ECE 458: Engineering Software For Maintainability

Senior Design Course

Spring 2015

Evolution 4 Analysis

Brian Bolze, Jeff Day, Henrique Rusca, Wes Koorbusch

Contents

1 Previous Design Analysis

During this evolution, we did not need to make too many edits to our previous design in order to meet Evolution 4 requirements. Since this evolution involved adding small features to already existing models, our previous modular design allowed us to add most of the new features relatively easily. Adding the ability to email a schedule to the user was relatively easy because all we needed to do was create a new form allowing the user to input a specific date field and then access the events themselves through the event controller. Also, our previous implementation of an easy to use mailing service within our Rails application allowed us to reuse that feature for sending text based schedules in HTML and PDF format. Expiration time and priority escalation involved addition of expiration and escalation fields to our model, implementation of useful methods in the model and controller to execute expiration and escalation, and a slight redesign of on the to do events view. However, none of this involved any rewriting of old code or major redesign of how the calendar system functions; it was primarily addition of new methods and features to already existing code. This simplicity of implementation for new features regarding PUD events stemmed from our ability design as modular as possible throughout the previous evolutions and our ability to use the Rails framework to its full potential.

2 Current Design Evaluation

2.1 Views

For views, Rails provides the use of partials, helpers, assets, and layouts. These features all come together to allow the developer to create a powerful interface. Rails motto of not repeating yourself allows for efficient interface implementation with a great deal of reusable code.

Layouts can be considered the beginning and end of the html file rendered on screen. With them we are able to render the navigation bar along with any error and notification that comes from an action on the application, since this view code wraps all the rest. On the body of the layout structure, the file yields to the corresponding html file that should be rendered on thee session. Within an html view, we can render several partials, which are basically the opposite of the layout. Instead of wrapping the html file, it is embedded in it. Therefore, we are able to render any piece of html.erb (embedded ruby) code within the html file. This proves to be extremely useful for generating forms. For example, the forms for generating and editing a to-do (how we called PUD event) should have the same fields with the exception of header information and the method called on the controller upon the form submission.

With this capability, we were able to create a highly modular and reusable front-end throughout each evolution, which made Evolution 4 much easier to implement. For example, we were able to implement the priority escalation feature with just a few lines of HTML and embedded ruby on the PUD form. We were also able to create a new interface on the home page. Before, we were using normal modals to render some more HTML and forms manually. After a while, however, the home page became slower and slower and we found that our code for the index view was becoming way too long and messy. We then figured out a way to render partials inside of the modals using AJAX. This change had two significant benefits for us. First, the page load time for the events index page (the home page) improved significantly, which is important because this action happens very often.

Second, this cleaned up our code by removing a lot of inline data manipulation and also reducing the repeated code by extracting the HTML snippets into Partials.

In order to implement mass creation of events, we built just one new view. This view contains all the instructions to create any kind of event, a text field for input, and the list of user and group ids for reference. The possible events that can be created are regular, allocated for to-do, to-dos, requests, time slots, preference-based time slots. Given all of the dynamic behavior required for error handling for mass creation, we thought we may need a more powerful tool than standard Javascript. One of the members on our team spoke with some other teams and did a lot of research on AngularJS. After researching, he found that it would be extremely useful and easy to capture the new functionality; however, given the time constraints we decided it would be more efficient to finish the rest of the requirements using standard HTML, ERB, and Javascript.

Assets are another important cog for a Rails application. All the js, css, and images are included within the assets folder. Rails make use of asset pipelining to precompile all the assets into a single optimized file so the browser can cache it and not make any further requests. The js in the application is used for popup forms and calendar presentation, while the css is, obviously, used for styling. One key issue where defining assets came into play was in converting HTML pages that listed the events for a specified time period into a PDF. The program we use to perform this conversion needs to have any CSS or Javascript that it intends to convert predefined in the asset pipeline. This allowed the program to not have to compile the information on call; rather it could easily access it in the asset pipeline and write it to a pdf.

