Automated Report Based System to Encourage a Greener Commute to Campus

Ronald Velasquez

Automated Report Based System to Encourage a Greener Commute to Campus

Automated Report Based System to Encourage a Greener Commute to Campus

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR
November, 2021

by

Ronald Velasquez

Abstract

This project consists of the design and implementation of a tool to encourage greener commutes to the University of Arkansas. Trends in commuting of the last few years show a decline in not so environment-friendly commute modes. Nevertheless, ensuring that this trend continues is vital to assure a significant impact. The created tool is an automated report system. The report displays information about different commute options. A Google form allows users to submit report requests, and a web app allows the sustainability office to process them in batches. This system was built in the Apps Script platform. It implements several Google services to make directions requests, store data, and send emails. It is composed of 6 major classes, functional code, and templates. Customizability and low cost were the two most significant design considerations. The current implementation of the system allows changing the takeaways, system constants, and email body. Also, it is possible to process an estimate of 200 reports/day at no cost.

THESIS DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this thesis when needed for research and/or scholarship.

Agreed Ronald Velasquez
Ronald Velasquez

Refused____________________________________
Ronald Velasquez

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

According to data reported on Second Nature [2], The Commuting Metric Ton Carbon dioxide equivalent emissions (MTCO2e) of the University of Arkansas have been steadily descending since the year 2018 as seen in Figure 1.1. This trend is significant when considering that university enrollment has been increasing since then. This trend is thanks to a portion of the university population moving to alternative commuting options. According to surveys conducted by the sustainability office at the University of Arkansas in 2015[3], driving was the commute mode preferred by 60% of the respondents. However, it dropped to 43% in 2020, and biking increased 6% [4]. Despite the 17% decrease in car commute preferences, other commuting forms such as walking dropped 3%, motorcycle/gas scooters increased 1%, and transit remained the same at 8%. Although the chosen alternatives are more environmentally friendly, they are still not optimal. There is a long way to go and great room for improvement.

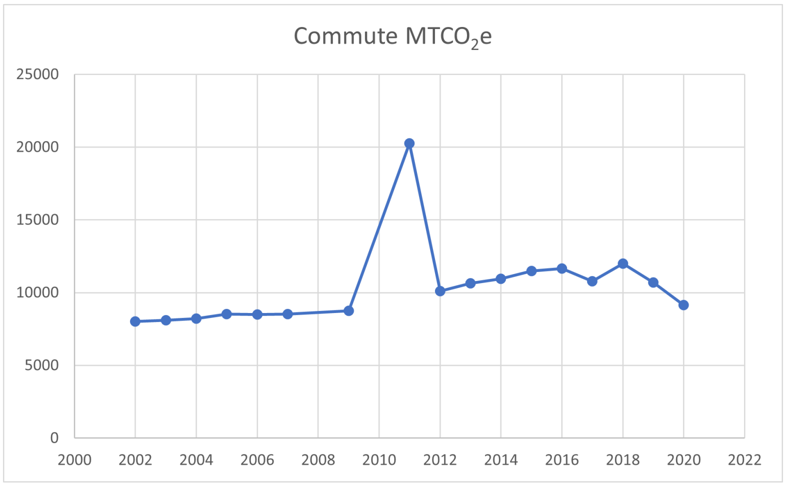


Figure 1.1: Metric Ton CO2e transportation emissions versus time for the University of Arkansas

The objective of this project is to create a new tool with the potential to encourage switching to greener commuting modes. This new tool would provide PDF reports. These reports would lay out the user's commuting options, duration, calories burnt, CO2 emissions, and expenses data. Also, based on the mentioned data, a commute modality is recommended. The possible commute modalities are walking, biking, transit, and driving. Manually generating these reports for the entire office staff at the university is time-consuming. It would require manually extracting data from google forms, performing google maps searches, and personalizing and sending reports to the respective users. Automating the generation of these reports would represent significant time savings for the university sustainability staff. Also, it would ensure that the user will receive it on a timely manner. The designed system is composed of two basic units: direct user inputs and batch processing. The direct-user-input unit processes entries provided through a form. On the other hand, the batch-processing system takes entries from a spreadsheet.

2 Overall Design

A set of technologies were chosen based on the necessities of the project. Having an overall knowledge of the available tools, classes that encapsulate them and provide functionality were defined. Through the implementation process, adjustments to them were made to fit the needs of the office of sustainability.

2.1 Project Component choices

The office of sustainability already possessed a manual system to process submissions. This system used a Google form as intake and made Google Maps searches with the provided addresses. Finally, it generated a PDF file report. The already in place system and the lack of future or constant system maintenance created constraints in the technologies and complexity of the system, leading to making the following choices.

2.1.1 Apps Script

Apps Script was chosen as the hosting platform. It is a service provided by Google that allows creating editor add-ons. Editor add-ons are JavaScript programs that can control and attach triggers to google services such as Google forms and spreadsheets. It also allows hosting web applications with simple systems to execute back-end functions. Apps Script only constitutes the platform of the system. Consequently, it was necessary to link a Google Cloud project to it. Therefore, APIs accessible through Google Cloud could also be implemented on Apps Script. The main advantage of Apps Script over other services such as Firebase is the capacity of having full access to basic cloud functionalities for free. Nevertheless, certain aspects of the platform are constrained by daily quotas and other limitations restricting the number of daily generated reports. The quota, shown in Table 2.1, that creates this limitation is the Google Maps direction query count due to each of report requiring up to 5 queries. Consequently, only 200 reports can be generated daily. This limitation applies to both batch and form submission processing. In addition, script run-time further limits batch processing as seen in Table 2.2. The average time of generating a report is 4.7 seconds, and 6 minutes is the longest a single script instance can execute. Consequently, a single call to the script would only generate roughly 76 reports.

Service	Daily Quota
File Conversions	2000
Properties read/write	50000
Spreadsheets created	250
Triggers total run-time	90 min
Google Map Direction query	1000

Table 2.1: Apps Script free daily quotas relevant to the project. Retrieved from Google Apps Script quotas [1]

Service	Limitation
Script Run-time	6 min/execution
Simultaneous executions	30
Email attachments	25/message
Email body size	200KB/message
Email total attachment size	25MB/message
Properties value size	9KB/value
Properties total storage	500KB/property stored
Triggers	20/user/script

Table 2.2: Apps Script free limitations relevant to the project. Retrieved from Google Apps Script quotas [1]

2.1.2 Direction Finder over Google Maps API

The most important part of the system is that of providing a preferred route. To do so, information about trip duration and distance is required. Initially, Google Maps Directions API was considered an option. Every query to this API has a cost of 0.5 cents. Its main advantage is that of having no daily limitations in the number of queries. Nevertheless, since this is a new system, reducing the cost of initial implementation is a major priority. This led to choosing Google Maps Direction finder. Although direction finder only allows for 1000 queries/day, they are free. The same number of queries would have a cost of $5 using Google Maps Directions API. Another difference between these two APIs is how queries are performed. While Direction Finder is accessible through an object, Google Maps API responses require making HTTP requests. The response structure obtained is essentially the same as that of Direction Finder. However, it is a JSON object that only needs parsing. Consequently, if future system versions require an increase in daily reports, switching APIs is simple.

2.1.3 Gmail API

All the available emailing APIs on Apps Script are limited to sending emails to 100 different addresses/day. This would halve the number of possible daily generated reports. Consequently, since Gmail API has fewer limitations, it was chosen as the emailing API. Gmail API uses quota units to define daily quotas. It provides a billion quota units/day. Sending an email has a cost of 100 quota units. Therefore, it is possible to send a million messages/day which does not limit the current implementation nor possible future migrations to different APIs.

