Eggplant Functional

Mykel Allen
Bethel Tessema
Bladimir Dominguez

CSM Field Session 2018

I. Introduction

Eggplant functional is a software company that offers several products that are used to test code. Eggplant's services are especially useful to non-technical companies by helping them test and improve their software. Customers can keep track of their purchased products through Greenhouse, a portal that allows clients to: view their products with Eggplant, generate licenses for said products, and transfer the licenses to a different host if the license has already been generated.

Our team was tasked with the recreation of the Greenhouse portal. At the moment, Greenhouse executes at a very slow speed, creating a negative experience for the users. It runs in this fashion because it was created over 5 years ago and has not been updated much in the years since. Greenhouse currently utilizes a deprecated Ruby gem called databasedotcom to make the API calls that store and retrieve the customer's data from Salesforce, the database used by the portal. Our job was to recreate the application from scratch using the most recent versions of Ruby/Ruby on Rails and to integrate the newest Ruby gem (called RESTforce) to update the API calls to the database.

We were also asked to completely rework the user interface. Greenhouse was created when the company was called TestPlant, so it currently has color schemes and logos that no longer correlate with the company's newest branding. The current app also uses static HTML pages, which our team was asked to update so that we could implement functionality like dynamic formatting when the browser is resized.

II. Requirements

In our first meeting with our client, we were provided with both functional and non-functional requirements to include in the project. Most of the functional requirements are already in effect on the current Greenhouse web portal, but they function using the deprecated versions of the Salesforce API. These features include the user login system, license generation, license transferring, and downloading license files. Our responsibility with these specific features was to update them to use the most recent Salesforce API calls.

Our client also requested that we develop some additional features that the Greenhouse portal does not currently offer. These features include better password storage and retrieval, error handling, loading screens, browser responsiveness, and grouping licenses by license order. When implementing each of the functional requirements- both those that need to be updated and those that need to be newly created- we were asked to keep browser compatibility in mind, specifically with Internet Explorer 11 as this browser is used by many of Eggplant's customers.

The non-functional requirements requested by our client largely revolved around communication about the project. We were required to use RedMine, the project tracking tool used by Eggplant, for scrum tasking. We were able to add our clients as watchers on our project so that they could view our progress and we could request feedback on specific issues at any time. We were also required to give our client weekly updates on our progress, which helped us make sure we were on the right track and allowed us to make changes with the requirements along the way as the client saw fit. Lastly, we were

asked to provide documentation for our code so that it could be easily expanded on in the future.

As we began to consider how we would approach the creation of the app, we outlined everything that could possibly go wrong along the way so that we could address those problems sooner rather than later. We grouped these risks into two sections, one being based on technical issues that we could potentially face, and the other looking purely at our experience with the tools used.

Technical Risks

We started off by looking at the data that we would be using, and when covering all of the information with our client, it seemed that there were a few issues with how we would interact with their database. Salesforce, the database in question, is not built like a traditional MySQL database, but instead uses its own querying language, SOQL (Salesforce object query language). At first glance, the database appeared to be extremely large and complex. After looking through the setup of the schemas within Salesforce, we concluded that many of the data items we would be using could be located in multiple tables. This could make it difficult to traverse the relational style of the database and find the correct data.

After reviewing the setup of Salesforce with our clients, we saw room for changes that could be made to the layout of the database. These changes, if done improperly, could mess up the existing data within their production schema, or render their app useless due to a missing field or another similar error. Outside of these issues with our database interaction, it seemed that the only other issues that we would end up facing would be unfamiliarity with some of the tools that were required.

Skills Risks

The most apparent and upfront issue that we anticipated coming into the project was that the app needed to be built using Ruby on Rails, and no members of our group had experience with the framework. There was also little experience within our group with making a web app, so there was some risk in understanding the design principles that go into creating a product like Greenhouse from scratch.

III. Architecture

Figure 1 displays all of the components, both inside and outside of the actual code, that are required for the functioning of the Greenhouse web application. The gray dashed line represents the areas within the code that our team recreated. Licenses for the different products offered by Eggplant are generated for Macs, Windows, Cloud platforms, etc. Each generation is slightly different based on the platform the license is being generated for. For example, node locked licenses require a host id, whereas floating licenses do not. The business logic for this license generation process occurs within the code for the application. It communicates with the Salesforce database to store and retrieve data, and with the UI to display that data to the user.

