectations of input, which were defined by the Evolution 1 requirements. The original event model was very complex and Initially, we wanted to have the name of the user currently logged in to be displayed on the navigation bar, but time constraints and unfamiliarity with Scala caused us to push this back to the next evolution.

Martha and I were responsible for creating pages and components on the front end. By defining a detailed API with the entire group at the beginning, we were able to get started on building form mockups and familiarizing ourselves with Bootstrap without having to wait for the models to be built. Even though members of the group had busy schedules, we were able to find time to meet to discuss important details while working on our portions of the project individually. There were times where changes had to be made to the API, but this was communicated effectively throughout the group.