ECE 458: Engineering Software For Maintainability

Senior Design Course

Spring 2015

Evolution 3 Analysis

Brian Bolze, Jeff Day, Henrique Rusca, Wes Koorbusch

Contents

1 Previous Design Analysis

In this Evolution, we learned even more about the power of the Rails MVC architecture and its DRY design principles. The separation of logic between the Model, View, and Controller, combined with best-practices in code development, allows for a highly modular, flexible, and maintainable code base. For example, the more sensitive model data and code changed minimally during this evolution due to the separation of much of the feature specific logic into other model modules or controllers. By further extending the event *type* parameter, we were also able to add in functionality for time-slot events very easily. In terms of the Free Time feature, we simply created a new controller to handle all of this logic. While the logic for this was fairly intensive, it was kept away from all of the other parts of the project, and was able to be implemented without needing to change any other module.

From a front-end perspective, the team enjoyed even more benefits from the previous evolutions. By dividing up HTML code into reusable partials, we were able to generate new index and show views, as well as forms, very quickly and easily. Also, because of this DRY approach, when we wanted to make a change to any or all of these views, we could do this quickly by editing just one or two files. Due to the relative simplicity of the required front-end changes, the team spent a lot of time adding more dynamic and advanced features like AJAX and jQuery to provide for a more responsive user interface. The calendar home page now renders modals with lots of useful data that a user may like to access often, and even allows them to create new events without needing to navigate away from the home page.

2 Current Design Evaluation

By this Evolution, our team was clearly much more confident with the inner workings of web development and the specifics of Rails, and it showed in the increased pace of our progress. On Evolution 3, we were able to reuse several models to tie up the new logic, keeping creation of new code to a minimum. The team also used AJAX and jQuery to add more dynamic behavior to views, all the while maintaining the cohesion of our Rails backend.

2.1 Views

The front-end has improved significantly since Evolution 2. We were able to group several features into forms in the home screen, but time constraints did not allow us to apply the technique to all pages. It is possible we migrate everything to a more user-friendly interface by the end of Evolution 4. For the two new functions, time slots and free time, partials are still a crucial feature. We still reuse similar HTML code and modularize it by leveraging partials.

2.2 Controllers

The `TimeSlot` controller is relatively short since the validation, creation, and deletion logic is moved to the models as always. The controller basically handles creation, deletion and slot signup. It does not handle selection of free slot times, however. That logic is moved to the models as well.

The `FreeTime` controller, however, handles heavy logic and computation. Since a FreeTime model is not necessary to know when not to have an event, this controller behaves differently than the others. There is no specific model tied to it. Rather, it needs to manage users and their subscription and filter them accordingly. One big advantage of the finding-free-time-logic is that every tangible event (i.e. a slot event block is only for time boundary display but a time slot actually represents a busy time frame) is tied to a subscription. So, after the user selects all the people he or she wants to know the free time of, it is just a matter of retrieving all the visible subscriptions of all these users. It does not matter whether the event is a request, a time slot, or a to-do allocated, they all have subscriptions tied to it. If we would compare user by user, it would become computationally impossible to map out each conflict. Consider if we chose k users each with an average of n events. The naive approach to compare each one would yield an $O(k!n^2)$ algorithm because every event of every user would be compared to each other. Given that this approach is not viable, we needed to come with more efficient algorithms and heuristics to provide a fast server response. To actually filter the right times, the logic used is the following.

Parameter Validation It is important to not let the user input any inconsistent parameter since the free time logic is extremely sensitive. Any date off its place could terminate a whole loop iteration due to aggregated heuristics. The parameter validation includes the existence of at least one selected user, one selected weekday, correct start and end periods, and existence of an appropriate until time if recurrence is requested. Note that even though the until date field presents time input, it is irrelevant since the time period is already defined by previous fields.

Date parsing We need to know all days that the user wants to look for. If the user does not specify any recurrence the next days of week checked will be used for the search. However, if the user specifies a recurrence period, all the very next days of the week are taken as a starting point and each one recurs for the amount of time given until the defined end date. However, suppose someone chooses M T W and an every other day recurrence. Obviously Monday will land on every day that Wednesday lands, so only unique dates are taken into consideration.