2.1.4 Drive APIs

The designed system uses a DriveApp API which is part of the Apps script platform and Drive API V2. Although DriveApp is a complete interface, it lacks file conversion capabilities. File conversion is needed when downloading templates or uploading a file. Although the Utilities class also provides means to make blob conversions, spreadsheet formats are not supported. Therefore, a different file management API that would allow converting a file is required. This led to also using Drive API V2 to perform file conversions.

2.1.5 Spreadsheets for batch processing

A suitable input system had to be chosen to implement batch processing. Part of the manual process was already being done in spreadsheets. Consequently, using this file format would ease the transition to the automated system. Managing spreadsheets is easily done through the SpreadSheetApp API on the Apps Script platform. The usage of spreadsheets across the system has a couple of advantages:

- It allows to easily split the data to be processed without data conflict issues. Consequently, concurrently processing a file is a possibility.
- Since every single cell in a spreadsheet can be accessed and modified individually, error messages can be strategically placed on the troublesome cells.

2.1.6 PDF over HTML

Once the system generates a report, it needs to be sent using a format that displays consistently. Initially, HTML was considered an option. Nevertheless, due to the variety of HTML rendering systems and their lack of support for some CSS customizations, PDF was chosen instead. PDF is not an exact displaying system since certain aspects such as zoom are completely dependent on app settings. However, it ensures that the file being sent displays similarly across different devices without any visual artifact.

2.2 Class distribution and interaction

Six classes, two sections of purely functional code, and several HTML and CSS templates comprise the system. Classes do not encapsulate the functional code portion of the system due to constraints regarding cloud function calls and trigger creation. The six classes include a utility class used across the entire system and five other classes whose relationships are described in Figure 2.1.

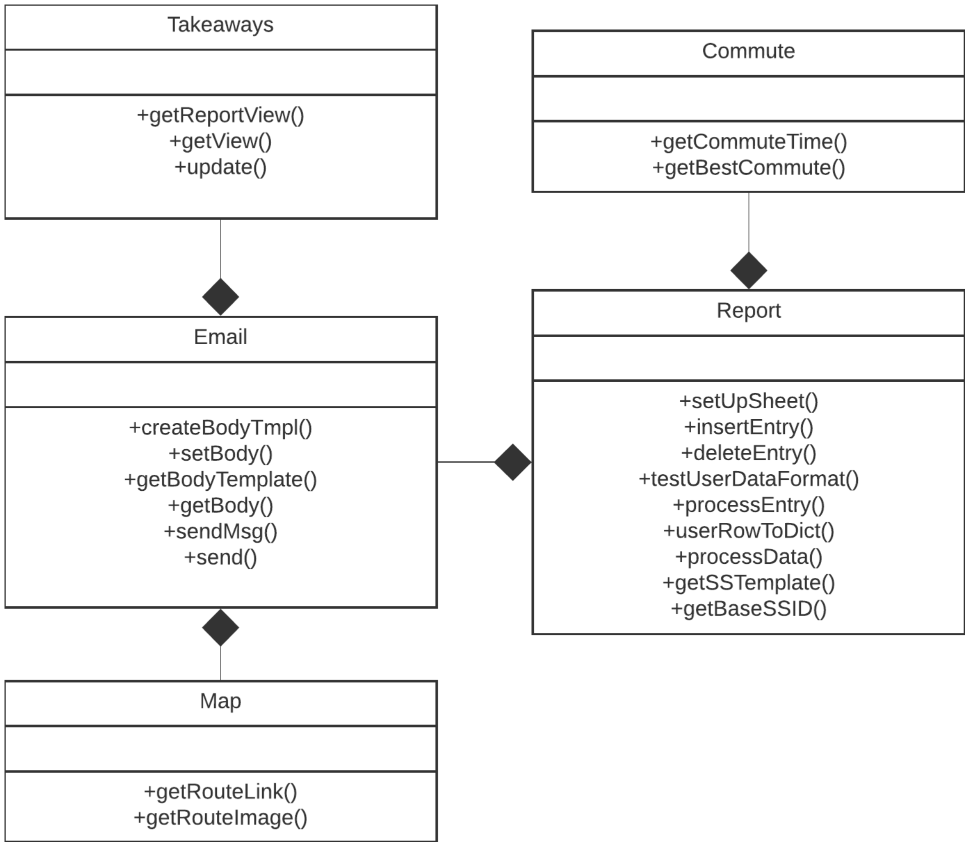


Figure 2.1: UML Class Diagram of commuter menu system (Only public methods shown)

Each class is utilized for a set of specific operations:

2.2.1 ProjectUtils

ProjUtils is the project utility class. Its methods are all static and used across the system. Some of the members of this class perform the following operations:

- Unit conversion for distance and time.

- Basic input testing routines for empty fields and specificity.
- Folder check and creation.
- File ID retrieval and creation.
- Quota tracking.

2.2.2 Takeaways

The Takeaways class is in charge of managing takeaway templates. A JSON file under the name takeaways.json stores them. This file is a dictionary of the various commuting modes, and each mode entry is an array of different takeaway templates. These templates may contain tokens of the form [⟨dataType⟩]. ⟨dataType⟩ refers to the specific data used to substitute the value. For instance, User.data, Driving.Time, etc. Regular expressions are used across the system to ensure the correctness of these substitutions. getReportView() makes use of them to generate the takeaways given specific user data. This class also provides a front-end view of the system through the method getView() and a way to modify the takeaways templates through update().

2.2.3 Map

The map class contains methods regarding visual aspects of the commute. Through the method getRouteLink(), a Google Maps link is created with the respective commuting mode, origin, and destination. The link is generated using the Google Maps API V1 format. Based on the type of addresses that the system is receiving, this function only needs to replace comas for "%2C" and white spaces for "+" to fit the correct format. Another relevant method is getRouteImage(). This method makes use of the static maps Google API to generate a view of the commute route. Using polygon information provided by DirectionFinder, the route is traced from origin to destination on a static map image. Nevertheless, the final project does not implement it. It provided little use to the system and had reliability issues. getRouteLink() was used instead since it delivered a cleaner and more useful experience.

2.2.4 Commute

The Commute class performs all operations related to DirectionFinder queries. It manages request errors and formats the requested data. getCommuteTime() is its main method. It returns an array of dictionaries that contain error messages, bus routes, commute duration and distance, and whether the requested route was found. getCommuteTime() requests directions calling other two private methods, _rqsCommuteData() and _rqsDriving(). _rqsCommuteData() is used for all modes but driving. Driving data requires making two requests due to the need for directions to and from parking lots in some cases. Therefore, _rqsDriving() accounts for these differences. In total, the Commute class performs five Google Maps API requests if a parking lot is present or four otherwise. The Commute class also selects the best commute mode. This is done through getBestCommute(). This method makes use of different time duration ranges to select the best commute. As seen in Table 2.3, if walking takes longer than 15 minutes and bicycling takes less than 30 minutes, bicycling would be chosen as the best commute mode. The order in which these modes are selected prioritizes the greener modes

Commute Mode	Commute duration (minutes)
WALKING	<15
BICYCLING	<30
TRANSIT	<45
DRIVING	No constraints

Table 2.3: Time ranges for best commute mode selection

2.2.5 Email

This class contains methods for report creation, emailing, and email body editing. It builds the report employing a template stored in commuteTemplate.html, and fills it out using data coming from the Commute, Takeaway, and Report Classes. The sendMsg() method creates and sends Emails. This method implements the method Users.Messages.send() from the Gmail API which receives a base64 encoded RFC2822 formatted string. This encoded string is a multipart/mixed message that includes the pdf report. In case of system errors, this method also sends error messages only as an HTML email by omitting the application/pdf part of the message. The sendMsg() function is implemented by the method send(). send() accepts user commute data and system calculation as parameters. Also, it obtains the best commute and creates a Takeaways object. It fills out the report template with this information and sends it. Email also maintains the body section of the email. It uses the method createBodyTemplate() to create a template file in case of deletion. It also possesses getters and setters to extract and edit a formatted version of the email body.

