Cumulyrics

Implementation Document

Team 19

Kyle Van Landingham, Alec Schule, Alec Fong, Andrew Zolintakis, Noah Bergman

Table of Contents

1. Executive Summary

This document describes in detail the implementation process carried out to create the Cumulyrics application. Ths document describes how each section of the application code base was implemented, and validates the implementation by comparing it to the design in the Detailed Design Document. Where deviations from that document occurred, this document justifies their necessity. This document also describes how the team adhered to the Project Management Plan throughout the implementation process. In particular, the Risk Management and Risk Planning sections of the Project Management Plan were closely followed during the implementation process. Finally, this document provides start up instructions to run Cumulyrics on the specified virtual machine.

2. Overview

2.1 Purpose

The purpose of this document is to validate the implementation of the system with the design specified in the Detailed Design Document and document the process followed during the implementation phase. The Implementation Document allows for a review of the process in order to gain an insight toward improving the process for future projects.

2.2 Intended Audience

The intended audience for the Implementation Document includes the team members of team 19 and other relevant stakeholders including the professor, TAs, and CPs. The team members should utilize the document in order to improve the process for future projects and the other relevant stakeholders should review the document to validate the implementation.

2.3 References

Reference	Relevant Information
Github Repository	https://github.com/kvanland/csci310Team19

2.4 Definitions

Term	Definition
SRS	Software Requirements Specification (Team 17)
PMP	Project Management Plan
NLTK	Natural Language Toolkit (http://nltk.org)

SVG	Scalar Vector Graphics. (Used to show visuals in a browser)

3. Mapping to Design

3.1 Front-End Implementation

The front-end portion of the implementation follows the front-end class diagram (Detailed Design Document 3.4.1) by dividing the javascript code in **javascript.js** with comments acting as dividers. Those divider comments are marked by long strings of "*" characters on the top and bottom of the title for each class section. Below each class section header is all functions and data members that are listed in the front-end class diagram. Each function has a comment next to the function declaration that declares the return type of that function. Most functions also have a comment directly below the function that declares the data types for the parameters passed into the function.

Some data members were removed from the **javascript.js** file because they served no purpose with regards to the functionality of the program. An example of one of the unnecessary data members is "backbutton: Button" which does not need a reference to since the button in the html file contains an onclick property that calls the appropriate javascript functions. There are a few data members which refer to html elements that are not useful in terms of functionality so they have been omitted. The full list includes *header:div*, *logo:div*, *backButton:Button*, *homeButton:Button*, *autoList:List*. Certain names were also changed to be more descriptive such as "*artistList:Array[JSON Object]*" which became "*currentArtistList*". The two listed JSON objects *Artist,* and *Song* were both omitted from the implementation because they represented unnecessary data.

Certain functions were introduced that were not foreseen during the design phase including an entire class named "Animation Functions" which include "s*hiftInputsDown(): void, shiftInputsCenter(): void, setHeight(id: string, height: int): void, setVisible(id: string): void, and setInvisible(id: string): void"*. These functions allow the javascript to manipulate elements within the browser and increase readability by simplifying browser manipulation through descriptive function calls. There is also a global variable *page:int array* and a function *setPage(page: int)* within the html class that aid in tracking the state of the application. Additions of jQuery functions to implement the autocomplete functionality have been added to the search class.

In addition to certain functions being added to the front-end code, there are also certain functions that have been omitted. These functions have been omitted because they served no meaningful purpose within the context of the front-end code. The functions that have been omitted include the following *showAutoComplete(search: string): void* and *shareAction(): void*. These functions were omitted because jQuery made the first function obsolete and the Facebook API was more accessed within **index.html** rather than in the javascript.

A major change to the design was locally hosting the Lyric API in order to obtain the lyrics for each song directly by using AJAX requests to the Lyric API directly. This made the **getLyrics.php** file obsolete and was therefore discarded from the final implementation. The original design plan was for the back-end **getLyrics.php** to act as a middleman between the AZLyrics and the front-end but that implementation gave unpredictable results. So, the entire

team realized that since the back-end did not communicate with the database to get lyric data it would be more efficient for the front-end to communicate directly with the Lyric API which proved to give more consistent results.

