Email Automation Application

Abagail Krostue

Aidan Funk

David Fraser

John Santiago

Lydia Smith

CSCI 370 Field Session

Donna Bodeau

Summer 2021

Table of Contents

Introduction

The vision behind our team's field session project originated from our client, Datava. Datava works with banks and other financial institutions to develop software solutions to meet their unique needs. This project was created by Datava so that they would be able to have an application to meet their customers' email needs. The ability for economic institutions to communicate with their large customer base is valuable in engaging with their customers and alerting them of important information. It is also valuable to these institutions to be able to quantify the success of these messages. In the project proposal handed to us by Datava, it detailed that our team needed to create a web application integrated with Datava's system that would allow users to design, send, and track emails with a large number of recipients.

This application would make sending emails much easier for companies with a large number of customers. Without email automation, companies would have to keep track of when to send out emails and who to send those emails out to. Additionally, they would have to manually send out these emails. With automation, users can simply define user groups and rules for when to send out emails, and our app will do the heavy lifting.

This application would also give valuable feedback to email senders. A company would want to know if a particular email caused a lot of unsubscribes, was often marked as spam, or was immediately sent to recipients' spam folders. In these cases, companies would know when to switch up their email strategy. Alternatively, a company would want to know if their emails were well-received and that they do not need to change their email strategy. For example, if a user of the app sees a high sent or opened ratio and a large number of links clicked within the email, they would know their email is performing well.

The functionality of the email application was broken into three main components: the back-end integration of an email provider, the scheduler process that keeps track of when to send emails, and the front-end user interface.

The email provider we decided to work with was Twilio SendGrid. It allowed us to efficiently send out emails and track the effectiveness of those emails. We were able to access Twilio's services through API requests in which the input and output were formatted as JSON objects. On the front-end, we created a plugin for Datava's Table Manager application. This plugin allows users to save the filters they applied to a given table and send emails to everyone on that filtered list. After clicking the plugin button, a user can set email schedules and view stats from within our email application. This design allows users to conveniently send mass emails to their clients and receive statistics on those emails.

Requirements

Taking a deeper look into the vision of the project led to the defining of the necessary functional requirements on both the front and back-end, which work together extensively. On the back-end, the application needed to be able to use a schedule to determine when emails were to be sent through a scheduler that was always running so that no email was missed. It needed to be able to interface with the chosen email provider, Twilio SendGrid, in order to successfully send emails with the correct content. Twilio SendGrid is a cloud based SMTP provider that allows businesses/companies to send an email without the cost and complexity of maintaining their own servers. Thus, in order to interface with Twilio SendGrid, we had to use API requests to send emails. Once the emails had been sent, data gathered from the sent emails, including the number of emails that were opened by the recipient, needed to be obtained, managed, and displayed. Gathering the statistics also needed to be done through API requests as Twilio SendGrid tracks the emails through their system and we only had to grab this data from the request.

Along with the requirements for our own application on the back-end, we were tasked with making sure that all of Datava's database information was thought through and implemented into our program so that our application could be easily implemented into their current system. We needed to make sure that we would be able to access the information in their preexisting tables as well as add our plugin into the options of the different actions that can be taken from those tables. However, our plugin is only an option if the table has a column with user emails, otherwise there is no need to send emails with the information that is in the table. All of these functional requirements needed to be implemented as both a desktop application and as a data table plug-in to make this process as user friendly as possible.

On the frontend, functional requirements included a user interface for creating email templates, modifying and sending the emails, and accessing the statis obtained from back-end API requests from Twilio SendGrid. Another front-end requirement was ensuring that the UI was user friendly as well self explanatory. To ensure this made it a requirement to add all visual functionality that Datava had been implementing so that there was no change between applications as to which visual options are available. Since the back-end works closely with the front-end in order to fill all of these requirements, we had to work together as a team to verify that both the front-end and back-end teams were on the same page. Miscommunication between the teams could have resulted in data not being properly exchanged.