The data in Salesforce is accessed through the RESTforce Ruby gem shown in purple in Figure 1 below. Since we did not want to perform migrations on their live database, our program was tested on a SANDBOX version of the actual database used in production. A SANDBOX is a recreation of all the same tables and fields as the original database that can be used for development purposes. We were able to add data into the SANDBOX and query this data to test our code without disrupting the day-to-day functioning of the current portal. We were also able to add columns to the database for new features we were implementing, such as certain information needed for password resetting.

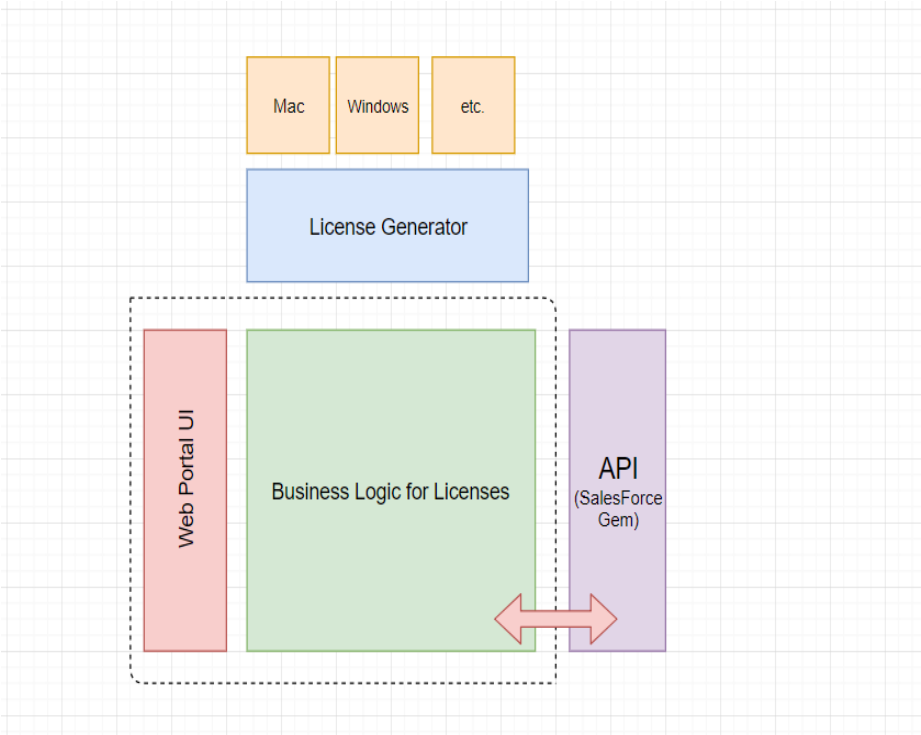

Figure 1: This image represents all of the moving parts required for the Greenhouse application to run. The gray dashed box represents the areas that our team was required to build/understand in order to recreate the application.

Web applications are structured to follow the Model-View-Controller (MVC) architecture. In general, MVC principles suggest that the models store the data used by the app, the views display that data, and the controllers act as a communication point between the models and the views (so additional logic can be placed here). Our code is structured to adhere to these principals. We followed the MVC guidelines by placing our queried data from Salesforce in the models, our UI code in the views, and our business logic, such as the logic for license generation, in the controllers. The model classes are where we made the majority of our updates to the API calls.

Figure 2 illustrates the general structure of the data in Salesforce. Each box represents a table in the database, and while Salesforce contains much more additional

information, the figure displays the tables that were most essential to our project. Each table is represented as a class in the model section of our code. Users are logged in via the Contact class, which holds the username and password information of the user. Each contact is tied to an Account class, which is used to access information about each user's license orders, license order items, products, and licensed servers; each of which exists as its own individual table in Salesforce and its own individual class in our code. The queries performed are placed within the class that represents the table the data is being accessed from, and the data is then stored within that same class. For example, the query to pull and store a user's products is performed inside of the product class. The code was structured in this format so that it would be clear on where to find any given piece of data and to improve the code's readability properties. Data that needed to be communicated between different models and controllers was passed through different functions to keep the desired structure.