3.2 Back-End Implementation

The back-end implementation partially follows the back-end class diagram (Detailed Design Document 3.4.2). We followed the general layout of the diagram. For example, the implementation has four different classes, each one gets the data for one of the four major features; WordCloud, SongsFinder, LyricsFinder, and ArtistSuggestion (the classes are named differently in the implementation and the design document). Each one of these classes has only the methods specified in the class diagram. Additionally, there were other files that handled the communication to the frontend and only called the four main classes to get the data to send.

There are also many ways the back-end implementation differs from class diagram. First of all, the requestHandler and Sender classes were not created. Instead, we created an api call for each of the four major features. Each one receives the request from the front-end, gets the data from one of the four main data classes and sends that back to the front-end. Furthermore, we added a class called DatabaseAccessor, which SongsFinder, WordCloud, and ArtistSuggestions all inherit from because they all access the database. Additionally, the implementation classes did not contain any data members that were specified in the design document, because they ended up not being necessary. Lastly, the API we used for getting the lyrics of a song's AZLyrics, which was not mentioned in the Design Document.

As described in the front-end implementation section the file **getLyrics.php** was omitted from the implementation because the API used to get the data was unpredictable in terms of consistently correct data. To increase the predictability of the API we hosted the API on our local machine and made AJAX calls to it. Upon further reflection the team as a whole decided that the gathering of lyric data should be left to the front-end to reduce dependency on the back-end and cut out an unnecessary step.

3.3 Scraper Implementation

The scraper implementation follows the second-level scraper decomposition. (Detailed Design Document 3.1.3.2) The implementation of the scraper was a fairly straightforward process. A complete list of artists is acquired from the LastFM API. Next, each artist is passed into the Spotify API to get a list of songs for each artist. Finally, an API based off the Lyrics Wiki was used to obtain the lyrics for each song. The lyrics are processed and stored in the database.The database schema was designed exactly as described in the Detailed Design Document. (3.8)

As part of processing the song lyrics, commonly used words were removed as required by SRS section 3.2.4.3 (REQ-1). As the SRS was ambiguous with regards to the exact list of words to be omitted, the NLTK list of english stop words was used as reference. The exact list of removed words is reproduced below:

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours',
'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers',
```