2.2.6 Report

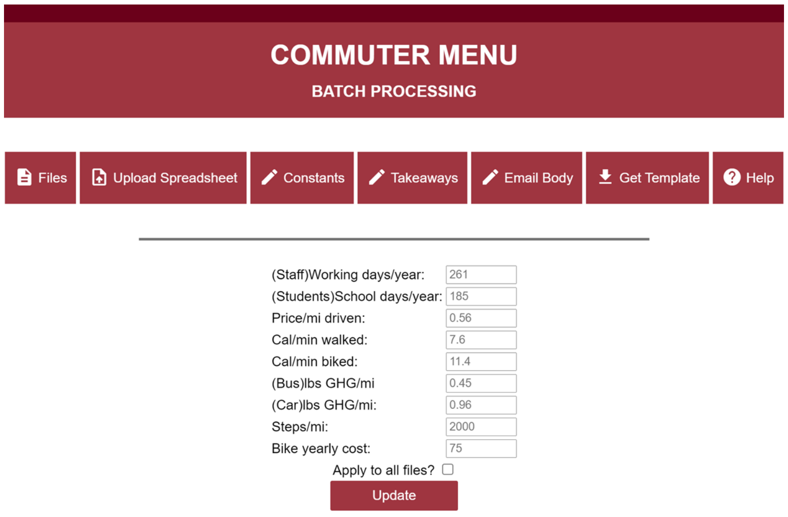
The report class is the entry point for all the processing done in the system. This class keeps a reference of the current spreadsheet being processed and a reference to an Email object. This class makes use of the method setUpSheet() to prepare spreadsheets for processing. Preparing a spreadsheet involves checking that it contains a proper output sheet where the results of the operations are stored. Also, whenever the constants sheet is absent, it is copied from the form submission spreadsheet onto the current spreadsheet. Report prepares values (rows) extracted from the spreadsheets. This preparation includes generating a dictionary named userData through the method userRowToDict(). This class also performs value testing and error reporting. In the case of batch processing and form submissions, if errors are present, an error dictionary is generated by testUserDataFormat().

This function also modifies values represented different internally. The method processEntry() processes individual spreadsheet rows. For batch processing, this function is called recursively by processData(). Batch processing progress is reported using script properties. Every time an entry is processed, a property increments. getSSCount() provides this script property to the front end. The Report class provides methods to securely insert and delete values in a spreadsheet. These methods are insertEntry() and deleteEntry(). When forms are submitted, there is a chance that the single spreadsheet used has simultaneous writes and deletes. To account for this, the Report class keeps a document-level lock to protect the spreadsheet from data corruption.

2.3 Built to be customizable

Another aspect taken into consideration when designing the system was its level of customizability. Since the system will lack constant or long-term support, the system must allow the user to make small changes to the report and values calculation. The first section of the system that is customizable is constants. Constants are stored in the system's form submissions spreadsheet. They are copied into any new spreadsheet uploaded to the system. This customization allows changing constants on only the submission spreadsheet or all the spreadsheets stored in the system. This is done by clicking on a checkbox at the bottom of the form as seen in Figure 2.2. Since most of these constants were obtained from web pages, scraping or using APIs that provide this information could be alternatives. Nevertheless, the lack of long-term support could cause issues with changes in web pages design and API lack of support. Therefore, having users make these changes is a more reliable option.

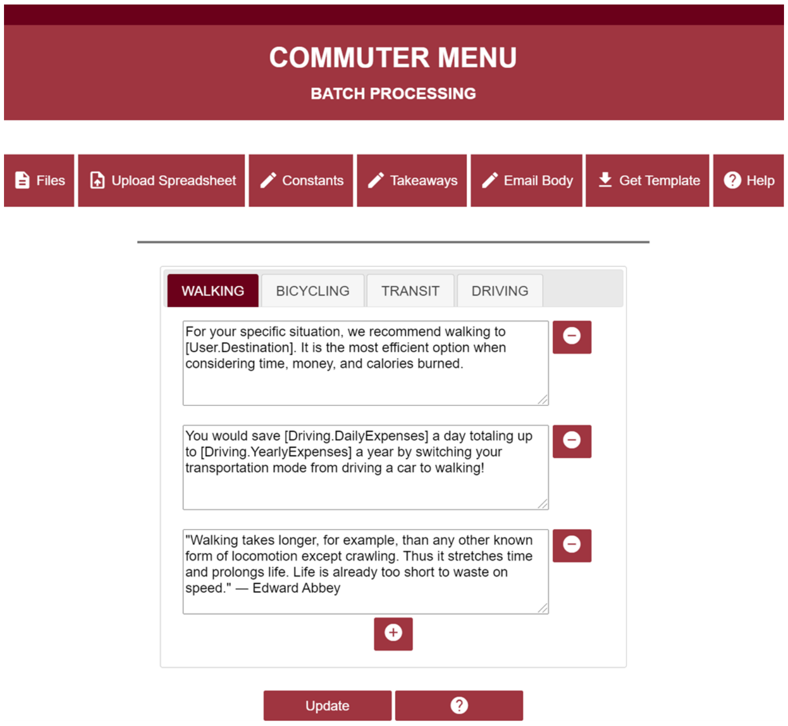
Takeaways are another editable section of the system. This section allows creating templates for every single one of the possible best commute modes as seen in Figure 2.3. The system allows to delete and add takeaways and display report calculations using tokens. Another relevant editable section of the system is the



(Staff)Working days/year:	261
(Students)School days/year:	185
Price/mi driven:	0.56
Cal/min walked:	7.6
Cal/min biked:	11.4
(Bus)lbs GHG/mi	0.45
(Car)lbs GHG/mi:	0.96
Steps/mi:	2000
Bike yearly cost:	75
Apply to all files?	☐

Figure 2.2: Web app view of system to edit constants

body of the email sent along with the report as seen in Figure 2.4. This field only allows accessing the name of the user to whom the report is being sent. In the same way as the rest of the editable sections of the system, the email body field allows for embedding extra HTML since inputs are unsanitized.