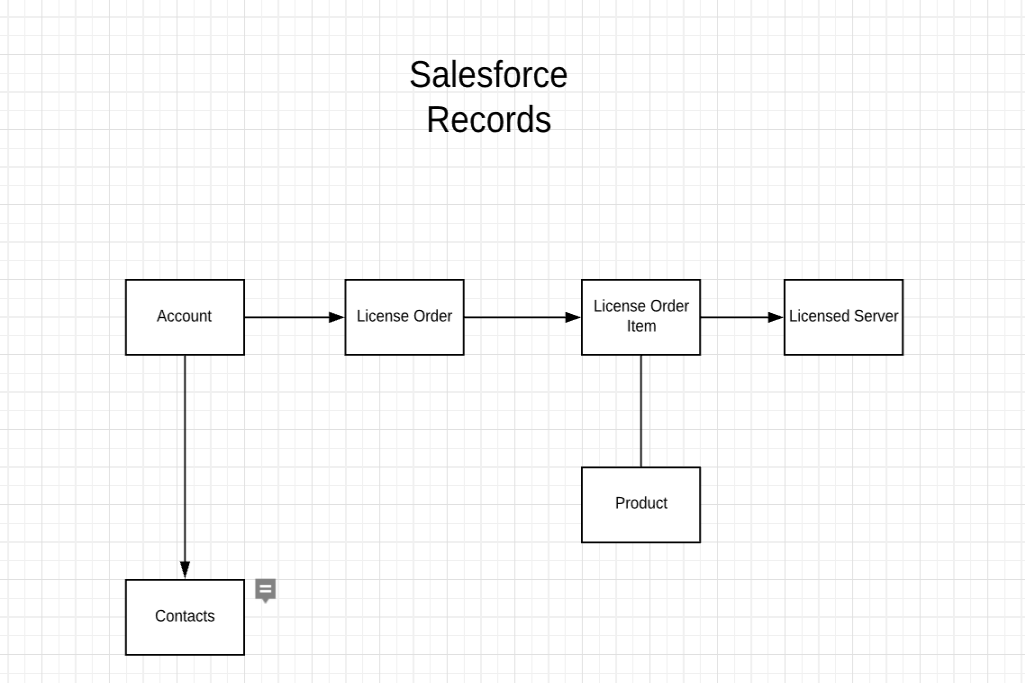

Figure 2: This image models the format that customer data is stored in Salesforce, the platform used by Eggplant to manage their customer licensing data. The records in Salesforce are accessed through the RESTforce Ruby gem.

IV. Technical Design

When a user interacts with the app, they have access to a few basic functions. The user will log on, see a list of their current licenses, and can choose to either download, copy the key from, transfer, or generate a license. We made a controller for each main portion of our app to perform the following logic: logging in, displaying all owned licenses, forgetting your password, changing your password, and generating a new license.

The login_controller is fairly straightforward. It authenticates the form submission it receives to ensure that a username and password have both been submitted. If one or

the other has not been received, a proper error message is displayed in the view. If both are present, it then looks for an existing contact in the database with a matching email address. If there is no user in the database with that email, another error message is displayed. Lastly, if a user is found, it takes in the password submitted by the form, hashes it, and compares it to the stored hash in the database. If they match, a session is created, and the user is redirected to their current licenses page. A session is a storage of information that can be used for the duration that the user is logged-in the app.

At this point, the app now exists within the license_controller, which finds all active licenses associated with the current user. A new header is now presented with links that allow the user to navigate to different pages within the app. These links allow the user to go to the home page, change their password, log out, or return to the licenses page. Underneath this header, a table exists with all of the important information regarding each license, such as its key, expiration date, the product it is for, etc. If a user has purchased a license but has yet to register their MAC address for it, there will be no key shown and they will have the option to generate a new one. This MAC address is important as it ties the license to a specific computer, and without it, the idea of distributing and requiring licenses would not be very easily achievable. If they have already generated the license once, then they will be allowed to transfer it given they have not done so more than twice already in the current year.

If the user chooses to generate or transfer a license, a new page will appear with the selected licenses' specific information, an input field for a new MAC address, and a submit button. The generate_licenses_controller is now active and is in charge of doing quite a bit of work. First off, it validates the MAC address submitted once the submit button is clicked. In order to do this, we utilized a regex statement that parses the submitted value and makes sure that it follows the proper format of a MAC address. These addresses are required to consist of six groups of two hexadecimal characters, separated by hyphens or colons (numbers ranging from 0-9 or letters ranging from A-F). In Figure 3, testing commands are shown that display what can and cannot be considered a proper MAC address. If the address passes, the controller then determines the type of license to be generated. Out of the four possible generators, the controller picks the right one based on various flags set to the product (such as whether the product is node locked, short term, etc.) and sends the needed information to the generator. The generator will then respond with a new key. After the key is finally made, a new license object is created in the database with the new key and a reference to the old key (for when we need to potentially count the amount of transfers performed).

```
# MAC Address validation
puts ('12:34:56:AB:CD:EF' =~ /^([0-9a-fA-F]{2}[:-]){5}[0-9a-fA-F]{2}$/i) #true
puts ('00:00:00:00:00:0P' =~ /^([0-9a-fA-F]{2}[:-]){5}[0-9a-fA-F]{2}$/i) #false
puts ('00 00 00 00 00 00' =~ /^([0-9a-fA-F]{2}[:-]){5}[0-9a-fA-F]{2}$/i) #false
puts ('00:00:00:00:00' =~ /^([0-9a-fA-F]{2}[:-]){5}[0-9a-fA-F]{2}$/i) #false
puts ('00:00:00:00:00:00:0P' =~ /^([0-9a-fA-F]{2}[:-]){5}[0-9a-fA-F]{2}$/i) #false
```

