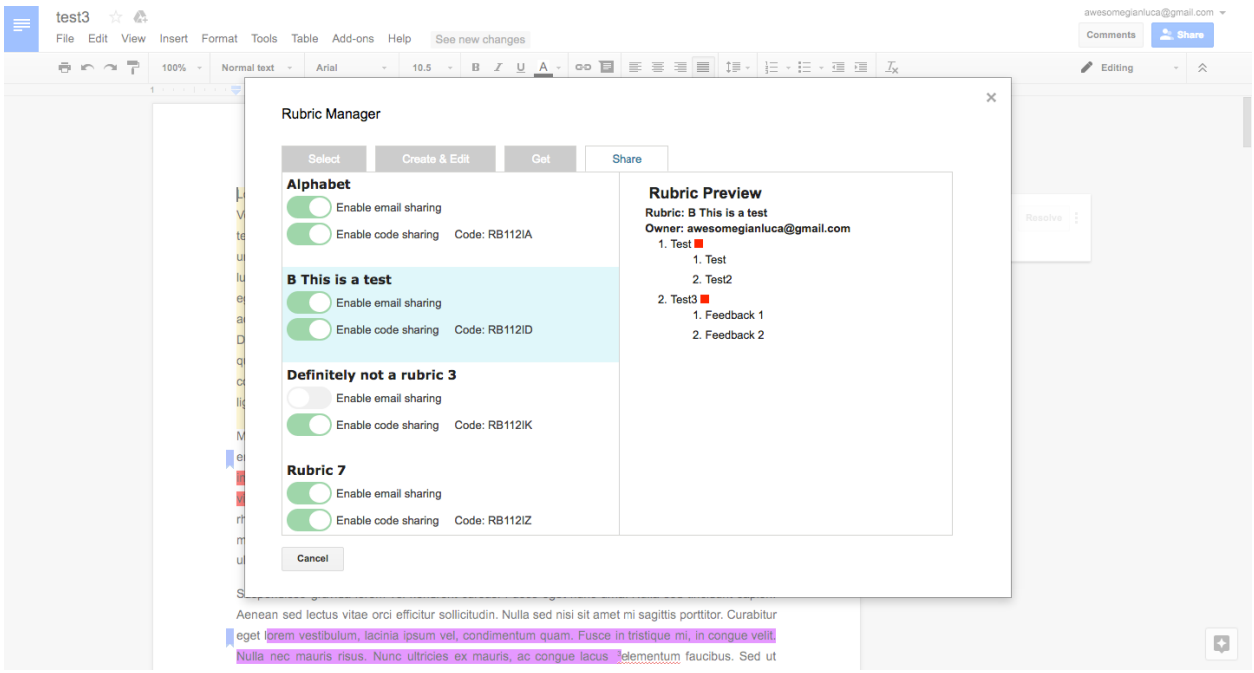
Adding to docASSIST: A Google Docs Add-on



By Gianluca Tarquinio

Adding to docASSIST: A Google Doc Add-on

An Interactive Qualifying Project Report Submitted to the Faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the Degree of Bachelor of Science

By

_____
Gianluca Tarquinio

Submitted on

Approved:

_____
Professor Neil T. Heffernan, Advisor

_____
Cristina Heffernan, Advisor

Table of contents

List of figures

Abstract

The goal of this IQP was to improve both the functionality and the usability of docASSIST. docASSIST is a Google Doc add-on, developed by WPI students, that makes it easier for teachers to grade student's papers and provide feedback on them. docASSIST allows teachers to use a grading rubric to grade a student's paper. During the grading process, the teacher can add feedback to the paper. The student can see the teacher's feedback, as well as their grade on each section of the rubric, and can make changes to resolve the issues pointed out by the teacher. In this IQP, the docASSIST UI was revised to be more clear and intuitive, and a number of features were added to docASSIST, including rubric sharing and the ability to keep a history of a paper's drafts during the revision process.

Aknowledgements

I would like to acknowledge the docASSIST developers before me, Nick McMahon, Sam La, Jean Marc Touma, Christian Roberts, and Zi Wang, for building docASSIST and getting it to where it was when I began work on the project. I would like to thank my advisors Cristina and Neil Heffernan for guiding me through the development process, providing me with feedback, and working with me to improve docASSIST. I would like to thank Andrew Burnett for all his feedback and help in finding ways to improve docASSIST. I would like to acknowledge Zachary Armsby, who I mention in parts of this paper, for working with me on docASSIST. I would like to thank the english teachers at Shrewsbury High School for the feedback they provided. Finally, I would like to acknowledge Cory Tapply and Trevor Valcourt for picking up the development of docASSIST where I left off.

Authorship

This paper was written by Gianluca Tarquinio, using Mark and Christian's docASSIST IQP paper and Zach's docASSIST IQP paper as references.

Introduction

I started working on docASSIST about two weeks after Zach started working on it, around the time that he finished working on improving the development setup for the project and refactoring the code. Our first task was to complete the rubric manager. The rubric manager is a tool that allows teachers to create, edit, save, import, and share rubrics, and then attach a rubric to the document. In its complete state, it is a very useful tool, but when I started working on docASSIST, it was still in the early stages of development. We spent about 5 weeks working on the rubric manager, with Zach mostly rewriting the backend Java code for storing users saved rubrics and accessing shared rubrics, and me mostly rewriting the HTML and CSS for the front end UI, as well as the Apps Script and JavaScript for the functionality of the rubric manager.