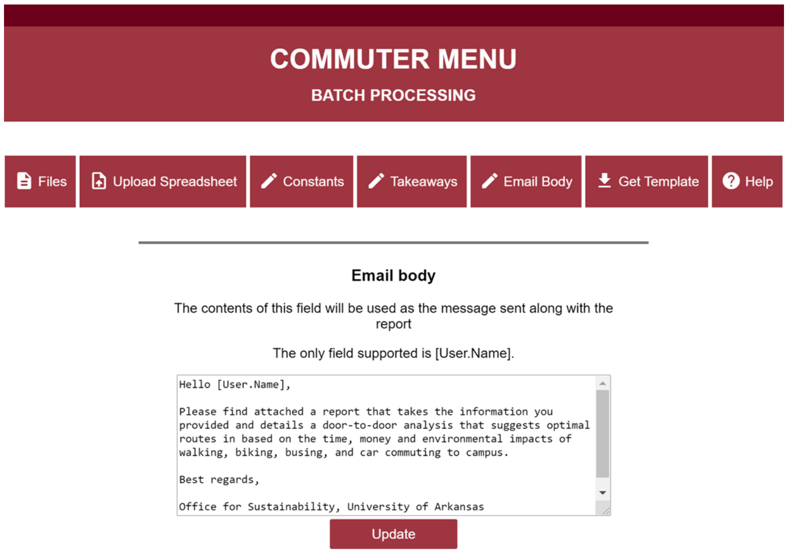


For your specific situation, we recommend walking to [User.Destination]. It is the most efficient option when considering time, money, and calories burned.

You would save [Driving.DailyExpenses] a day totaling up to [Driving.YearlyExpenses] a year by switching your transportation mode from driving a car to walking!

"Walking takes longer, for example, than any other known form of locomotion except crawling. Thus it stretches time and prolongs life. Life is already too short to waste on speed." — Edward Abbey

Figure 2.3: Web app view of the system to edit takeaways



Email body

The contents of this field will be used as the message sent along with the report

The only field supported is [User.Name].

```
Hello [User.Name],

Please find attached a report that takes the information you
provided and details a door-to-door analysis that suggests optimal
routes in based on the time, money and environmental impacts of
walking, biking, busing, and car commuting to campus.

Best regards,

Office for Sustainability, University of Arkansas
```

Figure 2.4: Web app view of system to edit email body

3 Design Constraints

The implementation of the system resulted in challenges due to constraints from the chosen platform and APIs.

3.1 Google Apps Script

3.1.1 Form Customization

Although Google Forms are easy to modify, their customizability is limited. It increases the difficulty of ensuring the correctness of user input. This is especially true for addresses. Due to the need for a certain level of specificity to ensure correct Google Maps queries, the user needs to provide more than just their street address. An option would be to suggest complete addresses as the user provides input. Nevertheless, Google Forms do not allow for that. Therefore, regex validation and comprehensive error messages are displayed to reduce user error.

3.1.2 Performance and lack of multi threading

The maximum function run time is 6 minutes long. Therefore, the number of entries that can be processed at once is limited. Batch processing is affected by this constraint due to it being a time-consuming process. A single call would not be enough to process a file with over 80 entries. Due to having several entries that go through the same process, parallelly processing a set of entries would represent a significant speedup. Nevertheless, since the system is JavaScript-based, this is not an option. This limitation is accounted for by making multiple queries with different row starting points. This halves the total processing time and increases the number of entries processed in a single click.

3.2 Google API

3.2.1 Time dependence of transit routes

Making commute requests in transit mode require a departure time. If not provided, it defaults to Date.now(). However, issues may arise for days when bus routes are not available such as Sundays. Therefore, the system moves all requests to the following Monday to mitigate the problem. Nevertheless, there is still a chance that the selected Monday has no bus routes available.

3.3 MailApp API

3.3.1 Manual message structure creation

Although MailApp API provides an easy and intuitive interface to generate email messages, the daily free quota is too limiting for the system. This led to using Gmail API. Consequently, email bodies and headers are manually set.

4 Detailed design of components

4.1 Script level properties

Script level properties are properties linked to the current project. Compared to other properties such as User level properties, different users accessing the web app share them. Script level properties are used instead of more specific user level properties since the information stored is mostly system settings and personalizations. These properties are no more than a dictionary that can store information between executions making its retrieval faster than if they were stored in Google Drive. Script level properties are implemented in the class ProjectUtils,

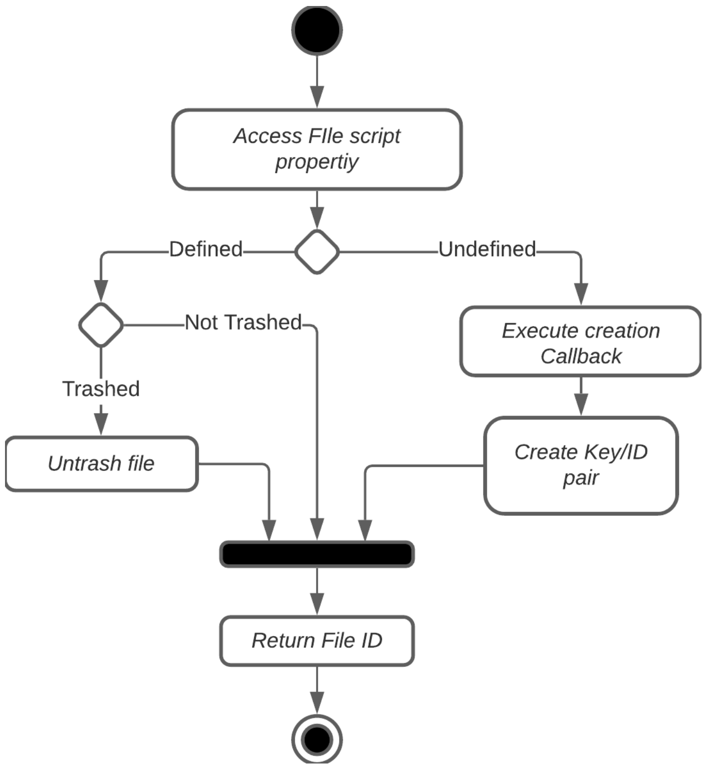


Figure 4.1: Activity flow of the method getFileID()

and they are used to keep track of file IDs and time quotas. The method getFileID() is used to obtain file IDs. This method accepts a property name (key) and a function callback. The callback is executed to generate a file, obtain its ID, and create a key/value pair in the script properties if it is not present. Keeping track of time quotas is done similarly. The main difference is that two properties are stored. One of them is a time property to keep track of the execution time used. The other property is the current date. The time quota restarts daily, so whenever the current date property is different from Date.now(), the time quota property needs to be restarted. Calling quotaTimeStart() sets up these properties and starts a timer. Then, quotaTimeEnd() stops the timer and stores the duration in the time property. The daily execution time is limited to 90 minutes. However, to allow some room to report errors, executions are not allowed after 80 minutes. Keeping track of the number of entries processed during batch processing is also done using Script properties.

