GOOGLE WORKSPACE ADD-ON FOR GOOGLE DRIVE NOTIFICATIONS

By

Tailon Russell

A Project Submitted in Partial Fulfillment of the Requirements

for the Degree of

Masters of Science

in

Computer Science

University of Alaska Fairbanks
May 2022

APPROVED

Dr. Glenn G. Chappell, Committee Chair
Dr. Orion Lawlor, Committee Member
Dr. Christopher Hartman, Committee Member
Dr. Jon Genetti, Chair
Department of Computer Science

1 Abstract

This project was designed to help collaborative teams receive timely notifications on changes that occur on shared resources. The end product of this project is a Google Workspace add-on that allows email messages to be sent when a Google Drive file is changed. Other services provide this feature are known as integration tools. Some such services are Zapier, Automate.io, Integromat. The issues with such tools are that they cost money the more they are used, having a large amount of files or folders in a cloud service like Google Drive will take more resources from the integration tool to track all of them, and notifications are not timely depending on how important instant notifications are to a user. The product of this project allows for instant notifications and for many files and folders, even if they are multiple folders deep, all while only taking up resources enough to contain information on the top-level folder or file.

Contents

2 Introduction

2.1 Goal

The goal of this project was to write a program that would allow users who are either owners of a Google Drive file or those that the file is shared with to be notified when there are changes to the file. Such changes include when a file is edited, added, deleted, and when the file is downloaded. The goal in regards to the notifications was to have the user be able to specify which service they would want to be notified on, such as discord, gmail, slack, etc.

The idea for this program was born out of the need for my team to be notified of changes on our digital assets made by members of the team. My team is a podcast team and our workflow is that each host records their own audio and uploads it to a folder in Google Drive. Then one person will download those files and edit them on their computer. We also have other files where it is beneficial to be notified when they are edited. To know when these files were edited required us to communicate with each other on a separate platform. Then the question came if there was a way to automate this; thus the idea for the program was created.

2.2 Background

Services such as Zapier, Automate.io, and Integromat allow for efficient ways to communicate when files are changed and downloaded. These services are known as automation services or integration-as-a-service, where they allow users to connect different apps together in a workflow so that events in one app trigger events in another app. An example of a workflow would be connecting Dropbox to Google Drive where a file created in a Dropbox would trigger the creation of a file in Google Drive.

The pricing for each of the services as well as the target audience, and even platforms (such as mobile apps, web apps, etc) are different. Moreover, each service offers differing levels of functionality depending on pricing.

For our uses, Zapier was the best option, however the downsides of this service were that it could not monitor subfolders, and for the free option, it would only allow for change checks at an interval of 15 minutes. These downsides led me to desire to create a program that does those things better, i.e., to monitor subfolders and to perform instantaneous change checks.

To further clarify, this program is in the realm of notification-as-a-service rather than integration-as-a-service. The difference being that the former allows for more functionality than notifications on changes (refer to the example of a workflow in section 1.2).

3 Google Workspace Add-Ons

The idea for working with Google Drive files necessitated the use of the Google Drive API. However, before starting this project, the specifics of how the program was going to be run was not considered. After some research, the platform called Google Workspace Add-Ons was discovered which allowed the program to be added to a user's Google Drive, and for the program to be run continuously so that there could be instantaneous responses to file changes, and for the creation of an aesthetic GUI without much effort.

Google Workspace (formerly known as G-suite) is a service that provides collaboration tools to teams (tools such as Google Drive, Google Meet, Google Slides, etc).[4] Google Workspace Add-Ons is a Google Workspace product that allows users to integrate third-party apps to extend Google Workspace tools and allows developers to create those applications[5]. It comes with a framework for all add-on GUIs so that every add-on has a similar look and feel. Google Workspace tools also have APIs that allow developers to access each tool.

4 Competition Overview

While the following services are competitors, this project is not meant to create a replacement for any of those services entirely, it is only meant to better perform for the specific task of notifications. They are competitors in that users can use them to do the tasks that this add-on can do and other things that users may want to do.

4.1 Zapier

Zapier is the market leader in automation services. It allows various apps to be connected in workflows called zaps in which the user must allow Zapier to access their accounts. Its library of apps that can be connected is large, with Zapier boasting more options being added each week[12]. It also allows

users to connect to apps via webhooks when it does not have connections for the app. Zapier focuses on automating users' tasks so that they can focus on more important tasks, without code, and to meet any challenge[11]. Zapier has five pricing tiers: free, starter, professional, team, and company[13], and allows for billing on a monthly or yearly basis. As discussed before, the limitation of Zapier is that it cannot monitor subfolders, and for the free tier it only allows change checks every 15 minutes.

4.2 Automate.io

Automate.io actually brands itself as an iPaas solution (integration platform as a service), and claims that it is a modern iPaas solution as such solutions were costly and complicated to use in the past[2]. Automate.io boasts no coding experience necessary and instead uses a drag and drop interface, and has pre-built automation templates. Automate.io has six pricing tiers: free, personal, professional, startup, growth, and business, and it allows billing at a monthly or yearly rate[1]. The limitation of Automate.io is that the number of apps that it can connect to is less than the amount that Zapier can. However, it also allows users to connect to APIs for apps that they do not have connections for.