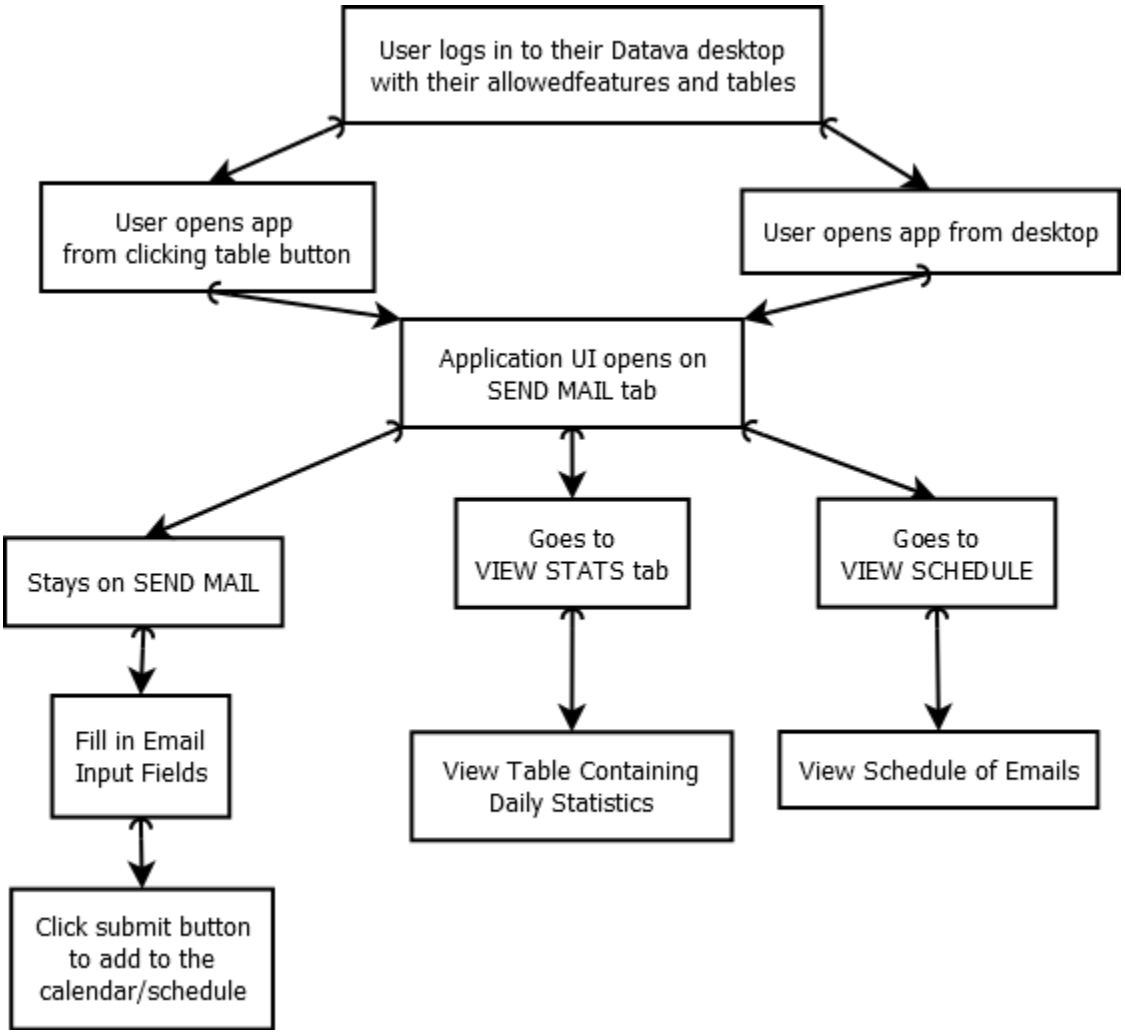
Additionally, this project consisted of several non-functional requirements. The code written for this application needed to follow Datava's coding standards. Along with the code, proper documentation, for instance quality comments, needed to be included. We also needed to make sure that our user interface was either extremely self explanatory or create a step by step guide on how to use our application and plug-in. When finished with our project Datava also requested some documentation of what files we worked with, if they were modified or created, and how they currently work so that for future development it is easy to understand how all of the code works.