Merge user events The first step is to merge all events of all users into a single array. For purposes of finding the free time available, it is not necessary to know who conflicted, only what times did not conflict. The conflict logic is reserved for another action call.

Filter event array After merging all events, we will need to eventually calculate the free time between them. To avoid iterating more times than normal, a filter is performed to extract only events that land on the given time frame. In case there is no event that meets the criteria, the whole time period is returned and terminating the free time parsing routine.

Iteration The idea is to build an array with all free times with the format `[[start1, end1], [start2, end2],...]`. This way we construct an array with all possible free time ranges. This algorithm can be accomplished in O(n) worst case time if we manipulate data decently. Firstly, it is important to notice that the resulting events are sorted by starting time, then the event array is iterated following the rules below along with a `current_time` pointer to keep track of our position in the time frame:

- If `current_time` is contained in the time frame of the iterated event, we move current time to the end time of the event

- In case there is a large event iterated that encompasses a whole other event, the next event will have an end time smaller than current time. So we can skip this iteration.
- If we find a `current_time` that is less than the start of the next event, we can add `[current_time, event.end_time]` to the free time array.
- As a precaution, if `current_time` passes the given end period, we break the iteration.
- Finally, if the event iteration finished with an event inside the given period, we still need to add one last time range to free times consisting of the final state of the current time and the end period provided.

Small duration elimination After mapping every free time interval available, we still need to take into consideration the duration of the event the user wants to create. Therefore, we run the resulting array into a routine to eliminate such short time ranges from the array of free times.

Parsing into start times After retrieving the free intervals for a given period, these intervals are parsed into all possible start times with time steps of 15 minutes until 15 minutes before the end of the free time range. This is necessary for the html select input field. Then, the start date, end date, and free time are added to a hash used by the view.

Repeat Repeat the free time parsing for each day

From the options generated of free time, the user can choose one day and a start time to automatically send a request to all other users chosen.

For the conflict logic, the idea is similar but simpler. For conflicts, the user can only specify one date with optional recurring search. For each calculated day that the user wants to check for conflicts, all the user events are merged and sorted for the given period as well. However, this time on the iteration, the event period is compared for an overlap with the given period. If a match is found, it is added to the array of conflicted events. To expedite the iteration, events that have their end time before the conflict period start time trigger a skip on the current iteration, and the loop breaks upon finding the first event that has a start time greater than the end time of the period. The parsed information is displayed on the view with each user and event. If there are no conflicts for a given time period, an appropriate message is displayed. It is important to notice that, to handle the confusing realm of time zones, several controller global methods were created to manage them. From rounding seconds, to converting time zones, to converting to seconds to sum times, our project now has a steady base for handling this issue.

2.3 Model and Database

The only novelty on the project here are time slots. The free time requirement did not bring any new models to the table. However, two new types of events arose: time slot blocks and time slot events. Time slot blocks are more like informational event to delimit a time frame used for creating time slots. Users do not specifically have subscriptions assigned to these types of events. The repetition scheme model was reused to bundle together time slot blocks with the same purpose. The time slot model works only as a buffer to modularize and manage the time slot logic. It takes into account all possible validations upon creating to prevent the database from storing corrupt data. The validations are:

- There is not a previously created time slot on this event block
- Alignment of time slot start time with a multiple of the minimum duration allowed and certification that the time slot period does not fall outside the slot block time frame
- Time slot duration is a multiple of 5 minutes
- Duration is not greater than maximum duration
- Time slot does not overlap with an existing one

Note that, since the time slot durations are flexible, we decided to appeal to lazy loading and only create a time slot when a user requests it. The event block keeps track of the available times to schedule an appointment and presents them to the view. This way we can allocate slots properly without any conflict. After creation, a TimeSlot creates a time slot event and a subscription to both the requesting user and the owner of the event block. If we were only left with the TimeSlot model creation, there would be no subscriptions tied to them and free time parsing would be significantly more complicated.

3 Design Process Notes

This section of the document describes experiences regarding the design and testing process for each member of the team. These experiences include analyzing data, design processes, and team management.