```
'herself', 'it', 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves',
'what', 'which', 'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are',
'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does',
'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until',
'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into',
'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down',
'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here',
'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more',
'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't',
'can', 'will', 'just', 'don', 'should', 'now']
```

While implementing the scraper, the team followed the Risk Management and Risk Planning sections of the Project Management Plan. Most risks did not end up being relevant. For example, the low priority risk “Database capacity not adequate” was not a problem, as the entire database only required 100 megabytes of storage space. The one risk did become realized is 3rd party APIs going down. The API for Lyrics Wiki was unresponsive too often to be usable in the application. Luckily, the original risk effect rating of “Catastrophic” turned out to be an overestimation, as a way to host the API locally was easily found.

As testing was carried out during the implementation process, several bugs came to light that were fixed. Originally, the LastFM API was used for both the list of artists, and the song list for each artist; however, testing revealed that the song list was quite lacking for some artists. This was fixed by using Spotify’s song lists instead. Another bug revealed through testing was some stop words not being omitted if they had a capital letter. For example, “I” appeared in many word clouds. This was easily corrected by using a case insensitive matching in the scraper code.

3.4 Overall Implementation

Certain steps to validate the implementation to the design require a view of the overall system as a whole. The previous sections describe how the implementation for the three different aspects of the system map to their specific designs, however the system as a whole needs to be validated against the design.