System Architecture

To gain a broader understanding of how our application will be accessed and utilized by users, the user experience story given to us by our clients has been transformed into the following flow chart. Not only does maintaining this flow chart aid in demonstrating how every front and back-end aspect works as a whole, but it also provides a basis to conform from in order to easily consider changes if and when different ideas or client preferences arise.

The below diagram details the application workflow of the front-end. It shows all the possible actions a user of our app could take and the corresponding decisions a user would choose from after taking a given action. Ultimately it shows the steps a user must take in order to view data, view an email schedule, update an email schedule, or send an email.

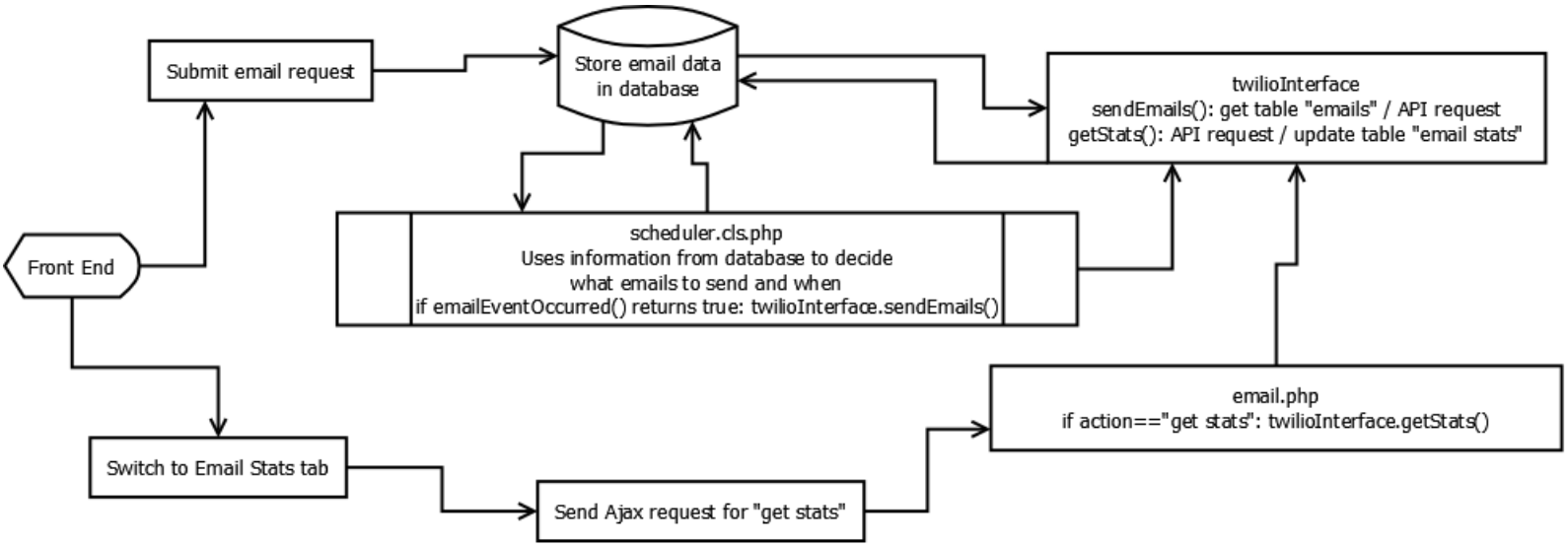
Figure 1 : User Experience Flow Chart



Our implementation of this email application lives inside of Datava's existing architecture. The tools that we used to construct this user experience were ExtJS, a component based javascript framework, and datava's pre-existing ExtJS components. Using these tools, we could quickly create our UI by composing these pre-existing and new components. As shown in the diagram above, we wanted interactions with certain front-end elements to trigger back-end functions; this includes saving data to the database and retrieving statistics. To accomplish this, we added a button listener to the "send email" button and a tab switching listener to the "view stats" tab. If these listeners detected an event, they would trigger an AJAX function which would make requests to Datava's server. AJAX stands for Asynchronous JavaScript And XML, and it allows web pages to be updated without reloading the entire page by exchanging data between the front and back-ends. These requests would then be handled by a main php file which would map front-end requests to