Figure 3: Testing of regex command for MAC address validation.

Lastly, we have two controllers with similar functionality, the password_change_controller and the password_reset_controller. We broke the password logic up into two different controllers because the actions performed by the two are fundamentally different. If a user wants to change their password, they are required to first be logged in. They simply must enter their current password, enter a new password that fulfills the requirement of being five characters long, enter the same new password as a confirmation, and hit submit. This new password will then be hashed and stored in the database. If a user is not able to login and change their password, they can click the “Forgot Password?” link on the home page, and an email will be sent to the email address they provide, given a user is found in the database with that matching email address. They will have two hours to visit the link sent to them and reset their password. Considering all of the form submissions and backend work required throughout the app, our group got a strong sense of how all of the different components of our code (models, views, controllers) work together and the flow of a Rails application.

The broad variety of different languages that were incorporated into our single app really helped with cementing our foundation of web applications. Our project alone uses HTML/Embedded Ruby, CSS, Ruby, and SOQL API calls, with the possibility to add JavaScript as well. Getting these languages to coordinate was easier than we anticipated, and it allowed us to easily run and test our code on a locally run server. This in turn was useful in ensuring that our logic was functioning as intended and that the UI appeared the way we expected it to.

Using multiple languages allowed us to add lots of functionality to the UI, making it look better and making it more responsive. Embedded Ruby is a system that embeds the Ruby language into HTML plain text code so that data from our backend could be displayed on the UI. By using embedded ruby, we were not only able to execute Ruby specific code within our view, but we were also able to create forms and instance variables using more compact and efficient code. This was useful in displaying essential data from Salesforce, such as the different licenses and information about those licenses, on the current licenses page of our project.

CSS, on the other hand, was used to enhance our UI by working with HTML to manipulate specific divisions and containers in the view. This manipulation can be simple, like changing background colors for certain portions of the page, but it can also be more complex and allowed us to add more intricate features to our layouts. Figure 4 shows one enhancement of standard HTML using CSS. The bottom half of the figure shows “filler links” created with standard HTML, while the top half of the figure shows links that have been enhanced with CSS. The links in the navigation bar in the top portion of the figure have dynamic hover highlighting. This allows them to show up dark gray when the mouse hovers over a specific link and show up blue to represent the link that the user is currently on. They have also been enhanced with ratio specific formatting depending on browser size. The combination of the benefits from Embedded Ruby, HTML and CSS were extremely helpful in UI design throughout the app.

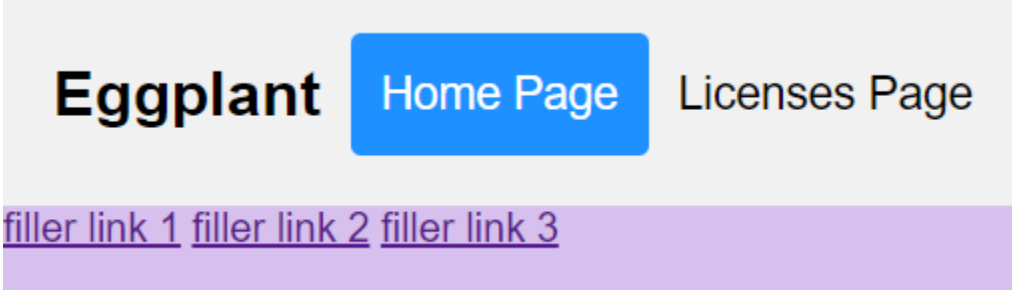

Figure 4: Top are links using CSS to format them like buttons, highlight grey when hovered over, and highlight blue when the active page matches the link. The bottom links are the standard html links.

