`

YOOBEE COLLEGES

Bachelor of Software Engineering

Diploma in Software Development

CS205

Investigative Studio 2

(NZQF Level 6, 45 credits)

Stage II

Implementation

Team members:
Anastasiia Karpova

Based on the research question:
“How can we enhance a scheduling application for busy, small business owners?"

This report belongs to:
Chris (Hui Hui) Chong
210758235@yoobeestudent.ac.nz

`

Table of Contents

`

Introducing Boostly

Our Production Process

We developed Boostly based on our [research proposal and PoC that we conducted]. To summarise our research:

- Our target audience are small business owners who use Timely (an appointment scheduling SaaS)
- Timely has a “waitlist functionality” that functions just as a text-only notes section. If the business owner wants to notify their waitlist, they need to manually message each client on their waitlist – An extremely time-consuming process.
- **Our proposal was to create Boostly** – An application built on top of the existing Timely application that knows when an appointment slot has opened up, and allows business users to easily send notifications to their waitlist.

Timely does not have any open APIs available for external developers to use, so syncing Timely and Boostly is not a straightforward process. As such, we have broken Boostly down to **3 main development phases**. A detailed list of our tasks and timelines have been included in the appendix at the back of this report.

Phase 1 - Serverless functions

[22nd April to 2 June 2023]
Syncing Timely <-> Google Calendar Google Apps Script -> AWS Lambda -> AWS Simple Email Service

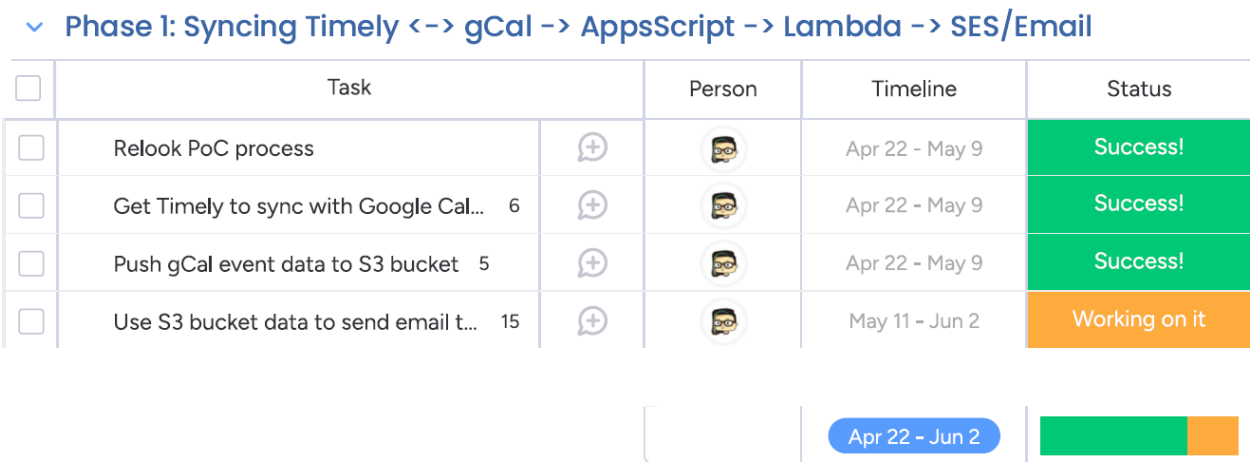


Fig 1. Breaking down our timeline into bite-sized phases[1].

[1] The full timeline with a detailed breakdown of our tasks can be found in the Appendix

`

This phase is comprised of the most moving pieces and the most uncertainty. It is also important because it triggers the start of each interaction with Boostly. Because of the number of moving pieces in this phase, we've drawn up a graphical diagram[2] to better display the sequence of events.

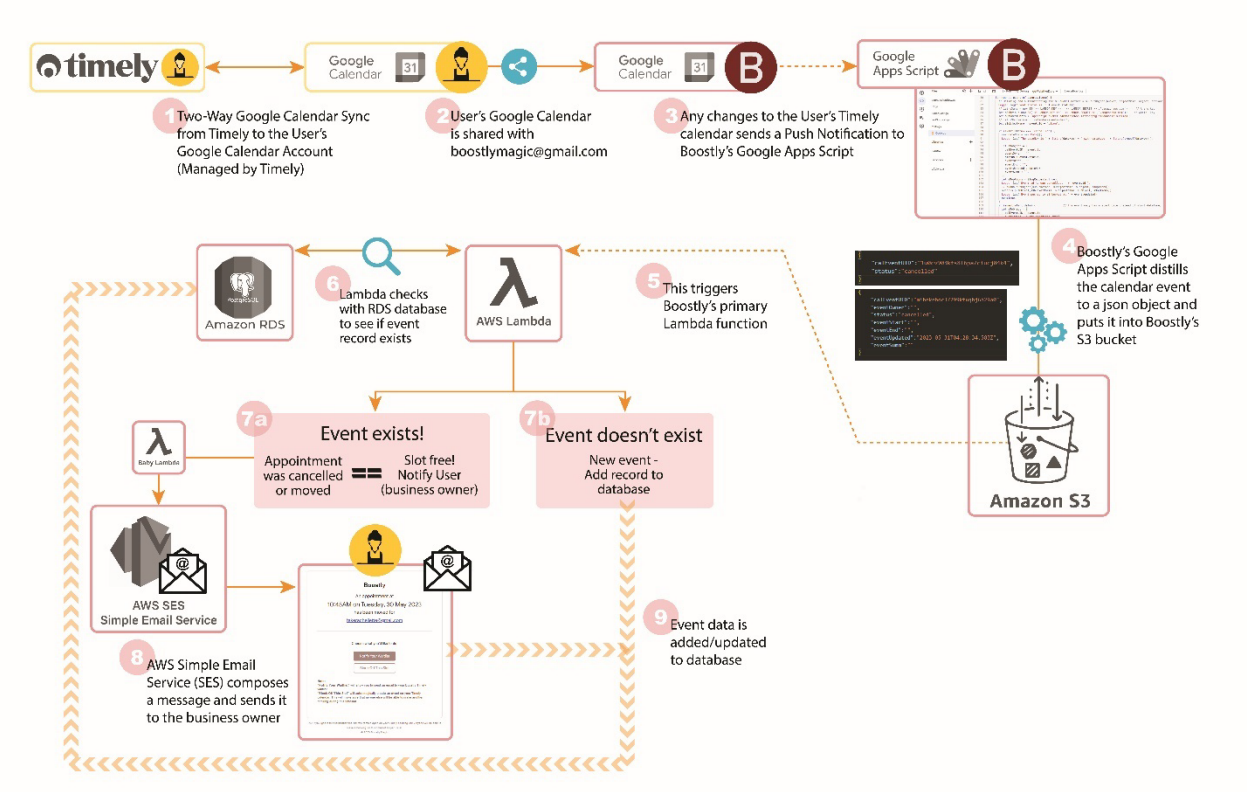


Fig 2. Graphical flowchart depicting the sequence of events for Phase 1 of our Boostly application.

More development detail and high-resolution screenshots of the scripted components can be viewed in the Technical Detail portion of the report below.

We broke these components down into bite-sized, manageable components that we would tackle daily. This stage also took the most time, as it involved trying to sync up three different software service providers (Timely, Google, & AWS) that were naturally not set up to communicate with each other.

There was a lot of uncertainty at this stage as to whether this project might be viable. To combat this, we did two Proof of Concepts (PoCs), limiting the amount of time that we would spend, so that we would be able to drop this project if absolutely necessary in order to complete our assignment on time.

Fortunately, we were able to complete both PoCs in a reasonable timeframe and proceeded to polish up the process flow and move on to Phase 2.

[2] A larger landscape version has been attached to the end of this report after the Appendixes

`

Phase 2 - Main application

[29th May to 28th June 2023]

Business owner (User) login, client management, sending notification to clients

Phase 2: Business owner login, client management, sending notification to clients

Task	Person	Timeline	Status	Comments
Setup + Boostly User functionality 11		May 29 - Jun 17	Working on it	Note: Business Ow...
Client management functionality 4		Jun 7 - 16	Success!	
Waitlist functionality 5		Jun 1 - 19	Success!	
Notification to clients 6		Jun 13 - 28	Working on it	
Testing 3		Jun 14 - 28	Working on it	
		May 29 - Jun 28		

Fig 3. Managing Boostly's User-facing functionality.

This phase consists of the components that Boostly Users will be able to see and interact with. The User would have to first sign up for an account before they would be able to receive the notification shown in Phase 1. We used a relatively standard registration and login process that we will not go into in this report.

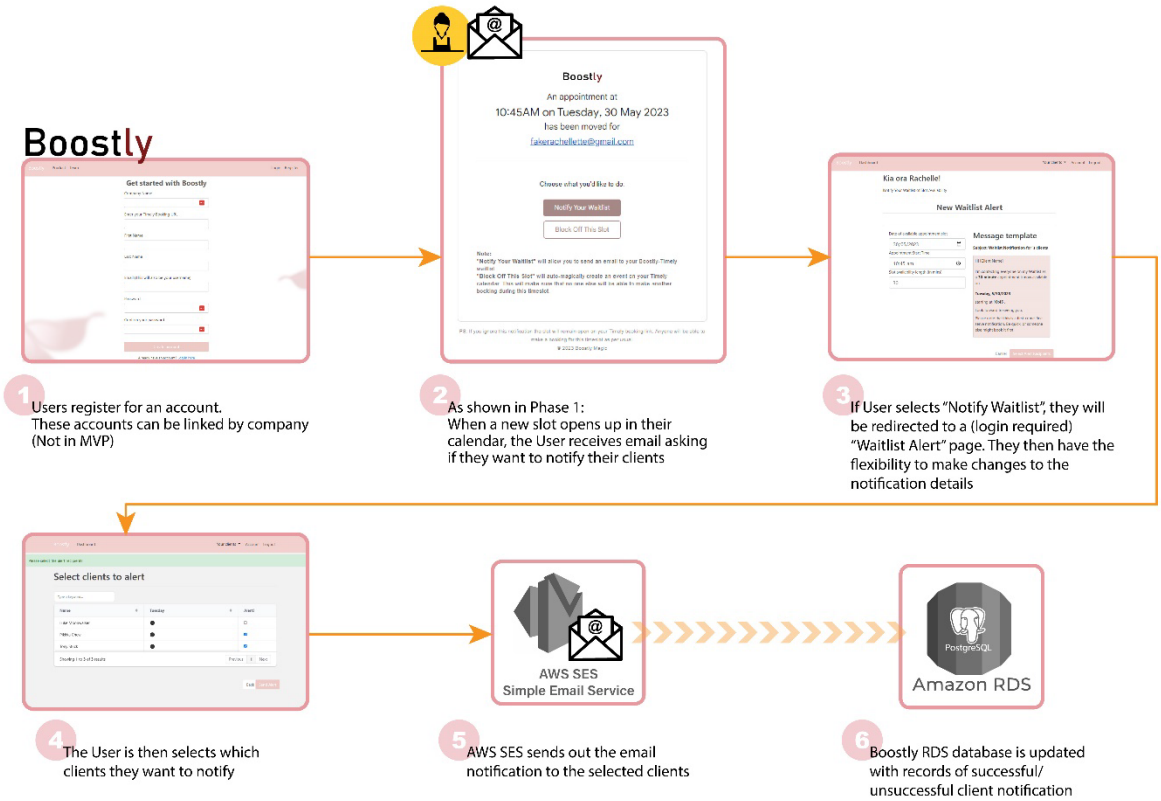


Fig 4. Screenshots depicting the sequence of events for Phase 2 of our Boostly application

`

While running this through with our test User, we found that there was a need to add a step between a new slot opening up, and their clients receiving a notification. The justification for this has been detailed below, where we validate the evolution of Boostly's capabilities.

Phase 3 – Packaging

[23rd June to 29th June 2023]

Tidying up, deploy, testing, report, and presentation

Phase 3: Tidying up, deploy, testing, report, and presentation

Task	Person	Timeline	Status	Comments
Live user testing with Lena		Jun 26		
Tidy up 4		Jun 26 - 29	To do	
Deploy using Elastic Beanstalk 1		Jun 26 - 28	To do	
Report		Jun 23 - 29	Working on it	
Presentation		Jun 29	To do	
		Jun 23 - 29		

Fig 5. Testing, deployment, and documentation timeline.

I personally would have preferred to have more time for this phase, but unfortunately, with the limited resources that were further depleted with me falling sick, we needed to be able to churn this portion out in the span of a week. We also had to find another live user to test our product as the schedule of our initial user (Lena) clashed with ours, and the testing was done very informally to simply ensure that they were able to do extremely specific tasks.

Unfortunately, because a large bulk of the functionality did not have a user interface, we were unable to get UI feedback for most of the project. While we did constantly check in with our small group of target users, describing the process flow is still very different from carrying out the activity.

We have highlighted only the major changes and justifications in this document. For a more comprehensive breakdown of changes, our SRS Documentation has been updated and is included with this report. Any changes to the documentation have been highlighted, with justification for those changes included.

`

Setting up a CI/CD pipeline using AWS Elastic Beanstalk and AWS CodePipeline

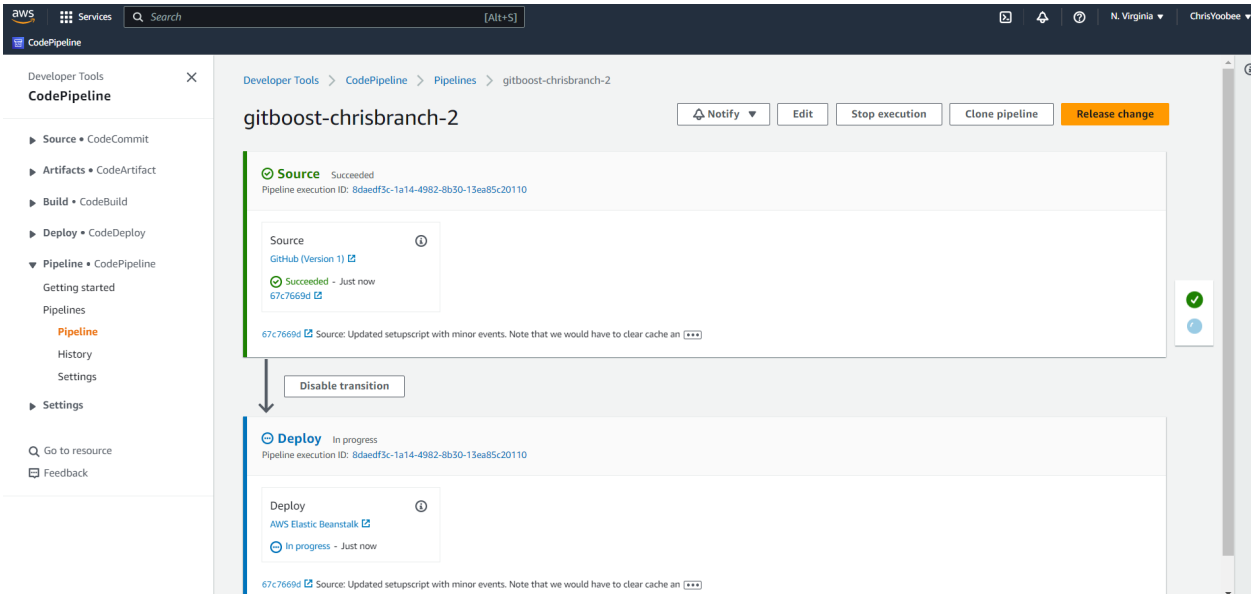


Fig 6. With CodePipeline, each push to a branch meant making an incremental change in our AWS Elastic Beanstalk-deployed application

The costs of Elastic Beanstalk are not huge when it comes to commercial expenditure, however they are quite significant to students with limited funds. As a result, we kept quite a lot of our deployment local.