Once the rubric manager was complete, and Zach was finished working on the project, I turned my attention to improving the main body of the docASSIST add-on. At the time, a docASSIST user could choose from two options for interacting with a document, review and revise. The review sidebar allowed teachers to mark up and grade a document, and the revise sidebar allowed students to view the feedback left for them by a user and make changes in response to it. In my remaining two weeks, I worked on improving the existing layout of these sidebars so that they were more clear and easier to use, fixed bugs, and added some additional functionality, including a way to create non-editable drafts of a paper in order to track its progress through editing process.

Rubric Manager

The rubric manager is a tool within docASSIST that allows a user to easily create and manage rubrics. The idea behind to rubric manager was to give users an easy way to create, edit, and share their rubrics, as well as get existing rubrics from other users. When I started working it, the rubric manager could only create and edit rubrics. There was a basic UI in place for the planned future functionality of the manager, but it needed to be changed. By the time Zach and I finished working on the rubric manager, its UI was vastly improved and its missing functionality was implemented.

Development Process

Before I could begin working on the rubric manager, I needed to look at the existing code to get an idea of how everything worked. Working on docASSIST was the first time I used Apps Script. I had some experience in JavaScript, but I had no experience in Apps Script. Additionally, the code for most of the project was very convoluted and difficult to understand, and there was a lot of it. By the time I understood enough to start working on the rubric manager, I still had no idea what most of the code in the rubric manager did. My understanding of the code increased continuously as I worked on it, but even by the time I had finished working on the rubric manager, there were still a few parts of it that were a mystery to me.

My process for building the rubric manager had three parts. The first part was rebuilding the functionality and appearance of the UI. As I didn't understand how most of the code worked at that point, this was a slow process. In order to make a change to an existing part of the UI, I first had to figure out how the existing code for that part of the UI worked, which often involved sorting through hundreds or even thousands of lines of the rubric manager code, as the code was very disorganized and, for any given part of the existing UI, most of its code was spread throughout the rubric manager code. This made debugging quite a challenge, as I didn't even begin to understand all of the interdependencies within the code. As a result, I made an effort to make my new code as similar as possible to the existing code in order to minimize the unexpected and difficult to find errors that any change might cause. As difficult as it was to debug the JavaScript, debugging the Apps Script was for more difficult. It turns out to be pretty difficult to get any sort of window into what is causing an error in the apps script, or even to find out where the error occurred. If there was an error somewhere in the Apps Script, there was no way to determine what line it was on, only that something that I changed since the last time it was working either caused a new problem, or caused an existing problem to surface. It wasn't until four weeks after I began work on the project that we finally found the location of the console that the Apps Script files logged their errors to. Up until that point, if I wanted to log something for debugging, I had to throw and error, and put whatever I wanted to log as the error message. As throwing an error causes execution to terminate, this method of logging debug information was far from optimal.

By the time I was finished with my main changes to the UI, Zach had implemented the new backend to the point that it was ready to be used. This allowed me to start the second part of building the rubric manager, which was adapting it to work with the new, and vastly different, backend. At this point, I understood a lot of the rubric manager code, but there was still a lot of it that I didn't yet understand. The main challenge in this task, in addition to the difficulty with debugging, was that there was a lot of duplicate code and unorganized code in the rubric manager. Making any changes required changing many lines of code, and often resulted in unforeseen consequences that forced me to familiarize myself with a new part of the rubric manager code. This was problematic, as the backend was not totally finished, and was changing. As I adapted the code to the new backend, I tried to clean it up as much as possible. This helped greatly when it came time to make more changes to parts I had already changed, in response to further changes to the backend, which mainly involved changes to the structure of stored data. By the time the rubric manager was mostly working with the new backend, the code, while still messy, was in a state much easier to make changes to.

The third part of building the rubric manager was a final round of debugging. While testing the rubric manager with the new backend, we discovered many bugs in both the frontend and backend code. The final process of finding and fixing significant bugs was long, and iterative. As we fixed large bugs and tested the rubric manager, more and more smaller bugs began to appear, some of which required significant changes to fix. As we approached the release of the rubric manager onto the published version of docASSIST, we frantically fixed bugs as quickly as we could, until everything finally worked, for the most part. There were still some minor bugs left over, most significantly the bug that caused category names within a rubric to be reordered seemingly at random, but they were not severe enough to warrant the time required to fix them.

Select

The select tab in the rubric manager allows a user to choose a rubric to attach to a document from all of the rubrics they have access to. In terms of UI, I didn't really make many major changes. It's just a list of the rubrics a user has access to, so there wasn't really much to change. All that I did to the UI fix any aesthetic issues with it, such as improper alignment of some of its elements, add some aesthetically pleasing effects and highlighting to the list, and make the changes necessary to distinguish between different type of rubrics (owned rubrics vs. shared rubrics). The majority of the changes that I made to the changes that I made to the select tab were to adapt it to the new backend. The structure of the stored rubric data, and the way that it was retrieved, changed completely. Adapting the existing code to fit this required many changes.

First, I had to change the Apps Script code associated with getting rubrics and their data. The old backend didn't really store much data, just rubrics and the emails associated with them. The new backend stored a lot more information. The Apps Script functions for getting all the

rubric ids associated with an email, and getting the actual rubric data associated with a rubric id were already there, but they were not at all compatible with the new backend, and had to be completely rewritten, and their signatures changed. The Apps Script functions for getting data were basically all of the form: request the data -> parse the data. Naturally, the new backend required both of these steps to be rewritten.