2.2 Controllers

Rails emphasizes how controllers should be kept with a minimum amount of work. Controllers are basically responsible for extracting the variables that the view will use and redirect the browser based on user action and input condition. Also, bear in mind that complex queries for the extraction of the views variables are all moved to the models and placed into methods. The controllers of the application perform their function as demanded however Rails does not fail to warn that the implementation practices are not perfect. Sometimes a controller has to handle more than one model at once. For instance, to created a requested event, the controller should create the event itself, a request_map, and the requests to be sent to the corresponding users. Nevertheless, we try to keep that logic to a minimum by letting the request_map take the responsibility of creating its own requests by just passing the ids of the users.

Another important point to notice is that Rails recommends that max amount of shared variables between the controller and the view is constrained to two. However, in some cases this is simply not possible. For instance, the events controller segregates arrays of events by visibility for presentation. It is better to extract and separate each type of event into a different array rather than letting the view make difficult queries and figure where to place the next event on the iteration.

The to-dos controller did not change its logic significantly. It only needs to account for extra parameters and some validations on priority escalation end expiration. But it does not need to handle the logic and handling of these new features. They are all performed on the model layer.

For preference based signups, upon an event creation the controller needs to account for a couple of flags, which are if the event is preference based and whether to-dos should be sent for this slot

event. Depending on the status of the repetition scheme that holds a set of events for a slot session, the controller will present different options for the user. In case the slot session is a regular one, the user will be able to simply signup to it. However, if the slot is preference based, can submit as many preferences configuration as he/she wants. Later the owner of the event may assign users to the slots with a list of all the given preferences. Keep in mind that this action is irreversible so once the slots are assigned, they cannot be edited. Most of the logic is also moved to the model to facilitate fetching and manipulating information on the project.

For mass creation of events, a new controller had to be designed. The controller simply takes the text input of the view and parses it completely. Several validation steps are performed before the events are actually created. Such steps are listed below. It is important to notice that all possible errors are gathered at each step but that the following step is never executed once the previous one was not parsed successfully.

In order to extract the events to be emailed to the user, a new method needed to be added to the events controller that filters out the events not in the specified time period. The controller then returns a list of the events in the specified time range, and sends them to the view to print them in an HTML table to be embedded in an email or attached as a PDF.

text parsing This step tries to separate the whole input into a hash of adequate strings. It parses each event by the delimiter
`n` and each parameter by `;`. It also validates if the type of event key is present on the input

key validation This step verifies if all argument keys are according to the given definitions. In case there is an invalid key or a mandatory one is missing, this step will log an error

value validation This step parses through the given values and converts all the string values into the appropriate data structure. For instance, string dates will be parsed as `DateTime` objects. It is responsible for validating and converting durations, dates, integers, id lists, and list of slot blocks with date ranges.

event creation Once the whole input has been validated and parsed, the events are created based on the type provided. Note that in this phase there can still happen errors, such as the user providing a wrong list of ids or specifying end times that are before start times

2.3 Models

Rails provides a solid framework for models to know where they stand. Rails make use of active records pattern to make a model communicate directly with its database counterpart. Also, ActiveRecords on rails offer validation calls to ensure the data inputted is exactly according to the developers conditions. If it is an integer, we can define a minimum and a maximum, if it is a string, we can constrain the length and so on. Therefore, we make use of those validations to ensure there are no surprises while retrieving data.

ActiveRecords allow to define a relation between themselves and other ActiveRecords on the database as long as the have the right column name for it. But Rails not only provides a relationship for one-to-many, one-to-one, and many-to-many, but also provides a through relation. It allows an ActiveRecord to reference another one through an intermediary table on the database. This design eases the use of complex queries and simplifies the code greatly upon implementation. For instance,

Events can refer directly to their subscribers (Users) without us writing any code to fetch data from the subscriptions table. The joins are made automatically. Another example is the groups table, which can access its members (also Users) directly without retrieving membership models.