V. Decisions

A lot of what we were tasked with implementing is already being performed by the previous app, such as how licenses are currently retrieved and stored, the layout of the session object, and the license generators themselves. The largest change we made, as mentioned, was improving the API calls of the app to a newer version. The structure of the code ended up being very similar to the previous version; however, we came up with a few design changes as a group to make the app more fluid.

User Interface

We were given lots of freedom regarding the user interface. The requirements specified by our clients included the following: the new portal should have the same functionality as the old portal, it should be responsive to web browser resizing, and it should be updated with the new company name and theme. The first step to creating our UI was developing a login screen that would set the theme for the rest of the website. As a team we sketched multiple outlines for the login screen and ran them by our clients. We took the feedback given to us and created our login screen, which would then outline the theme for the rest of the application. Screenshots of our final designs were sent to the User Experience team at Eggplant, and we were given feedback on those images so that we could continue to iterate on our layouts. Figure 5 shows the portal currently being used in production, which uses the company's previous name/color scheme. Figure 6 displays our updated layout with the company's most recent name, logo, and colors.

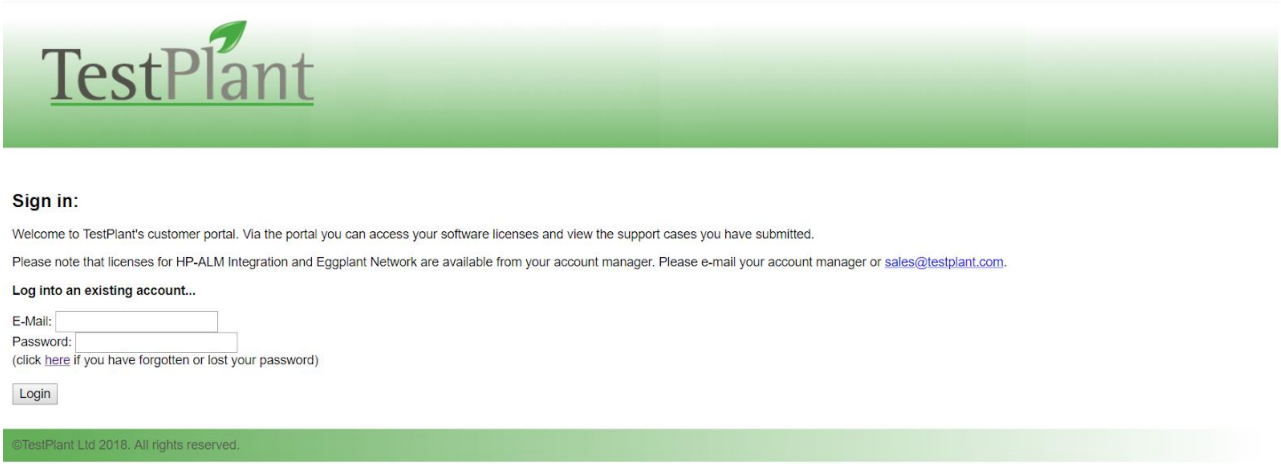

Figure 5: Current Greenhouse portal login screen.