3.1 Designed and Conducted Experiment

Brian's Contribution

An experiment I designed for Evolution 3 was in debugging repeated events. I decided to take a bottom up approach, working from the data, up to the model, controller, view, and then looking at the related assets. I began by writing a test to ensure the correct data was being stored and the validations were being asserted, and all checked out. I then opened the Rails console and created a new repeating event in the database, and it all worked. I then tested the Event modal methods to check the data fetching, and it all worked. However, one I reached the controller, I found the error. I traced through the data flow through the controller and found old unused logic in one of the methods that was causing the repeated events to return nil. After fixing this, I was able to get the events to regenerate on the Console. I then wrote a new jbuilder file to convert this data to JSON to be used by the graphical calendar, and everything was fixed. I was able to repeat this same process again and again across different parts of the project for debugging, and noticed a significant productivity increase.

Jeff's Contribution

One experiment that I needed to design was to confirm the functionality of choosing a single slot block to sign up for when a time slot event has multiple slot blocks with different start and end times. With my initial implementation of signing up for a time slot, I simply listed all the events

associated with the slot event and included a simple submit button for the events. However, it would not function properly when having to choose between multiple slot blocks (even if you only specified signing up for one block, it would still sign you up for the defaults on the other block). In order to solve this issue, I had to create a separate form for each signup block with its own unique submit button. Although this was a simple solution, I was not initially convinced that such a simple solution actually solved the problem. In order to test this, I took the outputs on the rails server from both of my implementations, and examined the SQL statements that resulted in creation of signup slots. In my first incorrect implementation, the rails server would create as many slots with the user signed up as there were sign up blocks for the slot event. In my second implementation, it would only sign the user up for the timeslot whose associated submit button was clicked. I was only able to confirm this behavior as a result of experimenting with the rails server and its outputs.

Wes's Contribution

In between evolutions 2 and 3, our app seemed to lose the ability to send email alerts. For a while, I could not figure out what was happening because our scheduler was properly logging times and placing reminders in our database, but then not waking up when that time passed. It turns out that the East Coast of the U.S. had daylight savings time at the beginning of Spring, and that meant that UTC and EST were four instead of five hours apart. Although we never hardcoded a time shift into our app, our email alerts were being sent an hour later than they were supposed to. I designed an experiment where I changed the time on the clock on my computer, and then set various reminders for myself to determine whether the scheduler would wake up at the designated time in EST or one hour later. After waiting an hour, the scheduler woke up and sent an email like I had anticipated, so I had to recalibrate our email timing accordingly.

Henrique's Contribution

Over Evolution 3, many of the same practices were preserved. Especially to validate the time slot logic, rails console was crucial. I have conducted several model experiments and tested passing several different start times and durations to ensure a time slot would not be created unless it was coping completely with the document specifications. Also, I was able to test time slot event creation by a time slot through the console and ensure that the respective subscriptions were being generated. Also the console was extremely useful to figure out time zone differences and methods, I was able to test and find out several points on the project that failed on time zone conversion and design methods to handle them appropriately, especially events coming from the server that would always be in UTC format. As for experimenting with free time, there was not a specific IDE to handle it except looking into print statements on the console. Building a working logic was cumbersome and took several server requests and I/Os to get it done.

3.2 Analyzed and Interpreted Data

Brian's Contribution

On of the issues I ran into again and again was with the Asset pipeline when moving to production. As the maintainer of the Heroku production server, I was responsible for pushing the latest working code to the production server, and ensuring everything ran the same way as in development mode. This was surprisingly difficult. Due to the way that the Asset pipeline concatenates Javascript and CSS files when moving to production, it is easy to break a lot of Javascript if not

all of your files work. Also, debugging Javascript can be difficult because it either works, or it doesn't, and it does not give you any hints when it doesn't. Therefore, I decided to use grep from the command line to parse through the compiled/concatenated Javascript and CSS files on the production server end, and compare it to what we had in development. By analyzing and identifying the differences across the two, I was able to narrow down the problems very quickly. Many times, it came down to errors at the end of one file, that when taken in pieces would allow most parts to work in development, but when concatenated, the code after the error cannot run.