back-end functions. These functions include reading and writing to and from system tables as well as interfacing with the Twilio API. To help visualize this process, we created a process flow diagram which models this relationship between the front-end, the request router (email.php), and the back-end functionality in the twilio Interface.

Javascript and PHP coding languages allow for communication between our front-end user interface, the back-end server-side functionality, and out to Sendgrid. This communication is essential for maintaining accurate data and ensuring that the logic the user expects follows through properly. This back-end process flow detailing the server actions associated with user interface components can be seen below.

Figure 1.1 : Overall Back-End Process Flow



Both the front-end and back-end communicated with the database stored in Datava's system in order to access and manipulate data. The front-end needed a way to save and display the data in a clean, user-friendly way, while the back-end had to access the data in order to create the emails and send them out at the correct times. The back-end also updated certain columns in the database to assist in keeping track of what emails have been sent out.

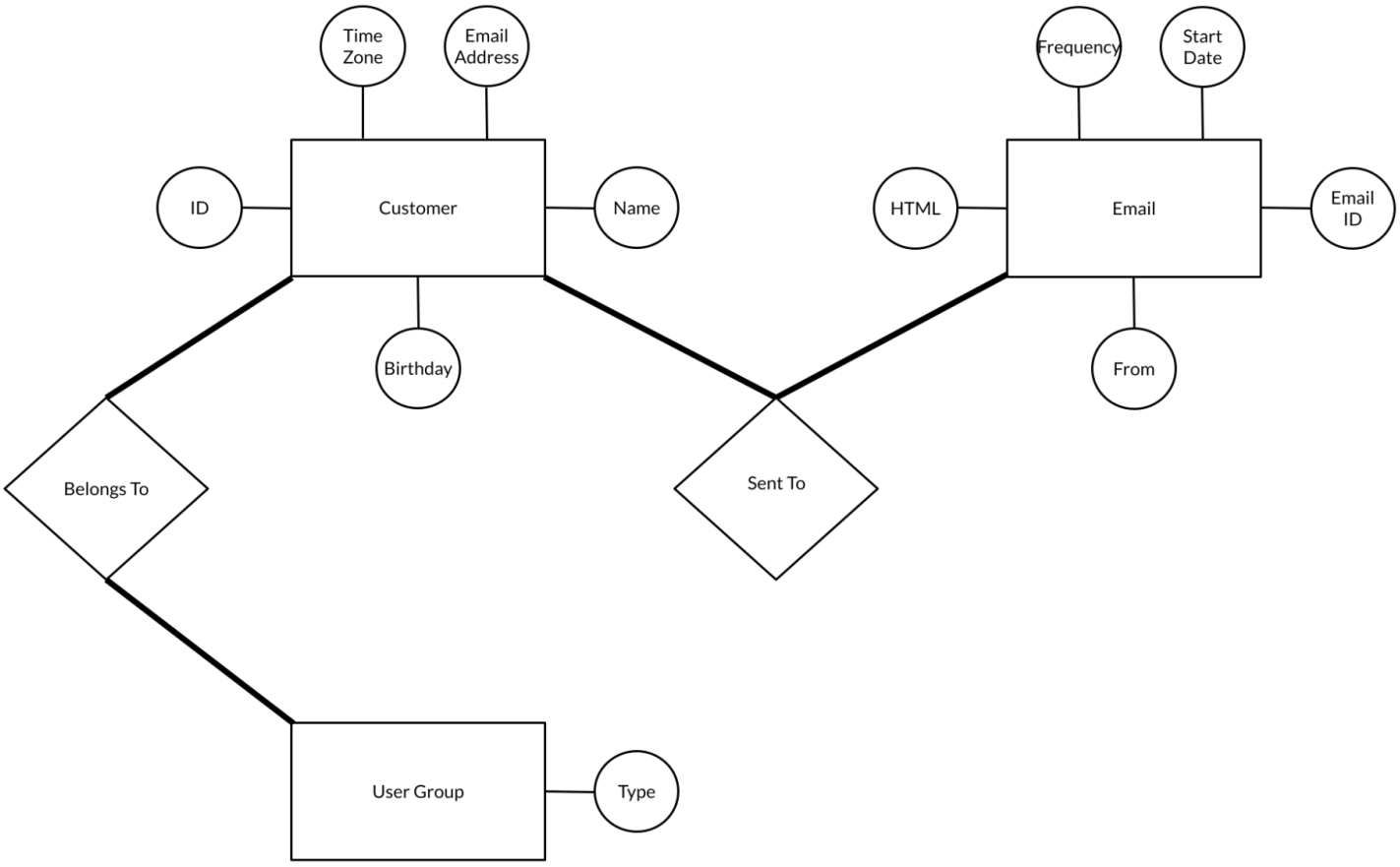
As shown in both of the flow charts above, the back-end and front-end had to work together extensively in order to make this application as user friendly as possible while still being able to complete all of the tasks. The Overall Process Flow Chart shows how the data is moved from the front-end to the back-end; the User Experience Flow Chart shows how the user will interact with the user interface, as well as how the user interface sends information to the back-end. Overall, these flow charts were helpful in understanding our approach before we created these systems so that the entire team was on the same page as to how the front and back-end would work together.