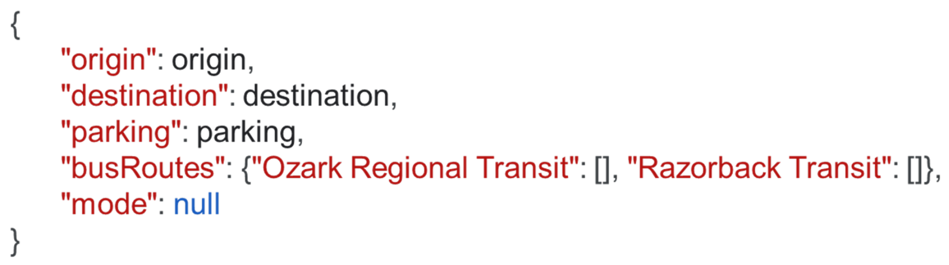
4.2 Data corruption protection

Data corruption can occur when different instances of the system try to write at the same time on the same data field. To account for these situations, Apps Script provides access to the LockService API. The LockService API allows controlling the concurrent access of critical code zones. This API is essentially mutexes, and as such, it is necessary to call tryLock() or waitLock() to attempt lock acquisition. Once the lock is no longer needed, releaseLock() must be called. LockService provides access to locks of different scopes. The designed system only uses document-level locks. However, they can also be script or user level. The locks implemented control editing script properties to avoid creating multiple files if the key is not present. Also, during spreadsheet row deletions, there is a slight chance that other system instances are appending rows. This could cause the deletion of the incorrect row. Finally, when setting up a spreadsheet, a lock is requested to avoid creating multiple sheets under the same name. In all cases, the lock is only attempted to be acquired for one second. The system uses both tryLock() and waitLock(). tryLock() returns false if a lock is not acquired. Therefore, less vital operations such as keeping track of execution quotas implement this type of lock. Stopping the system when a lock is not acquired would not provide extra benefits. waitLock() throws an exception when a lock is not acquired. Therefore, operations essential to generate the report implement this type of lock. When processing a form is the source of these exceptions, an email is sent to the user informing server issues. For the web app, lock exceptions are handled in the front end like other exceptions from the system. It prompts an error message to the user informing the error and requesting to retry. Not acquiring a lock is unlikely since one second is enough for all processes that require a lock to finish.

4.3 DirectionFinder Queries

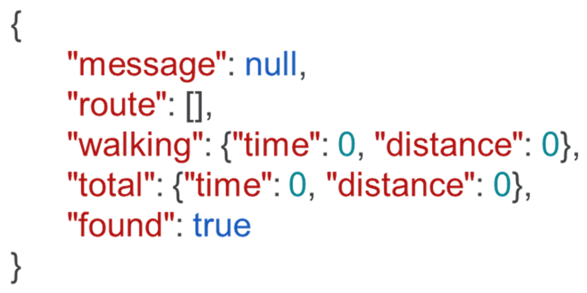
4.3.1 Formatting

The Commute class makes all DirectionFinder API queries. This class stores commute information in a dictionary as seen in Figure 4.2. This dictionary is an easy way of maintaining commute information and switching between the different commuting modes. This is done by only changing the value of the mode entry. The dictionary also stores the resulting bus route numbers as another dictionary under the name busRoutes. This dictionary contains entries for the two bus services in the area, and each one is an array on which the respective route numbers are appended. The response from a DirectionFinder query is an object. This object is a collection of routes each of which contains several legs which contain steps. Queries are made without requesting alternative routes, so there is always a single route entry. Legs are usually the result of commuting mode changes, e.g., if to catch a bus some walking needs to be done there will be a leg for walking and a leg for riding the bus. On the other hand, steps result from direction changes in a commute and contain distance, time, and transit information. To simplify handling commute information, the responses are mapped into



```
{
    "origin": origin,
    "destination": destination,
    "parking": parking,
    "busRoutes": {"Ozark Regional Transit": [], "Razorback Transit": []},
    "mode": null
}
```

Figure 4.2: Commute class main dictionary structure



```
{
    "message": null,
    "route": [],
    "walking": {"time": 0, "distance": 0},
    "total": {"time": 0, "distance": 0},
    "found": true
}
```

Figure 4.3: Dictionary structure used to store DirectionFinder responses

a collection of dictionaries of the structure shown in Figure 4.3. The route entry stores the query response used to create a static map using polygon information. Although no longer in use, it is still part of the structure for future implementation of this feature. This dictionary also contains the messages entry to describe errors and a boolean entry found to signal if a route for the given mode exists. The entries walking and total are used to keep track of walking and driving or riding. They store time in seconds and distance in meters. Driving or riding information is obtained by subtracting walking from the total.

4.3.2 User data Calculation

Once a set of DirectionFinder queries have successfully been made and formatted, commute data is computed. The formulas in Figure 4.4 are introduced into individual cells in the spreadsheet where the user data comes from. reportType-

Days represent the maximum number of days that a user will be going to campus. Depending on the type of report, the value of reportTypeDays may change. There are two basic report types: student and staff. The reportTypeDays for the student type is 185 days, and the staff type is 261 days. Since this defines the total number of days for each type, this must be divided by seven and multiplied by the commute frequency of the user to obtain their yearly commute days as seen in Figure 4.4 for the yearlyCommuteDays formula. In the front end, report types such as on-campus and off-campus students map to the student type while faculty and staff map to the staff type. This data is displayed on the report in the form

$$totalSteps = distanceWalking \times \frac{steps}{mile}$$

$$carYearlyExpense = parkingCost + \left(commuteDistance \times 2 \times yearlyCommuteDays \times \frac{cost}{mile}\right)$$

$$carDailyExpense = carYearlyExpense / yearlyCommuteDays$$

$$yearlyCommuteDays = \frac{reportTypeDays}{7} \times commuteFrequency$$

$$bikingCaloriesBurnt = bikingTime \times 2 \times \frac{calories}{minBiked}$$

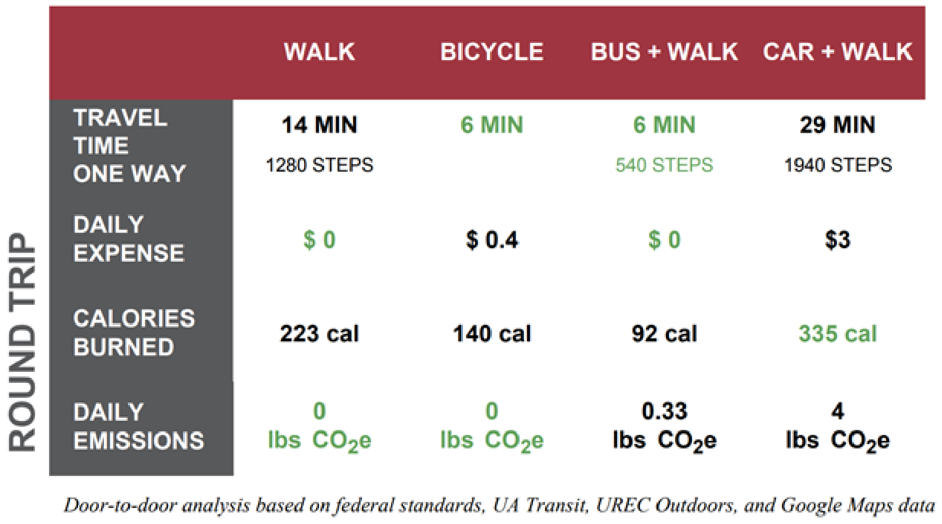
$$walkingCaloriesBurnt = walkingTime \times 2 \times \frac{calories}{minWalked}$$

$$dailyGHGCO_2 = [driving|busRiding]Distance \times 2 \times [driving|busRiding]\frac{lbGHG}{mi}$$

$$yearlyGHGCO_2 = dailyGHGCO_2 \times yearlyCommuteDays$$

Figure 4.4: Majority of formulas to perform user commute calculations. Red colored values are editable system constants.

of a grid as seen in Figure 4.5. Such a grid is a simple table HTML tag. However, some cells and values in this table were changed to fit the desired format. First, some of the calculations are rounded to the nearest integer, for instance, calories burnt. On the other hand, other values are also rounded to two decimal places if below zero. This is the case for daily emissions and daily expenses. The smallest value in each row is colored green unless the row is calories burnt in which case it is the greatest. In terms of structure, each cell in TRAVEL TIME ONE WAY is another table tag in which commute duration and steps are separate rows. This allows for a more consistent separation between the values and a better alignment with the BICYCLE cell which lacks step count.



