C-lyrics - A Word Cloud for Lyrics

Executive Summary

C-lyrics is a public website that will generate a word cloud for any given artist based on the most frequently used words that appear across all of the artist's published songs. This product will interface with the EchoNest API which will serve as the database from which we find and analyze the songs. By clicking on a specific word in the word cloud the user can see a list of all of the songs that word appears in and how frequently it occurs in each song. Furthermore, the user can click on any listed song title to see the complete lyrics for that song with the original word that was selected from the word cloud highlighted every time it appears.

C-lyrics is intended for use by the general public. There will be no login required and there is no stored history of previous searches. Because of this we will have very low memory requirements and can run the product off of one server. The user can access C-lyrics using any device running any OS, assuming it has an internet connection. After typing in the artist name and selecting the submit button, the word cloud will be generated and will be able to be shared via Facebook.

1 Introduction

1.1 Purpose

This software testing document is meant to explain, detail, and report on the white-box and black box tests that are run on the implemented software code in order to assure quality. In addition, coverage metrics, justifications and processes will be described. The document also includes project management plans and schedules that were used to accomplish the testing phase.

1.2 Overview

C-Lyrics is implemented using 2 languages, PHP and JavaScript. The backend is implemented using PHP while the front end is built using Java Script with the Angular JS framework. We used the PHPUnit framework for white box testing the back end PHP functions. However, we also needed to perform some whitebox testing on the AngularJS frontend implementation. To test code implemented using AngularJS we used the Karma testing platform which works conveniently well for the purpose. To perform our blackbox end-to-end tests we used the Protractor framework.

The document is therefore organized into two main sections, Unit testing, and Acceptance testing. Details about each testing scenario are explained in subsections of these two main sections. The project management documentation can be found in the appendices which comprise the last section of this document. ## **1.3 References** [1] IEEE. IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications. IEEE Computer Society, 1998.

[2] "word cloud". Oxforddictionaries.com (January 31, 2015)

[3] EchoNest API documentation (January 29, 2015)

[4] A document to remind us the definitions of each UML symbol UML Cheatsheet (February 17, 2015)

[5] Gliffy, a tool to create flowcharts and diagrams. Gliffy.com (February 17, 2015)

[6] Wikipedia definition of black-box testing. [http://en.wikipedia.org/wiki/Black-box_testing] (February 17, 2015)

[7] Wikipedia definition of white-box testing. [http://en.wikipedia.org/wiki/White-box_testing] (February 17, 2015)

1.4 Definitions And Acronyms

Fill In From Old Stuff MVC Protractor? FR-Functional Requirements PMP @@@ Add definition for "webscraper" - Code that gathers data from HTML code on a website URL

1.5 Scope

The intended audiences of this document are future development teams who will maintain or add to this software and the client. All testing code is available through the code base account on version control service Github.com. References to file paths within the repositories contained in the code base are given as needed for further study in each appropriate section.

1.6 Black-Box and White-Box Methodology

The techniques used in black box testing were that the test were administered based on the requirements given by each specification. The developing team examined the functionality of the C-Lyrics system without peering into its internal structures or workings. This was applied to specific levels of software testing, that being unit, system, and acceptance testing [6]. The techniques used in white-box testing were that the test were administered based on the actual code written by the development team. White-box testing tested the internal structures or workings of the C-Lyrics system. These tests needed an internal perspective system, as well as programming skills used to create design test cases. Certain inputs were used to exercise paths through the code and determine the appropriate and expected outputs for a certain function [7]. With these techniques, the overall system and quality of the code was improved in functionality and readability for the user. We decided to use Protractor instead of Cucumber for the black box testing of our system because it came built in to the tools, Yeoman and Angular, that we were using for development. Additionally, Protractor made the testing process easier to test functionality as well as was more easily accessible compared to the later. For white box testing we used both PHP Unit as suggested in class as well as Karma. The PHP Unit was used to test the back end of our system while Karma was used to test the front end. We chose to use Karama in addition to PHP Unit because it, like Protractor, is built in to the tools the development team used for development and was able to provide a more robust framework for testing our code than using PHP Unit alone.