Technical Design

One important technical aspect of our application is the data associated with it. The data that we collected throughout this project includes email template data, email schedule data, customer data, and email performance data. The email template data stores the messages inputted by the user, which act as an email's body. The email schedule data allows the user to view all of the emails set to be sent and provides the ability to delete a queued email. The customer data refers to the table columns from which contact information and details can be accessed. Lastly, the email performance data exists as the statistics retrieved from SendGrid. Our application stores this data in Datava's system by using specific entity relationships to aid effective email automation. This data will also assist Datava's customers in planning their emails as the statistics that they get back will give them access to information regarding the success of their past emails.

An interesting aspect of this entity relationship is how we related customers and emails in our back-end. We did this using two tables: a customer table, and an email table. We chose to give our email table a recipients column that referenced a subset of emails in the customer table. We call this subset a user group in our ERD models. With this approach, we were able to access customer information such as name, email address, birthday, etc. for all customers listed for a given entry in our email table. This design also allowed us to inject customer data into the HTML specified in the HTML column of the email table. With this approach, we could generate personalized emails that depended on the recipient of a given email. One simple use case of this was injecting the user's name in the greeting portion of an email. Another advantage to our schema is that it allows for emails to be sent depending on customer specific data such as Birthday. Because our customer and email tables are related, our back-end can simply iterate through customers on a "birthday" email list and send to those with a birthday. Ultimately, relating customer lists and emails allows for many practical use cases. Below is the ERD which shows this relationship between customers, emails, and user groups.