With the Apps Script code supporting the new backend, the JavaScript code that generated the list of rubrics had to be completely rewritten as well. First, all of the rubric ids they had access to needed to be retrieved. This information came in the form of three lists: owner rubrics, rubrics obtained through email sharing, and rubrics obtained through code sharing. A list of rubrics was then generated for each of these lists. To do this, the information about each rubric needed to be obtained. Once this was done, a list of all the rubrics (by name) that the user has access to, split into sections based on the nature of their access (owned vs. shared), could be generated by JavaScript and inserted into the document.

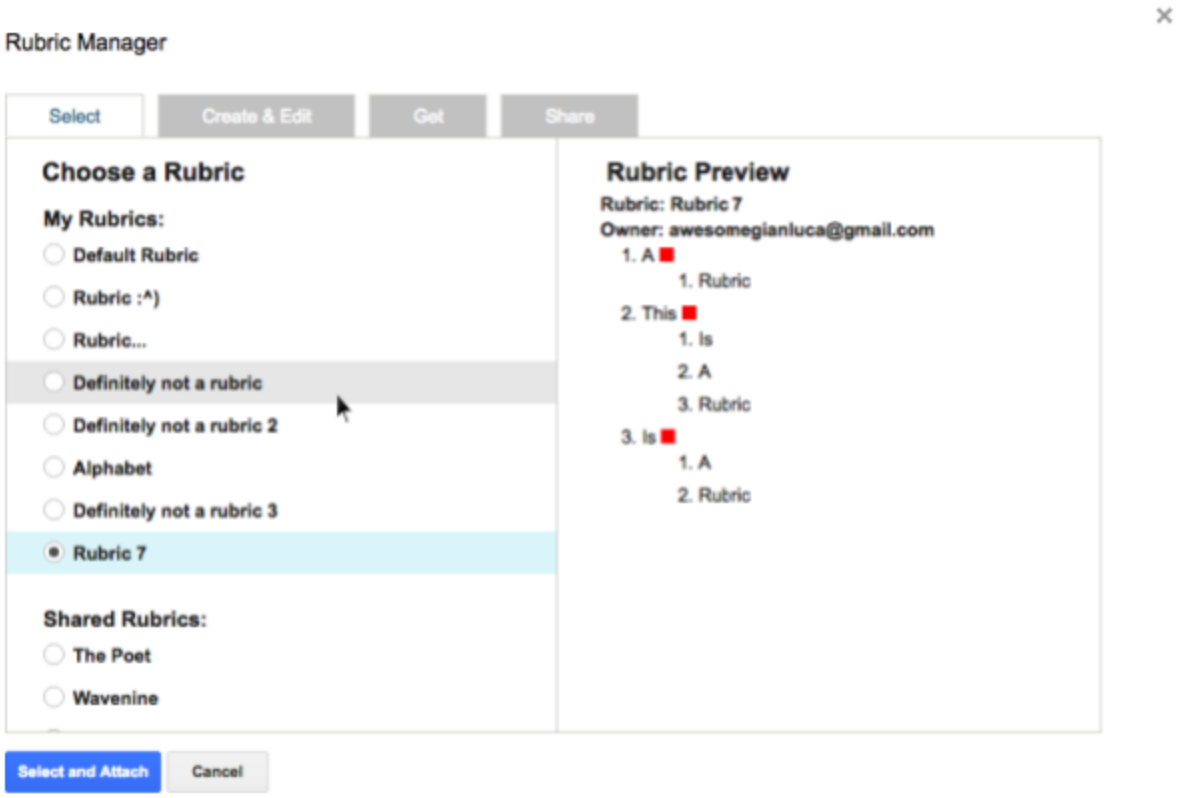


Figure 1: New Select Tab

Create & Edit

The create and edit tab in the rubric manager allows a reviewer to create new rubrics and edit existing rubrics. A new rubric can either be created from scratch, or can use an existing rubric as a starting point. I made very few changes to this tab. It's the only tab that I didn't make any UI changes to. There had been a completely rebuilt UI for this tab in the making, but it never actually made it into the rubric manager. The only changes I made to this tab were the changes necessary to adapting it to the work with the new backend, as well as bugfixes. The required changes to the Apps Script to adapt this tab to the new backend were similar to, and partially overlapped with, the changes I made to the Apps Script for the select tab.

