Cumulyrics

Detailed Design Document

Team 19

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

1. Overview

1.1 Executive Summary

This document thoroughly defines the design we have chosen for the system, which allows it to operate as best as it possibly can and satisfy all requirements stated in the SRS. The document contains many diagrams that vary in abstraction, information hiding, and modular grouping. There are higher level diagrams, such as software architecture and top level data flow. Our document also contains lower level ideas such as specific classes and display components in the class diagrams and the second level decompositions of the data flow diagram. In addition, there are other diagrams that focus on what data is being transferred between different modules and components. The sequence diagrams, communication diagram focus on the previously stated view point. There are also diagrams, such as the first level decompositions and component diagrams, that are intended to emphasize the organization of the system, showing how all of the components are connected. Other diagrams, use case and state machine, are in the document to show the different ways users can interact with the website. Finally, the document contains a diagram describing our database design. It includes the tables in the database and all of the fields in each table. Each diagram has an explanation highlighting how the design helps the system satisfy all requirements. All diagrams serve a different purpose in explaining and documenting the design, and together they make up all aspects of the system.

1.2 Purpose

The purpose of the Software Design Document is to provide a description of the design of a system fully enough to allow for software development to proceed with an understanding of what is to be built and how it is expected to built. The Software Design Document provides information necessary to provide description of the details for the software and system to be built.

1.3 Intended Audience

This document is intended for the stakeholders, which include the CPs, TAs, and professor, and the developers,Team 19.

1.4 References

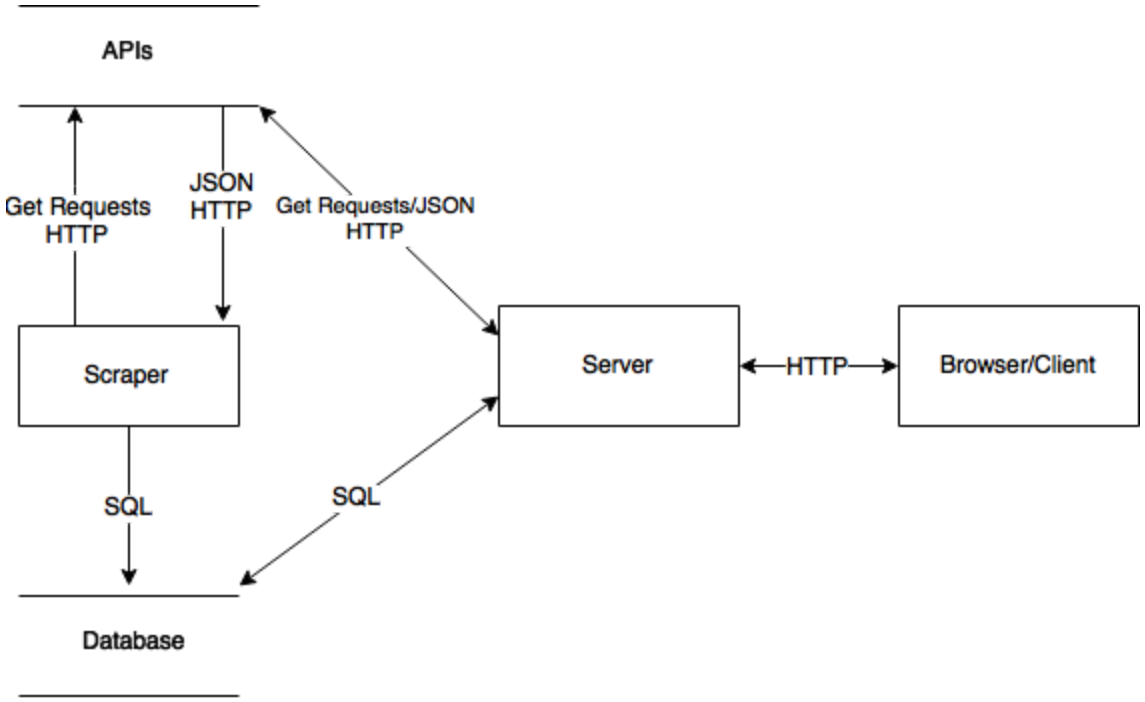
Reference	Relevant Information
Team 17 SRS	Available through blackboard under the Project 1 - Design section of the Assignments tab.
Lecture 7	Available through blackboard - Content tab