2 Unit Testing

The reason why unit testing was administered to all of the functions below was because they was tested against the requirements specified. The developing team decided to unit test certain functions because the main system was highly dependent upon them to create a robust system that is fully functioning. The majority of these functions were contained in the lyrics javascript code. The reason being that many aspects of the C-Lyrics system were dependent upon manipulation of the words and lyrics that were given by the API. Therefore, the majority of main functions within the C-Lyrics system were handled in the lyrics javascript document. As a future disclaimer, some of the tests do not pass because we did not meet the requirements met in the first implementation, therefore, there were some errors that could not be caught given the time constraints. Each test below will have a short description of what will be expected from the test and what will be tested based on the function. Additionally, each milestone that will be satisfied will be associated with that test.

2.1 Karma Testing Software

Karma is a testing tool that is used to test web-pages built with AngularJS. It is essentially running several compatibility tests to determine if C-lyrics is running and operating on various web browsers. Each Java Script function has a test suite applied to it. Each suite describes a particular functionality containing several tests. Each test contains several assertion clauses which test the functionalities of the software. The following list describes each test that was run with the Javascript Unit software:

- test/spec/services/lyrics.js
  - Test 1: countFrequency(word, lyrics)
    - Milestone G.2, H.1
    - Counts how many times a word will appear in the lyrics array and associates these words to a specific count.
  - Test 2: selectMostFrequent(words, N)
    - Milestone G.2, H.1
    - Tests if the function returns the top N words from the words array to be selected for the word cloud.
  - Test 3: getSongsTitle(words)
    - Milestone H.2
    - Tests if the function returns the title of the song that contains the word that is being taken in.
  - Test 4: formatTop(songs, N)
    - Milestone G.2, H.1
    - Tests to see if the function returns the N most frequently occuring words not including stop words.
  - Test 5: extractWords(songs)
    - Milestone G.2, H.1
    - Tests to see if the function will extract all stop words from the list so that they don't appear in the WC.
  - Test 6: removeDuplicates(words)
    - Milestone G.2, H.1
    - Tests to see if the duplicate words that are going to be in the WC are removed.
- test/spec/controllers/main.js
  - Test 7: main controller functionality
    - Milestone G, J
    - Makes sure that the main controller is working properly.

2.2 PHPUnit Testing Software

The PHPUnit software was used to run unit testing on the backend PHP code. These tests were mainly geared towards achieving milestones for the autocomplete functionality and gathering the word cloud data (milestones G.1 and G.2 of the PMP). The following list describes each test that was run with the PHPUnit software:

- Test 1: testQueryArtist()
  - Milestone G.1
  - Ensures that, given a correct artist name, the autocomplete returns results that are not empty
- Test 2: testQueryArtistSize()
  - Milestone G.1
  - Given a correct artist name, ensures that autocomplete returns the correct number of results
- Test 3: testQueryArtistError()
  - Milestone G.1
  - Given an invalid input, ensures that autocomplete returns null instead of incorrect data
- Test 4: testGetApiKey()
  - Milestone G.1
  - Testing to make sure the EchoNest API key is correct and not empty
- Test 5: testGetClient()
  - Milestone G.1
  - Ensures that a client connection to the databases can be established using the API key
- Test 6: testGetArtistApi()
  - Milestone G.1
  - Checks that a client connection to the artist API is valid
- Test 7: testGetArtist()
  - Milestone G.2

  - Ensures that, given an artist name, the webscraper receives the right artist name

- Test 8: testGetUrlList()
  - Milestone G.2
  - Confirms that gathering URL's of songs for a given valid artist name functions properly

- Test 9: testGetUrlListSize()
  - Milestone G.2
  - Given a valid artist name, ensures that the webscraper returns the correct number of URL links for songs