Jeff's Contribution

One key goal of the our work in this evolution was making our calendar that displays various types of events more reliable and user friendly. Brian spent a significant amount of time debugging the Ruby and the Javascript associated with our calendar that displays events. However, once he pushed his changes to it to our repository, I kept getting a confusing error causing our application to crash. The error was a result our program attempting to draw an event on our calendar whose creator_id field was nil. At first this seemed very confusing, considering that I believed every event should have a creator_id field. As usual with performing a lot of debugging, I pulled up the rails console and examined all of the events created in my local database. I then realized that any time slot event created by a user had no specified creator_id field, because in our design, we do not associate the event with a creator, but instead has a simple association belonging to the user who signs up for the time slot and the event associated with that time slot. Because Brian had not been working on developing slot events, he had not created any on his local machine and did not get the error, even when attemtping to display all of his events on the modal that displays them. After interpreting this data displayed by the rails console, I was able to perform the necessary edits to Brian's code that made sure to associate the time slot event creator with the event being drawn on our calendar.

Wes's Contribution

There were not many cases during this evolution that I think I analyzed and interpreted data. However, in testing whether time slots were working properly, I tested to see that slots were created, intervals were properly spliced, and that the requestee would end up occupying the proper number of slots when they requested a certain amount of time for a meeting. After the requestee filled out the form for time slots, other requesters properly see the correct number of time slots as being occupied.

Henrique's Contribution

In terms of handling time slots, there was not much data to analyze except for ensuring the database cohesion. For that, I made use of validation and after create calls to help maintain the state of the server. Also, it is important to note the addition of cascading dependencies to on the time slot controller. Suppose a repetition scheme for time slots gets deleted. It would leave idle on the server its respective event slot blocks, time slots, event slots, and subscriptions. However with cascading deletion, all these models will be cleaned from the database. Data was hard to interpret on the FreeTimeController. It was basically a lot of date math to verify if results were consistent. Some times I would get dates at years of 2064 because seconds were not explicit on date math. Therefore the system would think it was supposed to add days instead. Also, aligning times correctly led me to establish a standard among the dates I was working with. Rails has a DateTime, Date, and Time object in which I believe behave slightly different in terms of manipulation even though I can provide a date to the Time model. Therefore I had to standardize all date instances

into the current time zone and by zeroing the seconds so I could make reliable Boolean logic with them and be able to analyze and parse the dates correctly.

3.3 Designed System Component to Meet Desired Needs

Brian's Contribution

One of the more prominent components that I helped design was the controller logic for finding free times. While much of this is discussed in the design section, I will expand on my own specific contribution. Henrique and I began by whiteboarding a test scenario of checking for free time between three users (and/or groups). We drew out three timelines for each user showing their events as conflicts and the rest as free time. We realized that it would be an $O(k!n^2)$ operation to compare all events to each other in series, so we decided to collapse the three timelines into one sorted array, so that we could analyze this in one pass. The rest of the logic was fairly trivial. We then went on and peer programmed most of the controller code, before I split off and implemented the associated views and forms required.

I also built all of the modals on the home page to display all of a user's events, requests, and To-Dos. I also created a cleaner modal to submit a form for a new event. This component proved to be extremely useful and greatly improved the User Interface.

Jeff's Contribution

One key component that we wanted for our calendar is the ability to easily make multiple event blocks for our time slot events (have a time slot event that has sign up slots in multiple different time blocks.) One of my key jobs was designing the user interface to do this. The biggest challenge was incorporating the ability to display a dynamic number of event start time and end times so that a user could specify multiple sign up blocks. Because much of the input data associated with time slots are very similar to a normal event (Title, Description), we wanted to reuse as many of our rails partials as possible. In order to generate the form for time slot events, I created a form that contained fields for title, description, minimum and maximum slot durations, and then a dynamic number of partials that asked for event start time and end time. In order to make the number of input start times and end times dynamic, I designed a system where the rails view uses Javascript to initially ask how many unique time slots the user wants for the time slot event. After obtaining this number, the javascript renders the specified number of partials that contain input fields for the start time and end time. The controller makes sure to create an event for each number of slot blocks specified by the user.