A very important part of active records are their callbacks. It is possible do define a method to be called before or after any validation, creation, update, or deletion. These callbacks are fundamental for the cohesion of the database and are a great asset towards design. Methods and configuration can be called automatically over active records without the intervention of controllers. For instance, a To-Do verifies the next time it should be scheduled before creation in case it is recurrent, and reschedules itself if needed before an update (i.e. marked as done). For To-Do allocated events, they verify that their duration is a minimum of 15 minutes before validation and fetches the next to-do based on priority after creation. Finally, a reminder takes care of scheduling its notification task upon creation and unscheduling the task upon destruction to avoid emails of non-existent events.

Last important point of ActiveRecords are dependencies. We are able to specify the dependencies of relation between ActiveRecords and how the deletion of one should affect its related tables. Therefore, we are able to cascade deletions and get rid of potential memory leaks in the database without much trouble. For instance, if a user is deleted, it consequently deletes its created events, that deletes its related request map and requests, to-do items, and subscriptions. To-dos and subscriptions in turn destroy their respective reminders, which get unscheduled before destruction. Therefore, one could argue that the database has a considerably cohesive structure.

For To-Do escalation and expiration, five new columns had to be added:

expiration date containing the time for the to do to expire. Once the object is created, it checks for the existence of this property and sets itself to expire on the correct date

escalation_prior enum that represents how soon before the deadline escalation should start. It is scheduled right after the record creation in case the property is present

escalation_recurrence enum that represents the periodicity of escalation, it will keep escalating until the event is done or the priority has raised to 1

escalation step amount by which a to do should be escalated

job_id the id of the scheduled job for escalation. Useful for cancelling it if needed or if the record is destroyed

The TimeSlot model did not change significantly. A preference column was added and some of the validations were bypassed like allowing only one signup at once in case the slot pertains to a preference based event.

RepetitionScheme has had a considerable amount of code added. It is responsible for grouping the slot blocks and also for handling the preference logic. Two new relations were added to the RepetitionScheme. The first is a list of allowed users that can sign up to the event and the second is a list of to-dos that it could create in case of a preference based event. The logic to manipulate preferences is described below.

generate_to_dos_with_position it creates a to do for each allowed user with the chosen priority of the owner of the event

get_all_user_preferences returns a hash of all allowed users mapped to a list of their time slot suggestions sorted by preference

suggest_slot_assignment algorithm that tries to suggest an adequate initial setup for time slots. It starts with the hash of all user preferences, eliminates users who have not given any preferences, sorts the keys by amount of preferences given in ascending order. Then, it iterates through these keys and retrieves the first time slot on the list that does not overlap with a previously suggested one. Given that the hash is sorted by amount of preferences, users who did not provide many preferences will have suggestions before users who specified several ones. This way, there is a greater chance that everyone will be assigned a slot

user_allowed? returns a Boolean value determining if the user can see these slot events

resolve_preferences_for_slots finalizes a slot assignment. If any, it marks all the generated to dos as done. It deletes the other suggested slots and marks its own status as resolved

2.4 Database

The design of the database changed drastically since evolution 1. It got much more complex and with the help of active records callbacks and dependencies, it becomes self-maintainable in many points. Each ActiveRecord has its corresponding table and relations. Those are:

- User: holds user general information along with the current session. A user has many created events and subscriptions to other user created events. It holds many memberships to groups defined by other users and many self-created to-dos. It also has many visibilities applied to him specifically regarding someone elses event and several requests made by another person to him. A user model is responsible for retrieving any type of event with a certain characteristic related to him. Also, it creates and removes its own subscription.
- Event: Holds the general information of an event. It belongs to a creator, which is a User model. It holds many subscriptions made by other users to it and has many possible visibilities applied to it. It contains one to-do in case it is a to-do event or one request_map in case it is a requested event. A column event_type helps discern what type of event we are dealing with so we can present and consider it properly. We thought this design would be better rather than creating several event tables for each type and querying different tables to get essentially the same object. We let the event set itself up depending on its type. For instance, when a to-do event is created, it takes care of setting its title and description automatically based on the fetched to-do. Right now, the three possible types are regular, to_do, and request.
- Group: represents a group with a name to which one can assign several members for organization. A group is linked to many memberships as it is a many-to-many relationship with Users. A group also has many visibilities as a visibility rule of an event can be assigned to whole groups.
- Membership: table that maps the many-to-many relationship between users and grops.
- Subscription: table that maps the many-to-many relationship between users and events. If the user has a subscription, the event will be displayed for him if the visibility is not private. A subscription also holds a reminder, so each user can set his own reminder to a specific event.