4.3 Integromat

Integromat also provides a drag and drop interface. Integromat boasts that it provides more than 7,000 templates to automation workflows[6]. One of the features of Integromat is that it allows for users to duplicate processed data into multiple routes[9]. For example, if a user needed to create a file in Google Drive and duplicate it in OneDrive. Something that is unique to Integromat is that it has workflows to handle errors[7]. When an error occurs during the execution of a workflow, users have the option to either ignore the error, execute new logic, or break so the error can be fixed manually. The limitation of Integromat is that it counts the trigger as a task[8, 10]. This means that a workflow in Integromat will cost more (meaning it will consume more tasks) than the same workflow in another service like Zapier for instance.

4.4 Digital Asset Management

While solutions made under this umbrella are not direct competitors, they can provide an alternative to the use of automation in this context. Digital Asset Management (DAM) is a term that is used to describe software

that allows for a centralized location for all of a business's digital assets. Such software allows for better sorting of assets through the use of metadata and better distribution capabilities. Such a software would allow for easier access and notification of users, perhaps without the need of a third-party application. Some examples of DAMs are Brandfolder, SmartSheet, and Bynder.

5 Timeline

May 8th, 2021

The idea for the project was finalized and the committee members were notified about it. The idea was to create an application that would track Google Drive changes and downloads on files and notify users whom the file was shared with.

September 7th, 2021

My first graduate talk on my project, talk #0, was given. There were two ideas that were discussed: creating a Google Drive plugin or using GPT3 to create computer-generated promotional text for social media posts.

September 13th, 2021

The direction for the project was chosen, that direction being creating a Google Drive plugin. It was also determined that what will be developed is a Google Workspace add-onfor Google Drive. After this, time devoted to building the plugin by incorporating cloud functions and the Google Drive API.

October 12th, 2021

My second talk on the project, talk #1, was given.

October 15th, 2021

The decision to develop a companion app for the add-on was made. The reason for this was that the tracking of when downloads occurred could not be done because the API that was needed to track such things was only able to be used by administrators, a special Google account within the University organization that needed approval to be a part of. Thus, the decision was

made to track downloads by having the add-onallow users to download files through the companion app and then the app would change the metadata of the file after it was downloaded.

November 9th, 2021

My third talk on the project, talk #2, was given. During this talk, a deadline was set to implement the download tracking feature using the companion app by the end of the year 2021, otherwise this feature was going to be abandoned.

January 1st, 2022

Implementation of the download tracking feature failed to be done before the start of 2022. This feature was abandoned so more time could be devoted to finalizing the rest of the functionality.

January 21st, 2022

Finished implementing the use of Cloud Run instead of Cloud Functions. This involved creating a docker image and pushing that image to Google Cloud. As well as using Flask to define the add-on's paths.

February 9th, 2022

Reached my minimum viable product (MVP). Notifications for file changes worked and users were able to select files and enter email addresses. Those selected files would have channels created to watch for them.

February 15th, 2022

Finished updating the homepage to allow users to select whether they want to track files or if they want to track folders. Depending on what they click the next card would display a list of files or a list of folders.

February 27th, 2022

Finished implementing finishing touches such as customizing the add-on's icon and properly implemented Flask and a Procfile for use in Gunicorn.

March 1st, 2022

My fourth and final graduate talk, talk #3, was given.

March 25th, 2022

The semi-final version of this paper was submitted to my committee.

April 15th, 2022

The final version of this paper was submitted to my committee.

6 Technologies Used

The tools used for implementing this add-on were the Google Cloud, the tools within it and its SDK, Google Workspace Add-Ons, Google Drive and its API, Gmail API, Pycharm, Miniconda, Python and Flask, and Google Datastore and its API.

6.1 Google Cloud

This tool is a web-based interface where the services used for authentication, logging, and various others, are made available. This is also where the code for the add-on is hosted, as well as where the project associated with the add-on is created.

In order for authentication to work, Oauth credentials and a consent screen needed to be created in Google Cloud. The consent screen allows users to have a GUI for authorizing the add-on to have access to their data. Oauth credentials allow the add-on to access the enabled APIs.

Google Cloud is where APIs are enabled so that they can be used in the project. The Google Drive, Gmail, and Datastore APIs were enabled for this project.

When code is pushed to Google Cloud, the logs of which are stored in Google Cloud logging, which was useful for debugging and printing output to verify aspects of the add-on were working correctly. A Google Cloud product that was also useful for debugging was API Error Reporting.

Google Cloud is also where the service account used in the project was created. The service account is authorized to allow the app to do things.

For example, when the homepage was just a list of files, it needed to acquire a list of the files in my Google Drive account, the service account authorizes the add-on to do this.

6.2 Google Cloud SDK

The Google Cloud SDK provides libraries and tools for interacting with Google Cloud products and services. This needed to be installed on my machine so that gcloud commands could be used in the terminal. Gcloud commands allowed me to push code to Google Cloud, deploy my container image. The SDK provided access to packages that were needed to interface with the APIs.

6.3 Google Cloud Run

This Google Cloud product allowed docker images to be pushed, so that the add-on could be containerized and updated quickly.

6.4 Google Workspace Add-ons

This is the framework used to develop the add-on. It also provides docs on how to develop the cards (the term used to describe the pages of the add-on) and the files needed so that the add-on could be identified by Google Workspace. Google Workspace Add-Ons, on the user side, allows the app to be added to a user's Google Drive so that they can access it.