A large contributor to deviation from the design involved architectural erosion in the form of having the front-end code communicate directly with a third party API. As explained in section 3.1 and 3.2 it made sense given context why the team would deviate from the software architecture established in the design document. This erosion is illustrated in sections 2.1, 3.2, and 3.3 through the software architecture, UML Component, and UML Communications diagrams.

Through the UML Use Case diagram, we can map the use cases to actual interaction with the system and the <<extends>> use cases do accurately represent the use cases for the system. There is also only one actor because the system treats all users equally and grants each user the same privileges and access.

With regards to the Sequence diagrams the diagrams that match perfectly to the implementation are the *Artist Cloud Creation Sequence, Songs Containing a Selected Word Sequence, Auto Complete Sequence,* and *Database Creation Sequence*. The two sequence diagrams that do not map to the implementation are the *Share to Facebook Sequence* and *Song*

Lyrics Sequence. The share to facebook feature became much more complicated than the team expected and as a result chose to leave out that functionality and focus on the rest of the system. This choice was justified because the requirement for facebook integration was of low priority. The lyric sequence was not implemented according to the design document for reasons stated earlier in the document.

The system from the viewpoint of the finite state machine diagram was implemented according to the design with the exception of the Facebook state. The team was able to share a link to Facebook but failed to develop a way to upload a base64 image derived from a SVG to Facebook.

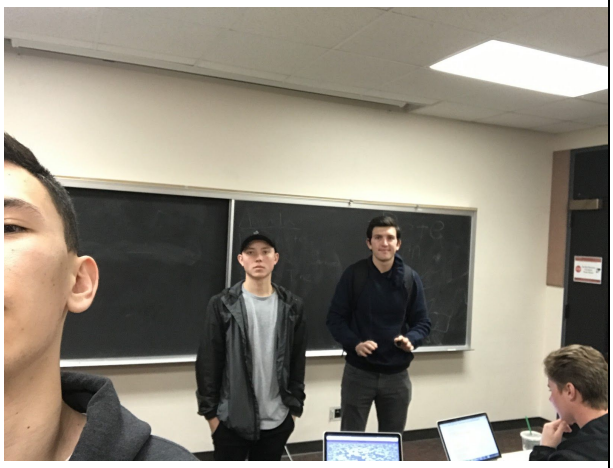
Overall, the system implemented closely matched the system described in the Design Document. The design principles used to inspire the design were correctly implemented into the final system. The code is well commented and organized clearly.

4. Description of the process

4.1 Project Plan Compliance

In accordance to the Project Plan this is a log of the meetings that occurred while the team worked on the implementation for the project.

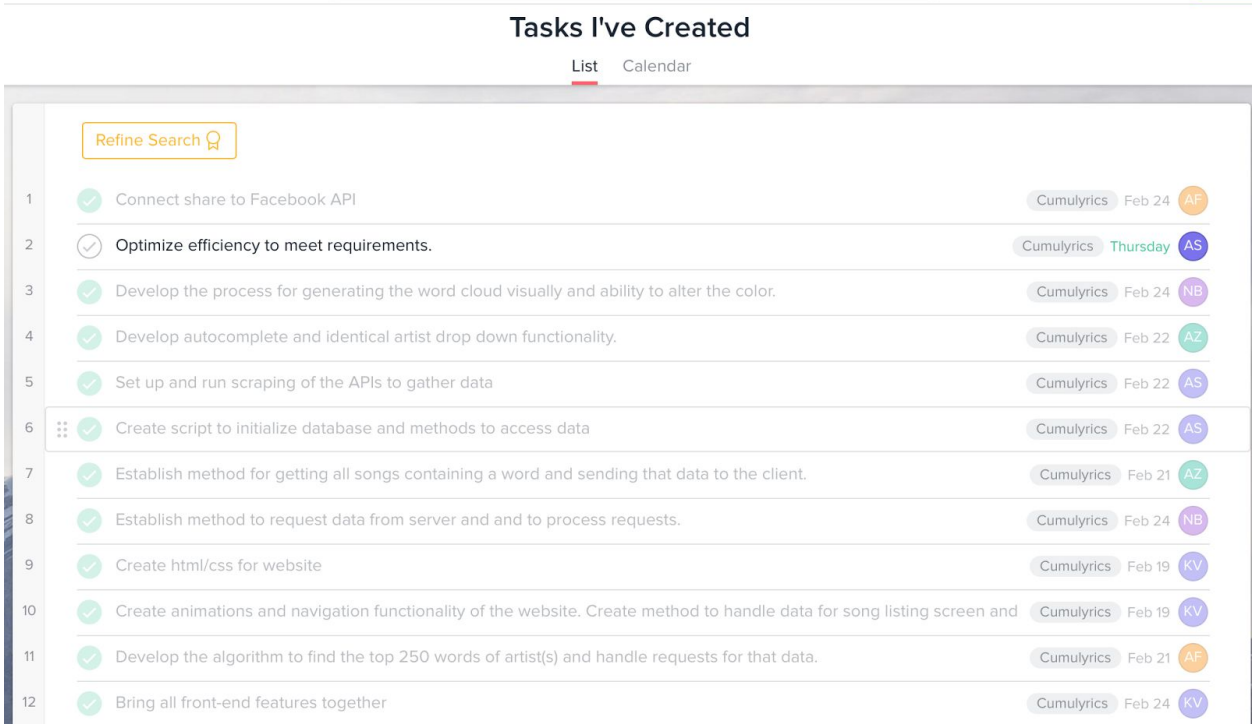