1.5 Definitions

Term	Definition
SRS	Software Requirements Specification, the document that contains the agreed upon requirements for the system.
API	Application program interface (API) is a set of routines, protocols, and tools for building software applications.
Decomposition	The process of breaking up large problems or systems into smaller more manageable and easier to understand parts.
RDBMS	This stands for a relational database management system, which is a database management system (DBMS) that is based on the relational model as invented by E. F. Codd. It is one of the most commonly used database management systems.

2. Software Architecture

2.1 Architecture Diagram



Explanation -

This diagram outlines the architecture for the proposed system design. Each rectangle represents a component that encapsulates certain functionality necessary for the system to operate. The arrows indicate the protocols and types of communication between each component. This diagram is representative of the system from a physical viewpoint to illustrate the physical representation of the system.

This representation of the architecture is meant to satisfy software design principles. Abstraction is considered because each module represents a smaller subsystem that needs to be designed which when merged together form the overall system. The architecture approaches modularity through logical cohesion and common coupling. Each module has some type of functionality different from the other modules and each module communicates with the others by simply sending data meant to be utilized by the receiving module. In terms of information hiding this representation does not show how data is being manipulated by each module or the form of the data being sent between modules.

Browser/Client - This component represents the front end section of the system which will handle what the user interacts with and views. This component communicates directly with the server to request and receive relevant data.

Server - This component represents the backend section of the system that handles all the data manipulation and requests made by the Browser/Client. The server acts as the middle made between the database and third party APIs to provide the Browser/Client with the required data needed to satisfy the system requirements.

Scraper - This component is what gathers all the data needed from third party APIs to be stored in the database. The scraper will be run prior to the system becoming functional and will update at yet to be determined intervals in order to ensure up to date data.

APIs - This component is the source of all data needed to satisfy the system requirements. The APIs will provide song lyrics, artist names, artist pictures, and song titles.

Database - This component will store all data that is subject to the one second performance requirement including artist names, artist pictures, song titles, and words that will make up each artist’s wordcloud (See 3.8 Database Schema).

2.2 Architectural Styles and Patterns

2.2.1 MVC Architectural Style (Model View Controller)

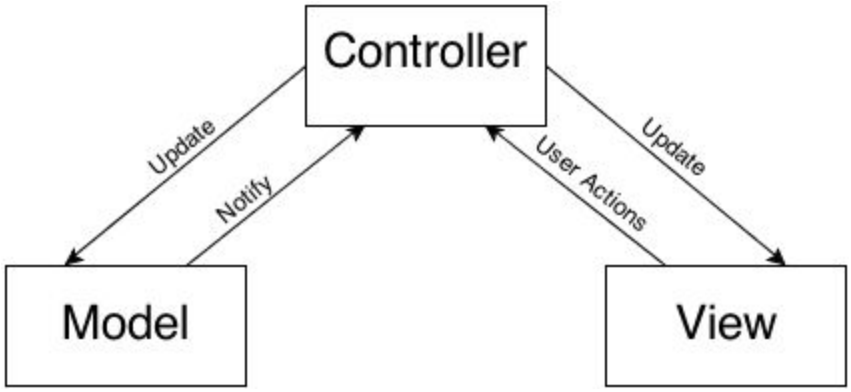
MVC Style separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other.

The model component - manages the system data and associated operations on that data.

The view component - Defines and manages how the data is presented to the user.

The controller component - Manages user interaction and passes these interactions to the View and the Model.

MVC will be used for Cumulyrics because of the multiple ways to view and interact with data. To increase modularity, facilitate maintainability, and augment readability MVC architectural style separates the application logic with interface.This allows the data to change independently of its representation and vice versa. It also supports easy presentation of the same data in different ways.