- Visibility: represents one visibility rule applied to one event by its creator. An event can have several visibilities and can have them ordered in whichever way the user wants. We make use of the act_as_list gem that orders the visibility by position, scopes unique values by its corresponding event, and rearranges all positions once one of the visibilities position is changed, allowing for always having a set of visibilities in a clearly defined order. A visibility can either belong to a user or group. We decided to not break a group in its members and assign several visibilities because if a user would try to move the position of a whole group visibility, it would be hard to keep track of the new position of each member that pertains to that group. We make use of enums mapped directly to a status column to discern the different visibilities.

- RequestMap: is basically a modularization level to manage the requests for an event. This way, an event that is not a request does not need to carry unnecessary behavior or information besides a request_map_id. It holds the relation to its many requests and is able to retrieve requests by status and create new ones for a given set of users.

- Request: model that most importantly holds a state for the views and controllers to handle the behavior of a user towards an event. A request holds its corresponding request_map and user, and contains a status column to discern whether it is pending, confirmed, declined, removed, or with a requested modification.

- ToDo: is an asynchronous model that can be optionally embedded in an event. Much like a visibility, it also makes use of the acts_as_list gem to order the to-dos of a user by priority in a cohesive manner. If a to-do event is allocated, it automatically fetches the next to-do on the priority list that is not done and is not assigned to another event already. It holds its own creator id and has a column to define its optional recurrence. After the event gets done, it verifies if the recurrence time has elapsed since creation. If so, it immediately creates a new to-do with the same parameters. If not, it schedules the creation of another to-do to the proper time. It contains an optional reminder as well with recurrent characteristics.

- Reminder: Handles email notifications for either a subscription or a to-do. After creation, it schedules the right reminder based on the presence of a foreign key to either object. It holds a start_time and a recurrence value for the initial and subsequent notifications. For scheduling, we use the rufus_scheduler ruby gem.

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

In evaluating our past implementations, I realized that one of the most common bugs we had were with handling data validation. There are two places you can handle bad form data, either in

the view or the model. This can make it difficult to keep track of which form has good validation or not. So in order to catch some of these bugs, I decided to write a collections of tests. One of the benefits of RoR is the integrated test environment. It allows a developer to write tests for models, controllers, and other components at varying granularities, as well as seed the database with sample data using fixtures. After writing a suite of tests for the model and running them against a few fixtures, I was able to quickly identify and resolve some errors when we put in bad data, particularly in the Time Slot form.

Jeff's Contribution

One aspect of this evolution that I had to do a significant amount of experimentation with was the pdf generator for the schedule email. When developing this feature for our application, I used a Rails Gem called WickedPDF that required the creation of templates for pages that were wanted to be made into a PDF. However, there were a wide variety of different tags associated with these templates to link WickedPDF to various javascript and CSS files. I first experimented with these tags using a very simple implementation of FullCalendar. After a variety of experimentation and further research into the topic, I realized that the Gem that I was using was unable to support the complex javascript associated with FullCalendar, as a result of asynchronous data transfer between the client and server. After further experimenting with all the different tags associated with the WickedPDF module, Wes and I came to the conclusion that although it may not be as aesthetic, it was in the best interest of the group to make a PDF that displays simple HTML that WickedPDF could easily handle.