2/20/17 All team members met at Leavey Library at 7pm-8:10pm. The team rearrange the schedule a bit to accommodate the shorter implementation phase allowed by the project timeline. Each team member was asked to determine a reasonable due date for each of their task which were updated on Asana. The team then briefly discussed how the front end and back end will communicate (AJAX) and coded until it was determined that the team could code and reconvene another day when merging was needed and tasks that interacted with each other would need to be sorted out.	

2/23/17 All team members met at VKC from 4pm-6:30pm. After two days of working on the code individually, the team met and each member worked on their respective tasks. The scraper was in progress, the front-end html and css were finished, and the php backend was in progress. The team discussed multiple issues that might arise including compatibility with the virtual machine.	
2/25/17 All team members met at VKC from 5pm-7pm. By this point the team had access to all the data the scraper provided. This data was used to test out the mySQL access from the php back-end. The front-end javascript code was complete with animations and displaying dummy data. Note: Alec Schule left before a picture could be taken.	
2/27/17 All team members excluding Alec Schule met at VKC from 2pm-5:30pm. The front end team members were working on implementing facebook share integration and converting the word cloud to a jpeg format. The scraper was run again with updated parameters and more up to date data was given to all team members. The php backend team members were testing internally with the updated data to ensure that their classes worked correctly.	

2/28/17 All team members excluding Alec Schule met at VKC 8pm-10pm. The front end had complete AJAX requests and the backend php files were being debugged. Facebook integration was still being tackled. Documentation for the backend and frontend also began. (Alec F. and Andrew not pictured)	
2/29/17 All team members met at VKC 6pm- . The team got the system working on the virtual machine and finished the documentation for the implementation phase.	