- Test 10: testGetUrlListEmpty()
  - Milestone G.2
  - Given an empty artist name, ensures that the webscraper returns zero URL links

- Test 11: testGetUrlListFaultyInput()
  - Milestone G.2
  - Given faulty artist name input, ensures that the webscraper returns zero URL links (no bad outputs)

- Test 12: testGetLyrics()
  - Milestone G.2
  - Given a valid artist name, asserts that the lyrics list will not be empty and equals the correct number of songs

- Test 13: testGetLyricsFaultyInput()
  - Milestone G.2
  - Given faulty artist name input, ensures that the lyrics list contains no results and throws an error code

- Test 14: testGetLyricsEmpty()
  - Milestone G.2
  - Given no artist name input, ensures that the lyrics list contains no results and returns an error message

- Test 15: testScrapeBetween()
  - Milestone G.2
  - Given a HTML page, ensures that the webscraper functions properly and takes the correct data from the site

2.3 Measures and Coverage

The tests run using Karma software sufficiently cover all of the functionalities for the JavaScript frontend. Only certain functions were tested since the majority of the code calls functions without specific testable return output. The testing code written actively covers these functions by ensuring that larger, encompassing functionalities are working properly.

The PHPUnit tests that were run have sufficiently explored each necessary function of the PHP backend. By checking how the code processes various types of inputs (valid, faulty, and empty) as well as checking the communication of different components of the code, we can be sure that the code will not provide functional errors in the processes of generating autocomplete results as well as pulling song lists and lyrics from lyric websites. In this test, we have run 15 tests to measure the robustness of the code. All of them have executed without errors, providing sufficient evidence that the PHP code runs properly.

3 Acceptance Testing

3.1 Protractor Testing Environment

Protractor is a testing framework used to run end-to-end type tests for AngularJS implementations. These tests will be used as acceptance tests since they reveal if the user defined requirements as described in the SRS are met. The nature of Protractor code makes it difficult to state tests by function name as was done in the previous section. To compensate, the criteria for each unit tests are listed below, the path to each test on the Github repository is also given for further reference.

- Main Page Layout (frontend/test/spec/controller/main.js)
  - Test that the first page when opened has search bar
  - Grey background
  - Submit button purple
  - Text field outlined in purple
- Search (frontend/test/spec/controller/main.js)
  - If input artist name that doesn't exist
    - Error "Artist not found"
    - Empty submit error
    - Facebook and Add to Cloud not on screen
  - If artist's name correct

        * Word cloud appears
        * White background appears
        * "Add to Cloud" appears
        * "Facebook Share" appears

- Autocomplete (frontend/test/spec/controller/main.js)
    - After pause in user's input to search bar autocomplete field becomes visible
    - Test scroll bar
    - Pictures for each artist
- Word Cloud (frontend/test/spec/controller/main.js)
    - Word size depends on frequency
    - Words multicolored
    - Words link to songs list page
    - Stop words filtered out (ex. "it," "the," and "a")
    - Word cloud generation is 10 sec - 1 min
- Song List (frontend/test/spec/controller/songlist.js)
    - Songs sorted from highest frequency to lowest
    - Ability to select a song title to go to lyrics page
    - List will be aligned left
    - Title is clickable
    - Navigation buttons work
        * "Back to Cloud"
- Lyrics (frontend/test/spec/controller/songlyrics.js)
    - Selected word highlighted
    - Lyrics will be aligned left
    - Navigation buttons work
        * "Back to Cloud"
        * "Back to Songs"

3.2 Measures and Coverage

There is one black-box test for every requirement. One test is enough since there are a set number of pages and functions (buttons) that the user has access to. The tests will figure out for example, if the add to cloud button works. Adding lyrics to the cloud is a requirement, the test will check if the connection is made. The correctness of what actually shows up in the cloud once new lyrics are added is a white-box testing problem. White-box testing concerns are handled in the previous section of this document. Therefore, our testing policy assumes that as long as each customer requirement is tested at least once, we have achieved satisfactory test coverage.