2.2.3 Three-Tier Client Server Architecture

In a client server architecture, the functionality of the system is organized into services, with each service delivered from server(s). Clients are users of these services and access servers to make use of them. We are using the 3- Tier Client Server Architecture because data is shared from a common database and accessed from various locations.

Data

Maintains the applications data such as songs, artist, artists url, etc, sql queries. This is stored in a relational database management system(RDBMS). API calls are also made in this tier to retrieve data such as lyrics. Connections with the RDBMS are managed in this tier.

Middle

The middle tier implements the logic, controller logic, and presentation logic to control the interaction between the applications' client and data. Dictates how clients can and cannot access data.

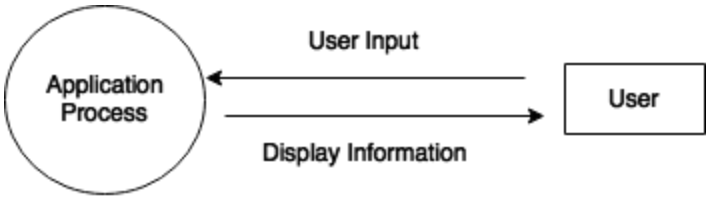
Client

The client tier is the application's user interface connecting data entry forms and client side applications. It displays data to the user. Users interact directly with the application through user interface. The client tier interacts with the web/ application server to make requests and to retrieve data from the database. It then displays to the user the data retrieved from the server.

3. Detailed Design Diagrams

3.1 Data Flow Diagram (DFD)

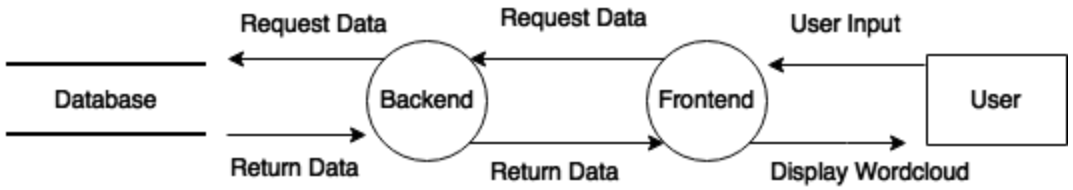
3.1.1 Top Level Dataflow Diagram



Top Level Data Flow Diagram Explanation:

The top level data flow diagram shows how data flows through the program at the most abstract level. The only external entity for this application is a single user, and this user interacts with the application process, by providing inputs and receiving a desired output.