In terms of using Asana here is screenshot of the tasks completed and assigned during the implementation phase. Note that these tasks are overarching and map with the tasks listed in section 3.2.2 of the Project Plan. These tasks complied to the schedule within the Project Plan even though the due date for the implementation phase was pushed back. The tasks listed in the project plan lacked any sort of tasks that ensured that all sections of the system worked together so, that was a task assign to all team members after all other tasks were created (Not included in

the picture).



According to the Project Plan a metric for tracking progress is task completion progress through Asana. The team had an on time completion percentage of around 83%. That percentage was calculated by dividing the amount of on time tasks completed by the total number of tasks. The main reason for the less than 100% completion percentage on time is that the schedule created in the Project Plan did not accurately account for the amount of time needed to complete certain tasks such as developing the autocomplete functionality and Facebook integration.

At the beginning of each meeting the Project Manager began with assessing the progress of each task, the quality of work so far, and the likelihood of finishing the each task on time. This involved one on one conversations with each team member resulting in suggestions of minor adjustments to the remaining work if necessary and reminders to adhere to the design document. Then, the Project Manager would assess potential likelihood of risks occurring and reviewing avoidance strategies.

Some risks that did occur were “3rd party API’s take too much time”, “Team Member gets distracted by other schoolwork”, and “The time required to develop the software is underestimated”. In the case of the first risk, the team pivoted to another API that allowed the scraper to gather all necessary data in about 10 hours of runtime. In the case of schoolwork distraction, as per the Project Plan section 8.3 Risk Monitoring other team members were open to absorbing a portion of the affected team member’s responsibility. In the case of the underestimation of implementation time necessary, the stakeholders granted an extension for the project implementation that allowed the team members to confidently implement the system with proper adherence to the SRS.