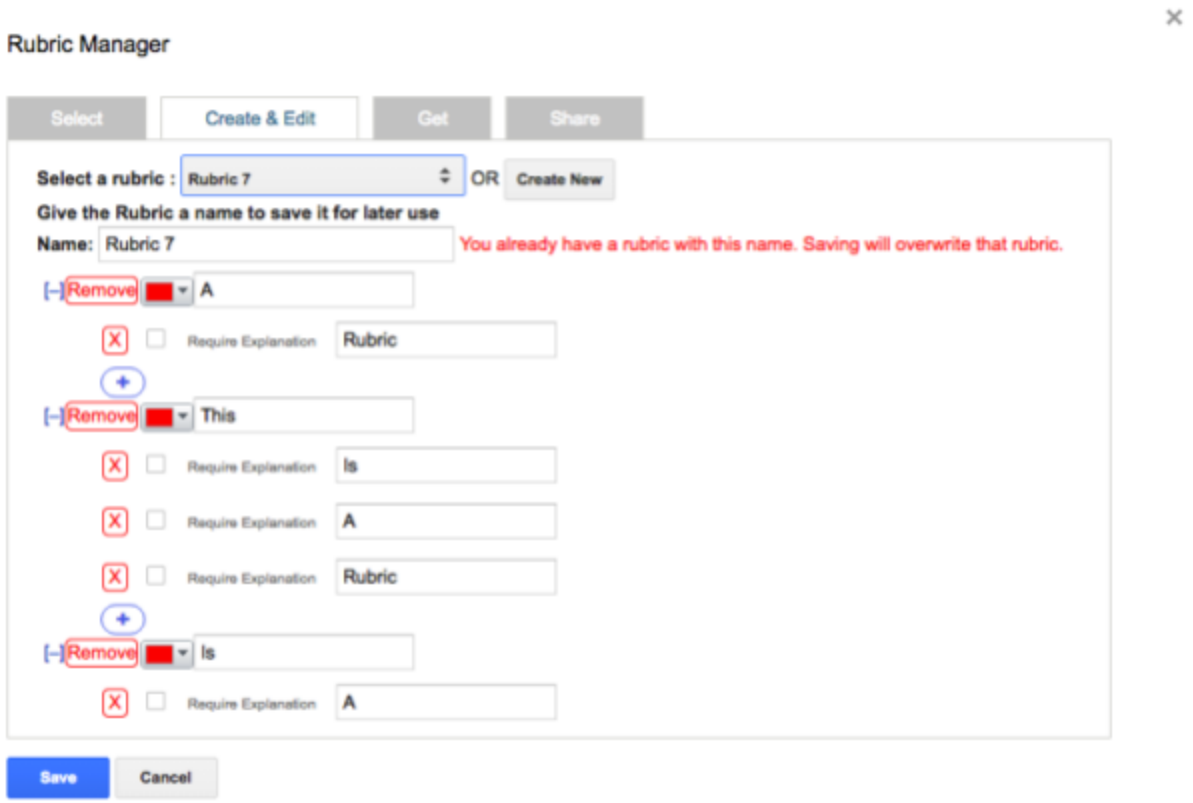


Figure 2: Create & Edit Tab

Get

The get tab allows a user to import rubrics that other users have shared, either by email or by code. When a user gets a rubric this way, it shows up in select tab as a shared rubric. Before I started working on it, this tab had some functionality and a basic UI. A user was able to

view the rubrics associated with an email and import any of them. In addition to the rubric preview panel, the tab consisted of a text field to enter an email, an area to list the email shared rubrics associated with that email, and a text field to enter a code for getting a code shared rubric.

The first thing that I did was program the functionality of the tab. This included things like changing the state of elements, like disabling buttons or checkboxes, and implementing basic logic to determine what preview should be showing and what rubric should be imported when you hit 'Get'. Throughout this process, I made changes to the layout of the UI to the layout of the UI to make it more clear and intuitive. This was the very first thing I did in the rubric manager, and it helped me to understand how UI code throughout the rubric manager worked. Most of the HTML code that made up the UI was generated by JavaScript as it determined the rubrics it would need to display. By the time I finished programming the functionality of the UI, I understood a great deal more about the rubric manager code than I had after my time reading through and trying to make sense of the code before I actually started making changes to it. Next, Zach and I both made significant improvements to the aesthetics of the tab. This included things like cleaning up the alignment of the elements, adding effects and highlighting to the rubrics in the email shared list similar to those found in the select tab, and rewriting the CSS to make everything from spacing to the text itself look much better. By the time we were done, the tab looked quite good, but still did not get email and code shared rubrics as intended. This is because, at the time, we were still using the old backend, and email and code sharing were not supported in the old backend.

As the new backend started to take shape, I was gradually able to complete the functionality of the tab. Email sharing was the first feature I was able to implement. Similarly to the other tabs, this first involved adapting the Apps Script function to work with the new backend. This tab used the same Apps Script functions as the select tab, as well as a function to add a rubric to the users list of email shared rubrics. With the Apps Script done, I was able to finish the implementation of email sharing in the UI. When the user entered an email, a call was made to the backend attempting to retrieve all of the rubric ids associated with that email. The new backend was implemented such that if a request was made for the rubric ids associated with a different email than the one making the request, only the ids of email shared rubrics would be returned. Once the ids were retrieved, the data for each of the rubrics in the list of rubric ids owned by the email entered were retrieved, information about the rubrics, most importantly their names, were retrieved from the backend, list in the select tab.

Frontend support for getting code shared rubrics came much later, and was one of the last things I changed in the rubric manager. Once code sharing was supported by the new backend, implementing it in the get tab was simply a matter of writing the Apps Script functions necessary for finding a rubric by code, and for attempting to get a rubric shared by code. The new backend was implemented such that the contents of any rubric could be viewed given the code associated with it, even if it had code sharing disabled, but only rubrics with code sharing enabled could be imported by code. The result was that very few changes needed to be made

to the JavaScript code for the tab to support code sharing, as most of the logic had already been implemented.