Wes's Contribution

This evolution I spent a lot of time making sure that the mass creation of events in a single text box worked. Although I did not design the controller that handles the logic, I had to be sure that we had covered all the corner cases we could come up with, and properly notified the user of any errors. I cycled through all combinations of events and possible omissions with a checklist, ensuring that errors would be caught and described properly. There were a few cases that did not work at first, like adding the possibility for preference-based sign up blocks, but eventually all issues were resolved. Finally, I accidentally pressed submit when the text box was empty, and the submission went through and gave a message indicating that my event was successfully created. Even though that error was not harmful because it didn't put anything into our database, it is nonetheless confusing to a user that may not know exactly what they are doing. I then went back and made sure that users aren't able to submit blank mass creations, and that a relevant error message is shown when that does happen.

Henrique's Contribution

In terms of to-do expiration and priority escalation, it required verifying if Rufus Schedulator would fire the event on the correct time. I had to use again the console to test several different cases of date inputs on a to-do and observe its behavior. For instance, I could set an expiration on the past or a recurrence prior that has happened already. On the first case I had to reject the input, but on the latter I would fire escalation right away.

To design mass creation I would not be able to troubleshoot it on the backend. This feature was rather graphical and needed error parsing to be sent directly to the screen. So I set up the environment to always delete any successfully created object so the database would not be overflowed with testing data. I was able to see the error messages and design the parsing algorithm

adequately after several trials and submissions. I had to ensure that all the parsing would occur successfully so I broke the process into several stages. Firstly, I parse the text into a hash of strings with keys and values. Then I verify if the keys are valid. Value parsing and conversion follows to finally then create the events.

For preference based slots, I had to create an environment with mock data to test the methods of the Repetition Scheme. Testing preference based slots was more cumbersome than other models because it demanded a set of allowed users, event blocks, and time slots within the event blocks.

3.2 Analyzed and Interpreted Data

Brian's Contribution

In all of the components that I worked on, the most important tool for analyzing and interpreting data was the Rails console and the server logs. One specific application was designing the forms for preference-based signups. In order to track which parameters were being passed and in which format, I used the server logs to look at the data format passed, and then the Rails console to check the database to make sure the correct data was being stored.

Jeff's Contribution

Throughout the entirety of this project, the primary data that I have needed to analyze and interpret has been outputs from the Ruby on Rails console. In this specific evolution, I developed the relatively simple algorithm of restricting a user's events to display in an email given the time frame from the user. In order to confirm the functionality of this algorithm, I had to utilize the rails console to make sure that given my input fields that the correct data was pulled from the database and either presented to the user in an text email, or further processed by the wkhtlmtopdf program in order to create a pdf attachement.

Wes's Contribution

The only time I can think of where I specifically analyzed or interpreted data was when I was ensuring that a user could select a certain date range and only have events sent to them in their schedule if they were within that date range. I created a lot of events around the same time, and then examined exactly what happened in cases where there was overlap between both the start and end times, just the start time, or just the end time. Our logic essentially excludes any event that has any period of time outside the exact window specified by the user.

Henrique's Contribution

When talking about models requirements, like to-dos and preference based sign ups, most of the data was analyzed through the console. Results such as changing the priority of a to-do or setting it to done after expiration were some of the data that had to be considered reviewing. For preference based sign ups, the results of the RepetitionScheme record were the most important. These methods were responsible for grouping and sorting the several time slot suggestions and for creating to-dos to other users.

Analyzing data for mass event creation was more complex. But since the process was broken down, I was able to step through it slowly. It took significant time because any input is fair game in a text field. So I had to try all combinations of wrong input on all types of event creation to

ensure data correctness. Data analysis was performed with several checkpoints on the state of the data throughout each parsing process.

3.3 Designed System Component to Meet Desired Needs

Brian's Contribution

One specific component I worked on was the collection of views for preference-based signups. I needed to create a new view for the creation of signup slots that are preference based, one to choose preferences, and then another to assign those preferences and give an assignment reccomendation. Given all of the new model and controller we needed to build along with this, I worked side-by-side again with Henrique to implement, test, and debug this iterface. This is one component where we could have really used the help of a Javascript framework like Angular, to help with dynamically adding HTML content and with form validations. Given the time constraints, however, I decided to implement all of this using just HTML, ERB, and basic Javascript.