At meetings where Asana tasks were completed, the team took some time to review the code to make sure it measured up to the standards we set in the project plan. The team looked at the code readability, checked for modularity, and made sure variable and function names were self-explaining. The team also measured the average length of variable names and function

names. For example, the WordCloud class had an average variable name length of 11 and average function name length of 12.3. The variable name measured up to our required 10 letter average length and the function name fell short of our required 20 letter average length. However, the team deemed the function names clear enough and did not require any change to the names. The team did this for each source code file.

After the project plan compliance at each meeting, the team worked collaboratively on their task for more than an hour on average. In between meetings the project manager would check in on the status of the tasks in progress and request adjustments or new tasks if necessary.

4.2 Ordering of Implementation

The implementation ordering followed the schedule found in the Project Plan 3.2 Project Tasks. The implementation process minimized task dependencies on other task completion in an attempt to reduce any possibility of process bottleneck. Team members were split into 3 teams, front-end, back-end, and scraper.

The front-end team began with designing the html and css for the website which did not have any dependencies on the back-end or scraper teams. After that task was completed the front-end team then tackled the implementation of the javascript which incorporated the use of jQuery, AJAX, canvg, d3, and Facebook API. During development dummy data was used to test the use of these third party technologies. Only once the back-end team and the scraper team were complete or at least had prototype versions of their files did the front-end team connect the browser based front-end to the server side code.

The back-end team began with designing multiple php files to handle AJAX requests whose parameters and return values were listed in the Design Document section 3.4 UML: Class. Similar to the front-end team the back-end team used dummy data to test their return values and format. Only once the scraper team had collected some usable and properly formatted data did the back-end team test and debug their code with real data.