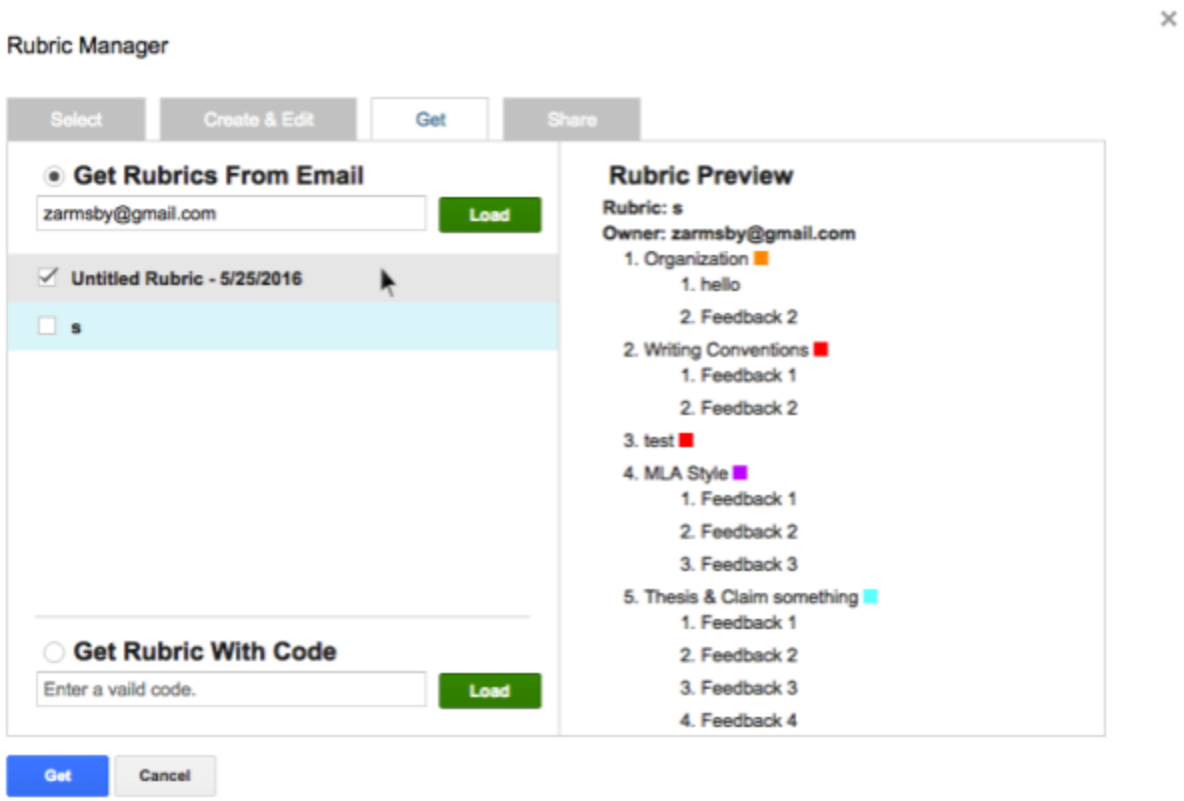


Figure 3: New Get Tab

Share

The share tab allows users to share their rubrics so that other users can get and use them. There are two different types of sharing: email sharing and code sharing. If a user enables one of their rubrics for email sharing, then that rubric will appear on the list of email shared rubrics in the get tab if somebody attempts to get rubrics from that user's email. If a user enables one of their rubrics for code sharing, then anybody will be able to get that rubric by its unique, alphanumeric code, given that they know it.

Before I started working on it, there was a simple UI in place for this tab. As for functionality, all it could do was list the rubrics associated with the current user email. There were UI elements in place for controlling email and code sharing, but they didn't actually do anything. I worked on implementing the complete functionality of the share tab throughout my time working on the rubric manager, as email sharing and code sharing became supported by the backend. When a new type of sharing became supported by the backend, there were a few things that I needed to do in order to implement support for that type of sharing in the share tab.

First I had to write the Apps Script functions that set the sharing values of a particular rubric and that checked the existing sharing values. Once the Apps Script functions were written, I had to alter the function that listed the user's owned rubrics so that it checked the state of each for the new type of sharing, and set the state of the corresponding UI element accordingly. Additionally, the function had to be attached to that element, so it could then set the sharing state of the rubric that it was associated with. For code sharing in particular, I had to add an element to display the code for a rubric when code sharing was enabled for that rubric. The function that constructed this list worked the same way as the other rubric list building functions that I've described in previous sections.

Once the share tab was fully functional, I began to work on updating its UI. This was the last thing I did in the rubric manager before focusing on bug fixing. I made big changes to the existing UI, and by the time I was finished, I looked quite good. As with all of the other rubric lists, I added highlighting to show which rubric was currently being previewed, and some nice mouseover effects. One of the problems with the previous UI for all of the tabs with a preview panel was that it was unclear which rubric was actually being previewed at a given time. The highlighting that I added fixes this problem and, in addition to the mouseover effects, makes the interface feel more responsive. I changed most of the existing CSS, and added more of my own, to make the interface look more polished.