Figure 2.0 : ERD of Customer Email Entities



Another interesting aspect of our design is how we measured the performance of the emails sent with our tool. We decided to track metrics such as opens, unsubscribes, reported spam, etc to give users insight into the effectiveness of their emails. For simplicity, we decided to store the aggregate, daily statistics across emails in a statistics table. This table can be viewed in our user interface under our ‘Stats’ tab. The user can additionally create a chart of the stats retrieved in order to visually display each of the metrics. This data is able to be used by the consumers to determine which email campaigns were most successful as well as how many people have actually seen their emails that are being sent. Without this information, the customers are unaware of the success of their emails.

Figure 2.1 : Email Performance Statistics

Quality Assurance

There are a number of steps that we have taken during the development of our project to ensure the quality of our work and product. The overarching guidelines outlined in Agile and Scrum development principles. We implemented these by holding daily standups with our client, using the tool Trello to maintain an updated backlog, and performing sprint retrospectives in order to provide Datava with continual pieces of working components.

Implementing pair programming proved to be another valuable portion of our quality assurance. This aided our team in ensuring concise code, as well as reducing roadblocks encountered through the input of multiple perspectives. By implementing pair programming we also have been able to ensure that our code is high quality and everything is being implemented correctly. Not only has this ensured our code had great quality, but it also helped us to come together as a team and bounce ideas off of each other so that we were all growing as programmers.

Another tool we implemented to strengthen the quality of our application included code reviews. During many of our daily stand up meetings, Datava's team reviewed our code and gave us comments and critiques. These reviews aided in our understanding of good javascript and php coding styles, showed us available resources to utilize, and gave us a better understanding of what our client envisioned for the final product. These reviews with Datava were extremely helpful as we were able to either confirm that our code was going in the correct path or the Datava team could walk us through what they were expecting so that we could understand how the code worked as well as what their vision for what the end product consisted of. However, we did not only do this with Datava, but sometimes if we were working alone and hit a blocker, we asked another team member to do a code review of our code. This helped so that we could try to work through as much as possible without Datava's team's assistance and so that we were all on the same page as to what was being created and how the code worked. It also helped ensure that we did not have any simple mistakes that could be solved easily with another set of eyes. Additionally, we did reviews as a whole team to ensure non-repetitive code and clear documentation.

In order to assure quality of our code for all users, we had to account for edge cases. For example, we had to account for users across different time zones. Our solution was to use the GMT international standard. In addition, we had to account for collisions in customer information like in their first and last name. We addressed this by making email the primary identifier for customers in the back-end. Finally, we had to consider all frequency options a user might want e.g daily, weekly, etc. We implemented our scheduler to account for all of these frequencies. Other edge cases we also considered included if users picked send dates that had already passed, if an email were to be removed from an email already in the schedule under an interval, and many other ways a user may use the application in a way different than what was intended.

We implemented user interface testing manually to review the functionality of our design by accessing the UI through Datava's server. Using the inspect tool, we were able to see responses and gauge which of our ajax requests are working and then

debug where we had problems. This was also helpful when attempting to populate tables, for we could check if the table existed and then see which information has been populated, if any. From there if any problems persisted, we returned to our code and determined where the error was located.

To assess the results of our back-end code, we utilized unit testing. For each unit of functionality, we created a set of punit tests that failed. Then we implemented our code and ran the punit tests. If they passed, we moved on to the next unit of functionality, otherwise we fixed the errors within our code and checked that the changes made met the expected results before moving forward to the next test. Another way we tested back-end functionality included using AJAX requests from the front-end and inspecting the result with chrome dev-tools. For the email aspect of our project, we sent test emails to our school and personal email addresses to see if they were sent correctly.

Results

Throughout the project our team has primarily used two types of testing: user interface testing and unit testing. User interface testing was very helpful throughout the development of our project as we were able to manually test our code through Datava's server using the inspect tool that is part of Chrome's developer tools. This helped us to see responses and gauge which ajax requests were working and then debug where we were having problems on the front-end. We also used the debugger that's built into Chrome's dev tools which allowed us to set breakpoints and step through our code line by line. User interface testing was especially helpful when testing the machinery of sending out email and retrieving stats before the front-end could call these functions.