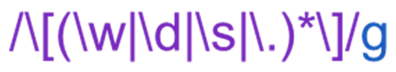
ROUND TRIP	WALK	BICYCLE	BUS + WALK	CAR + WALK
TRAVEL TIME ONE WAY	14 MIN 1280 STEPS	6 MIN	6 MIN 540 STEPS	29 MIN 1940 STEPS
DAILY EXPENSE	$ 0	$ 0.4	$ 0	$3
CALORIES BURNED	223 cal	140 cal	92 cal	335 cal
DAILY EMISSIONS	0 lbs CO_2e	0 lbs CO_2e	0.33 lbs CO_2e	4 lbs CO_2e

Door-to-door analysis based on federal standards, UA Transit, UREC Outdoors, and Google Maps data

Figure 4.5: Grid on report displaying some of the user calculated data

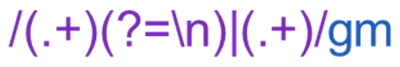
4.4 Customizable takeaways and email body

There are front-end and back-end systems relevant to the email and takeaways customization. In the front end, inputs are acquired through textarea HTML tags contained in a form, and regular expressions preprocess the data. For the takeaways section, tokens are extracted using the regex shown in Figure 4.6 for testing. This regex matches any value that is either a letter, number, space, or dot in between square brackets. Therefore, not necessarily that of correct tokens allowing displaying error messages for strings that follow the format of a token but are not correct. These matches are then compared to the acceptable tokens. For email body preprocessing, the regular expression in Figure 4.7 is used. This regex captures any line of text with a line skip or at the end of the text. The matched text is inserted inside of an HTML paragraph tag. This is done so that the email body is properly displayed. Properly displaying content requires some extra processing in



```
/\[(\w|\d|\s|\.)*\]/g
```

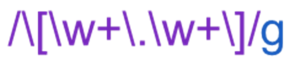
Figure 4.6: Takeaways front-end regex



```
/(.+)(?=\n)|(.+)/gm
```

Figure 4.7: Takeaways front-end regex

the back end. For the email body, it is only necessary to substitute the [User.Name] token if it is present. For takeaways, the process takes longer due to the length of strings. Therefore, using regular expression and replaceAll() is the most efficient way of performing this operation. The regex utilized is in Figure 4.8. This regex is a stricter version of that in Figure 4.6 and only accepts characters separated by a dot. To speed up the process, a dictionary is used as a hash table. Some of the entries in this hash table are the needed values. Nevertheless, some other entries are lambda functions that receive the commuting mode as its parameter. This was done to reduce the size of the hash table.



```
/\[\w+\.\w+\]/g
```

Figure 4.8: Regex for back-end Takeaway processing

4.5 Batch processing

4.5.1 Types of batch processing

The batch processing supports two forms of processing: process-all and process-new. To keep track of which entries have been processed, whenever a file is being set up, an extra column is added. This column is a collection of checkboxes named progress. These checkboxes are checked if an entry has been successfully processed. If there is not a checkbox present, it would be the equivalent of an unchecked checkbox. Checkboxes were chosen over the usage of spreadsheet properties since they would allow users to have greater control over the entries to be batch processed.

4.5.2 Front-end parallel processing

To achieve some speed up, the entries on a spreadsheet are processed concurrently. This starts by calling processBatch() which makes a back-end request to getSSCount(). Since getSSCount() signals the start for processing, certain setups take place here. First, all checkboxes are unchecked in the current file if the call is made in process-all mode. Second, this function restarts the entries processed counter for the current spreadsheet. Finally, it returns the number of entries present in the file. Next, partProcessing() is called. partProcessing() defines the start and end indexes of the concurrent calls using the entries count. Then, it makes queries to the back end to run the function processFile(), and it keeps track of the number of concurrent calls being made. A single call to the server can only process approximately 78 entries in a file. Therefore, whenever the server returns an error message signaling that run time has been exceeded, another call is made from partProcessing(). The new call is made as a process-new batch processing. Since process-all causes checkboxes to be cleared, process-new would only pick up where the processing was left. Currently, the speedup obtained is not so significant due to the limitation of entries that can be processed per day. Consequently, the number of entries is only split into two concurrent processes. However, the functions can be modified to allow for more concurrent queries whenever the system's daily quota needs to be increased.

4.5.3 Entry processed tracking

While entries are being processed, updateProgressVal() is called in an interval of 7500ms. This function makes a call to the back-end procedure getProcCount() which returns the number of entries that have been processed. Calls are made in this interval since it has been observed that at most a report takes roughly the same amount of time. This ensures that by the time the request is made there have been changes to report. These changes are displayed using a jQuery progress bar.

4.6 Form Processing

4.6.1 Uniformity in batch and form processing

The processing of form submissions implements the same code used for batch processing. Instead of having processData() as its entry point, it has processEntry(). Therefore, the data needs to be formatted as a dictionary which is done by calling Report.userRowToDict().

4.7 File Management

4.7.1 Spreadsheet Uploading

As part of the batch processing, spreadsheets need to be uploaded to google drive. This process is done using an input HTML tag and setting its type property to file. This field supports .xlsx, .xls, and .ods formats. Once a file is selected, its blob data is acquired. Sending this data to the backend in the most efficient way requires using base 64 encoding. Nevertheless, during the process, the default JavaScript base 64 encryption takes the entire integer buffer and converts it into a string which includes comas. To account for this issue, the function uploadFile() takes the encoded string, decodes it, generates a blob, gets the blob data as a string, splits the string, parses the integer values, and generates a blob for the spreadsheet. The generated blob file is then converted to the google spreadsheet type using the Drive API and moved to the proper folder.

4.7.2 Report Images

When the report template is converted into a PDF file, the resulting PDF file only displays content that was already present and not content that requires retrieval. Therefore, the source of images cannot be regular URLs. The data must be in the file at the moment of conversion. Consequently, data URLs are used instead, and the image data is encoded in base 64. To reduce encoding time, the images are stored in HTML files as image tags ready to use. The project files that are of this type are SOLogo.html, mapsLogo.html, and passiogoLogo.html.

4.8 Input testing and error handling

For the correct functioning of the system, the following set of entries must not be empty:

- Name
- Email
- Relation to the UofA (Type of report)
- Origin Address
- Destination Address
- Parking address
- Parking cost

Most of the input testing for form submissions happens in the front end. Built-in validations are used to test email and integer value fields. On the other hand, validations were manually defined for origin, destination, and parking address fields. To test them, regular expressions were defined as seen in Figures 4.9 and 4.10. The origin address regex requires that the address at least contains street and city which need to be at least three characters long. On the other hand, the regex for destination and parking address is a derivative of the origin regex. Since these two values can be described in terms of their names or street name, only a single value is needed. Whenever there are errors that the form cannot provide messages for,

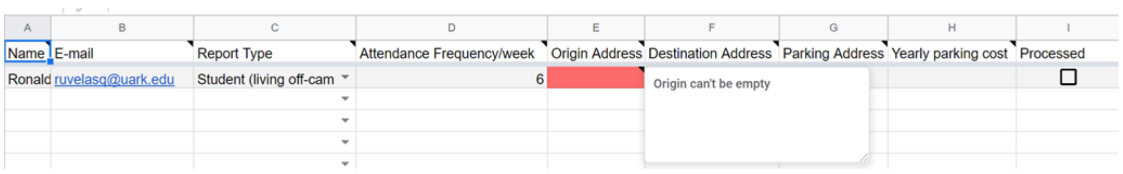
```
/^(\s*\w+\s*\-?\.?\s*){3,},(\s*\w+\s*\-?\.?\s*){3,}$/
```