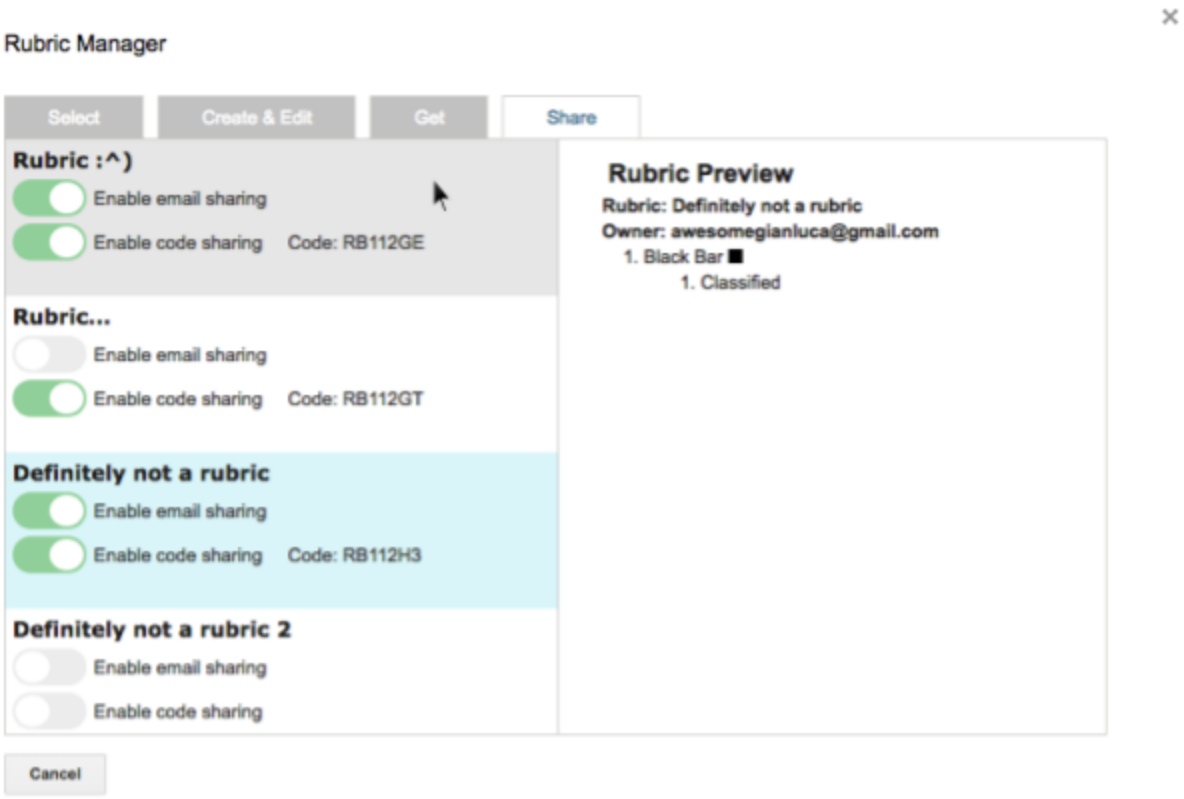


Figure 4: New Share Tab

Preview Panel

The preview panel appears in the right half of the select, get, and share tabs. It shows the user a preview of the rubric that they have selected. Over the course of my work on the rubric manager, I made some changes to the preview panel. These changes were mainly in response to the new backend, as the old preview panel code was not compatible with the new backend. Changes to the preview panel code were particularly painful, as that code was some of the least readable and difficult to change code in the docASSIST addon. In addition, I made a few changes to the information that was displayed in a rubric preview, like the type of access the user has to the rubric, and of course I fixed bugs.

Bug Fixing

At the end of my time working on the rubric manager, Zach and I turned our attention to fixing bugs. There were a great deal of bugs in the rubric manager, some of which were fairly severe. Many of these bugs had been present before we started working on the project, and many were introduced by our recent changes. As the rubric manager code was so convoluted, a minor change would often cause some strange, unexpected bug in a part of the code that I was not familiar with. As I mentioned before, there was a severe lack of debugging tools available for the Apps Script portion of the code. The combination of these two things made debugging a long, painful process. Often we found ourselves unsure of what function, or even file a bug was located in, or even whether it was in the frontend or backend. After quite a bit of time spent debugging, however, we eventually found the rubric manager in a state suitable for official release.

Sidebar and Menu

The main functionality of docASSIST is accomplished through the use of two main sidebars: the review sidebar and the revise sidebar. The review sidebar allows a reviewer to grade or suggest changes to a paper, and the revise sidebar allows a student to implement the changes suggested by a reviewer. I say reviewer, rather than teacher, because docASSIST is designed not only to be a tool for teachers to review/grade papers, but also for students to peer review each other's papers. Towards the end of my work on docASSIST, and with the rubric manager pretty much complete, I began working on improving the main sidebars and menu in docASSIST. My goals were to make the UI for the main sidebars and menu more intuitive, and add some more functionality to them.

Menu

The menu is the way a user accesses the various tools in docASSIST. Initially, the menu had one layer, and the review and revise sidebars where the first two items on it, right next to each other. The clear problem with this is that the words 'Review' and 'Revise' not only look and sound similar, but they can both be used to mean similar things. This could lead to confusion while using docASSIST, especially for a student. To make the menu more clear, I replaced the review and revise menu items with submenus called 'Teacher' and 'Student', each containing two items. The teacher submenu contains 'Review' and 'Grade', where review takes the user to the review sidebar, and grade takes the user to the grade sidebar, which was previously only reachable through the review sidebar. The student submenu contains 'Peer Review' and 'Author Revise', where peer review takes takes the user to the review sidebar, and author revise takes the user to the revise sidebar. This is a small change, but it has a significant effect. Using the keywords 'Student', 'Teacher', 'Peer', and 'Author', the new menu leads the user through the menu, which greatly increases its clarity, especially to users unfamiliar with docASSIST.

Sidebar UI Changes

Of the two main sidebars in docASSIST, revise was pretty good as it was, but review needed some changes. The main problems with the review sidebar was that the interface was a bit unclear in a number of places. The problems were fairly minor, but improving on them definitely made the interface more clear and loss frustrating to use. Probably the biggest improvement that I made was to add loading and success indicators to various parts of the sidebar. There were already some loading indicators, but there were no success indicators. This was a bit problematic because some of the actions that didn't have loading indicators actually took a relatively large amount of time, and without a success indicator, it was unclear whether anything really happened. Users shouldn't be wondering it actions they were

attempting to perform were actually working, so the loading and success indicators were necessary. Once the loading and success indicators were all in place, I briefly turned my attention to changing the location and word choice of some the various buttons on the tab. One example of this was changing the 'Hide Formatting'/'Show Formatting' button to 'Hide Feedback'/'Shmow Feedback'. It's a very minor change, but formatting was a pretty vague way of referring to something that was called feedback everywhere else in docASSIST. The other improvements I made to the review tab were very minor changes, and of course some bugfixes. I also made some changes to the grade sidebar. In addition to changes similar to the ones I made to the review sidebar, most of my work on the grade sidebar was fixing a fairly important bug with grading, which occurred when users attempted to leave some of the grading fields blank.