Prior to this, we had also attempted to set up an Elastic Beanstalk environment linked to Github but was unsuccessful and abandoned that path due to time constraints. The process of manually deploying to the cloud each time we wanted to make a change was frustratingly slow, so when we had reached the stage where we had a product that was worth deploying on the cloud, I gave this another shot.

This time, we were able to successfully deploy CodePipeline linking Github to our AWS Elastic Beanstalk environment. The difference it made when pushing changes was huge! Once the CI/CD pipeline was set up, I found the transition from local to cloud to be very smooth.

`

Validating the evolution of Boostly

As we developed Boostly, we found that we had to make changes to parts of the process flow and technical design. These changes were not taken lightly, and each change was made to help enhance Boostly's core functionality and/or architecture.

We decided to focus on Boostly's core purpose and capabilities - automating the process of business owners notifying clients on their waitlist. In return, we would compromise on the scale – Only business owners could add clients to their waitlist. Their clients would not be able to do so for our MVP. This is so that we have a solid foundation that we can build on in the future, reducing the need to refactor or re-architect our entire project later.
The changes that we made to Boostly involved:

1. Achieving seamless synchronisation between Timely and Boostly
    a. Getting reliable appointment data
    b. Modifying the customer onboarding process
    c. Adding a step in the notification process as a safety feature
2. Changing of terminology and model classes
3. Removing functionality for clients to self-manage their waitlist notifications

Our SRS documentation has been updated, with any changes highlighted in green, and has been included with this report.

Achieving a seamless sync between Timely and Boostly

The synchronisation process between Timely and Boostly was one of our largest challenges for two reasons:

Problem I: Getting appointments from Timely to Boostly in a clean and reliable way

Getting appointment information was an interesting challenge because Timely does not have an API for us to communicate with. In our initial proposal, the goal was to simply forward Timely notifications to AWS Simple Email Service and parse that information to register each appointment and change.

While trying to implement this, however, we realised that the parsing of email data would be an extremely unreliable process. Timely also occasionally has a huge lag time between the appointment cancellation and the business owner receiving the email notification. If Boostly received an email notification out of sync, all the database records could get messy very quickly.

`

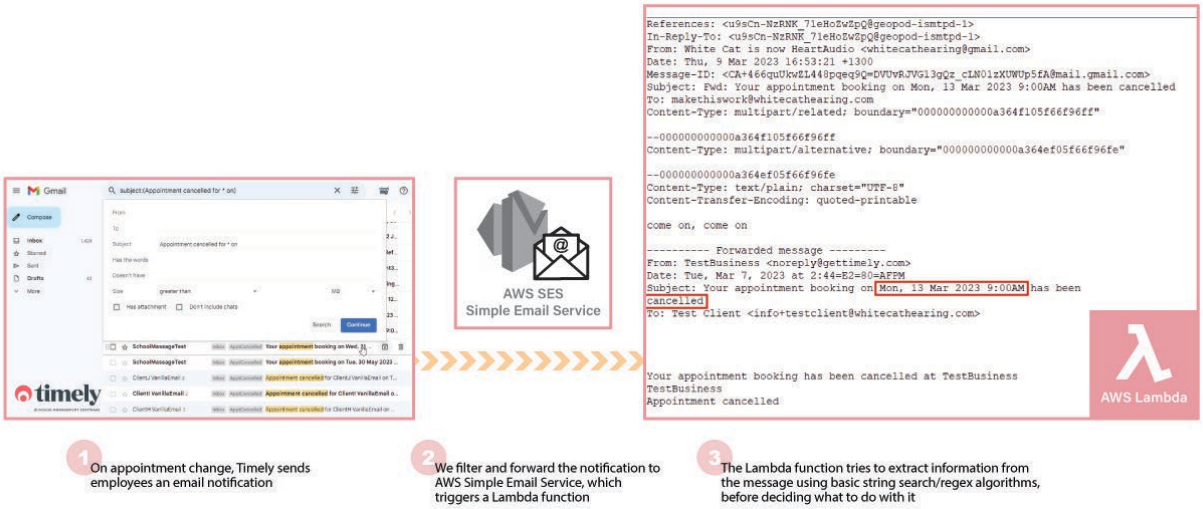


Fig 7. Initial process to get appointments from Timely was mistake-prone, especially if emails got delayed and bulk-sent an hour later.

```
Return-Path: <forwarding-noreply@google.com>
Received: from mail-oa1-f47.google.com (mail-oa1-f47.google.com [209.85.160.47])
 by inbound-smtp.us-east-1.amazonaws.com with SMTP id vkh0ueoa3v0sr1tkr0laqqck9mm1fph4trknmh81
 for boostcancelled@whitecathearing.com;
 Fri, 14 Apr 2023 04:33:07 +0000 (UTC)
X-SES-Spam-Verdict: PASS
X-SES-Virus-Verdict: PASS
Received-SPF: pass (spfCheck: domain of google.com designates 209.85.160.47 as permitted sender) client-ip=209.85.160.47; envelope-from=forwarding-noreply
Authentication-Results: amazonses.com;
 spf=pass (spfCheck: domain of google.com designates 209.85.160.47 as permitted sender) client-ip=209.85.160.47; envelope-from=forwarding-noreply@google.c
 dkim=pass header.i=@google.com;
 dmarc=pass header.from=google.com;
X-SES-RECEIPT: AEFBQUFBQUFBQUFGdkRpbndTRUZjZ3FEWXVhY29IOTVvUDg1RkhuVEdzSHk2djAzdXQzWjJoWjdPSE1kbjd1dG5VYi84TEpra2QwUk9zTnh3VXZUWjBXZGxrb3EwUGk4OVhTY1J4ZGh
X-SES-DKIM-SIGNATURE: a=rsa-sha256; q=dns/txt; b=hhQ/pIG3/zH/beUU76nS6R6I2Zb8XmzRIIRJjhkolGNvTVuMpQr4PA1jFyxbptKNc24M60oPaGbjT1oytw19g3rpnNt3KO4dfVx9IIyWL
Received: by mail-oa1-f47.google.com with SMTP id 586e51a60fabf-18779252f7fso8375081fac.12
        for <boostcancelled@whitecathearing.com>; Thu, 13 Apr 2023 21:33:07 -0700 (PDT)
DKIM-Signature: v=1; a=rsa-sha256; c=relaxed/relaxed;
        d=google.com; s=20221208; t=1681446787; x=1684038787;
        h=to:from:subject:message-id:date:mime-version:from:to:cc:subject
         :date:message-id:reply-to;
        bh=cPxIeupBuCcShA+beLwpWCUDZ5yPTlLonuHdd9/ZO4Q=;
        b=iUSreKIUaI+5ZhHbA+YMKNfwxlJqwl7FFDDgWoVOVUuY7v43+GmQomgRw0IH+KE9YJ
         D5G8U8zLk8xfJJPl32V4Wl1qS0oKOwdIWjMrH1/FFF323wXODBzmqvZfrHBWU4n3xmsw
         ++cSKZFFsHSYWH9dvyClfPlY10qnb9M4O2WvJDuAVLSO9FrcsfldPaGu8p3cRcb+D/gV
         Gr+gjuPA9Bh5LP91L5gptZ6hD1L8QEovxyWF8QtXyMddH+qgx+XJbGwxc07q9KqF/qXZ
         O8N114NJNmu6r3kvQnF+E162d81cjVdzHou540I2Vm9L4gef6EiXegq5t5aq2eq/2aWy
         zIFQ==
X-Google-DKIM-Signature: v=1; a=rsa-sha256; c=relaxed/relaxed;
        d=1e100.net; s=20221208; t=1681446787; x=1684038787;
        h=to:from:subject:message-id:date:mime-version:x-gm-message-state
```

Fig 8. Raw message format that we would have received and need to decode from forwarding emails to S3 using the initial process.

Our Solution: Leveraging Timely's two-way sync

To combat this, we created a Gmail account to investigate Timely's relatively new two-way Google Calendar sync (*Using the Google Calendar Two-way Sync*, 2021). By syncing the business owner's Google Calendar with Timely, and then sharing that calendar with our Boostly Google Calendar, we found that:

1. We could reliably grab all the calendar event information
2. We would not have to parse huge chunks of raw message formatted data to try and extract key appointment information
3. The data received was much cleaner and far more reliable

`

There is still an occasional delay between Timely syncing calendars with Boostly, but the time difference was cut down from (occasionally) more than half an hour to an average of less than 10 minutes.

The difficulty now lay in trying to push Google Calendar event data to AWS.

After some experimenting, we found that the Google Calendar API's Push Notifications ("Push Notifications," n.d.) seemed to only work reliably within the Google ecosystem. Not using this meant that we would have to frequently query Google Calendar, which seemed extremely inefficient. We explored using Amazon AppFlow (*Google Calendar connector for Amazon AppFlow - Amazon AppFlow*, n.d.) to facilitate this integration but found that their features were limited.

After conducting a secondary PoC and largely thanks to the AWS S3 Client Library created by Erik Schultink (*Apps Script – Google Apps Script*, n.d.), we were able to:

- Confirm that Google App Scripts was able to receive push notifications/triggers from changes made to a Google Calendar
- Push each event's information as an object to an AWS S3 bucket, which would then trigger a Lambda function
- Implement incremental sync of Google Calendar – The function would send a list request with a sync token, which would help retrieve only the resources that had been changed since the last request (as opposed to doing a full sync for each Google Calendar query)

Fig 9. Data received from Google Calendar's API comes in nicely formatted JSON

`

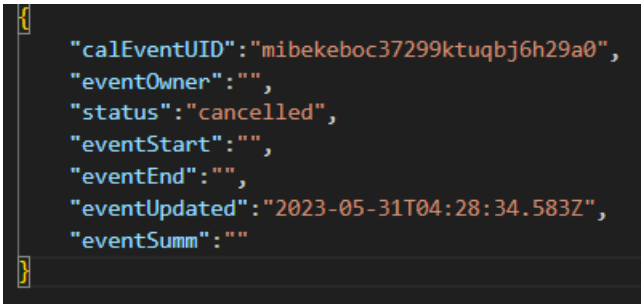


Fig 10. Vital information is condensed using Google Apps Script and sent to our S3 bucket

The weak points in this method are that:

- We are almost completely reliant on the two-way sync to trigger Boostly's serverless functions, and do not (yet) have a way to run checks to ensure that the sync is working
- This still adds an extra processing step (and thereby point of failure) to the application

However, we deemed that this process would be far superior and reliable when compared with our initial process flow (forwarding and parsing emails) and chose to proceed down this path.

Problem II: Customer Setup/Onboarding Process

During the testing of our Boostly setup process, we realised that our Activate Addon[3] process was rather convoluted and had too many technical steps that might introduce confusion along the way.

Our Solution: Compromise on a manual setup process

We finally decided that, for the MVP, we would provide a "consulting setup service" where "A friendly Boostly team member would set your account up for you". While this isn't ideal, we felt that this would mean that we could focus on the meat of the project – Helping business owners provide automated notifications to their clients.

Problem III: Sending Email Notifications Indiscriminately

While testing the initial process with our live tester, we realised that either:

- The User might be the one making changes to their own calendar
- The User might want to use that slot as a break for i.e., catching up on admin work.

Our initial plan was to trigger a notification to the User's Clients each time a slot opened up, and we realised this would not be viable in real life.

[3] This has been extracted from the initial SRS documentation and included in the Appendix

`

Our Solution: Adding an extra step in the notification process as a safety feature

We added an extra step to the activity flow as a safety feature – If a slot opened up, the User would first receive an email asking if they want to notify their clients. They could then have the flexibility to adjust the appointment time, select their clients, and send the alert.

Putting in an extra step here might take up a little time but offers the User a lot more control. After some testing, we found this a key functionality as it prevents the accidental sending out of notifications to clients. This change is reflected in green in our SRS functional requirements.

Changing of terminology and model classes:

This is my first time using a database and many of the methods and principles of database design and querying were completely foreign to me. While designing the database, I asked for feedback from a friend who works with databases and realised that the design in our initial proposal was too naïve. Some of the key changes made include:

- Changing “Business owner” to just be a “User” tied to a “Company”
    - I initially designed the database so that the User would be a Business Owner, or someone who oversees many “staff accounts”, with the ability to manage all the staff accounts on their behalf. This started to become quite confusing, and after visualising the relationships between the Business Owner, Staff, and Clients, I did an overhaul of that part of the database
    - More detail has been provided in the Database Relationships segment of the Technical Implementation Details

    Note: The Company feature isn’t properly developed but allows for future expansion.
- New model classes – AvailTimes, TempWaitAlert, SentWaitAlert, and MsgTmpl
    - While trying to figure out how to configure the client’s preferences to allow for different units of time, I realised that attaching the client’s preferred time directly to the Client’s preferences would soon become quite messy. Instead, AvailTimes was created to form that link more efficiently while providing flexibility for future change.
    - TempWaitAlert and SentWaitAlert were created to help store alert data in the most efficient way possible. A cron job can be run regularly on these tables to remove outdated and unnecessary records.
    - The MsgTmpl model class was created to try and capture notification message data in an efficient method (no repeating entries of the same data) that allows for flexibility (future ability for Users to customise their own alert message).

    The changes in model classes have been documented in more detail in the Technical Implementation Details portion of this report, under Planning for Future Flexibility.

Each of these changes have also been documented in our modified SRS documentation, attached to this report, with changes from the original SRS document noted in green.

`

Removing the functionality for clients to add themselves to the waitlist

The changes above are quite substantial but key to the smooth operation of Boostly. Given the limited resources of the group, we deemed it impossible to produce all of the features originally proposed without compromising on quality.

The functionality for clients to add themselves to the waitlist is important but not vital if we still provided business owners with the ability to add clients on their behalf. We decided that the trade-off in reducing functionality was worth building a more robust application and process. These changes have been reflected in purple in our functional requirement specifications table, and in green in our SRS document attached to this document.

These limitations were discussed with our tutor and this was accepted as a reasonable compromise.

Justification for using the US East Region for AWS:

At this time of writing, only limited locations can receive email via AWS SES. For example, if you use Amazon SES in the US West (Oregon) Region, then any Amazon SNS topics, AWS KMS keys, and Lambda functions that you use also have to be in the US West (Oregon) Region

The following table lists the email receiving endpoints for all of the AWS Regions where Amazon SES supports email receiving:

Region Name	Email Receiving Endpoint
US East (N. Virginia)	inbound-smtp.us-east-1.amazonaws.com
US West (Oregon)	inbound-smtp.us-west-2.amazonaws.com
Europe (Ireland)	inbound-smtp.eu-west-1.amazonaws.com

Fig 11. Amazon SES only receives email in limited locations, ruling out use of the Sydney region (the ideal region as it's currently closest available to New Zealand)

Functional requirement changes at a glance

This table has been extracted from our initial proposal document to provide a quick summary of how Boostly has evolved since our proposal, and what functions we've managed to develop.