Google workspace Add-Ons uses the deployment.json file to determine which URL to call when a homepage trigger is fired and when a contextual trigger is fired. A homepage trigger is simply the URL for the add-on called when the add-on is opened and no file is selected (the card for this trigger is the homepage). A contextual trigger is a trigger that is fired with a certain context, when a file is selected for instance. The trigger is defined in the deployment.json file depending on the Google Workspace tool the context is in. For example, the contextual trigger for selecting a Google Slide file is defined separately from the trigger for selecting a Google Drive file.

6.5 Google Drive API

This API provided methods that were used to acquire a list of files, as well as creating a channel to watch file changes, and acquire recent changes.

6.6 Gmail API

This API allowed the add-on to create and send emails.

6.7 Datastore API

This API allows the add-on to store tokens so that they would not need to be created each time a function is run. It is also where resource_ids are stored for created channels, and email addresses to be notified for a specific file.

6.8 Python

Python is the programming language chosen to write the add-on. This was solely based on personal preference.

6.9 Pycharm

This was the chosen IDE to write the add-on in. It was chosen because of personal preference towards the UI for Jetbrains products and previous success in integrating Miniconda with it.

6.10 Miniconda

Miniconda is a lighter version of Anaconda which allows developers to create Python virtual environments. It was chosen because of previous success in integrating it with Pycharm. Virtual environments were used to avoid running into the possibility that installing and uninstalling packages could break my machine.

6.11 Flask

It is a microframework that allows developers to easily create web applications. For this project, it was used as a way to create paths in the add-on. When a user opens the add-on there is a post request made to the URL provided to the deployment.json file, that URL is defined as the URL for the container image and the path defined in the flask app appended to it.

7 Implementation Narrative

7.1 Major Components

The major components of this project are the user interface, the notification trigger, the Google Datastore, and message delivery. When the user opens the add-on they interact with the interface. If they have no files or folders selected, then the interface shows them the list of folders they can select and the list of files they can select. It then takes them to a card (the term used to describe the pages of the add-on) where they can enter any email addresses they want notifications to be sent to regarding the selected files or folders. Going back a few steps, if the user opens the add-on and they do have a file or folder selected, the interface shows them the card where they can enter those email addresses. The final step in the interface is when the user presses the "submit" button and then the tracking begins, which is the domain of the notification trigger.

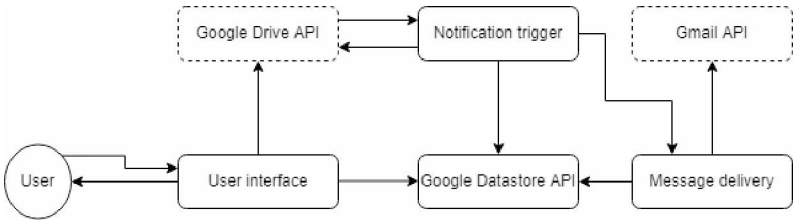
The notification trigger is a function in the add-on which is not user-facing. This function communicates with the Google Drive API, where the API sends an empty-body message which signifies that there are new changes. When the notification trigger receives the message it communicates with the Google Drive API again to get the list of changes that have been made. Then the notification trigger communicates with the Google Datastore and the message delivery components. The notification trigger uses the key of the desired Google Datastore entry to retrieve the information. The notification trigger then gives the message delivery component the file id and name of the file or folder that has been changed; whether the file has been added, edited, or trashed; and the parent id (the file id of its parent) if the file itself is not being tracked (as such is the case when a folder is being tracked and a file within that folder has been changed). The notification trigger determines if the file or folder has been added, edited, or trashed based on the trashed attribute in the metadata of the file or folder, and the createdTime attribute.

The Google Datastore is a database that contains the list of email addresses associated with the files and folders that are being tracked, tokens needed to authenticate the add-on to do actions on behalf of the owner of the project (this is not important to users), channel information needed in the event that the user wants to stop tracking, and page tokens needed for the notification trigger to request the list of changes. When the add-on is opened it needs to be authenticated, if there are no valid tokens in the

database, then a new one is created. When the user clicks the submit button, the email addresses that they gave are added to the database along with the file id of the file or folder being tracked. When the list of file changes are retrieved, the new start page token is stored in the database which overwrites the previous token. Whenever any of these resources are needed the components either make a query to this database or get the entity directly if they know the entity's key.

The message delivery component sends an email message to the desired email addresses with the given subject and text. It first must retrieve the email addresses from the Google Datastore so it communicates with the database. If the message delivery component was given a parent id, then it retrieves the email addresses associated with the parent id, as the information on the specific file or folder that has been changed will not be in the database, but the parent will.

The image below is the architecture diagram for the major components.



7.2 Project Phase: Pre-Planning

There was a lot of Google searching and guessing-and-checking done for this project. The first effort was on finding an easy-to-follow document that would be an introduction to using the Google Drive API. The Quickstart for using Javascript was found which hand instructions for first creating authorization credentials for a desktop application. Then the instructions gave steps to create an "index.html" file by copying and pasting the code provided. This was done in a Google Cloud Shell folder. The instructions said to start the server, but an error occurred with this happened saying there was no valid origin for the client. After some googling, it was found that the reason for this was because desktop authorization credentials are only for non-javascript projects (this was stated on their docs at the time but it is not there anymore). Therefore, Oauth credentials for a web application were created.