Jeff's Contribution

The main system that I needed to design was not a particular piece of the applicaion, but more how various aspects of the program needed to interact. In order to create a PDF attachment of the schedule, I needed to properly query the database, filter out the events in an efficient manner, display that data in an convenient html format, and send it to another program to generate the pdf. The hardest aspect of this process was incorporating the program to convert the html list of events to a pdf. Not only were many open source programs not compatible with the version of rails we use, but the program that I ended up using had very poor documentation and examples. Once I got the program working within our system, I then needed to properly configure the binary of wkhtmltopdf to run on our heroku server (all of our local testing was on OS X system where as the Heroku server runs Linux). As a result of this, I had to rewrite our application configuration file to make sure to check the operating system that the program is being run on in order to utilize the appropriate wkhtmltopdf binary file.

Wes's Contribution

I handled the requirement that a user shall be able to get an email version of their schedule. This feature was a natural fit for me because I handled all of the email notification logic in our last evolutions. The emails that our app sent previously were exclusively plain text, so sending an HTML-formatted email was something I had to figure out. However, Mailgun makes it really easy to send HTML and attachments in emails, so I didn't have too much trouble sending a list view of a user's events once I figured out the formatting. The HTML that gets rendered is very similar to what we used to display a list of events to our users on their home page, with a few adaptations that eliminated duplicated events.

Henrique's Contribution

The to-dos dealt mostly with scheduling tasks. Therefore, it was reasonable to let each individual record handle its own task. Designing this way made it easier to abstract the logic away from the controller.

Brian took lead on the front end of slot preferences, which made I easier for me to only focus on

the model logic. To facilitate the process of gathering the correct preferences, I have added several methods to the RepetitionScheme model. These methods are helpful for easily retrieving data and delegating certain special actions. For instance, there is a method to gather a hash that maps the user record to its respective list of slot suggestion sorted by preferences. Another method takes care of generating the to-dos for each allowed user. The RepetitionScheme can also suggest an allocation of slot preferences as well as performing the final touches on resolving a slot assignment just by providing the selected slot ids.

For mass event creation, the focus was to have descriptive error messages. Therefore, there is a lot of different kinds of error checking and the line of the event in matter is passed around as an argument between method calls. To facilitate the process of building the query, the errors are broken down in phases and checked for correctness on each process. Therefore, key validation does not occur if text parsing finds a problem and so on.

3.4 Deal with Realistic Constraints

Brian's Contribution

After speaking with other teams I decided that our team may be much better off using a framework like AngularJS for our front-end. I did all of the online tutorials and read through the documentation, and built a small test application to try out the features. While I found it very useful and easy, I discovered that it was simply not fitting for what we needed at a stage this late in the project. AngularJS is much more appropriate for a one-page application, and does not lend itself well to partials. The use of directives and modules in Angular act much like Partials do in Rails, however, these components clash when used together. Given the time we had, and the list of requirements we were given, I decided to cut Angular and continue using normal HTML, ERB, and Javascript.

Jeff's Contribution

One key aspect where I eventually had to cut my losses on progress that I had made and realize that the time spent on the attribute was not helping the group overall was with PDF schedules. My overarching goal was to allow the user to specify a date range, and then utilize Fullcalendar to create a visually pleasing graphical schedule for a specified weekly period. However, most of the Rails Gems that were capable of converting a specific web page into a PDF file were very sensitive to web pages that involved a lot of javascript. Not only does Fullcalendar have a significant amount of javascript, but using it in the rails framework in very specific ways has been somewhat difficult. Although I implemented the ability to create an instance of the weekly Fullcalendar, my experimentation with the html to pdf converter was quite unsuccessful. When Brian emphasized to me that more work was needed on preference based sign ups, I decided that I was probably not going to get the complicated pdf generator working, and instead opted for a simpler version of the pdf to be attached to user reminder emails.