FR#	Functional Requirements	Want/Must	Achieved?
FR001	~~Business owners~~ Users can view/add/remove clients to/from their waitlist	Must	Yes
~~FR002~~	~~Business owners can indicate a cut-off notification time/policy~~	~~Must~~	No

`

	Business owners can customise their notification message to their clients	Want	No
~~FR003~~	~~Business owners get a waitlist registration URL that they can share with their clients~~	~~Must~~	No
	Each team member in the business has their own waitlist	Want	No
~~FR007~~ FR002	The application automatically notifies Users when a slot becomes available	Must	Yes
FR003	Users are linked to each company via their clients	Want	Yes
~~FR004~~	~~Clients can add themselves to a business' waitlist~~	~~Must~~	No
FR004	~~Clients can indicate what types of slot cancellations they would like to be notified of~~ Users can set client notification preferences	Want	Yes
	Clients can indicate if they only want to receive notifications for a specific team member in the business	Want	No
~~FR005~~	Clients indicate that they want to receive notifications and agree to T&Cs	Must	No
~~FR006~~	Clients can remove themselves from notifications	Must	No
FR005	The User can view and adjust the notification's details before sending	Want	Yes
FR006	The User will only be shown clients whose preferences match the notifications' details	Want	Yes
FR007	The User can select which clients to notify	Must	Yes
FR008	The User can create a new waitlist notification from scratch	Want	Yes
FR009	The Client receives an email with a link to the User's Timely booking URL	Must	Yes
FR010	The application (Boostly) is fully synced with the scheduling application's calendar	Want	Yes
	Notifications that are not sent successfully should be queued for resending	Want	No

`

Technical Implementation Details

This section highlights some of Boostly's more interesting technical implementation pieces.

Serverless Applications

Syncing Timely's appointments with Boostly

Our goal here was to ensure that Boostly would be up to date with each appointment made, so that it could process the appointment and know whether to alert the business owner.

The Timely-to-Boostly two-way calendar sync is managed by Timely (*Using the Google Calendar Two-way Sync*, 2021b). We wanted the calendar to trigger our AWS Lambda function each time the calendar was updated. However, trying to implement this link straight to AWS was tricky since Google (naturally) wants us to stay within their cloud services. We managed to overcome this by using Google Apps Script as an intermediary between the two. Google Apps Script comes with an inbuilt Google Calendar push notification to trigger a script.

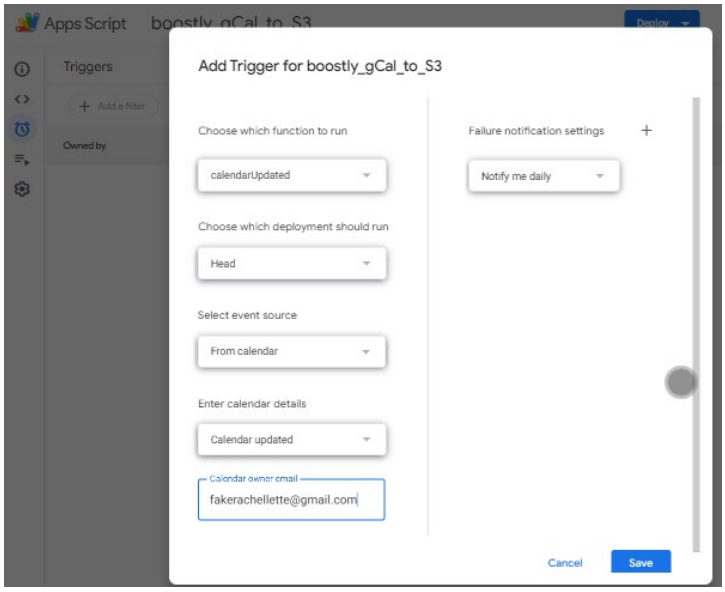


Fig 12. There is inbuilt functionality to trigger a Google Apps Script on Calendar change

The difficult part was being able to push the event data to AWS. Fortunately, we found a "Google Apps Script to S3" binding created by Erik Schultink (Eschultink, n.d.), which made this process possible. This binding was last updated 4 years ago, which meant that it didn't work out-of-the-box as AWS has changed the way in which they form a connection, so the entire package was copied over instead of being added as a library.

We then used documentation from the Google Calendar API ("Push Notifications," n.d.-b) to implement their incremental sync, which uses a token that allows us to only retrieve the latest calendar changes (as opposed to a full sync).

`

```
for (const event of events.items) {
  // Defining and authenticating the S3 bucket method = s3.putObject(bucket, objectName, object, options)
  Logger.log('Event status is :' + event.status);
  // let s3ana = new S3('AKIA6E24ECZ3SMHBMECE','JOa+ue/towXfbA45FJsQ6ceIGDaLHaC6VVsEvpG1','.amazonaws.com');    // Ana's key
  let s3chris = new S3('AKIAYUUMGNZMDJUJKE4C','5Xp1q6pgc+STB1BI0xMK1Ex0P5f9oq50YXwYH7O/','.amazonaws.com');    // Chris' key
  let s3bucketChris = 'appscript-bucket-s48mairqe6op31stxhqdpxg1e43aouse1a-s3alias';
  // let s3bucketAna = 'calendar-events-test';
  let s3objectName = event.id + ".json";

  if (event.status === 'cancelled') {
    var dateNow = new Date();
    Logger.log('The dateNow is ' + String(dateNow) + ' with datatype ' + String(typeof(dateNow)));

    let s3object = {
      cal_event_uid : event.id,
      event_owner : "",
      status : event.status,
      event_start : "",
      event_end : "",
      event_updated : dateNow,
      event_summ : "",
    };
    let s3options = {logRequests:true};
    Logger.log('Event id %s was cancelled: ' +  event.id );
    // s3ana.putObject(s3bucketAna, s3objectName, s3object, s3options);
    s3chris.putObject(s3bucketChris, s3objectName, s3object, s3options);
    Logger.log('Event pushed to s3 bucket at ' + event.updated);
    continue;
  }
  if (event.start.date) {                // The event only has a start date instead of start datetime, therefore it lasts all day
```

Fig 13. Key event data is extracted and formatted into a concise JSON object, which is then put into our AWS S3 bucket.

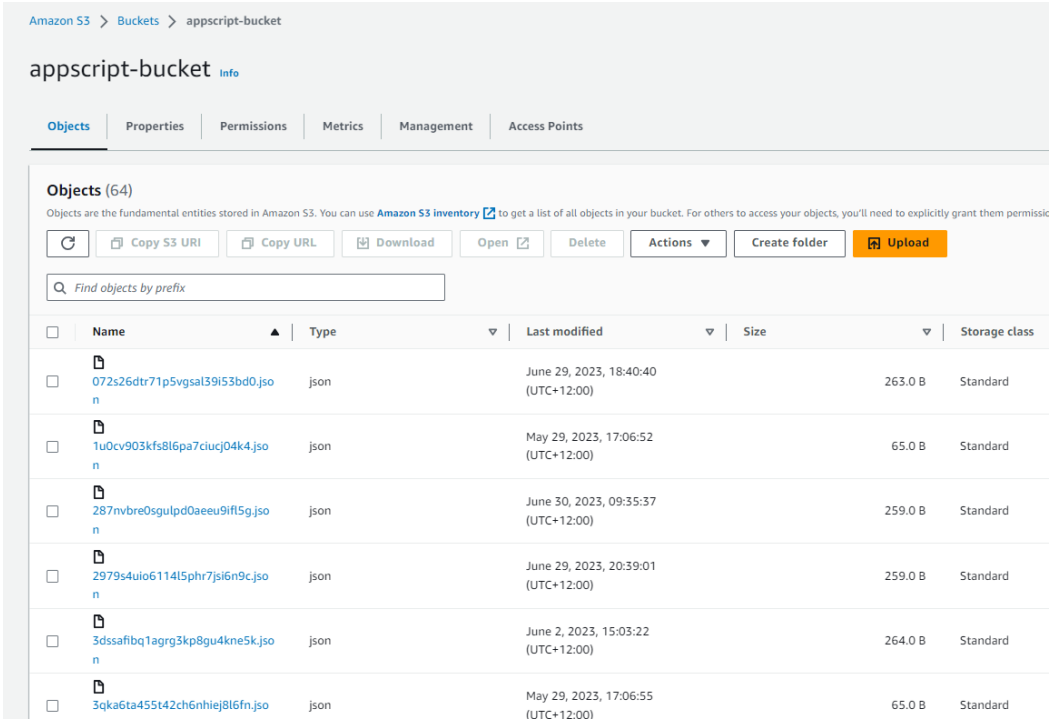


Fig 14. Objects arriving into our S3 bucket are configured so their event UID is their filename

AWS makes it simple to configure the S3 bucket to a Lambda script each time an object is placed in it. The only thing we need to watch out for is to dedicate this bucket solely for the purposes of getting concise event data, otherwise the Lambda script would trigger and malfunction.

`

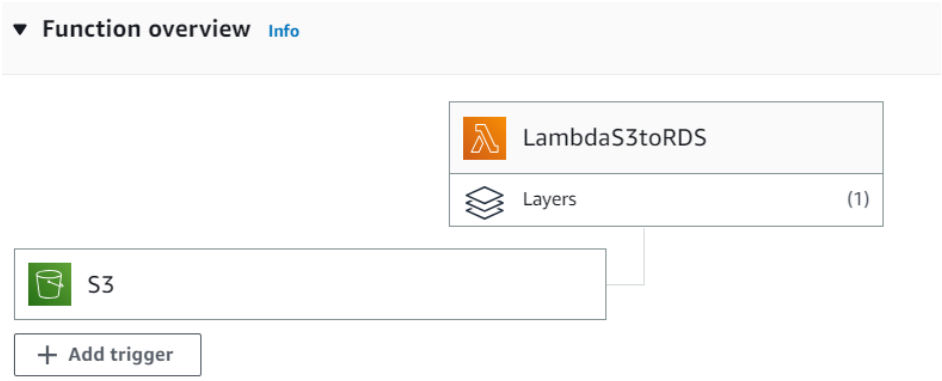


Fig 15. Inbuilt ability to trigger a lambda function when objects arrive into our S3 bucket

Putting data into RDS was more difficult. A large part of it involved importing the psycopg2 library as well as the AWS distribution to allow us to talk to RDS:

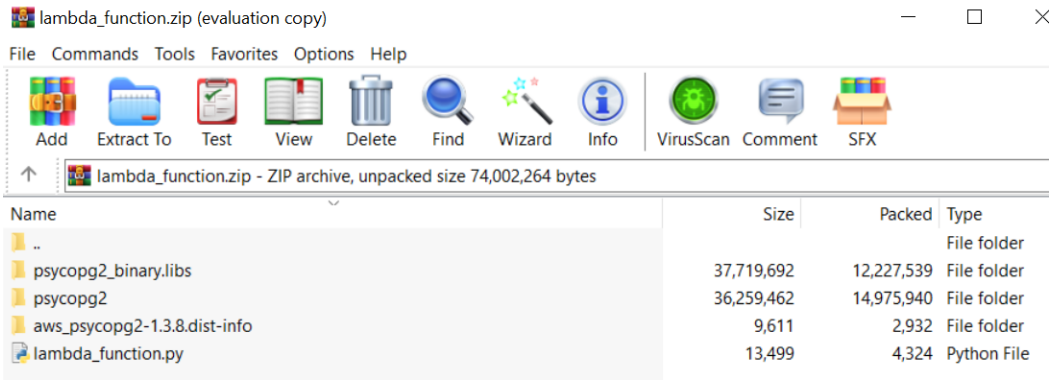


Fig 16. Importing the psycopg2 library into our lambda package

```
##############################################################
# SECTION 3: CONNECT TO RDS POSTGRESQL DB

def connectToRDS(tablename):
    try:
        conn = psycopg2.connect(
            database=db_name,
            user=user_name,
            password=password,
            host=rds_host,
            port=port
            )
        print("DB connection established")

    except psycopg2.OperationalError as e:
        # pass exception to function
        print(e)
        # set the connection to 'None' in case of error
        conn = None
    return conn
    # DON'T FORGET TO CLOSE THE CONNECTION AFTER!
```

Fig 17. Using psycopg2 was key to being able to connect with our PostgreSQL RDS database

`

```
########################################################################
# SECTION 2: CLASSES
# -- DESERIALISING JSON STRING TO AN OBJECT
# This class turns json keys into objects, but is specific to the Event table
class Event:
    def __init__(self, json_def):
        s = json.loads(json_def)
        self.cal_event_uid = None if 'cal_event_uid' not in s else s['cal_event_uid']
        self.event_owner = None if 'event_owner' not in s else s['event_owner']
        self.status = None if 'status' not in s else s['status']
        self.event_start = None if 'event_start' not in s else s['event_start']
        self.event_end = None if 'event_end' not in s else s['event_end']
        self.event_updated = None if 'event_updated' not in s else s['event_updated']
        self.event_summ = None if 'event_summ' not in s else s['event_summ']
```

Fig 18. Deserialising JSON string (created in Apps Script) to an object for easy extraction of data

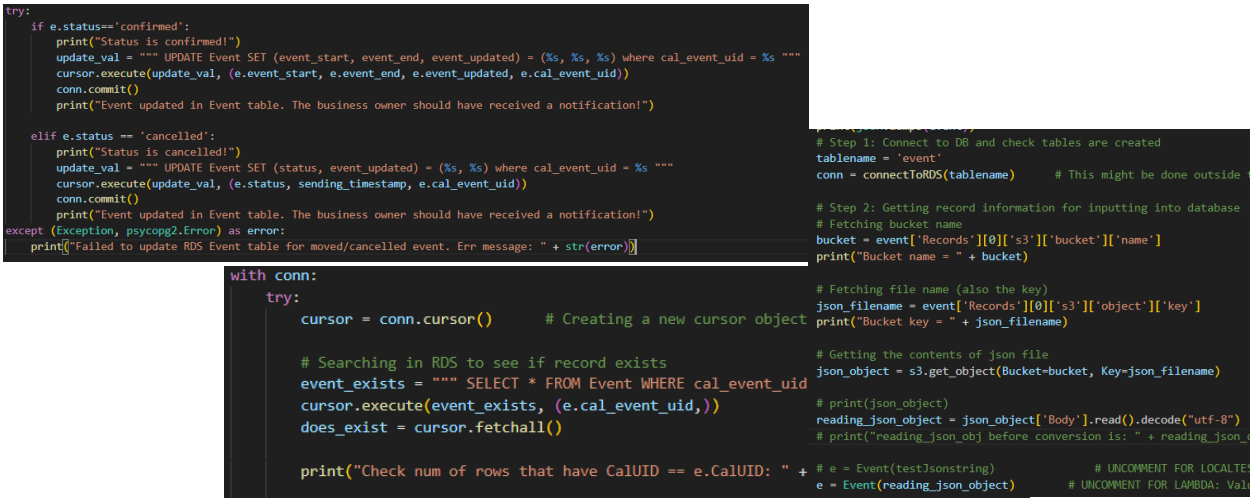


Fig 19. Snippets of code that shows the connection between reading data from Apps Script and pushing it to RDS depending on the logic involved

This made the script very heavy and required an import each time we wanted to test the code. I was unable to successfully import psycopg2 as a layer and/or figure out how to import only the key portions of psycopg2 due to time constraints – This is something that I'd like to be able to do given more time.