3.1.2 First Level Decomposition

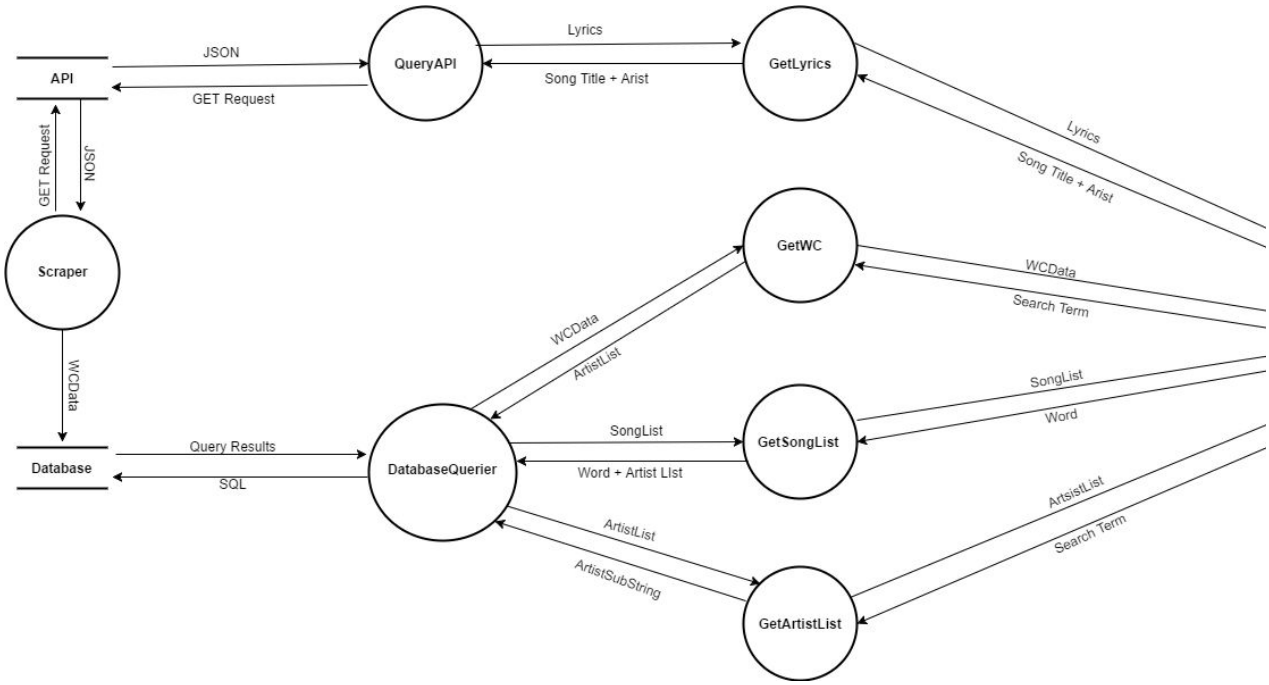


First Level Decomposition Explanation:

The first level decomposition for the data flow diagram provides a closer look at the design of the overall application process. As this is a web application, there must be a frontend (the user's browser) and a backend to the application. The backend will be responsible for communicating with the database, while the frontend will interact directly with the user. A request for data reaches the database via the frontend and backend processes, and requested data flows back along the same path.

3.1.3 Second Level Decomposition

3.1.3.1 Backend Decomposition

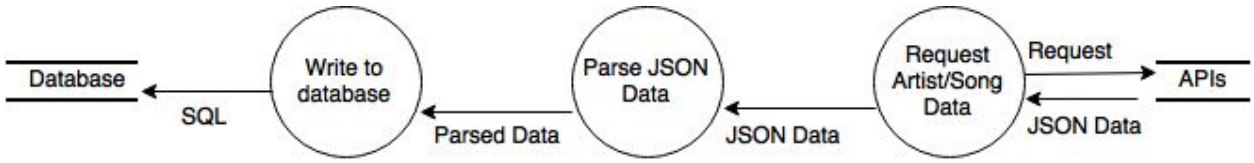


Backend Decomposition Explanation:

This second level decomposition data flow diagram expands the backend process to take a close look at the flow of data through it. The four main requirements for the backend are to return a list of artists matching a search term (SRS 3.2.1), return data to create a wordcloud from the lyrics of one or more artists (SRS 3.2.4), return a list of songs containing a word in the word cloud (SRS 3.2.8), and return the lyrics for a specific song (SRS 3.2.10). In this design, word cloud data, songs belonging to an artist, and a list of all artists can be retrieved directly from the database, so a database query process is responsible for retrieving those pieces of information. Full lyrics for each song will not be stored in the datab

ase, but will instead be retrieved from an external API via an API query process. The scraper is a separate process that will not run at the same time as the main application process, and is responsible for initially filling the database with artists, songs, and word cloud data. (See section 3.1.3.2 for scraper decomposition)