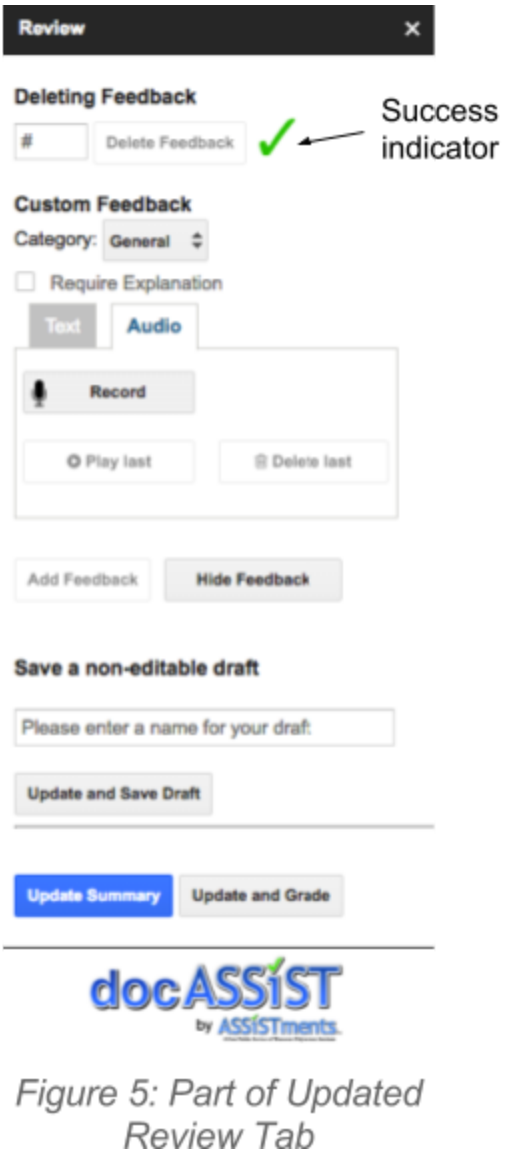


Figure 5: Part of Updated Review Tab

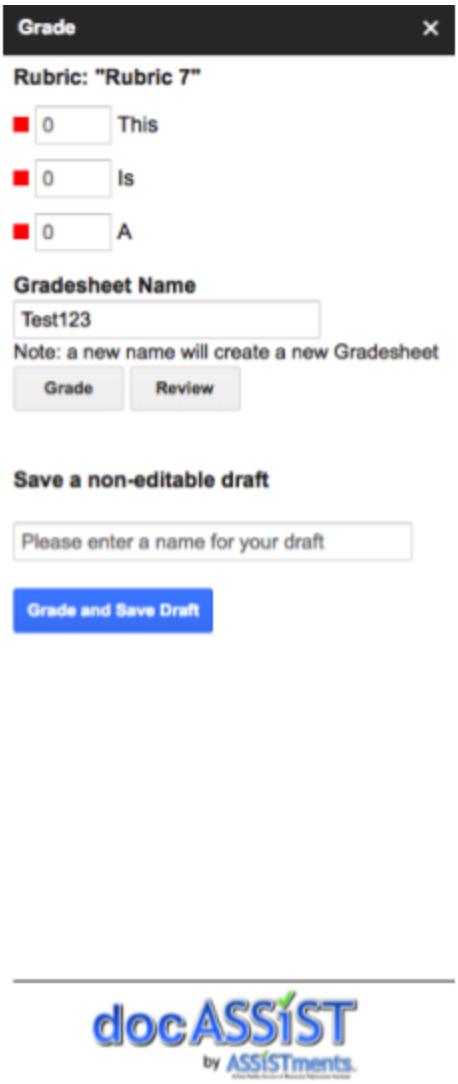


Figure 6: Updated Grade Tab

PDF Drafts

By far the biggest change that I made to the to the review and grade sidebars was to add non-editable PDF drafts, found at the bottom of both tabs (see figures 5 and 6). This feature allows a reviewer to save a PDF of the document as it is, and attach it to the document so that it can be viewed easily by anybody with access to the document. The purpose of this feature is to act as basic revision history with respect to docASSIST. It allows users to track a paper's progress through multiple waves of reviewing and revising.

The way it works is very simple. When a user chooses to save a non-editable draft, a PDF version of the document is created through the Google Drive API. If they do not provide a name for a draft, it defaults to "[Document Name] - Draft #.pdf", where [Document Name] is the name of the document, and # is the number of the draft (number of existing drafts + 1). As the new document is a PDF, it cannot be edited. The PDF is then saved in a folder in the reviewers Google Drive. By default, the folder is docASSIST/Drafts/'[Document Name]' Drafts, where [Document Name] is the name of the main document. Through the docASSIST options menu, a user can change the location that the PDFs are saved to anything that they want (although this feature seems to have been removed in the current version of docASSIST). Once the draft has been saved, a link to it is retrieved and, along with the date, is stored in the document's properties, as are many other things in docASSIST. The dates and names of each of the drafts of a document are listed at the bottom of the document. The drafts are automatically enabled for link sharing, which means that anybody with access to the document can view any of the drafts, as the links are part of the document.