`

The logic of how to capture the right event data for a notification (TempWaitAlert table) vs for storing into RDS' Event table was also quite tricky.

```
if cursor.rowcount > 0:
    print("Event exists! The client has either cancelled or moved the appointment slot. \nWe need to notif
          "\nCollecting details of the outdated appointment...")
    # Update Event table with the event's start and end times, then trigger the alertBusinessOwners lambda

    # Pulling out the event's details. We do this because, for cancelled events, Google Calendar does not
    # send over the details, just a "status=cancelled" line. For events that have been moved,
    # Google calendar would be sending over the new appointment info not old. We need the old info to
    # know what timeslot has opened up
    print("Print each row from RDS that fulfils calUID == calUID conditions")
    cal_id = ""
    free_owner = ""
    free_start = ""
    free_end = ""
    for row in does_exist:
        cal_id = str(row[1])
        free_owner = row[2]
        free_start = row[4]
        free_end = row[5]
        print("calId = " + cal_id + "<-- This ID should NOT be NULL!" )
        print("eventOwner = ", free_owner)
    print("The calEventUID in the row retrieved is: " + cal_id)
    print("Free slot start time is " + str(free_start) + " and the datatype is " + str(type(free_start)))
```

Fig 20. Google Calender event data contents change if appointments are cancelled, requiring additional logic to handle this

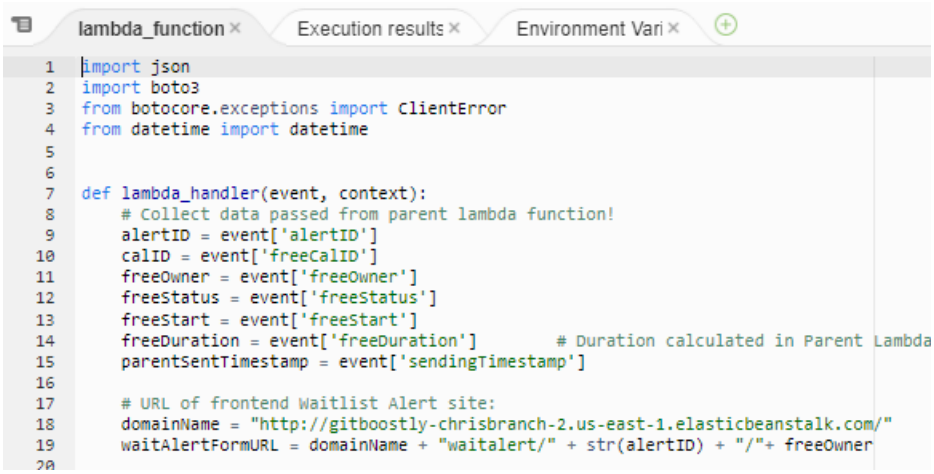
To send the email notification, we decided to trigger a secondary lambda function dedicated to that cause. Triggering a secondary lambda function from our primary lambda function was quite simple:

```
# Trigger child function and pass payload
try:
    response = lam.invoke(
        FunctionName = 'arn:aws:lambda:us-east-1:594073251416:functi
        InvocationType = 'RequestResponse',
        Payload = json.dumps(inputParams)
    )
    responseFromChild = json.load(response['Payload'])
    print("The response from the alertBusinessOwners lambda functior
except (Exception, psycopg2.Error) as error:
    print("Failed to trigger 'alertBusinessOwners' child lambda func
```

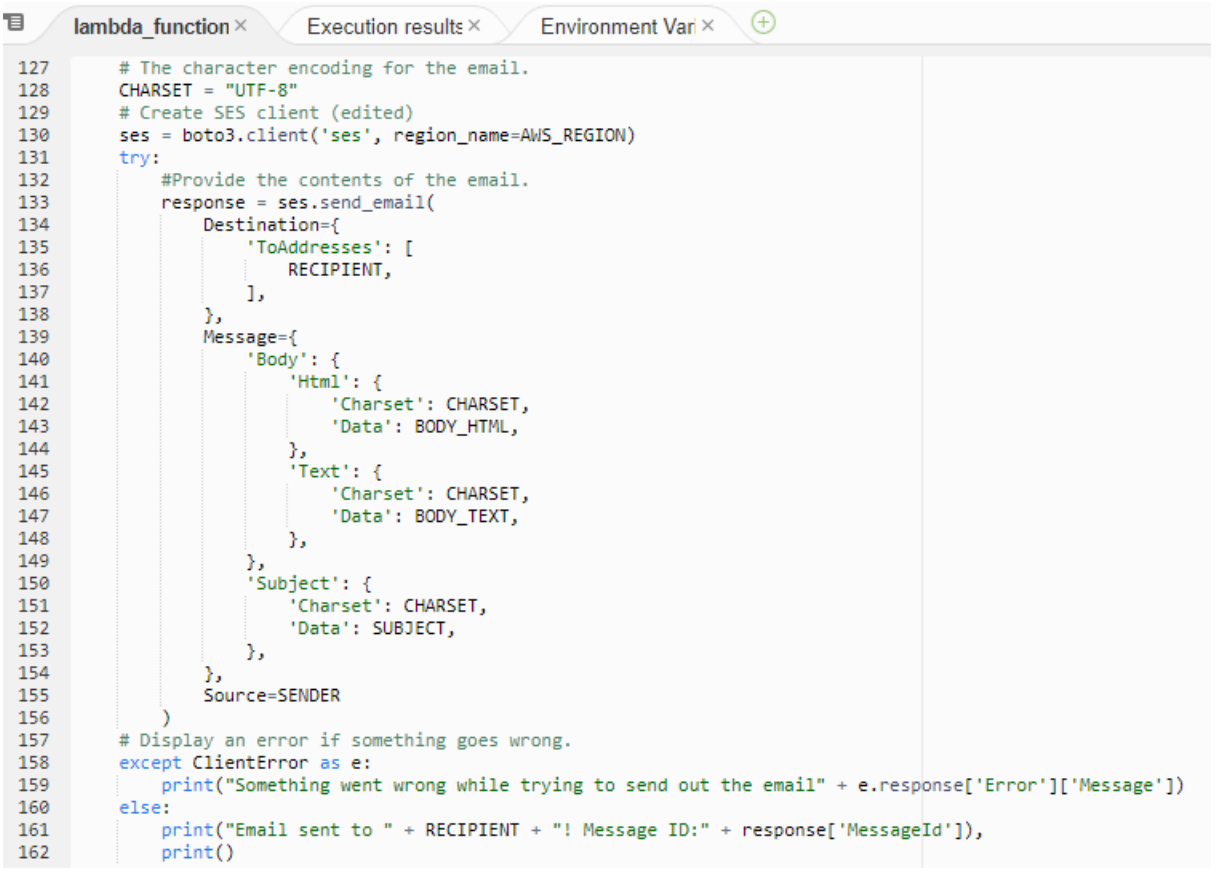
Fig 21. Triggering a secondary lambda function and passing over a payload, then getting a response to determine whether it was successful

Doing so helped to break up our code into smaller packages, allowing the actual email notification to be conducted by our child lambda function.

`




```python
import json
import boto3
from botocore.exceptions import ClientError
from datetime import datetime


def lambda_handler(event, context):
    # Collect data passed from parent lambda function!
    alertID = event['alertID']
    calID = event['freeCalID']
    freeOwner = event['freeOwner']
    freeStatus = event['freeStatus']
    freeStart = event['freeStart']
    freeDuration = event['freeDuration']        # Duration calculated in Parent Lambda
    parentSentTimestamp = event['sendingTimestamp']

    # URL of frontend Waitlist Alert site:
    domainName = "http://gitboostly-chrisbranch-2.us-east-1.elasticbeanstalk.com/"
    waitAlertFormURL = domainName + "waitalert/" + str(alertID) + "/"+ freeOwner
```




```python
    # The character encoding for the email.
    CHARSET = "UTF-8"
    # Create SES client (edited)
    ses = boto3.client('ses', region_name=AWS_REGION)
    try:
        #Provide the contents of the email.
        response = ses.send_email(
            Destination={
                'ToAddresses': [
                    RECIPIENT,
                ],
            },
            Message={
                'Body': {
                    'Html': {
                        'Charset': CHARSET,
                        'Data': BODY_HTML,
                    },
                    'Text': {
                        'Charset': CHARSET,
                        'Data': BODY_TEXT,
                    },
                },
                'Subject': {
                    'Charset': CHARSET,
                    'Data': SUBJECT,
                },
            },
            Source=SENDER
        )
    # Display an error if something goes wrong.
    except ClientError as e:
        print("Something went wrong while trying to send out the email" + e.response['Error']['Message'])
    else:
        print("Email sent to " + RECIPIENT + "! Message ID:" + response['MessageId']),
        print()
```

Fig 22 and 23. The child lambda function is small, comprising of sending a largely static message body inserting dynamic variables

`

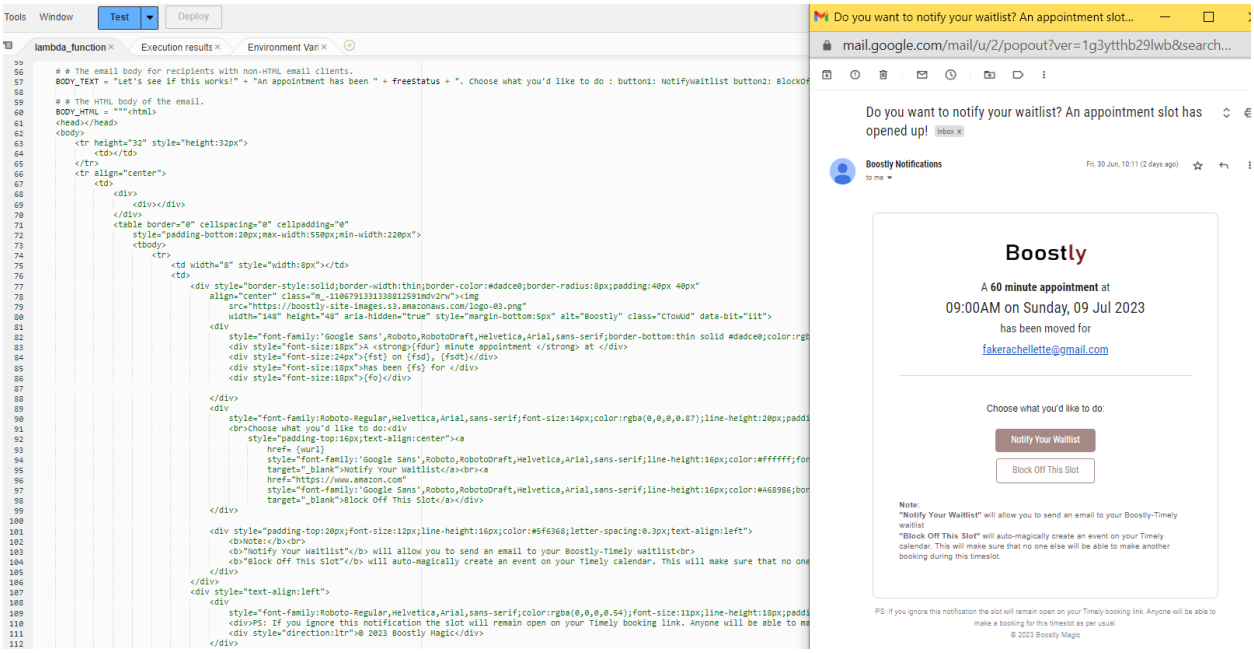


Fig 24. The email notification received as a result of the child lambda function - The trickiest part here was actually trying to figure out how to pass variables within an inline HTML stylesheet

Main Application

The skeleton of our main application (registration, login, account) was largely modeled against our CS204B project.

This included functionality such as hashing passwords, reducing profile image file size, and renaming the image to a unique/random 8 byte hex so that there were no conflicting file names in the database.

This is functionality that supports Boostly, but has not been included in this report so that we can focus on Boostly's key features.

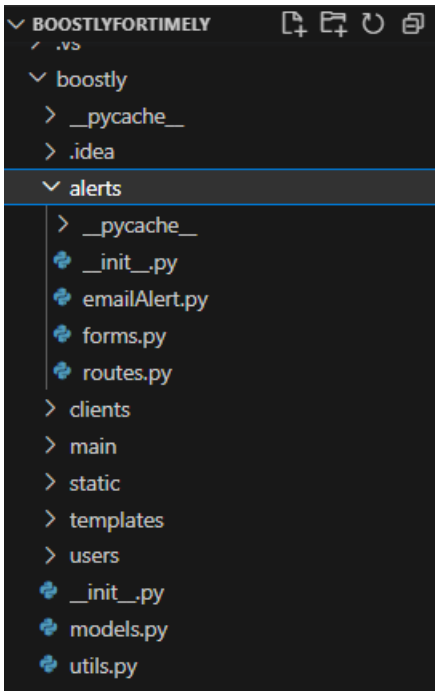


Fig 25. Using what we learnt while doing our CS204B Dopeticks project, we have split Boostly up into smaller packages for better organisation and reduction of dependencies.

`

Database relationships

I haven't worked with a database prior to CS204B, and my understanding of database relationships was limited to a purely theoretical perspective. Actually creating those relationships was an interesting challenge, and one that involved refactoring multiple times. A lot of the relationships are probably not necessary for the MVP, but putting them in means that future development work would be far easier, and on a much more flexible system. I spent most of the time building the main application on this, because I felt that it was important to create a good foundation to build on.

The User used to have a definition of "a business owner with multiple staff within that account", with Users having a one-to-many relationship with Staff, who had a one-to-many relationship with their Clients.

After (crudely) drawing out the relationships, though, I realised that there should be a many-to-many relationship between clients and companies (since each company can cater to many clients, while each client can also belong to multiple companies), and Users are linked to the Clients via their Companies.