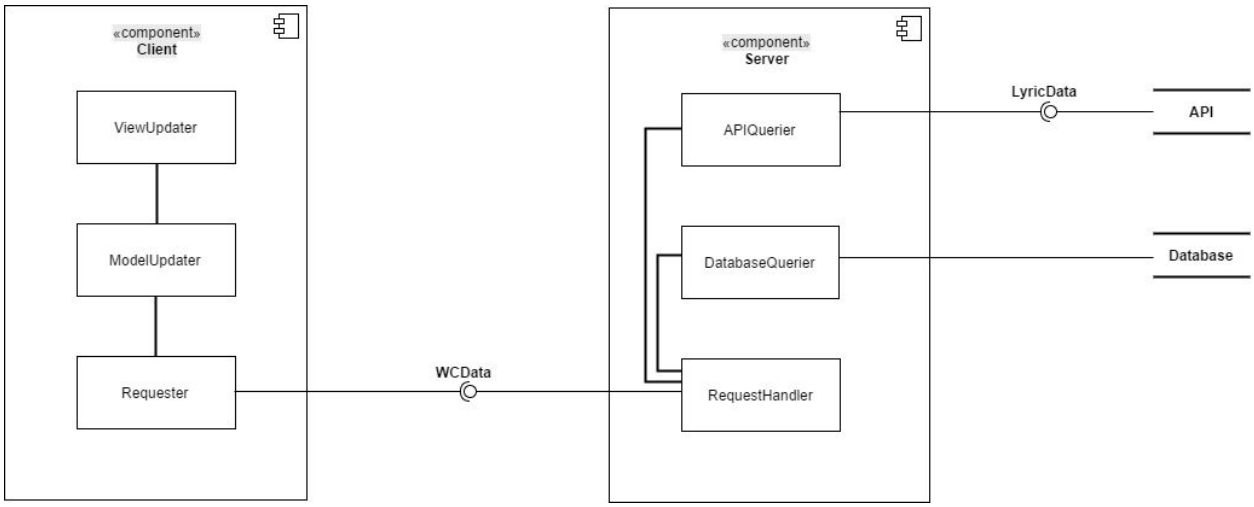
3.1.3.2 Scraper Decomposition



Scraper Decomposition Explanation:

This second level decomposition data flow diagram provides a breakdown of the flow of data through the scraper process, which runs separately from the main application process. The scraper communicated with APIs to retrieve lists of artists, lists of songs, and song lyrics, all of