Figure 7: List of Non-editable Drafts

Better Feedback

Attempting to improve the UI for adding new feedback as a reviewer was the last thing that I worked on. To add feedback in docASSIST, a user highlights text in the document, and clicks on one of the parts of the rubrics to associate the highlighted text with. If the user wants to leave feedback with a custom audio or text message attached, they have to do that through the 'Custom Feedback' section of the review tab, which is located below (not directly below) the part of the tab where normal feedback is entered, even if they are associating that custom feedback with a particular part of the rubric. This system fine for adding standard feedback, but unintuitive for users unfamiliar with docASSIST who want to add custom messages to their feedback.

I spent the last few days of my time with docASSIST working on an improved version of the existing feedback system. In the new feedback system, while the custom feedback section remains unchanged, allowing users to associate custom feedback with a top level category of the rubric or even independent of the rubric, the normal feedback system at the top of the review tab has increased functionality. When a user clicks on a part of the rubric in the normal feedback section, rather than immediately associated the highlighted text with that part of the rubric, a text field and some buttons show up directly under the part of the rubric that the user clicked on, giving them the option to add a custom message to the feedback before creating it. This custom message would be associated with the rubric subcategory that the user had selected, which was not previously possible. By the time I was finished working on docASSIST, text feedback was working in the new system, but audio feedback was not fully implemented. Of course, all of the text and audio feedback in the custom feedback section were still working as intended. Unfortunately, this system has not actually been added into the official version of the code, although it still exists in the BetterFeedback branch of the git repository.

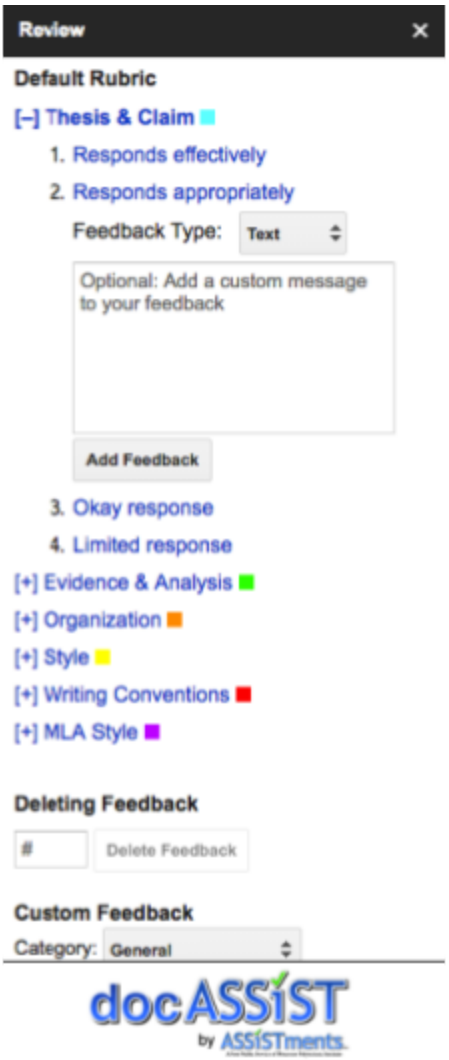


Figure 8: Improved Feedback System

School Visit

After we moved the new rubric manager to the official release of docASSIST, Zach and I visited Shrewsbury High School with Andrew Burnett to meets with the english teachers there and get feedback about docASSIST. Our goal in this visit was to hear what real users had to say about docASSIST, and what they thought we should add to it. As developers of docASSIST, it is very difficult to understand how a real user approaches it and expects from it. Therefore, it was important for us to get the input of real teacher either already using or potentially interested in using docASSIST. The other goal that we had in this visit was to identify bugs that we had not found during the development process. At the school, Andrew presented docASSIST to the teachers, and walked them through using it themselves. As they used it, Zach and I answered questions about docASSIST, identified bugs, and received suggestions for potential improvements and new features. The two most common suggestions that we got from the teachers were more for data analysis tools for grades and for some sort of success indication when a user performs an action. Of course, we identified some new bugs as well. Following the school visit, I implemented success indicators throughout docASSIST, and Zach and I discussed the data analysis tools that were requested. More specifically, the teacher wanted a way to automatically calculate weighted averages for grades. We considered two options, adding weights as part of the grading process, and adding weights to the actual rubric categories as part of the rubric creation process. We ultimately decided that the weights should be part of the rubric itself, and eventually passed the task of implementing this on to Cory and Trevor, the future developers of docASSIST who would work on this as part of integrating Google Classroom into docASSIST.

Conclusion and Future Development

Having finished my work on docASSIST, I'm satisfied with the improvements that I made to it. The completed rubric manager is a highlight of docASSIST, and docASSIST as a whole is, in addition to the other added features, now cleaner, easier to use, and less buggy. For future developers of docASSIST, I would recommend a large scale refactoring of the code. In its current state, the code for most of docASSIST is convoluted and very difficult to make changes to. Attempts to change the code often result in strange and significant bugs in unexpected places, that can be very difficult to find without a detailed understanding of the code in its entirety. This will only get worse as time goes on, which is why I think that totally refactoring the code would be wise for anybody who intends to make serious changes to docASSIST in the future. Despite this, I think that docASSIST is in a good place, and has many areas that can potentially be improved upon in the future.

Appendix

Docassist Website

https://sites.google.com/site/assistmentsfeedbacktool/

Github Repositories

https://github.com/docASSIST will be the future location of the docASSIST repositories

Docassist Frontend

https://github.com/zarmsby/docASSISTAppsScriptProject (must be member to view to request access email zarmsby@gmail.com)

Docassist Backend

https://github.com/nmcmahon1215/docASSIST (must be a member to view, my work is in any branch with gianluca in the name, as well as PDFThingy and BetterFeedback)