Then Miniconda and Pycharm were installed. This was used because the target programming language for the add-on was Python and and due to prior positive experiences using it. A virtual environment was desired because if something went wrong with using "pip" (specifically "pip uninstall") my machine would not be adversely affected, and Miniconda was chosen because of prior positive experiences using it.

The next focus on on following the Python Quickstart instructions, which provided the code that was then copied and pasted into the project an a running of the program was attempted, which failed. The reason it failed was because the credentials.json file was not included in the project folder. The instructions gave steps on how to do that, but after that another problem occurred. It turns out that since the Oauth credentials created for the Javascript code was being used but the project was no longer incorporating Javascript, the non-valid-origin-client error appeared again. The code ran successfully and with the correct output after Oauth credentials were created for a desktop application.

During the development of the application, an error kept occurring that was something along the lines of missing "GOOGLE_APPLICATION_CREDENTIALS". The error presented a URL to learn more about this. The article walked through creating a service account and creating the service account key. Then the article talked about adding the path of the key to the machine's environment variables, but this kept failing. Turns out, that the location of the key needed to be added to the environment variables field in Pycharm in the configuration settings.

Up to this point, development was solely on a Google Drive API desktop application, but then Google Workspace Add-Ons was discovered. This discovery answered the question of how the application was going to run. Was it a desktop application that will run in the background? Will it work on the browser? Google Workspace Add-Ons provides a framework for developers to develop third-party applications, and for me, a way to host the application so that anyone can simply add it to their list of Google Workplace apps (the term used for the apps created for Google Workspace) and it would be able to do its job there. At this point, the decision to create a Google Workspace Add-on was made.

7.3 Project Phase: Developing the Addon

After this discovery, the development of the add-on began. The Google Cloud Shell Editor was then not being used in the event that internet connection was lost, development still needed to continue. Then after some research, it was discovered that the Google Cloud SDK could be used to push code to a Google Cloud project. There was then a medium article on how to install it on a machine for use with Pycharm.

After a little bit of finagling Github to push to Cloud Run (which ended up being unimportant), the focus shifted to following the instructions for building an add-on in alternate runtimes. Google Workspace Add-Ons can be created in two ways: through Apps Script (Google's own coding language for Add-Ons) and alternate runtimes (like Javascript, Java, Python, etc). The quickstart employed cloud functions so that the code could be run but also so that the deployment.json file could have a valid URL for the entry point (which is a URL to the function, referred to as a trigger). The deployment.json file contains the directives for what URL to fetch when the add-on is opened to the homepage (this function is referred to as a homepage trigger) or for contextual triggers (a function called when a file is selected in a Google Workspace tool such as Google Drive and the add-on is open). After following the quickstart, the add-on was successfully created with very crude functionality: it could be opened to display a random picture of a cat. First there were two errors: the function which is run did not have the required parameters, and there were fields in the deployment.json that were in the wrong place. The alternate runtimes quickstart is for Javascript, so there was some guessing on how to make it work in Python. Eventually documentation was found that showed that in the Python runtime, a trigger takes one parameter rather than two as it does in Javascript. The deployment.json file contains json describing directives for multiple Google Workspace tools, and for each tool there are unique attributes within them. First the deployment.json file was constructed with some attributes in the wrong area, which caused an error.

The focus then moved on to following the quickstart for the Gmail API. It quickly realized that the instructions did not include sending emails, so research was done in API documentation for how to do so. Once the correct documentation was found, an error kept occurring when following the instructions that prevented the sending of emails. Eventually it was figured out that the problem was that the email needed to be a string rather than what it was, which was a MIMIText object.

Next came tackling the issue of storing and retrieving tokens. Up till this point, the tokens had been created using the Oauth Python package and they were in the form of a file called token.json (this was how it was done in the quickstarts). However, it was known that the tokens needed to be saved somewhere in the cloud, so the first solution was to try storing the information in App Engine. However, an issue persisted where files could not be saved anywhere except in a temp folder, but this could not be figured out, therefore google-searching was conducted on how to achieve the goal via other methods. Eventually a decision was made to use Google Datastore, a database that is less functionally rich than Firestore. This was chosen because there were docs for using it, it was simple, and it was free.

Once the functions for storing and retrieving tokens from the Google Datastore were written, the token file became unnecessary. Then the development of sending emails was finished. At this time, emails were sent using the contextual trigger.

Eventually it was discovered that the code had occasions where different versions of the Google Drive API were used (sometimes it was version 2, other times version 3), so to make sure that the add-on could be the most current it could be the most recent version was used (version 3). This broke the code for file tracking, which resulted in an the error saying the method changes.watch of the Google Drive API in version 3 had a required parameter that it did not require in version 2: a page token. This was rectified by using the getStartPageToken method in the Google Drive API to get the page token of the most recent changes and then calling changes.watch.

The first time the homepage was updated it showed a list of Google Drive files and a switch next to each file name. The idea was that when a user turns the switch on then the channel will be created for that file. More research was required and the "build an add-on in any coding language" page was found which had a bunch of examples of common UI elements or functions used in cards. The way cards are structured in Python is by JSON objects, necessitating the use of JSON attributes to structure the switches and the file names.