which are returned in the form of JSON data. The “parse JSON data” process reads the JSON data, storing the information as local variables, before passing the information to be written to the database. The “Write to Database” process formats the data for the database, and subsequently writes it.

3.2 UML: Component



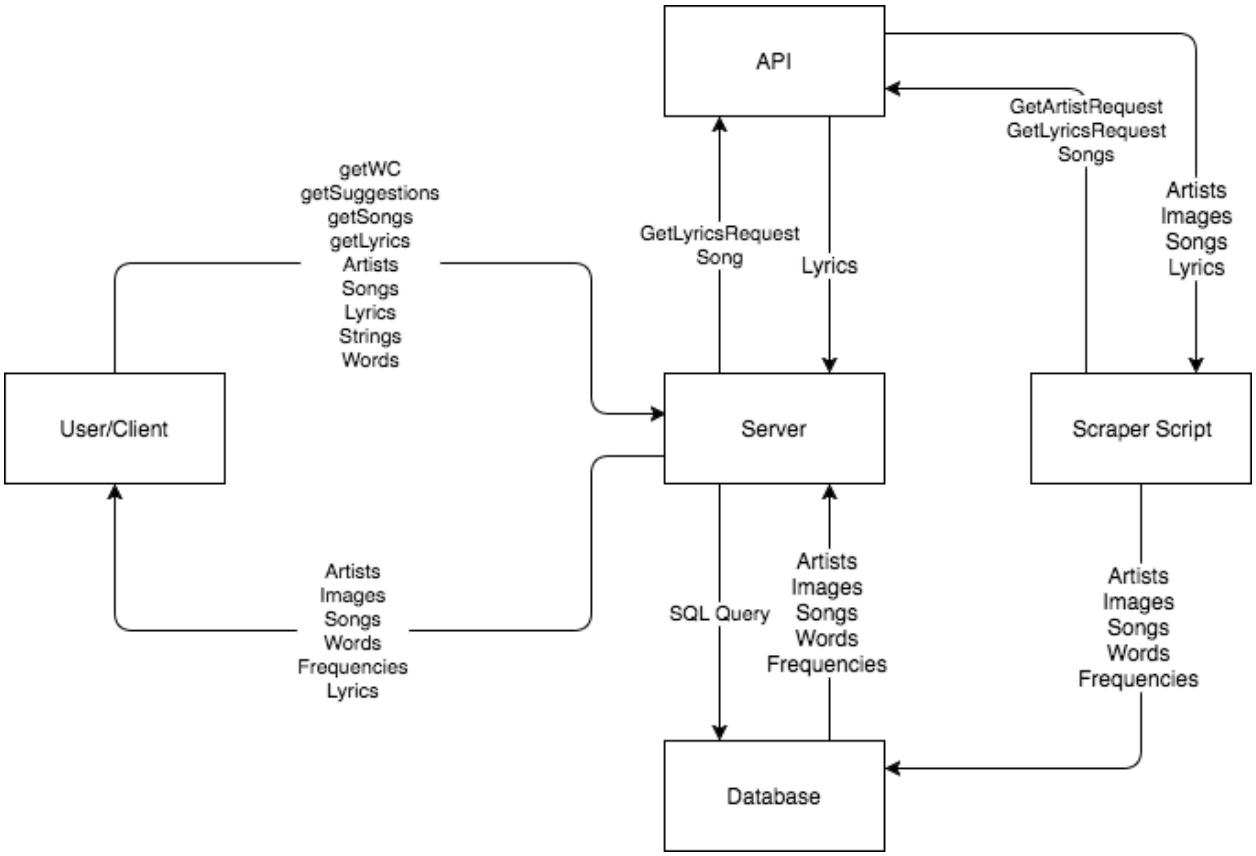
Component Diagram Explanation:

The component diagram represents the system as a relationship between large over arching entities. This diagram highlights the client server architecture of the overall system. The client is comprised of a view updater, model updater, and requester classes that clearly separate the client’s functionality into separate classes that aid in the abstraction of the design. The same abstraction applies to the server component. The classes contained within each component are separated based on functionality. This diagram utilizes information hiding to abstract the details of each class and component so the developers can understand the system at a higher level. The

viewpoint illustrated by this diagram shows the interfaces between components and how the classes in the class diagram relate to the overall system.

This diagram was chosen to represent the client server architecture to give more context to the more detailed viewpoint represented in the class diagrams (See section 3.4).

3.3 UML: Communication

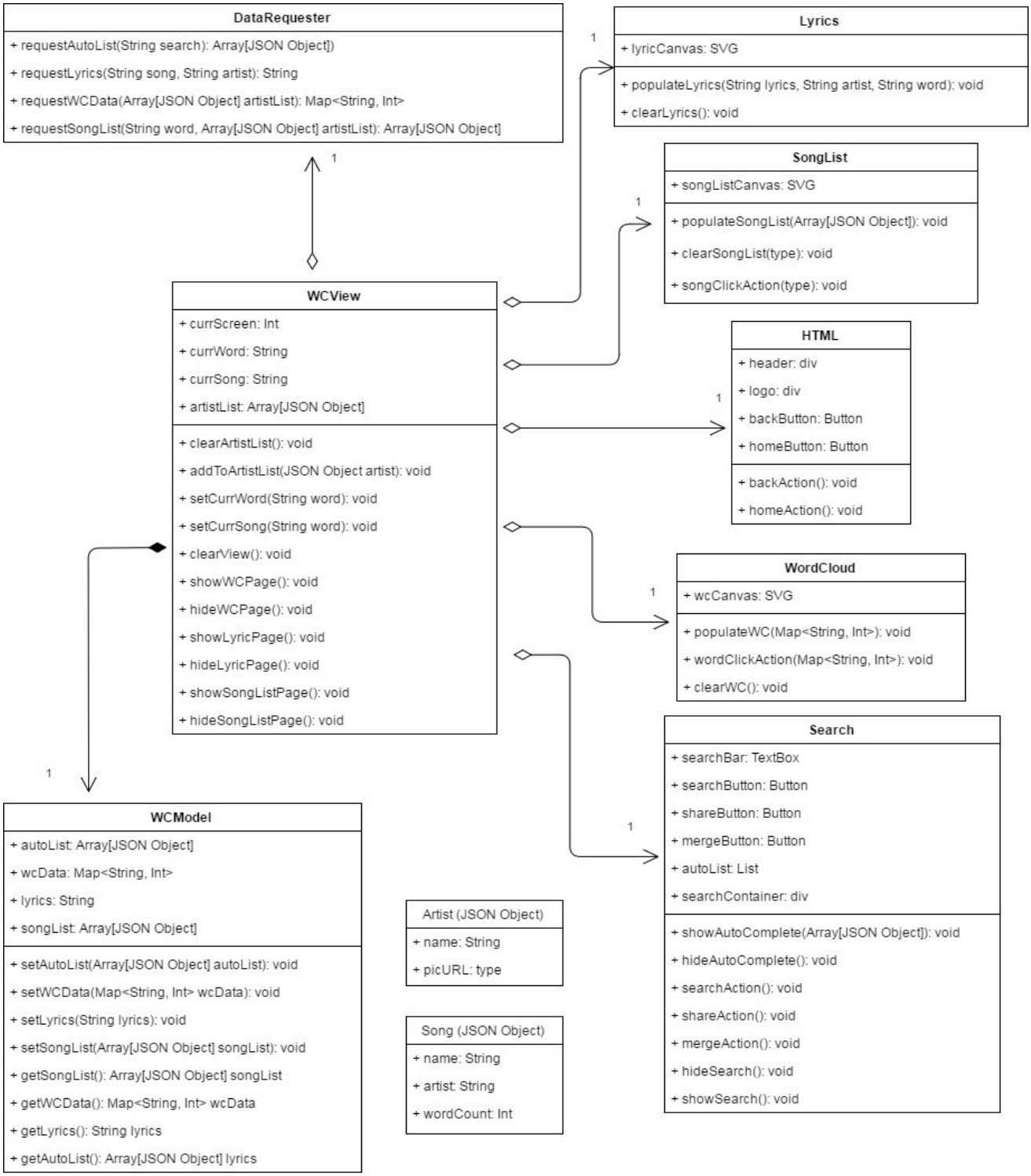


Communication Diagram Explanation:

The communication diagram shows all of the different lines of communication between the modules of the system. It also shows the data that modules will be requesting and sending. The client will request data from the server using the system's get methods and providing parameters. The server and scraper script will use the get API calls from Genius.com and Lyrics.com to retrieve data and send it to the server or enter it into the database, respectively. The database sends data to the server based on SQL query's, which is then sent to the client. All the modules are held together by logical cohesion and intra-module communication is based on common coupling. The actual methods and processes within each module are abstracted away because those are not the main focus of this diagram, but instead it focuses on the different types of data and queries being passed between modules. The design of the communication throughout the system will allow it to satisfy all functional requirements and the performance requirement. It is especially important in completing the performance requirement because we only rely on a third party API for lyrics (the database will be set up before the website is up and running) of a single song. In the other cases we will be accessing our own database which will be much faster and more reliable than third party APIs.