Figure 4.9: Regex for origin address testing

```
/^(\s*\w+\s*\-?\.?\s*){3,}$/
```

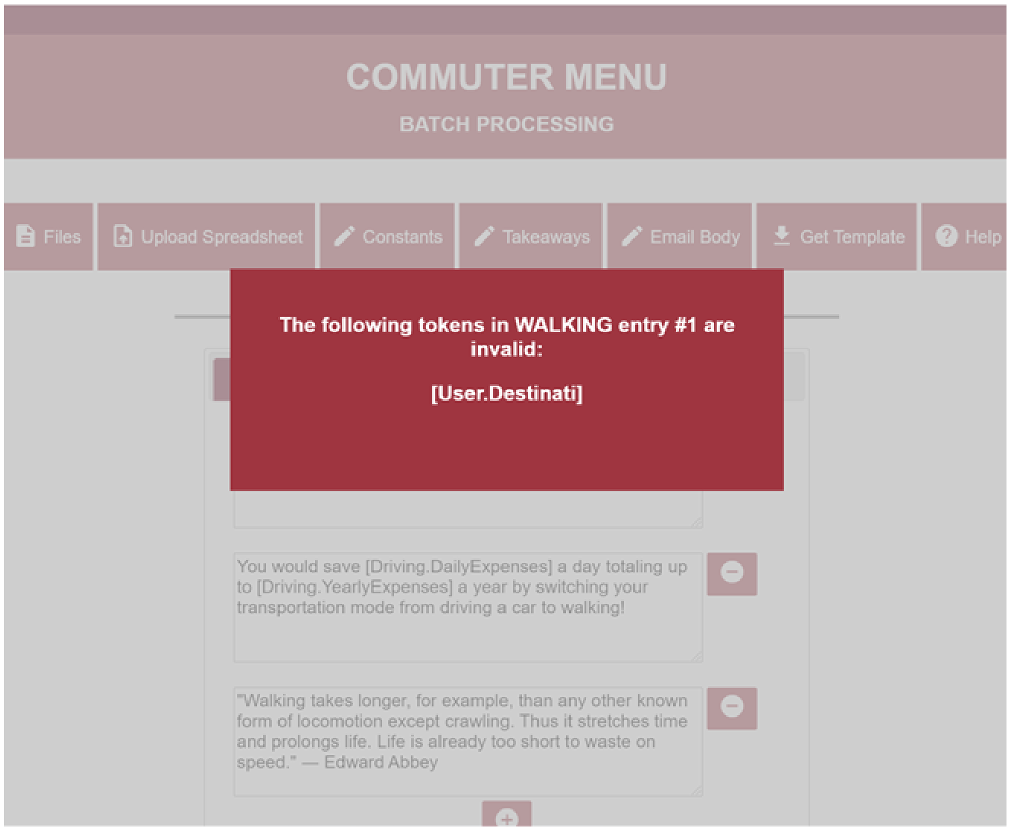
Figure 4.10: Regex for destination and parking addresses testing

an email is sent. It describes the errors along with a link to the submitted form. This reduces the time it would take users to correct their submissions. In terms of spreadsheet input testing, integer value validation is built-in into every numeric cell of the batch processing template. It was attempted to implement regex validations for origin, destination, and parking in the template. Nevertheless, this testing was not compatible with file formats other than Google spreadsheet. Consequently, when these addresses come from a spreadsheet, they are split wherever there is a comma, and the resulting number of values is counted. Based on this number, city and state strings are appended as needed. The email cell is tested in the backend. However, due to the complexity of email testing, the regex defined by [5] is used. To avoid any error providing the report type, cells for this column behave as a drop-down list. In case errors are present in any of the cells, their color is changed to red, and a note is created describing the error as seen in Figure 4.11. The web app also requires error handling due to the customizability options. The intakes of the editable constants are number-type input tags with minimum values set to zero. Consequently, values coming from this form will always be valid. Also, whenever there are empty entries in the editable takeaways, these entries are ignored. Red boxes are used to display error or updates of the system as seen in Figure 4.12. These boxes are timed and disappear after 5 seconds.



A	B	C	D	E	F	G	H	I
Name	E-mail	Report Type	Attendance Frequency/week	Origin Address	Destination Address	Parking Address	Yearly parking cost	Processed
Ronald	ruvelasq@uark.edu	Student (living off-cam	6	🟥	Origin can't be empty			☐

Figure 4.11: Error displaying in origin cell due to value not being present



COMMUTER MENU

BATCH PROCESSING

The following tokens in WALKING entry #1 are invalid:

[User.Destinati]

You would save [Driving.DailyExpenses] a day totaling up to [Driving.YearlyExpenses] a year by switching your transportation mode from driving a car to walking!

"Walking takes longer, for example, than any other known form of locomotion except crawling. Thus it stretches time and prolongs life. Life is already too short to waste on speed." — Edward Abbey

Figure 4.12: Error messages on web app

5 Case scenarios

The system provides two entry points: a Google Form and a web app.

5.1 Form

Users of the form are only allowed to submit a form to receive an email result as seen in Figure 5.1. On the other hand, the Office of Sustainability staff can access the web app and restart the Google Form to its default set of input fields through a menu.

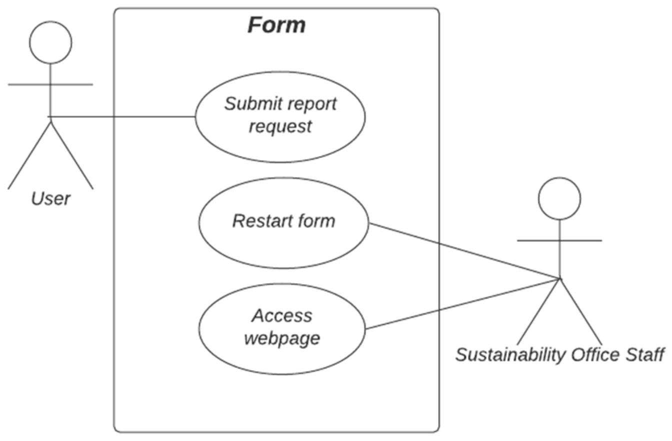


Figure 5.1: Form use case diagram

5.2 Web app

Most of the ways of interacting with the system are through the web app as seen in Figure 5.3. Through the web app, users can submit a spreadsheet. The just submitted spreadsheet and all previous spreadsheets are displayed on the home screen. From there, the user can view all the options available by right-clicking on the chosen file. It will display a contextual menu with the different options as seen in Figure 5.2. This menu allows the user to delete the file, open it, process all the entries or only the new entries. The user can also edit elements of the system such