The scraper team began with creating the script to scrape the viable APIs for relevant data. This process was quick and took less than two days to complete. The quick rate at which the scraping was completed allowed the back-end developers to test their code with real data to ensure that the front-end developers would have valid data returned from their AJAX requests. The scraper team did create multiple iterations of the scraper to optimize the space used and the quality of the data scraped from the APIs.

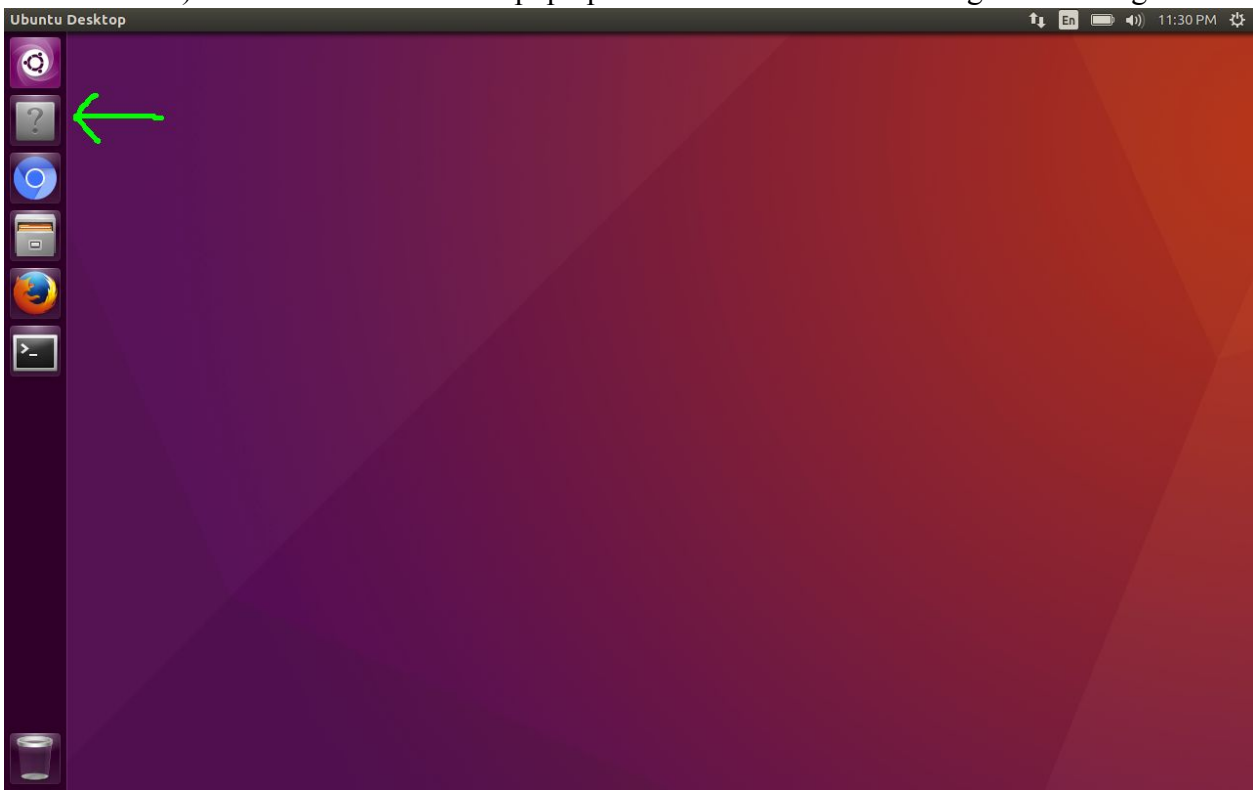
The implementation process had an emphasis on reducing dependencies between each team which allowed the project to progress with little risk of bottleneck. The system was purposely divided along those three aspects because each team could accomplish the maximum amount of work required before needing to connect their code to another team. This approach was used as opposed to the team all working on one feature or aspect of the system at a time since the implemented process allowed the developers to create solid and maintainable foundations that the rest of the code was built upon.

Throughout the process the team was constantly reminded to keep readable and maintainable code a top priority. As a result the implemented code has minimal nesting, ample information hiding, and small module size. This can be visually observed through the system code hosted on the team github repository linked in the reference section of the document.

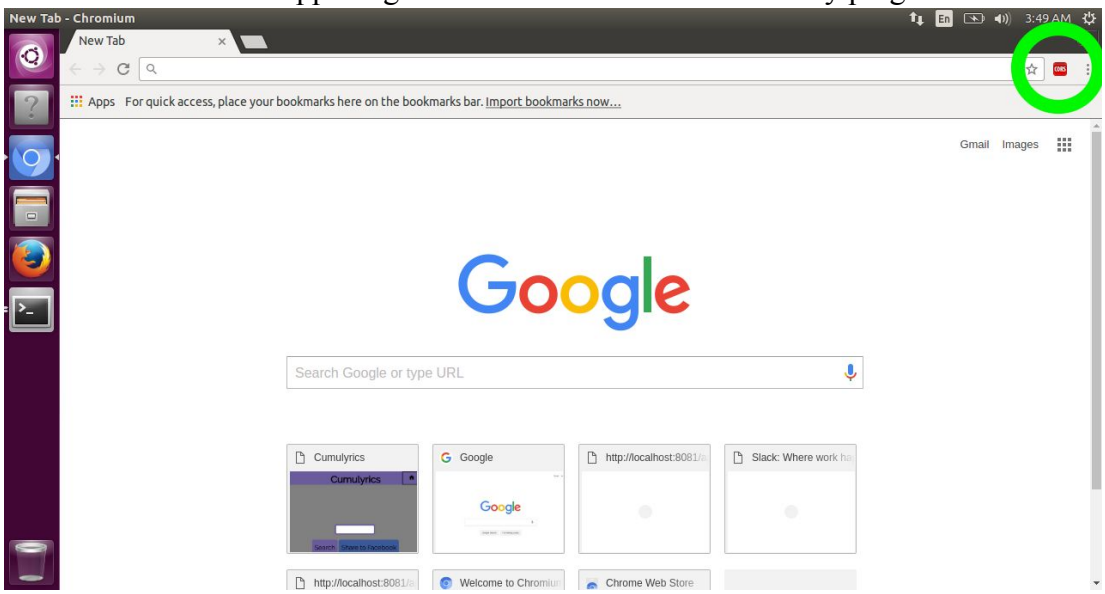
5. Startup instructions

Upon logging into the provided virtual machine, carry out the following steps:

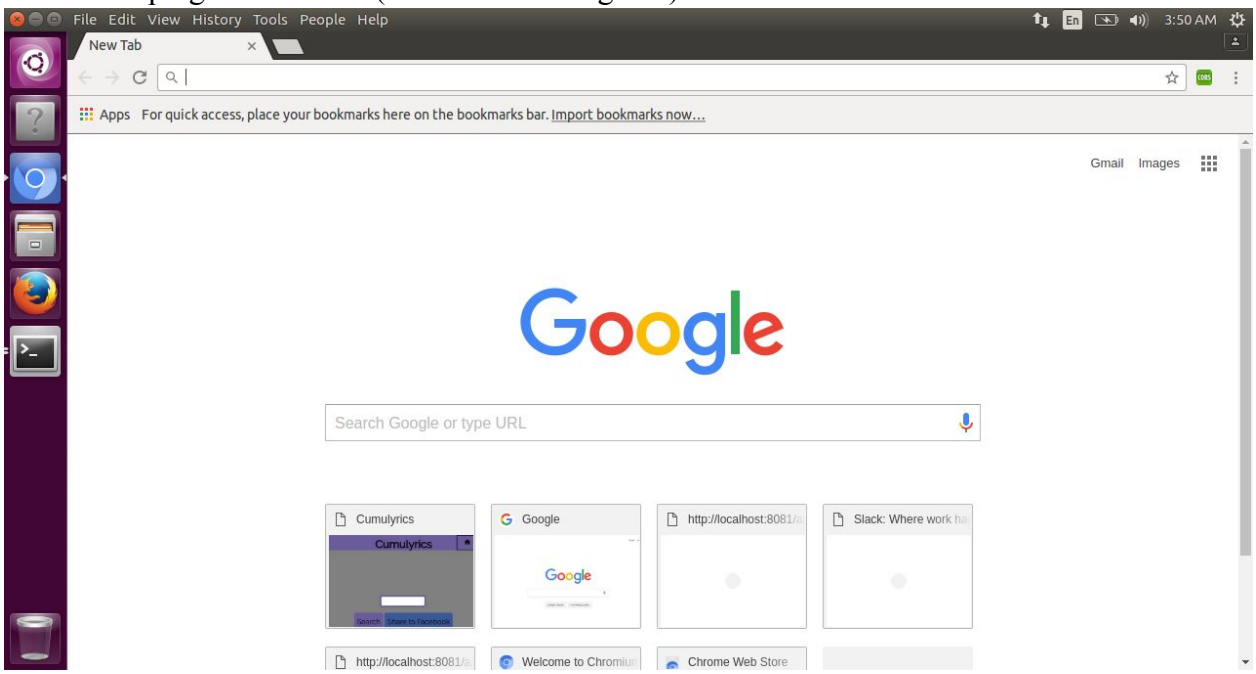
1. Click the application with a question mark icon in the sidebar. (Marked in the picture below.) A terminal window will pop up. Allow it to continue running in the background.



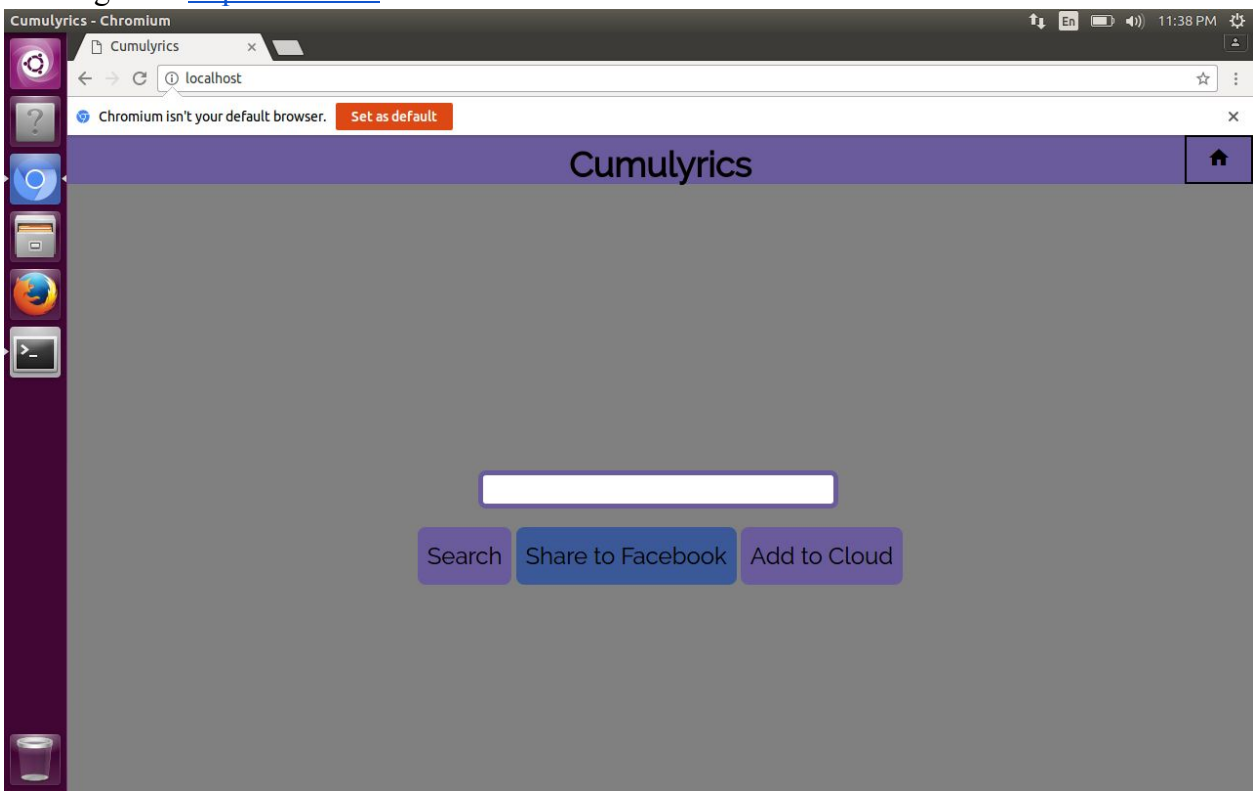
2. Open chromium. (Also visible in the sidebar)
3. Click red button in upper-right hand corner to enable necessary plugins



4. Confirm plugin is enabled (Button will turn green)



5. Navigate to http://localhost . Finished!



Note: Login password for student account is student-pw . “password” was too weak to be used