When finalizing the sending of emails, researched was done on how to acquire the email addresses of Google Drive users whom the file is shared with and it was discovered that this may not be possible as Google Drive keeps that information hidden. Rather than putting in effort to get the information on shared users, another card was added to the add-on that would

allow the user to add the email addresses of individuals that they would like to be notified on changes to files. With this functionality implemented, the project was approaching a minimum viable product (MVP).

It was during the research on trying to acquire information on users whom a file is shared with that a note that was written at the top of the quickstart for building an Google Workspace add-on in any coding language was heeded. The note stated "This quickstart is intended for example purposes only. To build a feature-rich add-on outside of Apps Script, we recommend you use Google Cloud Run instead of Google Cloud Functions." So far, google cloud functions had only been used, but after that reminder, research was conducted to figure out what it would take to transition to Google Cloud Run. What helped out the most with this transition was the quickstart for building a Google Workspace Add-on with Node.js and Cloud Run. While this quickstart was not written with Python being the runtime, implementing the equivalent things in Python was figured out. The main difference was that in Python, Flask was needed to implement the app routes. This was great for one reason: a domain did not need to be registered nor was an html tag needed to be included in the code. This will be explained further in section 6.5.

Some further issues that were encountered while doing this (other than the one described above) was figuring out how the routed functions are called, i.e., which request method. Initially, routes were only accessible via a GET method, but this was not how the add-on calls the functions, which is via the PUT method. Another issue was that all URLs referenced in the add-on needed to be changed to to use the URL of the container image.

Development on the homepage began a second time. The way that the list of files is acquired is by using the file.list method of the Google Drive API, and the response that is returned contains information including the files and the next page token. It was realized at this time that page tokens were not being utilized and therefore not every file in the users' Google Drive was being acquired. Therefore, pagination was implemented to allow users to get more results and to go backward to get previous results.

7.4 Project Phase: Trying to Track Downloads

It was decided after talk #1, that focus needed to shift towards the tracking of downloads, because research had shown that the Google Drive API does not monitor downloads, and if this was not going to work, then

that would affect the choice in the platform to develop the add-on for (at this time there was consideration in developing an add-on for Dropbox in hopes that its API would offer a way to monitor downloads) and it was early enough in the project that this could be changed without considerable cost. After researching to see if Dropbox was a better idea, it was determined that it was not. Therefore, focus shifted to figure out how to make download tracking work for Google Drive. The fruit that was born from that effort was that the Reports API of the Google Admin SDK was thought to allow for the monitoring of files. However, effort in this direction was stopped due to the fact that admin credentials from within the organization were needed to use it, that organization being the school-wide Google account, admin credentials for which was unlikely to be granted. Therefore, the focus changed to create a companion application for the add-on. A medium article walked through creating a desktop application with Python and PyQT. The application was developed to the point that it was able to access the Google Drive API and display all the files in my drive to the screen.

The idea for the companion app was that since the Google Drive API could not be used to track downloads, that a companion application could be used to implement said functionality. For this idea to work, another card would be added to the add-on that would allow users to click a download button and then have the file downloaded to their machine. The companion app would then know when the file was downloaded. It would then make a call to the Google Drive API to edit the file's metadata with data on who downloaded the file. In practice, this would then send a notification and then messages would be sent to users whom the file was shared with about who downloaded the file.

A deadline was decided in talk #2 about when to have the downloading functionality working. The deadline was that if download monitoring was not finished by the end of the year 2021, that the functionality would be abandoned in the final version of the add-on. At the time, it did not have more functionality than accessing Google Drive API and displaying file names to the screen.

It was soon realized that there was something that needed to overcome: how was the companion application going to communicate with the add-on. Eventually it was settled that using the Requests package in Python to make requests to the cloud function would be the way to go about this. Then it was realized that this would not be a continuous connection; it would

end as soon as the function returned a response. Then effort was made to create a server that had one endpoint on the URL for the cloud function and the other endpoint on the application. The first step in that direction was creating a server in a different Python application and then having the desktop application connect to the server, which did end up working. After this success, effort was made to create the server using the URL for the cloud function, but this effort failed. Then it was discovered that if development was going to continue on the idea of opening a connection, then the cloud function was actually what needed to open a connection with the desktop application. This effort also failed. Then came the new year, 2022, and the decision to abandon the downloading aspect of the add-on as well as the desktop companion app was made.

7.5 Project Phase: Getting Notifications

This aspect focused on being notified when a change occurs in a file. The first effort was on retrieving changes, which is done using the changes.watch method, discovered only after a frustrating back-and-forth between the same Google Drive API docs and then clicking on a link in the header titled "reference" which finally had the answer, which was how to retrieve changes. At the bottom of the page, was code for how to use the changes.watch method to create what is referred to as a channel, which is like a subscription to a specific resource, the resource in this case is changes.