Wes's Contribution

The modals on our front page that read "All Events", "To Do List", and "My Requests" were created to simplify the process of debugging our features from the previous evolution, but I think they ended up becoming valuable tools for streamlining the user experience on the homepage. The modals originally displayed the relevant fields with an empty table under them, but I also changed that so that they would display a message stating that the user doesn't have any of that type of event.

Henrique's Contribution

Initially, I have designed the TimeSlot requirement so that the necessary information should be fetched from that model. However, when getting into coding the free time I realized how hard would be to handle TimeSlot and Event times separately. But simply another event type and making an event behave as a time slot would yield poor design. An event would have to hold a lot of validation logic that would only be executed if it were actually a time slot. Therefore, a better solution was to make TimeSlot a buffer class to manage a time slot event. I really liked how this turned out. It was interesting that, even though I had not accounted for certain functionality integration, it was possible to add more logic to the design to enhance it without modifying it. In terms of free time logic, there is not much design to be talked about. I knew it would be a view to display all the controller parsing and validation. However, the design here would be the algorithm used to find all possible free times. I will not get in details as it is already explained in the design section. It is important to note that the algorithm runs in O(dn) time in which n is the total number of events that fall in the given time frame and is the number of days that need to be parsed through (given recursion).

3.4 Deal with Realistic Constraints

Brian's Contribution

During the last Evolution, as a team we talked a lot about wanting to go back, refactor old code, and clean up bugs from the previous evolution. However, due to time constraints this dropped to a lower priority and it ended up not happening. Realizing this, I tackled this first, before working on anything else. Although it took some time and investment, I realized a huge difference in productivity later in the Evolution. Also, when working on the new functionality (like the new reveal modals on the home page) I worked slowly and carefully, to make sure that all of the AJAX worked as expected. When it came down to the deadline, I realized that I was not going to be able to implement this on all of the pages before Wednesday. However, I still stand by my slower, more meticulous development process because I am now able to reuse a lot of this code using the partials that I generated.

Jeff's Contribution

Before attempting to implement adding multiple time slot blocks in the current method through asking the user how many different slot bocks they need, we wanted to implement a way to dynamically add them one by one using an HTML button and complex javascript. I initially wrote the javascript code to do this in a separate file from the view in order to have a button that would call this javascript and then append a new partial on to the view page to allow for the user to input another start time and date time for the block. Not only was this process very complicated, but even when I believe I implemented it correctly, I still came across some errors with the javascript that did not allow for it to add the partial to the view page correctly every time. This issue was primarily involved with the passing of variables from the rails server to the client in order to utilize javascript in adding the form to fill out in the browser. As a result of the constraint that it needed to work every time and I could not find an efficient solution to the javascript problem, I decided to create a new method for specifying the blocks. This ended up working out well, and better than the previous implementation specifically for error checking the user input.

Wes's Contribution

I initially had the goal of changing each home page (Groups, Subscriptions, Requests, To-Dos, Time Slots, and Search Free Time) so that creating and editing would happen in a modal popup instead of a new page rendered by Rails. Unfortunately, with the way that our project is structured, the partials and forms used to create new objects in the database are not very easily updated with values from Javascript, especially because it assumes that you have created an entry to be placed in the database before rendering. I received a lot of errors stating that I was calling a nil object when trying to create the object in the first place, and so decided that I would put the desire to have more forms in modals on the backburner to help with other features.

Henrique's Contribution

There were several problems that arose along evolution 3. When I first designed the Time Slot logic, I ended it up on this model. In other words, to find a slots start and duration I would only fetch from this model. However, when I starting developing the design for free time, I realized that I would have to fetch and parse all event subscriptions separated from time slots. This would not only make the design less robust but also undermine the parsing performance of free times. Therefore, it was possible to come up with a solution that would not harm the current design. By letting a TimeSlot hold and manage a time slot event, the project was able to maintain a concise design and I was able to avoid more complicated logic on the free time controller. Therefore, before taking free time I had to go back and make additions to the design. Nevertheless, it worked as expected and now time slots can be detected by the free time logic. When writing code for the free time controller, most of the time I spent debugging time zones. Some important points I found regarding time conversion were:

Time zones from server Every object that is fetched from the server and has a DateTime field present the time in UTC format even though it was saved in Eastern Time. I had to create a global `to_eatern_time(time)` method to solve this problem.