Furthermore, the development team does not have enough resources to test each requirement more than once. The product will also not be delivered on time if a more comprehensive testing regimen is applied for acceptance testing procedures.

4 Quality Testing

4.1 Reliability Tests

In the SRS we stated that we would test the reliability of our product based on the percent accuracy over 1500 searches, the minimum required percent being 95 percent and the goal being 98 percent. During testing we decided that it was not feasible and unnecessary for us to run the program 1500 times in order to gauge the reliability of our system given time and resource constraints. Instead, we tested our system by searching for distinct 15 artists across different genres running each search 3 times to ensure that it is reliably accurate. Because we are grabbing all of our artist and lyric information from the API, it can be assumed that our system will also work for all remaining artists in the API's database.

4.2 Availability Tests

The SRS states that we will test the availability of our system by the percentage of time that it is available out of 80 hours of monitoring the system, 95 percent of the time being the minimum required and 97 percent of the time being the goal. This was simulated by having each of the six members of the development team monitor the system for roughly 13 hours adding up to a total of 80 hours that the system was monitored. The system was not only monitored using different OSs but also using different web browsers such as Chrome, Firefox and PhantomJS. We feel that this is sufficient proof that our system will be available as needed and that this test also fulfills all application testability requirements.

4.3 Other Quality Tests

Other forms of quality testing such as maintenance tests and security tests were not administered because they did not fits the needs of this project. Because there is no projected maintenance for this project and there are no foreseeable security risks, we feel that it is appropriate to omit them.

5 Appendices

5.1 Declaration of Milestones

The milestones below were conceived for and reported on the PMP document. The below table is an excerpt from the original PMP document. They are reproduced here for clarity and reference.

- Testing and Final Delivery. 3/11/15*
- Milestones G-J
  - G: Testing of the Home Page
    - G.1: Search bar with autocomplete functionality when typing in an artist's name
    - G.2: WC generation with words that can be selected to take the user to the Songs Page
    - G.3: Share button to upload the WC to Facebook
    - G.4: Add to Cloud button to create a new WC based off of words commonly used by both of the specified artists
  - H: Testing of the Songs Page
    - H.1: List of songs sorted by how frequently the selected word is used in each song
    - H.2: Song titles in list able to be selected, taking the user to the Lyrics page
    - H.3: Back to Home button takes the user back to the Home Page with the WC still displayed and the artist's name still in the Search Bar
  - I: Testing of the Lyrics Page
    - I.1: Lyrics displayed on page with the selected word highlighted every time it appears in the song
    - I.2: Back to Songs button takes the user back to the Songs Page with the same list of songs still displayed in the same order

    - I.3: Back to Home button takes the user back to the Home Page with the WC still displayed and the artist's name still in the Search Bar
- J: Testing of the entire product to ensure all pages work together as specified in the SRS

5.2 Deviations From PMP

5.2.1 Deviations from Schedule

Originally, milestones were organized according to the table below, where each letter corresponds to a mile stone in each phase of the Waterfall software development cycle. The Milestones of interest are letter G-J. A description of milestones G-J can be found in the previous appendix 5.1. Significant deviation has occurred in project schedule cause by changed deliverable due dates.

The client pushed up the deadline for the testing document delivery to Wednesday March 11 2015. Therefore, milestones G-J should be extended until that date in the image above, representing the change in our PMP.

5.2.2 Deviations From Milestone Planning

Team member Milad Gueramian was unable to contribute to testing activities due to loss of development equipment (laptop computer). Therefore, he was unassigned from milestone G and was given the task of completing and preparing the testing document. A milestone was created for this task in the Github Issue tracker tool which, as described in the PMP, is use to coordinate and keep track of development team activities.

Furthermore, team member Jeff Kang was also reassigned to create the testing document with Milad Gueramian as the difficulty and resource usage of this task was not considered in the PMP.