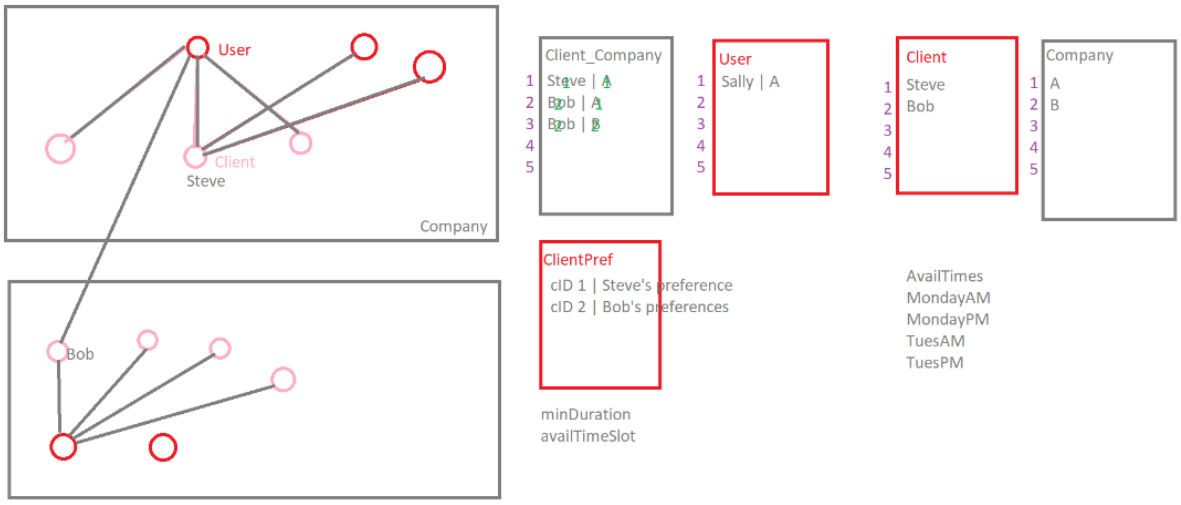


Fig 26. Crude graphical mapping out of relationship between the User, Client, Company, and client preferences

`

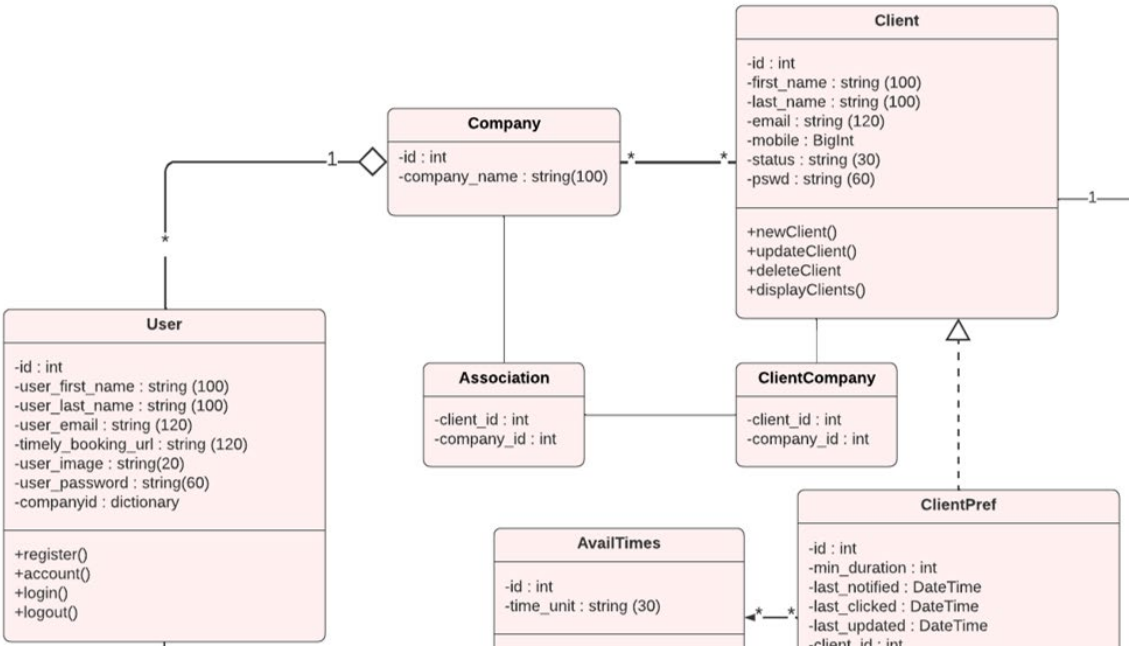


Fig 27. Snippet of Boostly's class diagram, taken from our SRS documentation

This resulted in a complete overhaul of the class diagram, and I'm hoping that this is a more logical way to build User access management. If you're reading this and have ideas on how I can improve, please do reach out[4]!

```python
# Associating ids from the Client with the Company
ClientCompany = db.Table('client_company',
    db.Column('client_id', db.Integer, db.ForeignKey('client.id')),
    db.Column('company_id', db.Integer, db.ForeignKey('company.id')),
    )

# The company model isn't really used in the front end for the MVP but is created to form the relationship between the User and Clients.
# It is the equivalent of your "Company Admin Account"
class Company(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    company_name = db.Column(db.String(100), nullable=False)
    # Association table allows us to find out which clients belong to this company using Company.clientsof <<backref>>
    userstaff = db.relationship('User', backref='coyowner', lazy=True, uselist=False)  # 1 Company --> * Users (staff)


class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key = True)
    user_first_name = db.Column(db.String(100), nullable=False)
    user_last_name = db.Column(db.String(100), nullable=False)
    user_email = db.Column(db.String(120), unique=True, nullable=False)
    timely_booking_url = db.Column(db.String(120), nullable=False)
    # A hash will be generated for user_image to be 20 Char, and userPass as 60 Char
    user_image = db.Column(db.String(20), nullable=False, default='default.jpg')
    user_password = db.Column(db.String(60), nullable=False)
    # Each User can only belong to a Company. So 1 Company --> * Users
    companyid = db.Column(db.Integer, db.ForeignKey('company.id'), nullable=False)   # Attaches user to company.id
    # Linking User with their alerts (1 User --> * Alerts)
    waitalerts = db.relationship('TempWaitAlert', backref='alertbelongsto', lazy=True, uselist=False)

    def __repr__(self):
        return f"User('{self.user_email}', '{self.user_password}', '{self.user_image}')"


class Client(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    first_name = db.Column(db.String(100), nullable=False)
    last_name = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(120), nullable=False)
    mobile = db.Column(db.BigInteger)
    status = db.Column(db.String(30))
    pswd = db.Column(db.String(60))                                    # For when we want to give clients a way to make their own changes
    companies = db.relationship('Company', secondary=ClientCompany, backref='clientsof')        # Refer to client_company table for client/company relsp
    clientprefs = db.relationship('ClientPref', backref='client', uselist=False)          # Forming a 1 Client --> * ClientPref relationship
    sentalerts = db.relationship('SentWaitAlert', backref='alerted')
```

Fig 28. Implementation of User having an indirect relationship with the Client, linked by the Company

[4] The easiest way to reach me would be via LinkedIn - https://www.linkedin.com/in/chrischonghuihui/

`

This made for very interesting database queries further down the line, since I needed to extract the clients only belonging to that company based on the current_user's id (or company id), and is a part that I need to keep practicing.

```
# Filter preftimes table to get all clientpref_ids of clients matching those days
curr_companyid = current_user.companyid
print("The current company is " + str(curr_companyid))
# Querying the database to get the intersection between the Company and the Client using the current_user's company id, then adding status filtering on top of that
clients = Client.query.join(ClientCompany).join(Company).filter(Company.id==curr_companyid, Client.status == 'active')
```

Fig 29. Learning how to implement SQL joins using SQLAlchemy queries

Planning for future flexibility

Another decision that I made during the architecting of the database was to extract "AvailTimes" as a separate table. AvailTimes refers to the timeframes in which the Client would like to be notified. For example, if I have taekwondo training every Monday, Wednesday, and Friday, I might not want to have any massages on those days, and would not want to receive alerts for those periods.

Determining the flexibility in time units for the client's preferences is difficult – Ideally, a client should be able to block off specific chunks of time in their preferences. However, the front-end development required for this is quite large, and I wanted to keep it simple for the MVP.

The AvailTimes table makes it possible for us to later (quite effortlessly) change the time units (i.e. from being able to select "Notify me only for slots that open up on a Tuesday and Thursday" to "Block off 5.30PM to 9PM on Mondays, Wednesdays and Fridays"). It also standardises storage of client timing preferences.

While doing this, I also found out (quite happily) that pairing a time unit with an integer not only saves space and computing power, but also prevents storing strings of time variations (i.e. monday, mon, Mon).

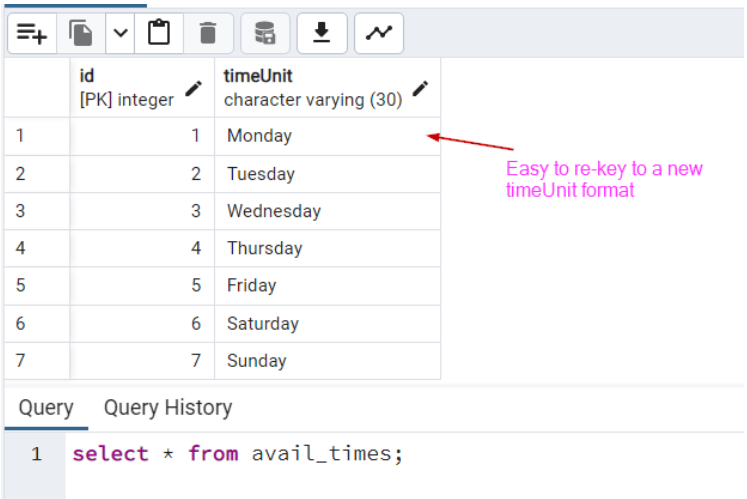


Fig 30. AvailTimes table can be easily changed to represent a different time unit, i.e. by hour

`

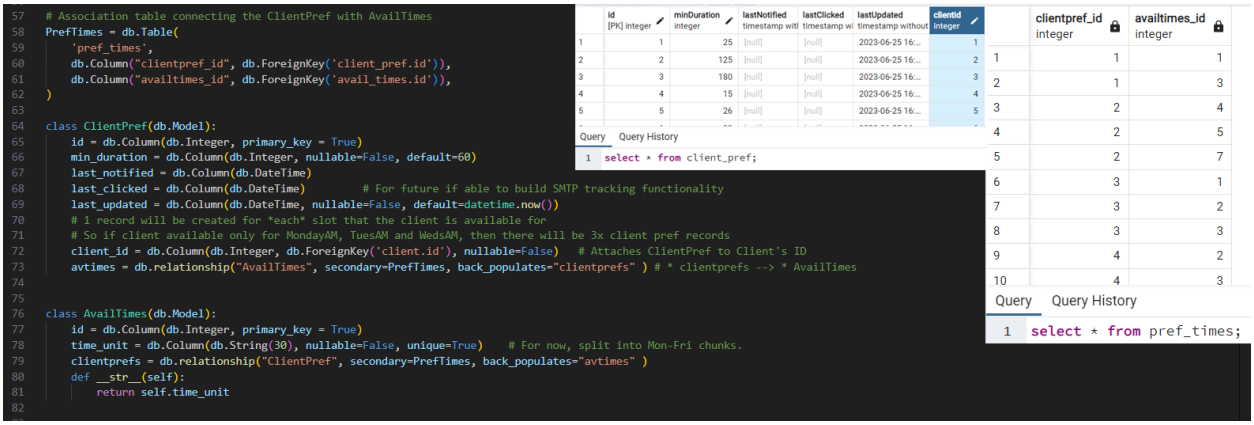


Fig 31. Querying a database of integers takes significantly less computational time, particularly for tables that potentially can have huge amounts of records.

Flexibility and Recordability for Alerts

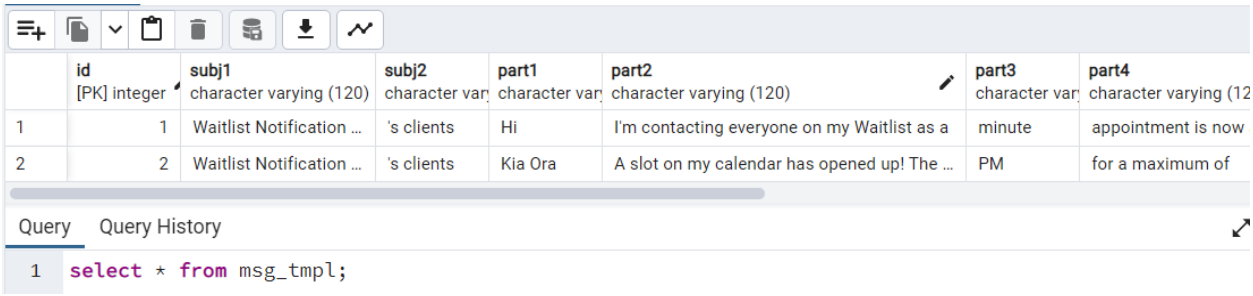
I spent quite a bit of time trying to design a system that would allow for future customisation, while making the storing and sending of alerts more efficient and planning for a more reliable system.

```python
# The Message template table store default "canned messages" that are used to alert clients,
# broken down into parts to allow inserting i.e. date/time/username data in between
class MsgTmpl(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    subj1 = db.Column(db.String(120))
    subj2 = db.Column(db.String(120))
    part1 = db.Column(db.String(120))
    part2 = db.Column(db.String(120))
    part3 = db.Column(db.String(120))
    part4 = db.Column(db.String(120))
    part5 = db.Column(db.String(120))
    part6 = db.Column(db.String(120))
    part7 = db.Column(db.String(120))
    part8 = db.Column(db.String(120))
    waitalerts = db.relationship('TempWaitAlert', backref='msgtemplate', lazy=True, uselist=False)

# This is a table that stores temporary alert data. A cron job can be run daily to remove data that is no longer useful/necessary
# There are 2 types of alerts --> One registers that a slot is available/created,
# the other registers that an alert has been sent (and who it's been sent to)
class TempWaitAlert(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    # Alert data
    slot_start_date_time = db.Column(db.DateTime, nullable=False)
    slot_length = db.Column(db.Integer, nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))        # Attached when business owner submits waitAlert form part 1
    msg_tmpl = db.Column(db.Integer, db.ForeignKey('msg_tmpl.id'))    # Attached when business owner submits waitAlert form part 1
    # For cron job to look at to know whether to clear out or not
    last_updated = db.Column(db.DateTime, nullable=False, default=datetime.now())
    status = db.Column(db.String(30))

    def __repr__(self):
        return f"TempWaitAlert('{self.id}', '{self.slot_start_date_time}', '{self.slot_length}','{self.sendStatus}')"


# There are 2 types of temp alerts --> One registers that a slot available,
# the other registers that an alert has been sent (and who it's been sent to)
class SentWaitAlert(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    # Alert data
    slot_start_date_time = db.Column(db.DateTime, nullable=False)
    slot_length = db.Column(db.Integer, nullable=False)
    sent_user_id = db.Column(db.Integer, db.ForeignKey('user.id'))        # Attached when business owner submits waitAlert form part 1
    msg_tmpl = db.Column(db.Integer, db.ForeignKey('msg_tmpl.id'))    # Attached when business owner submits waitAlert form part 1

    # Send data
    client_id = db.Column(db.Integer, db.ForeignKey('client.id'))
    send_alert_id = db.Column(db.Integer)  # This will be null for the parent Alert when sent. If populated, the id should match with
    send_flag = db.Column(db.Integer)
    # For cron job to look at to know whether to clear out or not
    last_updated = db.Column(db.DateTime, nullable=False, default=datetime.now())
    status = db.Column(db.String(30))
```

Fig 32. Splitting the message into parts so that Users can have custom messages in the future.

`

Having a custom notification message is a very common request/feature for notification applications. I wanted to find a way to allow users to customise their messages without having to store huge chunks of repetitive data. Because the variables in the message are fixed (client name, date, day, time, etc) it means that we can define the maximum number of parts that the message can be broken down into. By creating a separate table to store these pieces, we can assign each template to an id and attach it to the alert id.



	id [PK] integer	subj1 character varying (120)	subj2 character var	part1 character var	part2 character varying (120)	part3 character var	part4 character varying (12
1	1	Waitlist Notification ...	's clients	Hi	I'm contacting everyone on my Waitlist as a	minute	appointment is now
2	2	Waitlist Notification ...	's clients	Kia Ora	A slot on my calendar has opened up! The ...	PM	for a maximum of

Fig 33. Planning for easy changing/customising of notification body by breaking notifications down to segments

The other area that I wanted to tackle was maximising the storage of event data. This is because we will be receiving heaps of event records – We should expect to have to store many appointments for each User! The first thing that I did (in our Google Apps Script) was to cut down the event data that is transferred and stored to the bare minimum required information needed. I then decided to split the alert data into two tables – One for all the *potential* alerts, and one that would track all the alerts that have been sent. There are several reasons for doing this:

- Bearing in mind that most of these appointments will *not* be cancelled, a lot of the event data that we are storing will *never* actually become an actual alert. If we separate out the sent alerts from the potential alerts, we can then run an efficient cron job on our TempWaitAlert table that will purge the dataset from all the irrelevant/outdated events.