The first issue encountered when implementing the changes.watch code was that the value needed to give the address attribute of the request body was unknown. The address attribute is simply a URL that the notification will be sent to. The issue with this is that it was unknown what to put for the URL, there was nothing in place that had a URL associated with it. After a lot of google searching, a medium article was discovered that walks through how to use a cloud function to create a webhook for instant notifications. The process for this involved registering a domain in Google Search Console using the URL for the cloud function and then adding an html tag to the file containing the function definition. A function called trigger had already been created within the project that would gather all the recent changes when the code was run, and this was the function that needed to be run when a change occurred on a file, therefore it was the URL for this function that was registered as a domain to Google Search Console. Once this was done, the URL for this cloud function was used for the address field of the request body. Then the trigger function and the

function for creating a channel (called create_channel) was pushed to Google Cloud, and then when the code was ran a channel was created.

Previously, the method for creating and sending a email messages was implemented, so all that was necessary to do next was work on the workflow to get the information ready for a message to be sent. Starting when the user clicks the submit button on the item tracking card (the card where users enter email addresses), the add-on would save the email addresses to the Google Datastore and would create channels for each of the files selected, the address of which was set to the address of the container image followed by "/trigger." The "/trigger" was a path to the function called trigger (the same URL used to create channels), which would then call the function that handles the sending of emails. Then, whenever a change occurs on a watched file, a request to the trigger function would be made and then that function would get the email addresses associated with the files from the database and then another function would send a message to those email addresses. Once it was verified that this was working, by tracking a file and making a change to it, the project had reached its MVP.

7.6 Project Phase: Post-MVP development

The first two things developed after the project reached its MVP was the contextual card and the tracking of folders separately from files. First, the contextual card. At this time, the contextual card was the same as what it was when development was focused on the download functionality. This meant that it displayed the file name, the icon URL, and the download button. The new idea was to allow users to click on a file in their Google Drive and then set up tracking that way, which meant that the contextual card needed to be changed to be the item tracking card.

After the contextual card was updated, development focused on tracking for a folder. The first effort made was to figure out how to divide the files into folders and non-folder files. Google Drive defines everything in terms of files, so a folder is a file type that only contains metadata, whereas most other files have the type of blob, which is any file that contains binary text such as images, videos, and PDFs, and there are more file types than these. How this was accomplished was by filtering the files into files with the type folder and files that are not a folder type. Then two cards were created, one that displayed the list of folders and another that displayed the list of files, and updated the homepage card to allow users to select whichever card they desire to go to.

Then came the actual tracking of folders. First it was tested if creating a channel to watch the folder would result in getting notifications about files inside the folder; however, this was not the case. So, then the decision was made to create channels for the folder and then all the files inside that folder. This worked, however it was soon realized that if a user were to create a new file, that file would not be tracked. In response, an additional channel was created that just watched for all changes in the Google Drive and then have the address for that channel be a different path inside the add-on. After that was working, it was quickly realized that since tracking is being done for all changes, it was not necessary to have channels for specific files, a single channel could be created for all changes and then emails were to only be sent if the change occurred on a tracked file. The saving of channel information was implemented during this development.

Finally, it was down to implementing the finishing touches to the add-on. A button was added to the homepage labeled "stop tracking" which would take the user to the Stop Tracking Card and allow the user to choose which folders or files to stop tracking. This was implemented with mostly the same code as the item tracking card, except the list items are only the tracked files. Another way for users to stop tracking was added via the contextual trigger which is a button labeled "stop tracking" next to the "submit" button.

The next finishing touch that was implemented was how the email addresses and channels were stored. Other file information needed to be stored such as the file name and it was decided that since Google Datastore API calls could be minimized by including the email addresses to be notified in the file information. Storing channel information was changed to include the expiration of the channel, so that channel expiration could be checked. If a channel did expire, then the channel would be recreated.

Along with these changes the tracking of folders and their children were updated, along with changes to how they were stored. It was realized that if channels were created that tracked all changes for a folder and all of its children then the add-on would send multiple email messages about the same event. For example, if file_1, which lives in folder_1, was changed, then a notification would be sent due to the folder_1 channel, which recognizes that file_1 is its child. At the same time, a notification would be sent due to the file_1 channel because a change happened on it. This was unnecessary, so it was decided to only create one channel which tracks all changes.

The next bit of changes implemented were adding a message to the user

about how tracking folders work, updating how acquiring recent changes worked, and basing the email message text on what happened to the file. The message to the user was simple, a decorated text widget was added to the list cards and the item tracking card. Updating how acquiring recent changes worked was a little more involved. In order to get changes a page token must be provided. The way this was being done was by getting the most recent page token (via the getStartPageToken method from the Google Drive API) and subtracting 1 from it. The reason for this is that when providing a page token and getting the changes that have occurred, those changes are the ones that happen after the page token. Think of this as a book, with words being written up to page 11, the most recent page token in this case would be page 12, which is empty. So the idea was that by subtracting 1, the add-on would get the previous page of results, it would get page 11, so to speak. However, this is not the case, it is more like getting the previous paragraph. When there were many changes that occur, only a few of the changes were noticed because they would happen too quickly that by the time the most recent page token was retrieved, there were changes that occurred prior to it that did not have notifications sent for them. All this to say, acquiring recent changes was updated to be done by storing the next page token of the current change and then when getting the recent changes, the add-on would get the page token stored in the Google Datastore, rather than calling getStartPageToken.