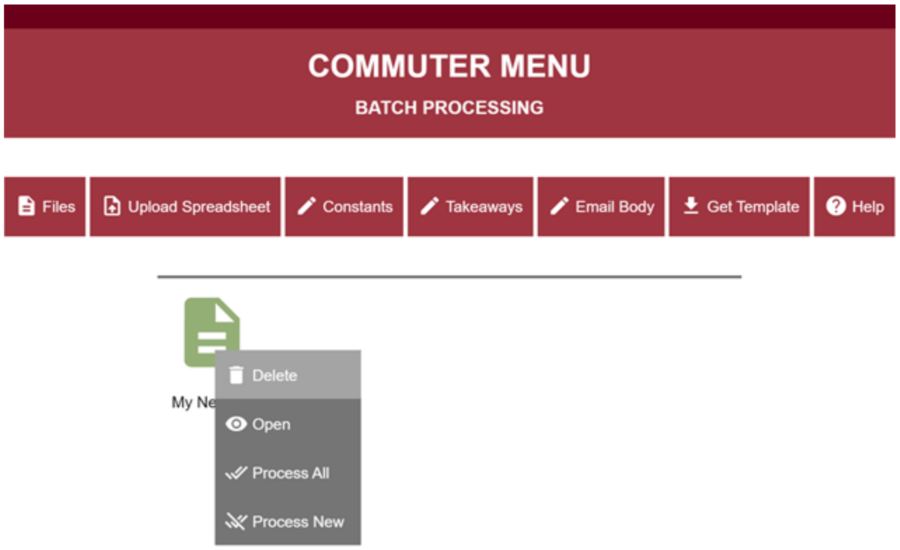


Figure 5.2: Web app home screen displaying contextual menu

as constants, takeaways, and the email body. In addition, a spreadsheet template with the correct structure for batch processing can be downloaded by clicking on Get Template. Finally, the Help section contains step-by-step information on how to use the system.

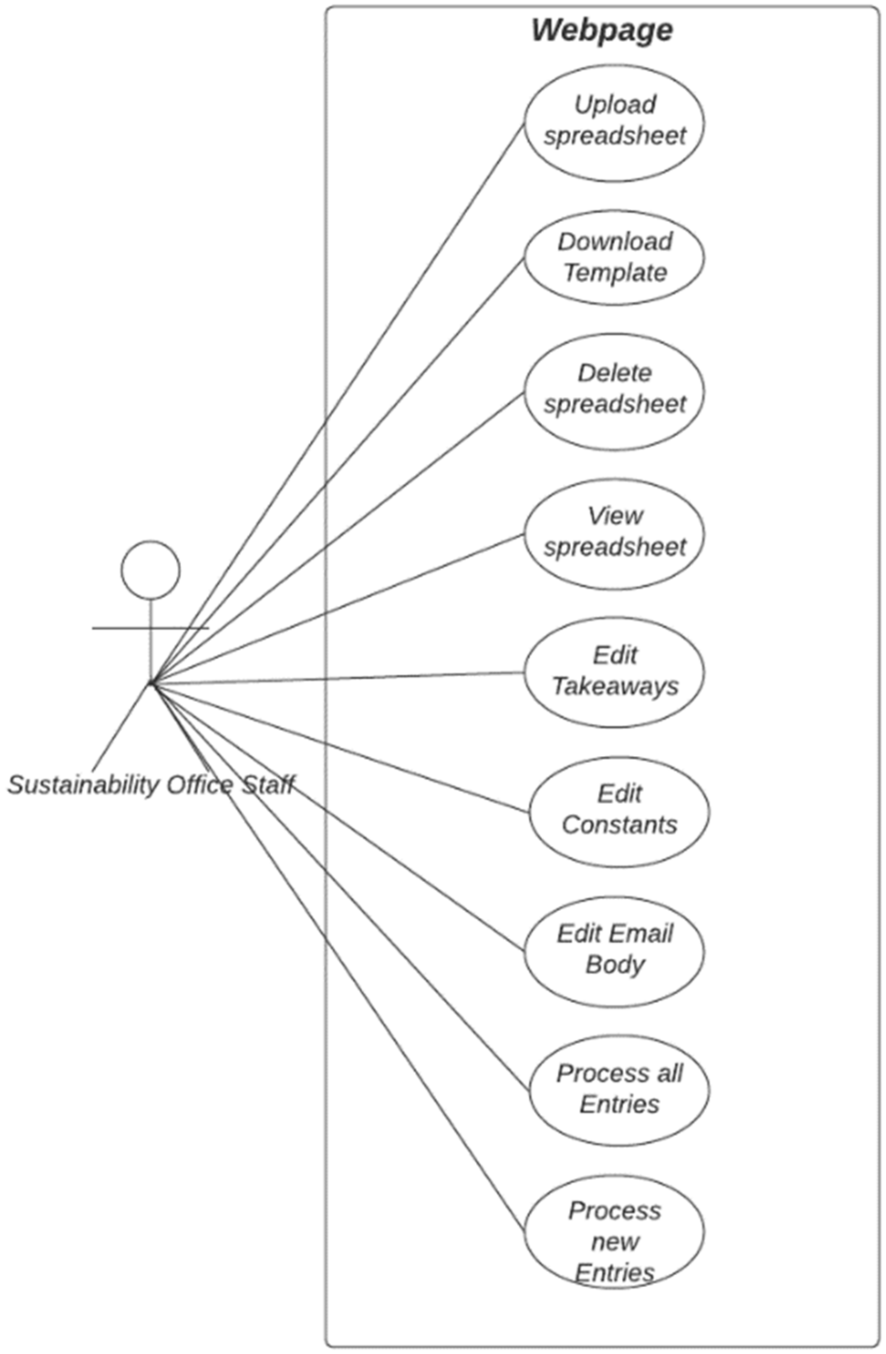


Figure 5.3: Web app use case diagram

6 Future Work

- Publishing the system. This would allow for the app to be linked to different accounts and further reduce privacy concern issues.

- Substituting Google Forms for a customized system. This system could allow implementation of the Places API to perform address completion.Also, the front-end Maps API could be used to display the start and end of the commute. This would provide the user submitting the form with a more intuitive experience which is the current system's weak point.

- Improvements on best commute heuristics. Having a system that only accounts for commute duration instead of including emissions does not necessarily select the best commute mode.

- Speed up the spreadsheet submission process. Currently, the encoding system is time-consuming. Implementing a correct base 64 encoding in the front end would speed up the process.

- Implementing the paid version of Google Maps directions API. The current version of the system implements DirectionFinder. DirectionFinder only allows for 1000 requests/day which translates to 200 reports/day. Currently, this daily report limitation is sufficient. Nevertheless, if the demand for reports increases, moving to a paid version of this API would roughly allow for 1148 reports/day.

7 Conclusion

This project is the result of following the requirements defined by members of the sustainability office. By focusing on customizability and cost reduction, the system can run without requiring constant support and at no cost. Despite the system having limitations, they do not constitute constraints in the early stages of the project. The current system demands are not high enough to require more than 200 daily reports. Consequently, it successfully satisfies their needs. Finally, if there is a need for changes in the future, the system was designed in a way that would allow for easy modifications. It is modular, and most of the system's designs are stored in templates.

Bibliography

[1] (2021, Mar.) Quotas for google services. [Online]. Available: https://developers.google.com/apps-script/guides/services/quotas

[2] (2020, Jul.) University of arkansas main campus emissions report. 1 University of Arkansas, Fayetteville, AR 72701. [Online]. Available: https://reporting.secondnature.org/institution/detail!129129

[3] U. of Arkansas Office of Sustainability, "Commute preferences survey," 1 University of Arkansas, Fayetteville, AR 72701, 2015.

[4] ——, "Commute preferences survey," 1 University of Arkansas, Fayetteville, AR 72701, 2020.

[5] (2021, Nov.) Html standard. [Online]. Available: https://html.spec.whatwg.org/multipage/input.htmlvalid-e-mail-address