- It would be easier for us to track sent messages for alert system reliability, all forms of data analysis, record keeping, and to deduce i.e., if there's an error with the client's email address, or if there was a system failure somewhere and we might need to resend some messages.

Because each SentWaitAlert record corresponds to an alert to a client, it means that a "single" alert notification from the User would generate quite a large number of records. This is another reason for trying to pair message templates against integers.

	id [PK] integer	slot_start_date_time timestamp without tim	slot_length integer	sent_user_id integer	msg_tmpl integer	client_id integer	send_alert_id integer	send_flag integer	last_updated timestamp without time zone	status character varyi
1	1	2023-07-01 13:45:00	60	1	1	2	6	[null]	2023-06-29 07:02:09.552078	sent
2	2	2023-07-01 13:45:00	60	1	1	4	6	[null]	2023-06-29 07:02:09.70993	failed
3	3	2023-07-01 13:45:00	60	1	1	7	6	[null]	2023-06-29 07:02:09.891479	sent
4	4	2023-06-30 09:00:00	60	1	1	5	7	[null]	2023-06-29 07:04:08.614479	failed
5	5	2023-06-30 09:00:00	60	1	1	2	7	[null]	2023-06-29 07:04:08.836031	sent
6	6	2023-06-30 09:00:00	60	1	1	7	7	[null]	2023-06-29 07:04:09.069858	sent
7	7	2023-06-01 11:30:00	180	1	1	2	2	[null]	2023-06-29 08:43:42.173383	sent

Fig 34. Most of the values in the SentWaitAlert table are kept to integers to as to make it easier to search.

`

Changing status sent=1 and failed=0 would make the database even more efficient, but might also be confusing if more codes come in. I had a thought to change this to reflect SMTP codes but decided not to do so for the MVP.

Frontend Implementation

Dynamic Waitlist Message

I am fortunate to have my teammate, Ana, who took charge of most of the design and front-end work.

One of the few things that we did together was the dynamic changing of message preview variables. This functionality was important to us as it helped the User get a clear understanding of what they were sending.

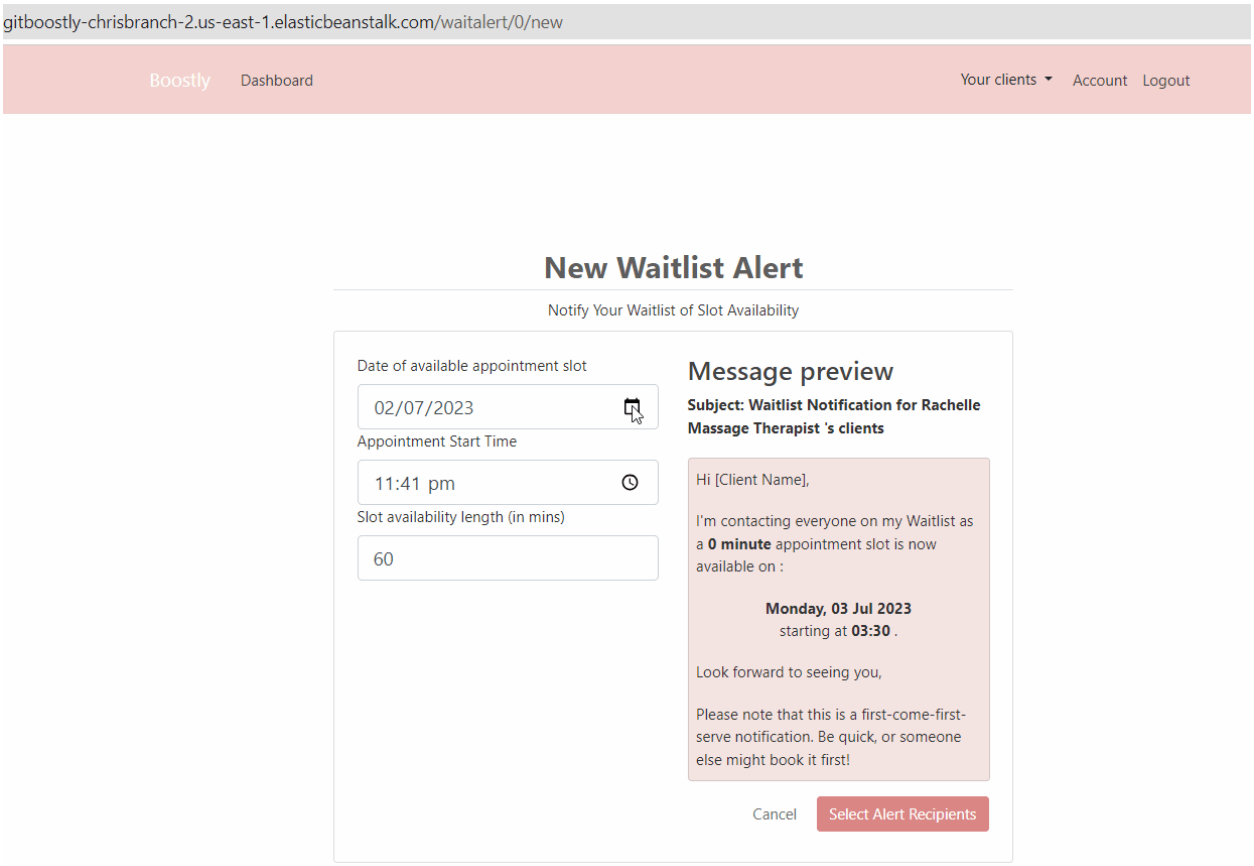


Fig 35. Changing values on the form dynamically changes the values in the message preview. Click on image or go to https://s12.gifyu.com/images/SQJ2w.gif *to view the animated version*

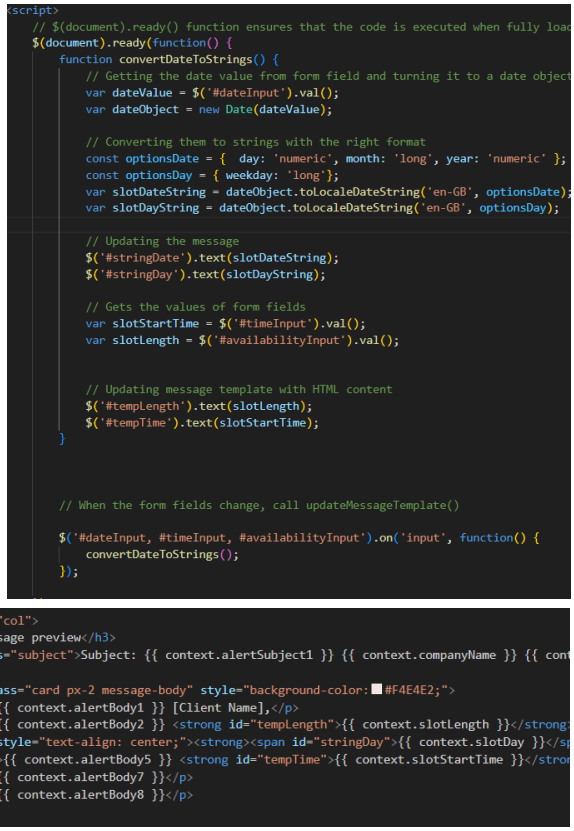
`

Frontend: Experimenting with GridJS

I stumbled across Miguel Grinberg's resources (Miguelgrinberg, n.d.) on building beautiful, interactive tables using grid.js and decided to give it a try.

I learnt a little more about creating jQuery scripts and how to incorporate them with html and Flask, and later how to insert checkboxes into the table, but honestly, Miguel Grinberg did most of the work and I cannot claim credit for this.

I have included this in my report simply so that I can document this for me to look back on in the future. I was surprised to learn that there seems to be no standard way of creating tables. This is something I'd like to explore more in the future, on the side.



```
<script>
    // $(document).ready() function ensures that the code is executed when fully loa
    $(document).ready(function() {
        function convertDateToStrings() {
            // Getting the date value from form field and turning it to a date object
            var dateValue = $('#dateInput').val();
            var dateObject = new Date(dateValue);

            // Converting them to strings with the right format
            const optionsDate = {  day: 'numeric', month: 'long', year: 'numeric' };
            const optionsDay = { weekday: 'long'};
            var slotDateString = dateObject.toLocaleDateString('en-GB', optionsDate);
            var slotDayString = dateObject.toLocaleDateString('en-GB', optionsDay);

            // Updating the message
            $('#stringDate').text(slotDateString);
            $('#stringDay').text(slotDayString);

            // Gets the values of form fields
            var slotStartTime = $('#timeInput').val();
            var slotLength = $('#availabilityInput').val();


            // Updating message template with HTML content
            $('#tempLength').text(slotLength);
            $('#tempTime').text(slotStartTime);
        }



        // When the form fields change, call updateMessageTemplate()

        $('#dateInput, #timeInput, #availabilityInput').on('input', function() {
            convertDateToStrings();
        });
```



```
v class="col">
 <h3>Message preview</h3>
 <p class="subject">Subject: {{ context.alertSubject1 }} {{ context.companyName }} {{ cont

<div class="card px-2 message-body" style="background-color:#F4E4E2;">
    <p>{{ context.alertBody1 }} [Client Name],</p>
    <p>{{ context.alertBody2 }} <strong id="tempLength">{{ context.slotLength }}</strong
    <p style="text-align: center;"><strong><span id="stringDay">{{ context.slotDay }}</sp
    <br>{{ context.alertBody5 }} <strong id="tempTime">{{ context.slotStartTime }}</stro
    <p>{{ context.alertBody7 }}</p>
    <p>{{ context.alertBody8 }}</p>
```

Fig 36. Learning the basics of jQuery to the message preview when the User creates an alert

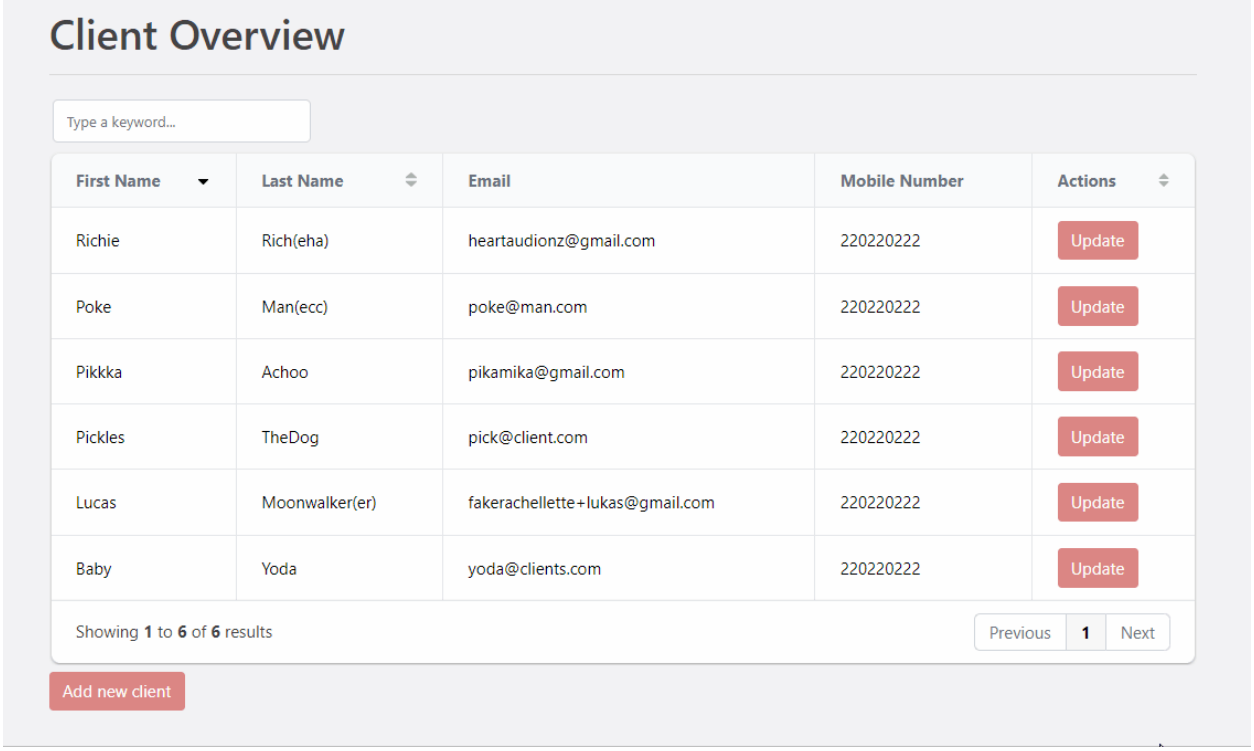


Fig 37. Beautiful Flask tables using Grid.js

Click on image or go to https://s11.gifyu.com/images/SQJSg.gif to view the animated version

`

Teamwork & Communication

Ana and I have done projects together over the last two years and were able to collaborate quite seamlessly together.

One of the main issues was that we both are working part-time. Unfortunately, our schedules conflicted, and we were seldom able to work on the project at the same time.

To combat this, we split the work so that I did most of the backend and Ana did most of the frontend, which helped ensure that we didn't have a lot of merging conflicts.