Updating the email message text involved searching for whether the file was trashed (in the trash can) and when it was created, and adjusting the email message text accordingly. The hardest thing needed to implement this was converting the time given by the Google Drive API, which was in human readable format and in UTC time. The human readable format needed to be converted into a Python datetime object in the UTC timezone and then converted to the system's local time (making sure that it was not just converting to AKST in the event that someone in another timezone would use this add-on).

Other changes made were editing the name of the container image and updating the add-on icon. When the transition to using Google Cloud Run was made the default name "helloworld" was used for the container image, so this was updated to be more specific. In doing so all the URLs in the code needed to be changed to use to the container image URL. To update the add-on icon, an icon was made and was saved as an image in the project. Turns out that the "logoUrl" attribute in deployment.json needed a proper

url, and not a path to a file in the project. To accomplish this a path within the add-on was added for a function that returns the image using the "send_from_directory" method in the Flask library and a directory was made for the image to live in.

The final two aspects that were updated in the project were properly serving the add-on and tracking for sub-directories. During the transition to Cloud Run, there was an article that aided as it was addressed to help in developing a Flask app for Cloud Run. In that code, there is an if statement for checking for the main function, this was where the code for recreated a channel once it was expired originally resided. However, it was not being run. What was failed to be realized was that in the quickstart for Python and Cloud Run it was noted that that code should only be used for development purposes, and that for production a Procfile is needed. So, the code was updated accordingly, and once that was finished the checking for channel expiration worked.

The tracking for subdirectories was originally discovered as a bug, that is the lack thereof. The bug was that if there were several levels of folders (folders within folders) then any change on a file more than one level down would not have messages sent about it. This was because in the code was only checking for the immediate parent of the file, and not its ancestors. This meant parent of the file had to be found and if that parent was not being tracked, the the parent of that folder had to be found, and so on and so forth until a folder that was being tracked was found or none at all.

8 Project Status

The project is finished. The core of the add-on does implement most of the original idea. Messages can be sent and are specific to what happened on the file. A user can add the emails they want the message to be sent to. Tracking of files within directories multiple levels down is supported. There are some things that have not been implemented, namely the monitoring of downloads. Other aspects of the goal that was not implement were the sending notifications to applications other than Gmail, as well as sending notifications to users whom the file was shared with, as that information is not something that Google provides to developers.

9 How to Use This Project

The following instructions are for developers who want to work on this project in the future. In section 8.1, the instructions will walk through cloning the project on their machine and creating the necessary accounts within Google Cloud and configuring the correct settings on their machine in order to commence development. This section will also give instructions for installing the add-onon their Google account. Once the initial set up is done, then developers can follow the steps in 8.2 to make changes to the code and push those changes to their Google Cloud.

9.1 Set Up

Once these instructions are completed, developers should be able to see the add-onon all their Google tools such as Google Drive, Google Slides, etc (some tools may not have a right sidebar, like Google Meets, and therefore the add-onwill not appear in those tools).

1. Clone the repo from my git repository: `https://github.com/TailonR/Google_drive_addon_masters_project.git` and open it with Pycharm or some other application to write Python in.
2. Create a Google Cloud account and create a project.
3. Enable the Gmail, Google Drive, and Datastore APIs
4. Follow the steps found in https://developers.google.com/workspace/guides/create-credentials to create OAuth 2.0 Client ID and a service account.
   (a) Create an OAuth 2.0 Client ID for a desktop application
   (b) Download the ID to the machine, rename it "credentials", and move it into the Backend directory of the project
   (c) In the "Grant this service account access to project" section, assign it the role of "owner" which can be found in the basic option of the dropdown
   (d) In the "Grant users access to this service account" section assign a email of a gmail account
   (e) When creating the service account keys, download them to a machine and add their location to the environment variables on either

5. Install the Google Cloud SDK, instructions for which can be found in https://cloud.google.com/sdk/docs/install

   (a) May need to add it to the PATH environment variable

6. Follow the instructions on step 4 of the "Build a Google Workspace Add-on with Node.js and Cloud Run" quickstart, starting on the section titled "Build and deploy the add-on backend" and end at running the command under the "Install the add-on for testing"

   (a) After running "gcloud run services list –platform managed" paste this URL to replace each reference to the container URL in the code.

      i. The affected files will be deployment.json, channel.py, and cards.py

      ii. This is only the base url, keep anything past the last "/"

   (b) Any references to "todo-add-on" can be replaced with any name

9.2 Development

Follow these instructions to push changes to the code to Google Cloud.

- Run "gcloud builds submit" whenever a change to the code is made,
- Run "gcloud workspace-add-ons deployments replace name-of-deployment –deployment-file=deployment.json" whenever a change to deployment.json is made
  - Replace "name-of-deployment" with the actual name

10 Future Work

This section will discuss areas of future work, starting with the simple additions to more extensive changes. Firstly, implement a better way of storing OAuth 2.0 tokens. Currently, when an OAuth token expires a new one is created and then is stored in the datastore, but previous tokens are not removed. Therefore, to minimize the amount of data being stored in the datastore, it would be best to only keep the most recent OAuth 2.0 token. Secondly, make the add-on extensible for other Google Drive events. Such events could include when permissions on the file change, or when the file's metadata changes, etc.