Date picker time The date picker was a complicated one. It parses and sends the time the user chose but in UTC format. Therefore, I would have to convert it back to Eastern Time and subtract the UTC offset from it before retrieving any date from this jQuery component.

Seconds in date calculation To perform date math I needed to take into account that one date could have been created at 14:50:07 and another at 14:50:35. If I would try to compare these dates for equality, it would yield `false` even though I would like these dates to be matched. Therefore I have added another global method to zero the seconds of a date for better date comparison logic

3.5 Contributed to Team Work and Interacted with Team Members

Brian's Contribution

Throughout the course of this project, I have been acting as the curator of the Trello page. This proved to be highly useful to keep track of the Evolution requirements, but also to track bugs. During this evolution, I finally took the time to go back, record all of our applications bugs, and then assign them to new members and check them off on Trello. Beyond this, I spent a lot of time with Ricky whiteboarding and designing solutions for the new requirements. We spent many hours brainstorming how to write the logic for finding free times. We began by coming up with a brute

force search and then finally iterated to an algorithm that runs in O(n). When had fleshed out all of the details of the design, we spent the rest of the time peer-programming. Going off of good results from Evolutions 1 and 2, we decided this was the most efficient approach, especially when dealing with the sensitive database and model code.

Jeff's Contribution

As with every evolution, I did my best to communicate with my team members as efficiently as possible. This meant that I was always sending updates/issues to our facebook group so that the entirety of the team knew the status of the work I was doing. I also did my best to attend all full group meetings (often at Henrique or Brian's apartment), as we have found that we work most efficiently when the entire team is working together. This is primarily because we have the ability to easily ask questions and get second opinions on design decisions. For example, I would often get clarifications from Henrique on how a certain aspect of the model was meant to work that would save me loads of debugging time in implementing an efficient view.

Wes's Contribution

I have tried to be very communicative with the rest of the team in order to make sure that we don't have multiple people doing the same thing at the same time. Part of that effort involves using Trello, which Brian set up and maintains, and all of our team adds to regularly. Ricky and Brian worked a lot on creating the models and controllers and their associated logic for the new features. After created, there were a lot of combinations of logic that needed to be tested for bugs, because there are a lot of situations that can arise in terms of conflicts and visible/invisible events. I helped test these features so that they do not have any bugs when used properly, although we do need to do more work to ensure that users can't break the app so easily, since small errors can cause big issues.

Henrique's Contribution

I have focused on crossing off the functionalities required by the design document. Also, I have defined the design of the database up to evolution 3. My contributions over this sprint were:

Sign up block Allowing a user to create sign up blocks. The logic and database design to parse multiple blocks were already done, however I did not implement a dynamic view to do so

Slot duration Adding logic to handle minimum and maximum slot durations

Slot validation Not allowing slots with mismatching parameters to be saved to the database, such as start time alignment with the event block

Slot view I have put up the basic CRUD for managing time slots to have the functionality ready and working. I have left to the front-end team the tasks of making the views more user friendly and dynamic.

Slot management User can change and delete a signed up slot

Free Time parameters Basic CRUD with necessary validations to initiate a free time search. Parameters include choosing any combination of users and groups, time range, time slot size, weekday selection and recurrence.

Free Time parsing I have written all the logic for finding free times and conflict times at the `FreeTimeController`

Free Time display and request creation View for displaying possible free times in which a user can choose from. Also, creation of a new request to all involved users from the selected free time.

Conflict search View for analyzing which periods present a conflict once the user has inputted the right parameters

Time manipulation Added global time manipulation methods to make it easier to handle time math and time zone conversion

Design documentation Wrote the design section of Evolution 3 Analysis document

I have also made myself available for my team and offered my apartment for coding sessions