Wes's Contribution

One of the other contributions I made to our project this evolution was making sure that text and numerical inputs given by users are properly formatted. Our app was VERY fragile after Evolution 3, as evidenced by our botched presentation and other issues on our production server. So, I made sure that it was impossible to submit blank text fields for certain forms, restricted

the email sign up to actual email format, and handled a few other basic cases where a simple user misstep could break the app altogether. Although the checks are not fool proof, our app is definitely much safer now that we have some of them in place. I did these sanity checks using form validations and then specific checks where necessary. If the models or controllers sense an error, a relevant error message will show, and the form will not submit anything to the database.

Henrique's Contribution

Again, time zones were a big problem on this assignment. Throughout my coding time I have realized that some places were left without time zone check. That is because I did not realize it was a real problem over evolution 1 and 2.

Debugging mass creation was significantly time consuming. I had to constantly check the loggings on the server in case the program would not output the correct error messages. I had to write descriptive logging messages at almost each step to verify that everything was working properly. Matching the correct times to create the slots was particularly difficult because that is a lot of time window validation going on in the model. And with the wrong time zone, the slot would never get created.

Priority escalation end to do expiration had some problems in terms of the scheduling. Some times if I would try to unscheduled an already unscheduled job, or reschedule a new job, I could get a crash. As a consequence, I had to add several checks for corner cases so the Rufus Scheduler API would not break the project.

3.5 Contributed to Team Work and Interacted with Team Members

Brian's Contribution

Having had a bit of trouble with communication amongst the team in the past, I was a bit more dilligent about delegation and task tracking. We continued to use Trello for this, but we definitely saw some improvements overall with communication over Facebook chat and email. I again worked closely with Henrique to build some of the new features, and pair-programming helped a ton here. When we came together as a team, however, we did see a lot of improvement in the general productivity of the team.

Jeff's Contribution

Throughout this evolution, one aspect of teamwork that I tried to improve was my communication with the rest of the team. In previous evolutions, I would sometimes neglect informing my teammates on my progress (whether it be good or bad) on various aspects of the calendar application. Instead, any time I made any significant progress at all, or any time I didn't meet deadlines or implementation expectations, I did my best to inform the group as soon as possible. I believe this resulted in a better group dynamic overall and a more efficient working team. Within the team itself, Wes and I spent a significant portion of this evolution on implementing schedule emails. Because Wes had set up the infrastructure for sending reminder emails, he was very helpful in guiding me how to format emails correctly, and how to associate the correct data with Mailgun (our email service). We did a decent amount of pair programming that allowed us to catche each other's bugs and implement schedule email functionality relatively efficiently.

Wes's Contribution

Jeff and I both worked together on the schedule emails; he worked on the graphical view creation and I worked on the HTML within the text of a regular email. After I had finished my part, Jeff was still having trouble generating and attaching a PDF (our choice for graphical format) to an email. In order to help, I went over the logs from our email server, Mailgun, to see what was actually making it through and what wasn't. The example that Mailgun has online for attachments in Ruby on Rails uses something called a MultiMap, which is no longer supported. But through looking at our email logs and surmising the MultiMap was really just a hash, we were able to figure out how to add the PDF attachment to an email.

Henrique's Contribution

I have worked mostly on the back end and less on the front over evolution 4. The team was much more responsive and worked together to ease each others workload. My contributions are listed below.

To do priority escalation designed the priority escalation logic. Added the necessary columns and wrote the periodic firing of priority change

To do expiration designed expiration logic for to dos

To do view added the necessary fields to the to do view for expiration and priority escalation

Preferences sign up block creation allowing a user to choose a preference sign up block

To do sending allowing a user to send to-dos for a created preference based sign up block

Assignment suggestion short algorithm to suggest an optimal slot allocation for preferences

Slot assignment final logic to effectively assign slots to users. This includes deleting unassigned slots, marking sent to-dos as done, and updating the status of a RepetitionScheme

Mass creation view wrote the view for mass creation events

Mass creation logic defined the text format and code for parsing events in mass

Design document contributed to the controller and model sections of the design document