It would be convenient to have a way that users can add to the emails of files that are already being tracked. Currently, if a user wanted to add an email to the list of emails to be notified on file changes, they would have to enter all the emails they would want to be notified, the new email address and the previous email addresses. In lieu of this change, a note could be added that tells the user that they would have to enter previous email addresses along with any new email addresses they want to add.

Another area of future work is to add a time delay between email messages about files being edited. Currently, an email message would be sent whenever Google Drive has a chance to save the changes (the note on the top left that says "Saved to Drive"). This results in a lot of emails being sent. Perhaps an alternative to this, although it is uncertain if it would be easier, would be to send messages with a specific label attached to them, this way in the user's inbox would just see the name of the label and if they wanted to see the message they would click on it.

In order for non-developers to use the add-on, or anyone who does not have access to the Google Cloud project, the add-on needs to be published. It is unknown how extensive this would be as publishing the add-on was not the focus of the project. However, before any major changes occur (i.e., the future work that will be discussed in the next few paragraphs), the add-on should be published to see how it would perform when it is more widely-used.

More work could be done to get downloading of files working. The idea that was considered during the final push to get downloading working was to give the user the download url for the file, so the app can monitor when downloads occurred and alter the metadata of the file once the download was completed. It in unknown as to whether or not this is the approach that will work, but assuming it is, there is a video that discusses how to make that download link. That YouTube video is found on the following link with the timestamp where the speaker starts talking about it: `https://youtu.be/1y0-IfRW114?t=1303`.

Once downloading is working, altering the metadata included, then the next area of work would be to alert others that the file was downloaded. Followed by deleting the file once the download is completed, or some other response to a completed download.

For the final two areas of work, these were ideas that were tossed around in the beginning of development before a clear direction for the add-on

was found. However, implementing these changes would be an interesting challenge. The two changes: sending notifications to other applications and making the add-on extensible for new applications. Sending notifications through other applications is something that was the goal in the beginning but development was not made for it. The idea behind this is to allow users to choose which service they would want to receive notifications in, such services could include Discord or Slack. It is unknown how this would be done, but there is an article on the Google Workspace for Developers website[3], that discusses connecting to non-Google services, which is most likely a good place to start.

The final area of work is making the add-on extensible for other applications would most likely require an entire overhaul of the existing code base as the way that Google Drive specifies their add-ons to be made will most likely be different than other applications; like Dropbox for instance. It could be possible that multiple different add-ons could be made for each specific application, which would allow for the existing code base to persist.

11 Lessons Learned

The first lesson learned during this project was that no matter how simple a project may appear, there are things that can make the project complicated. It first came down to how the app was going to be build, and when the decision was made to use Google Workspace there were implications along with it such as how to represent the UI elements, the fact that a lot of the documentation was written for Google Apps Script and figuring out how to translate that information into Python. Figuring out how to authenticate the add-on and creating service accounts were also some things that needed to be figured out, as well as how to get the Google Cloud SDK working on my machine.

Another lesson learned during this project is the difference between a gmail account and a Google Workspace account. A Google Workspace account requires a domain for an organization, which means registering a domain was required, which is not free, in order to create the account. Then in order to get the account to work, Cloud DNS settings on Google Cloud needed to be set up, which did not end up working.

It was reinforced that "pip uninstall" should be avoided at all costs and that virtual environments are awesome. There came an issue where it

confusing with all the Google Cloud SDK accounts that were created in the course of the project. Prior to this, a package that was no longer needed was uninstalled using "pip uninstall" which caused things to break that were not broken before. Since this project was done in a virtual environment, a new virtual environment was created and it was assigned the to the project.

Moreover, a lesson learned was that documentation is great when it fits a need but is not helpful when it does not. While the Google API docs did have a lot of information about using Python with its APIs, there was also a lot of information that essentially needed to be transcribed over to Python because the documentation was for a different programming language. What was discovered is that even if the docs were not for the language being used, they were still helpful for pointing in the right direction.

References

[1] Automate.io. *Pricing*. URL: https://automate.io/pricing.

[2] Automate.io. *Work super smart with automate.io*. URL: https://automate.io/.

[3] *Connecting to non-google services from a google workspace add-on — apps script — google developers*. URL: https://developers.google.com/apps-script/add-ons/how-tos/non-google-services.

[4] Google. Accessed Feb. 28, 2022 [Online]. URL: https://workspace.google.com/.

[5] Google. Accessed Mar. 1, 2022 [Online]. URL: https://workspace.google.com/products/add-ons/.

[6] Integromat. *Achieve more in less time with fewer people*. URL: https://www.integromat.com/en.

[7] Integromat. *Automatic error handling*. URL: https://www.integromat.com/en/feature/error-handlers.

[8] Integromat. *Counting the number of operations help docs: Integromat help center*. URL: https://www.integromat.com/en/help/counting-the-number-of-operations.

[9] Integromat. *Features*. URL: https://www.integromat.com/en/features.

[10] Integromat. *Pricing parameters help docs: Integromat help center*. URL: https://www.integromat.com/en/help/pricing-parameters.

[11] *What is Zapier and How Does Zapier Work?* URL: https://zapier.com/how-it-works.

[12] Zapier. *Connect your apps and automate workflows*. URL: https://zapier.com/.

[13] Zapier. *Plans & Pricing — Zapier*. URL: https://zapier.com/pricing?gclid=.