In addition to using Chrome dev tools, we also used unit testing with the pUnit PHP testing framework. We created many pUnit tests to initially fail before we implemented our code. Once our code was implemented we were able to ensure that the functionality of our code was running smoothly and if not we were able to move on to the next piece of the project. This was particularly useful in verifying the accuracy of retrieved email statistics. Both of these types of testing were critical to the development of our product because without them we would have had major blockers throughout the process.

As we worked through this project we learned a lot of lessons as a team. One of the lessons learned throughout this project was the importance of the debugger, both in the vscode for the back-end and Chrome's developer tools for the front-end. Using the debugger aids in tracking down bugs and logic errors in a way that builds understanding of what is actually occurring and reduces time that could be wasted attempting to resolve the error through alternative routes.

This project also served in strengthening our utilization of version control. In comparison to using github for classes between groups of two or three students, learning how to effectively use github between larger project groups and within companies is important in communicating changes and maintaining code through potential conflicts.

Another lesson learned from this project was the importance of understanding the company's database before beginning any coding. Though the syntax of different programming languages are similar, the implementation of each is different based on the industry. Therefore, the more exploration we did throughout the company's database we were able to learn about how they were implementing certain elements on the front-end and back-end as well as avoid reproducing functions that may have already been implemented in their system.

The next lesson we learned was understanding the integration between the front and back-end of web applications. The importance of this lesson was to understand how both ends worked to be able to connect everything all together. Throughout this project, our team had to split between the front-end and back-end teams in order to efficiently be productive. However, there was a moment where both teams did not understand what the opposite team was doing. Therefore, pair programming and code

review helped very much get everyone on the same page to be able to move forward and connect everything together.

Future Work

There are several additional considerations that could be implemented in future work in addition to the features that we did not have time to finish. One of these is the potential for adding the functionality of cc and bcc recipients to an email. This pertains to adding additional emails to the send list that do not currently exist in the chosen table, which could be useful to, for example, add a supervisor to keep them informed on relevant messages.

The team began adding a more advanced email filtering system, but this wasn't able to be fully integrated with the UI due to time constraints. By adding in this feature, more comprehensive tables of emails could be created. It would also be beneficial to monitor the number of emails being sent out as a security measure for unusual activity. Several artifacts of code relevant to this future functionality still reside in our branch along with comment documentation of its potential use to the next developer to further improve the application. This should be fairly easy to accomplish since the advanced email filtering system is there. All that is needed to do would be to save the query, rows of names selected, or save the post request into a table that can be read to be used later when it is needed.

In addition to more features, having the ability to autofill message templates through pulling information from a client table would be beneficial to sending out mass emails without having to individually type each message out to each individual. This would include having the ability to insert a variable where a recipient's name or account information could appear within the email body.

Through SendGrid, when user's unsubscribe from an email, SendGrid blocks emails from being sent out to these recipients. However, in our database these recipients will still exist in the table. It would be more space efficient to remove them from the database, which would also eliminate the need to rely on SendGrid to block these emails from going out.

Another potential data security measure might be to control which users have the ability to send emails and view the gathered statistics based on a user's admin level; this may already be handled by Datava's login system.

Currently, our application works through having a user select recipients from a datatable containing user email information, from which they open our app, create the email, and then schedule it to be sent. From giving Datava a demonstration of the most current version of the application, they explained how it would be beneficial to send different emails to different people based on information stored in a given table; an example case of this is sending a user a 'Happy Birthday' email, which would not be the sent on the same date for many users.

Appendix

Sources Cited:

Information pertaining to our client Datava.
Datava. 2021, https://datava.com. Accessed 10 June 2021.
Twilio Send Grid Email Provider.
Twilio. Twilio Send Grid, 2021, https://www.twilio.com/docs/sendgrid. Accessed 11 June 2021.