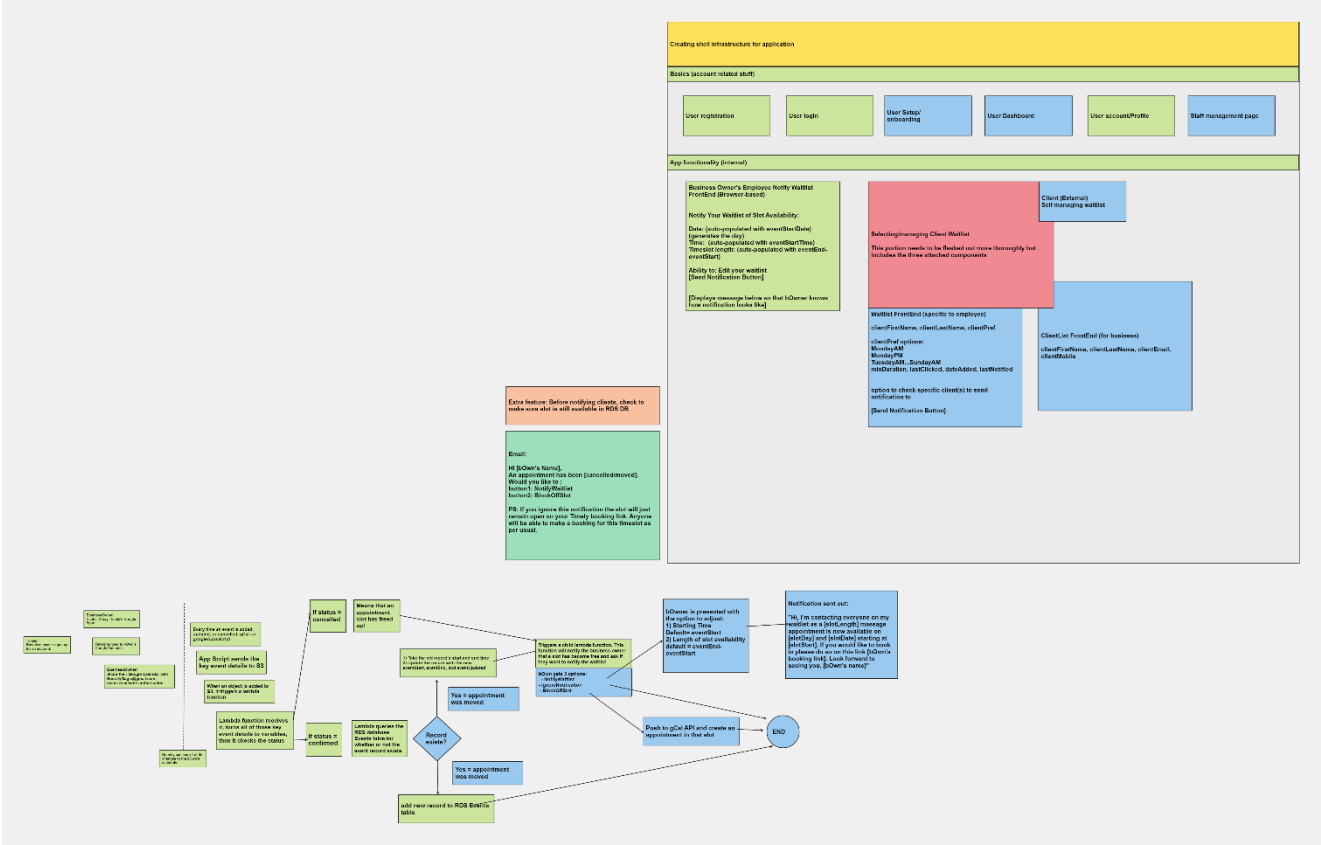


Fig 38. Mapping to communicate what needs to be done for an application with no frontend

However, this project was unusual in that it had a large chunk of work on serverless functions where we couldn't seamlessly push changes to the other person. A lot of work in the beginning was also very uncertain, and there was a period where we felt quite out of sync. We tackled this by using Microsoft Whiteboard to map out and break down all the steps and functions that we needed to accomplish, and used color codes to indicate which areas were in progress, and which were done. The functions were not fixed, and this gave a visual aid as to where we were going and how each function or step connected to other parts of the application.

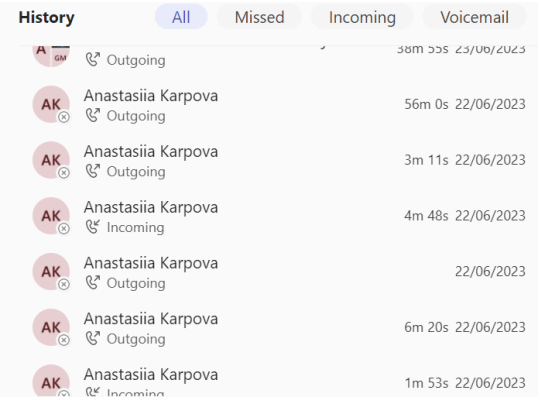
Once we moved to the main Boostly application, we relied a lot on updates in Teams and Git commit messages to communicate where we were at. We would do short bursts of calls to

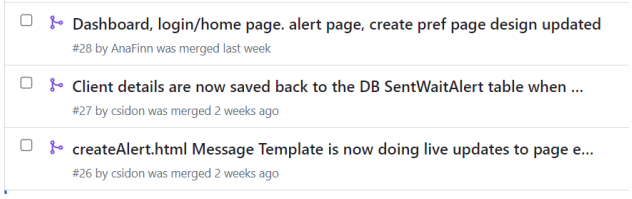
`

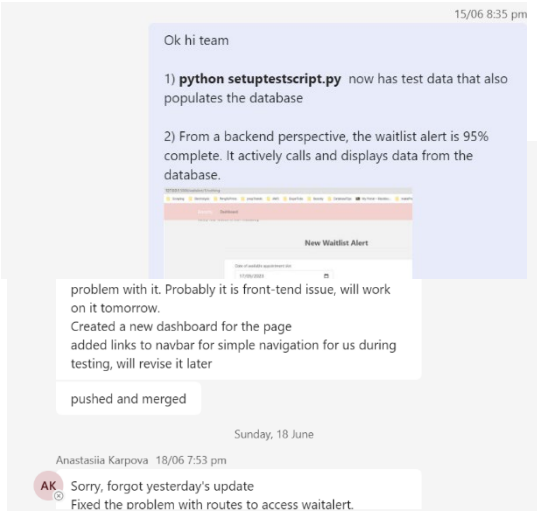
clarify areas and would occasionally have short pair-coding sessions to resolve an issue.

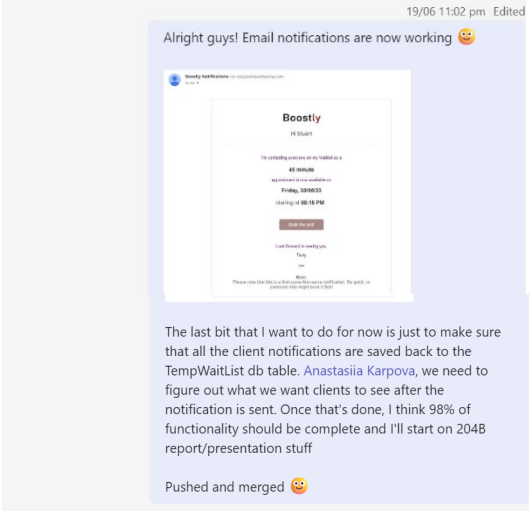
In the middle of the project, Ana had some disruptive personal issues, and I fell quite sick. During both times, we both tried to help each other out and keep going.

All in all, despite the hiccups, given the unique nature of our project, I think we did a pretty good job!









`

Thoughts and learnings

This is a short section for my personal learning – a little note to myself documenting my path as I gain more insight into software development. I would really like to carry on this project, but the reality is that it would be far more useful to i.e. offer to integrate it into Timely's existing application, to maintain and support the product as technology evolves.

Learning how to sync with an application without an API has been an interesting challenge – Doing this has helped me realise how many dependencies there are in even the simplest of applications. There are so many potential points of failure, and there is a lot of room for things to break. I would really like to learn how to implement a database on the cloud securely, and hope that the next time I read this, I'll have managed to achieve this to some level.

This project has taught me heaps, and I'm very fortunate to have had the support of my tutor, Arthur Lewis, to provide us direction and keep us on track. I'm also very grateful for the patient guidance of my friend, Matthew Schmidt, who answered all my inane questions and engaged in all those long discussions and debates on database tables!

PS: Note to self - AWS RDS does not like camelCase. Use python_standard snake_case or you will regret this later!

`

Links

GitHub:
https://github.com/csidon/BoostlyForTimely

Recorded Presentation (from 0:40 to 25:25):
https://myacg.sharepoint.com/:v:/r/sites/BSEPresentation/Shared%20Documents/General/Recordings/View%20Only/Meeting%20in%20_General_-20230630_115129-Meeting%20Recording.mp4?csf=1&web=1&e=d7ccGn

References

Using the Google Calendar two-way sync. (2021, July 13). Timely. https://help.gettimely.com/hc/en-gb/articles/360062947833-Using-the-Google-Calendar-two-way-sync

Push notifications. (n.d.). *Google for Developers*. https://developers.google.com/calendar/api/guides/push

Google Calendar connector for Amazon AppFlow - Amazon AppFlow. (n.d.). https://docs.aws.amazon.com/appflow/latest/userguide/connectors-google-calendar.html

Apps Script – Google Apps Script. (n.d.). https://script.google.com/home/projects/1Qx-smYQLJ2B6ae7Pncbf_8QdFaNm0f-br4pbDg0DXsJ9mZJPdFcIEkw_/edit

Eschultink. (n.d.). *GitHub - eschultink/S3-for-Google-Apps-Script: a binding for using AWS Simple Storage Service (S3) from Google Apps Script*. GitHub. https://github.com/eschultink/S3-for-Google-Apps-Script

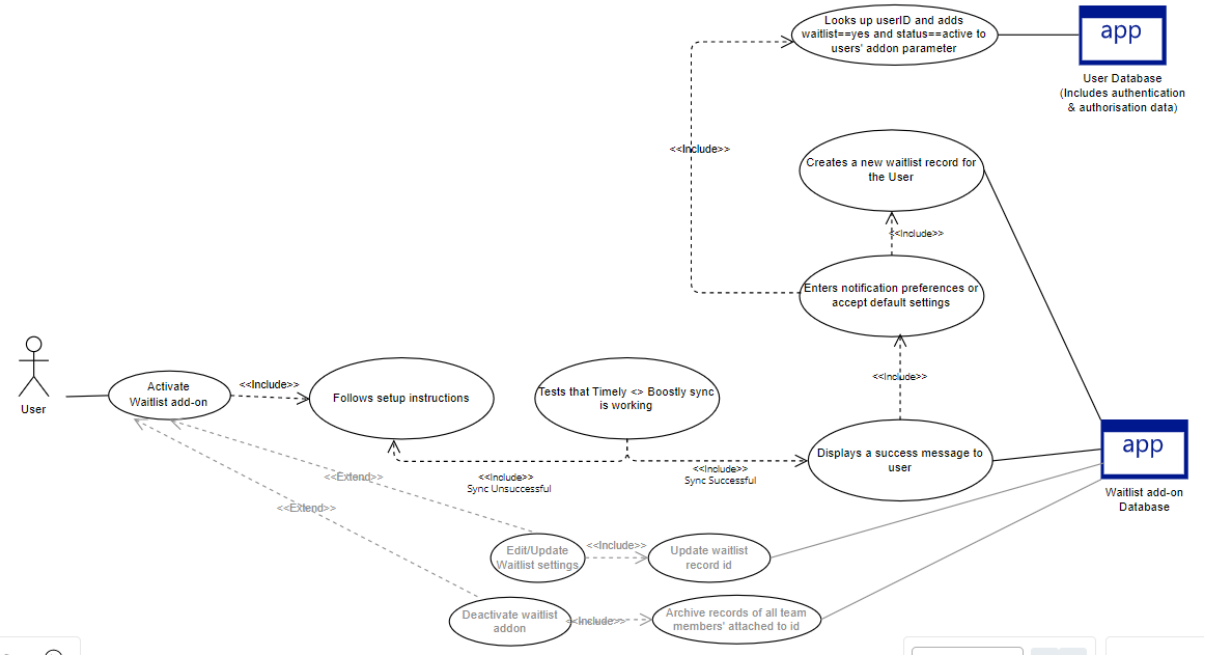
Miguelgrinberg. (n.d.). *GitHub - miguelgrinberg/flask-gridjs: Beautiful Interactive tables in your Flask templates using grid.js.* GitHub. https://github.com/miguelgrinberg/flask-gridjs

`

APPENDIX

Access addon use case

Use case: Activate addon



Brief Description

The User activates the Waitlist addons on Boostly

Initial Step-By-Step Description

Before this use case can be initiated, the User will have logged into their Boostly account.

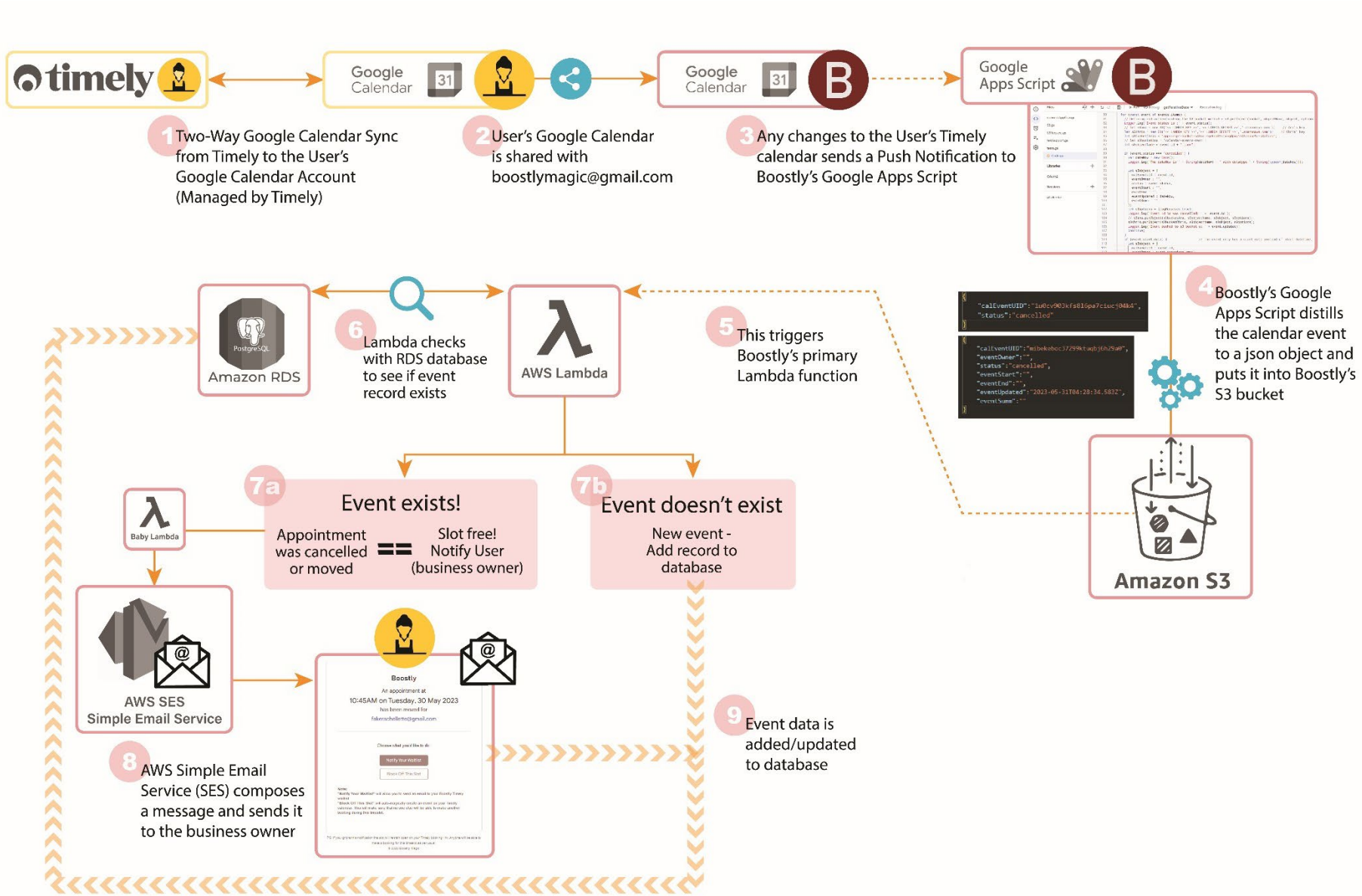
Xref: Log into Account

1. The User selects the “Activate Waitlist” button
2. The system will displays setup instructions to sync Timely with Boostly (exact interaction to be confirmed)
3. The User will follow the instructions
4. The system will initiate a test (or request that the User initiate a test) to ensure that settings are correctly captured
5. The system will display a successful sync message if synced and redirect user back to step 2 if not.
6. The system will request the User to enter their notification preferences
7. The User enters notification preferences and select “Complete Setup” after testing (exact interaction to be confirmed)