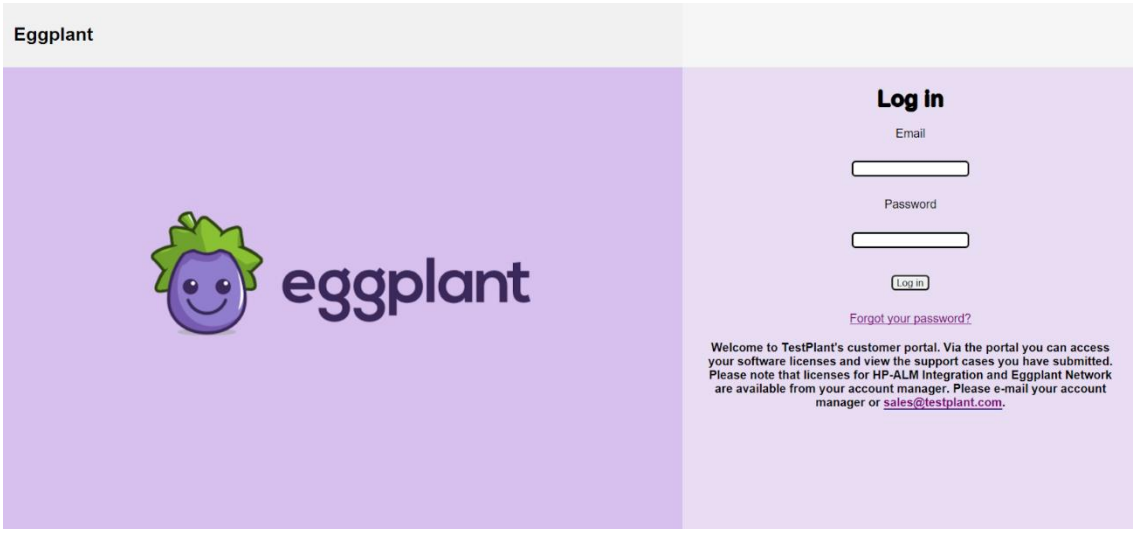

Figure 6: Our design for the new Greenhouse portal login screen.

VII. Results

The end goal for this project was to create a web portal application for clients that would replace the current one being used by the company. We were able to reach that goal and provide our clients with a completely new app, which we hope they will build on in the future.

This web application makeover was meant to increase performance in two main categories: speed and user experience. The new layout for the Eggplant portal has been modernized and updated with the company's latest images/color schemes to be more visually appealing to those using it. Additionally, this new layout is intuitive and has the same functionalities as the previous application so that users can comfortably transition to the new portal. When users log in to the new portal for the first time however, they will be asked to change their password. This is because currently, users have their passwords stored in plaintext in Salesforce, and when they forget their password, it is sent back to

them via email. We added a new hashing algorithm to the login and user saving processes, so with them resetting their password, it will then be encrypted in the database for an extra layer of security.

Since our project runs on a SANDBOX, it does not have the same amount of data as the actual database used in production. As a result, we were not able to compare the timing of the previous portal to the new portal with the same data. When timing the two portals on different datasets, however, we saw that the new portal ran considerably faster. When logging a user in, for example, the previous app took 69 seconds while the new app took 23 seconds. Although these tests were conducted with different users and different data, we expect to see similar results with our project run on the production database.

One of our client's important requests for the final product was that the portal would be compatible across different web browsers, specifically Internet Explorer (IE) 11 as this browser was the one expected to give us trouble, if any. We tested the product on several browsers including Chrome, Firefox, and Microsoft Edge and only found one issue; Internet Explorer was not displaying the background colors properly. We believed that this happened because this browser runs the app in compatibility mode (when active, it forces IE to display the page as if it were in IE7). Because of this feature and our upgrade to a newer version of Rails, a few attributes we had used, such as how we displayed our colors, were lost in the process of the page rendering. We resolved this problem completely by initializing the height of our body container to 98% of the viewport height, so our application is compatible across all of the desired web browsers.

Unfortunately, the only feature we were not able to implement due to limited time was downloading license files for some licenses. We have some ideas for both short-term and long-term future work that could further improve the application. In the short future, the UI could be enhanced by installing Bootstrap so that JavaScript functionalities, which are more advanced and have more features, could be integrated into the application. In the long-term future, we think the Salesforce database schema could be cleaned up to adhere to the Boyce-Codd normal form guidelines for database normalization (this provides multiple advantages such as reduced redundancy, increased data quality, and a better understanding of how the business is laid out). With all of this in mind, we believe that we have delivered a quality product with increased performance that can easily be expanded on in the future.

Lessons Learned

Field session was a unique experience because it allowed our team to get a taste of what our personal futures will most likely hold as software engineers. The lessons we learned throughout this summer also helped us in preparing for problems that we will most likely run into at least once throughout our careers.

Firstly, we should always be prepared to learn new concepts by ourselves. Coming into this project, our team had little to no experience when it came to working with Ruby on Rails and web apps in general. In school, we are used to having resources like lectures and homework assignments to guide us in learning new languages; however, we noticed that when working with a client, it is going to be our responsibility to find resources and

learn these new technologies on our own. We must take initiative to ask for assistance along the way from friends, coworkers, and within our group. While this difference in learning techniques felt strange at first, our group quickly learned how to become comfortable learning independently. We spent the first week of field session finding and reading online textbooks and watching tutorials about the new technologies we would be using, which greatly helped us in the following weeks of developing the product.

One of the most technically challenging aspects of our project was working with the Salesforce database. It is a much larger database than anyone on our team had previously encountered. Through working with this database, we learned that when starting a project or collecting data for the first time, it is important to make sure that the database schema makes sense to the developers and that it can easily be expanded on. This is just as important as understanding the new languages being used for the product. It is important to spend time watching videos, reading documentation, and writing sample code to fully understand the database and how it functions before you begin writing code for the actual product.