`

8. The system will create a new waitlist record for the User (and the User's team/employees) in the WaitlistAddon Database
9. The system will look up the UserID in the User Database and add "waitlist : active" to the "addons" field
10. The system will redirect the user back to their dashboard, which will allow them to easily update system preferences
11. The system will send a "Is Boostly synced with your Timely account?" notification to the User if it has not received any booking emails after 48 hours

Explanation:
This interaction is subject to quite a bit of user testing, and is also dependant on whether we are able to sync with Timely's calendar, or have to rely on the email notifications that Timely sends. This will be updated in the final report.
Note that, depending on the amount of time that we have, the Edit/Update Waitlist settings and Deactivate waitlist addon use cases might not be implemented. They are therefore included but greyed out at this stage.


timely
Google Calendar
Google Calendar
Google Apps Script
1 Two-Way Google Calendar Sync from Timely to the User's Google Calendar Account (Managed by Timely)
2 User's Google Calendar is shared with boostlymagic@gmail.com
3 Any changes to the User's Timely calendar sends a Push Notification to Boostly's Google Apps Script
4 Boostly's Google Apps Script distills the calendar event to a json object and puts it into Boostly's S3 bucket
5 This triggers Boostly's primary Lambda function
6 Lambda checks with RDS database to see if event record exists
Amazon RDS
AWS Lambda
Amazon S3
7a Event exists!
Appointment was cancelled or moved == Slot free! Notify User (business owner)
7b Event doesn't exist
New event - Add record to database
Baby Lambda
AWS SES
Simple Email Service
8 AWS Simple Email Service (SES) composes a message and sends it to the business owner
9 Event data is added/updated to database

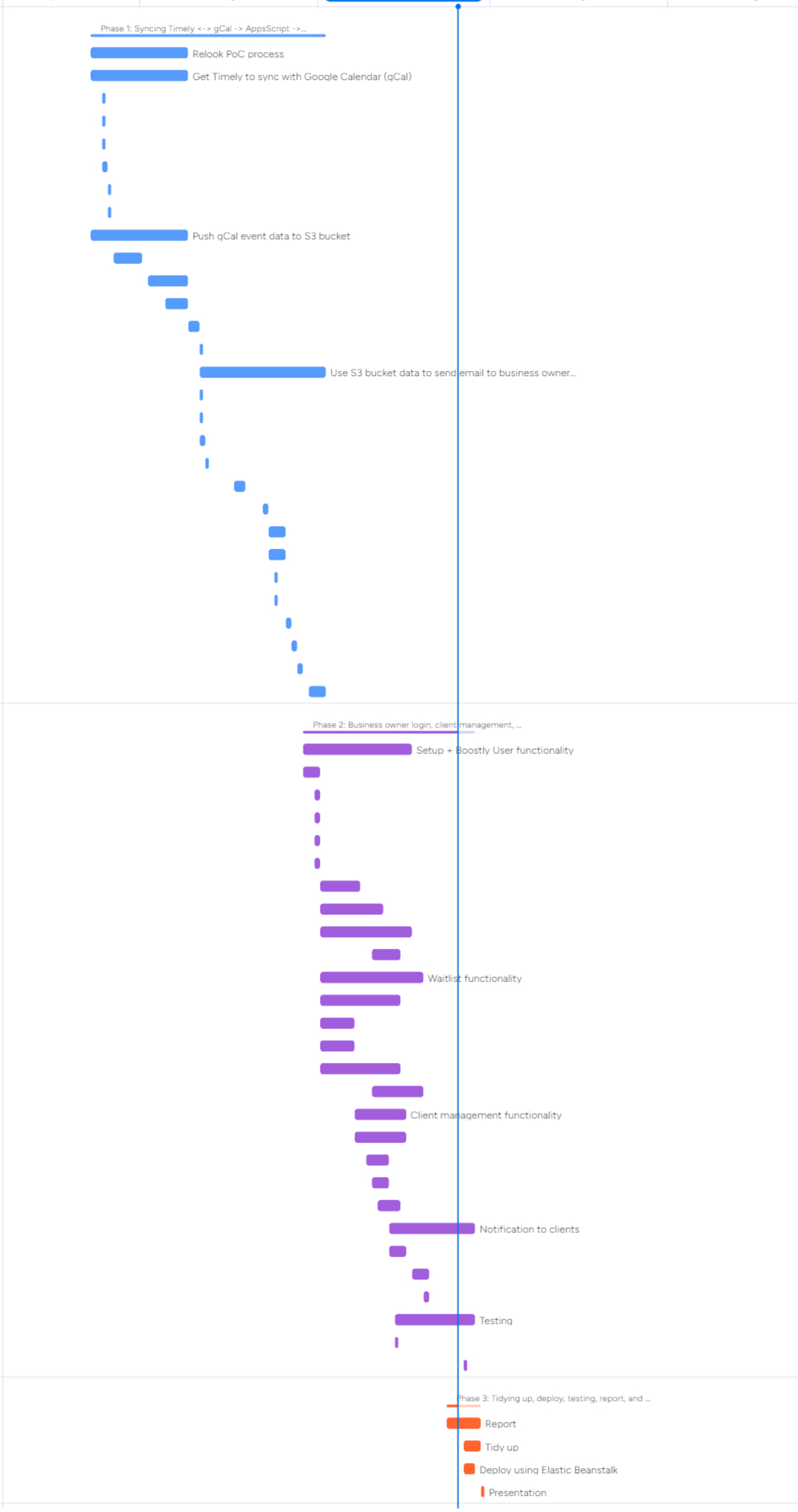
Gantt

June 25, 2023 | 13:44:05

Q2 2023 | **Q3** 2023

April | May | June | July | August

Phase 1: Syncing Timely <-> gCal -> AppsScript -> L		
Relook PoC process	Apr 22 - May 9	
Get Timely to sync with Google	Apr 22 - May 9	
Create test Timely Account	Apr 24	
Create test Gmail account to	Apr 24	
Create Boostly Gmail accoun	Apr 24	
Try out 2-way sync with Time	Apr 24 - 25	
Share FakeRachelle's cal to B	Apr 25	
Test sync between Timely<->	Apr 25	
Push gCal event data to S3 bucl	Apr 22 - May 9	
Try using AWS EventBridge	Apr 26 - May 1	
Try using Google Apps Script	May 2 - 9	
Send data to S3 bucket	May 5 - 9	
Get AppsScript to only get in	May 9 - 11	
Analyse event status and extr	May 11	
Use S3 bucket data to send em	May 11 - Jun 2	
Get S3 Bucket to trigger lamb	May 11	
Get Lambda to iterate throug	May 11	
Setup RDS database	May 11 - 12	
Connect database to Lambda	May 12	
Get Lambda to retrieve appro	May 17 - 19	
Commit records to RDS using	May 22 - 23	
Design logic for different han	May 23 - 26	
Extract and process specific	May 23 - 26	
Refactor code	May 24	
Make code more testable	May 24	
Query RDS to see if Events ta	May 26 - 27	
Create child Lambda for ever	May 27 - 28	
"Tag" business owners's time:	May 28 - 29	
When event is updated/cance	May 30 - Jun 2	
Phase 2: Business owner login, client management, s		
Setup + Boostly User functional	May 29 - Jun 17	
[Frontend] Database Waitlist	May 29 - Jun 1	
[Backend] Setup local postgr	May 31 - Jun 1	
[Backend] Setup basic Pytho	May 31 - Jun 1	
[Backend] Register functiona	May 31 - Jun 1	
[Backend] Login/logout funct	May 31 - Jun 1	
[Backend] StaffAccount func	Jun 1 - 8	
[Frontend] Overall CSS + Reg	Jun 1 - 12	
Chris sick	Jun 1 - 17	
[Backend] Refactor code	Jun 10 - 15	
Waitlist functionality	Jun 1 - 19	
[Backend] WaitAlert create, e	Jun 1 - 15	
[Backend] Figure out how to	Jun 1 - 7	
[Backend] Create waitlist aler	Jun 1 - 7	
[Backend] Sync alert page to	Jun 1 - 15	
[Frontend] Waitalert + previe	Jun 10 - 19	
Client management functionalit	Jun 7 - 16	
[Frontend] Grid table design	Jun 7 - 16	
[Backend] Client Preferences	Jun 9 - 13	
[Backend] Client Overview	Jun 10 - 13	
[Frontend] Overview page an	Jun 11 - 15	
Notification to clients	Jun 13 - 28	
[Backend] Created MsgTmpl	Jun 13 - 16	
[Frontend] Verified email noti	Jun 17 - 20	
[Backend] Parse information	Jun 19 - 20	
Testing	Jun 14 - 28	
[Backend] Create automated	Jun 14	
Testing MVP with live user	Jun 26	
Phase 3: Tidying up, deploy, testing, report, and pres		
Report	Jun 23 - 29	
Tidy up	Jun 26 - 29	
Deploy using Elastic Beanstalk	Jun 26 - 28	
Presentation	Jun 29	



Phase 1: Syncing Timely < | Phase 2: Business owner l | Phase 3: Tidying up, deplc

Add your board's description here See More

Main Tab... | Timeli... | + Integrate Automate

Phase 1: Syncing Timely <-> gCal -> AppsScript -> Lambda -> SES/Email

Task	Person	Timeline	Status	Comments	Date
Relook PoC process		Apr 22 - May 9	Success!		Apr 22
Get Timely to sync with Google Cal... 6		Apr 22 - May 9	Success!		Jun 25

Subitem	Owner	Status	Timeline	
Create test Timely Account		Success!	Apr 24	
Create test Gmail account to simulate business owner (FakeR...		Success!	Apr 24	
Create Boostly Gmail account		Success!	Apr 24	
Try out 2-way sync with Timely<->FakeRachelle calendars		Success!	Apr 24 - 25	
Share FakeRachelle's cal to Boostly cal with right permissions		Success!	Apr 25	
Test sync between Timely<->FakeRachelle<->Boostly calenda...		Success!	Apr 25	

Push gCal event data to S3 bucket 5	Apr 22 - May 9	Success!		Jun 27

Subitem	Owner	Status	Timeline	
Try using AWS EventBridge		Abandon	Apr 26 - May 1	AWS EventBridge fur
Try using Google Apps Script		Success!	May 2 - 9	AppsScript has built-
Send data to S3 bucket		Success!	May 5 - 9	
Get AppsScript to only get incremental event data from gCal		Success!	May 9 - 11	More efficient than p
Analyse event status and extract event data and push to S3		Success!	May 11	

Use S3 bucket data to send email t... 15	May 11 - Jun 2	Working on it		

Subitem	Owner	Status	Timeline	
Get S3 Bucket to trigger lambda function		Success!	May 11	
Get Lambda to iterate through S3 bucket files and extract onl...		Success!	May 11	We need to store eve
Setup RDS database		Success!	May 11 - 12	
Connect database to Lambda function		Success!	May 12	Require
Optimise psycopg2 - Pull out only required modules		To do!	-	
Get Lambda to retrieve appropriate data from S3		Success!	May 17 - 19	
Commit records to RDS using JSON		Success!	May 22 - 23	
Design logic for different handling of eventStatuses and how t...		Success!	May 23 - 26	
Extract and process specific variables to use for bOwn email n...		Success!	May 23 - 26	
Refactor code		Success!	May 24	Changed structu
Make code more testable		Success!	May 24	Add auto-database c
Query RDS to see if Events table contains unique CallID		Success!	May 26	

? Help

Create child Lambda for events that require alerting business ...		Success!		
"Tag" business owners's timezone to entry		Abandon	May 28 - 29	Spoke with Arthur an
When event is updated/cancelled, business owner should rece...		Success!	May 30 - Jun 2	

Apr 22 - Jun 2

Phase 2: Business owner login, client management, sending notification to clients

Task	Person	Timeline	Status	Comments	Date
Setup + Boostly User functionality 11		May 29 - Jun 17	Working on it	Note: Business Ow...	

Subitem	Owner	Status	Timeline	
[Frontend] Database Waitlist design		Success!	May 29 - Jun 1	
[Backend] Setup local postgres database		Success!	May 31 - Jun 1	
[Backend] Setup basic Python/Flask/SQLalchemy skeleton		Success!	May 31 - Jun 1	
[Backend] Register functionality		Success!	May 31 - Jun 1	
[Backend] Login/logout functionality + Profile		Success!	May 31 - Jun 1	
[Backend] StaffAccount functionality created		Abandon	Jun 1 - 8	We now have 5 mode
[Frontend] Overall CSS + Register/Login/Account		Success!	Jun 1 - 12	
Chris sick		Unplanned Delay	Jun 1 - 17	
[Backend] Refactor code		Success!	Jun 10 - 15	Changed structure o
[Backend] Add user Waitlist URL		Success!	Jun 26	
[Backend] Change all camelCase to snake_case		Success!	Jun 26	

Client management functionality 4	Jun 7 - 16	Success!	

Subitem	Owner	Status	Timeline	
[Frontend] Grid table design for Overview pages		Success!	Jun 7 - 16	
[Backend] Client Preferences		Success!	Jun 9 - 13	
[Backend] Client Overview		Success!	Jun 10 - 13	
[Frontend] Overview page and ClientPref page		Success!	Jun 11 - 15	

Waitlist functionality 5	Jun 1 - 19	Success!	

Subitem	Owner	Status	Timeline	
[Backend] WaitAlert create, edit, and select alertees		Success!	Jun 1 - 15	
[Backend] Figure out how to manage alerts		Success!	Jun 1 - 7	
[Backend] Create waitlist alert page		Success!	Jun 1 - 7	
[Backend] Sync alert page to DB		Success!	Jun 1 - 15	
[Frontend] Waitalert + preview functionality		Success!	Jun 10 - 19	

Notification to clients 6			Jun 13 - 28	Success!	

Subitem	Owner	Status	Timeline
[Backend] Created MsgTmpl		Success!	Jun 13 - 16
[Frontend] Verified email notifications to clients are working		Success!	Jun 17 - 20
[Backend] Parse information from database and send out emai...		Success!	Jun 19 - 20
[Backend] Get Business Owner's email-to-alertPage send over...		Success!	Jun 26
[Backend] Make sure that before allowing notification to client...		Success!	-
[Backend] Incorporate User Waitlist URL to notifications - Mak...		Success!	-

Testing 3			Jun 14 - 28	Working on it	

Subitem	Owner	Status	Timeline	
[Backend] Create automated database setup tables and data f...		Success!	Jun 14	
Testing MVP with live user		Success!	Jun 26	
Create component testing breakdown		Abandon	-	Abandon du

			May 29 - Jun 28		

Phase 3: Tidying up, deploy, testing, report, and presentation

Task	Person	Timeline	Status	Comments	Date
Live user testing with Lena		Jun 26	Stuck		
Tidy up 4		Jun 26 - 29	Success!		

Subitem	Owner	Status	Timeline
AppsScript		Success!	Jun 26 - 27
Parent lambda		Success!	Jun 26 - 27
Child Lambda		Success!	Jun 26 - 27
Python/eb script		Success!	Jun 26

Deploy using Elastic Beanstalk 1			Jun 26 - 28	Success!	

Subitem	Owner	Status	Timeline
Change waitlistURL to reflect AWS URL		Success!	-

Report			Jun 23 - 29	Working on it	
Presentation			Jun 29	Success!	


Jun 23 - 29