3.4 UML: Class

3.4.1 Frontend Class Diagram

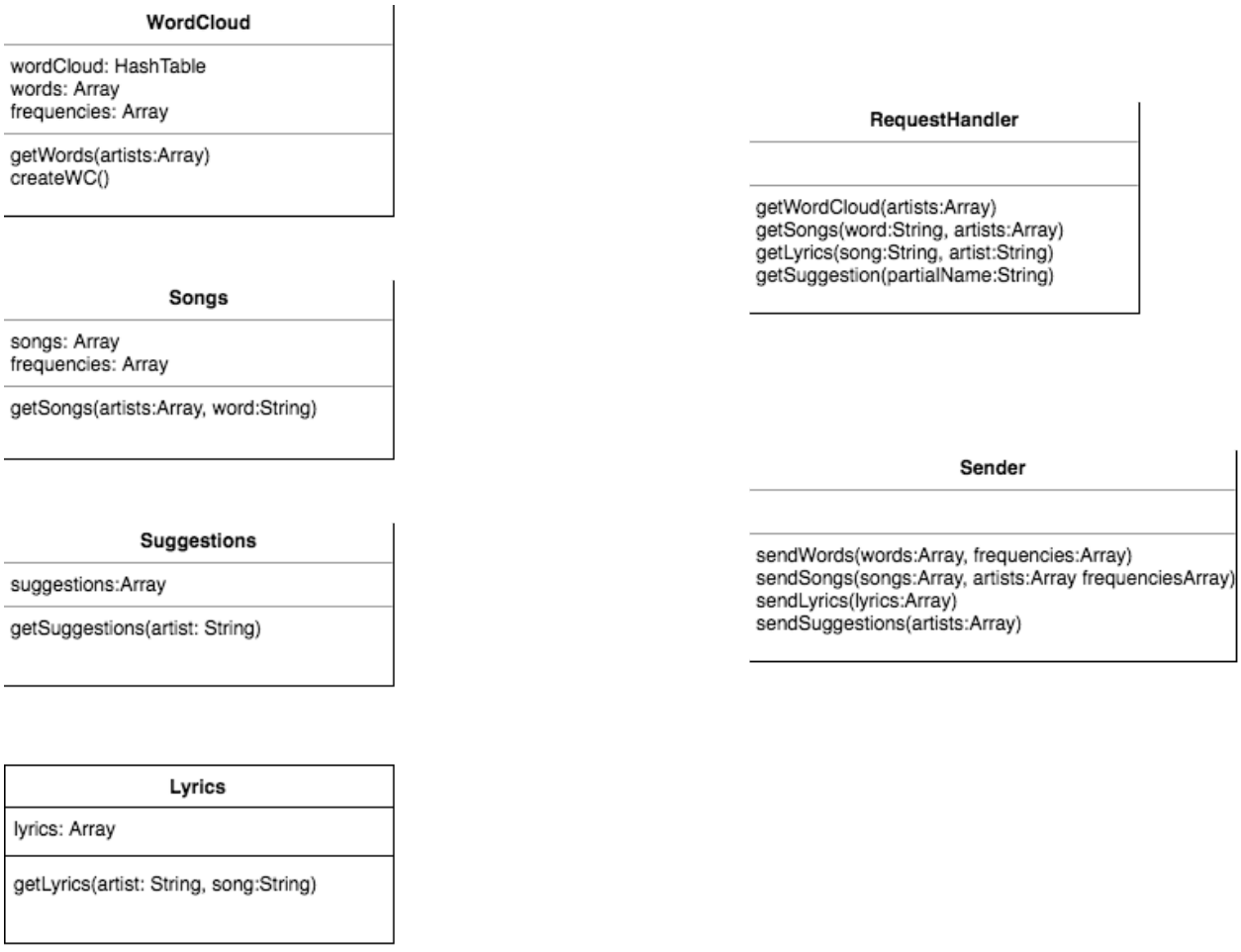


Front end class diagram explanation -

The class diagram has three main classes which are the WCModel, WCView, and DataRequester. These three classes are created to handle specific functionality with regards to the frontend of the system. This diagram is meant to outline the classes needed to accomplish the frontend requirements outlined in section 3.1.1 of the SRS. The diagram also utilizes information hiding coupled with descriptive function names to abstract the functions within each class.

The DataRequester class is the set of functions that interact and communicate with the server. This class handles requesting and receiving data from the server based on user interaction with the system. The WCView class is the set of functions and data members that handles the user interface and is concerned with showing the user relevant data. The classes that are connected with aggregation are concerned with directly manipulating the user interface. The WCModel class handles and manipulates the data that the front end acts upon. This class uses the data obtained by the data requester class and sets the data that the WCView will handle. The JSON objects listed on the bottom of the diagram represent the types of data being transferred between the backend and frontend sections of the system.

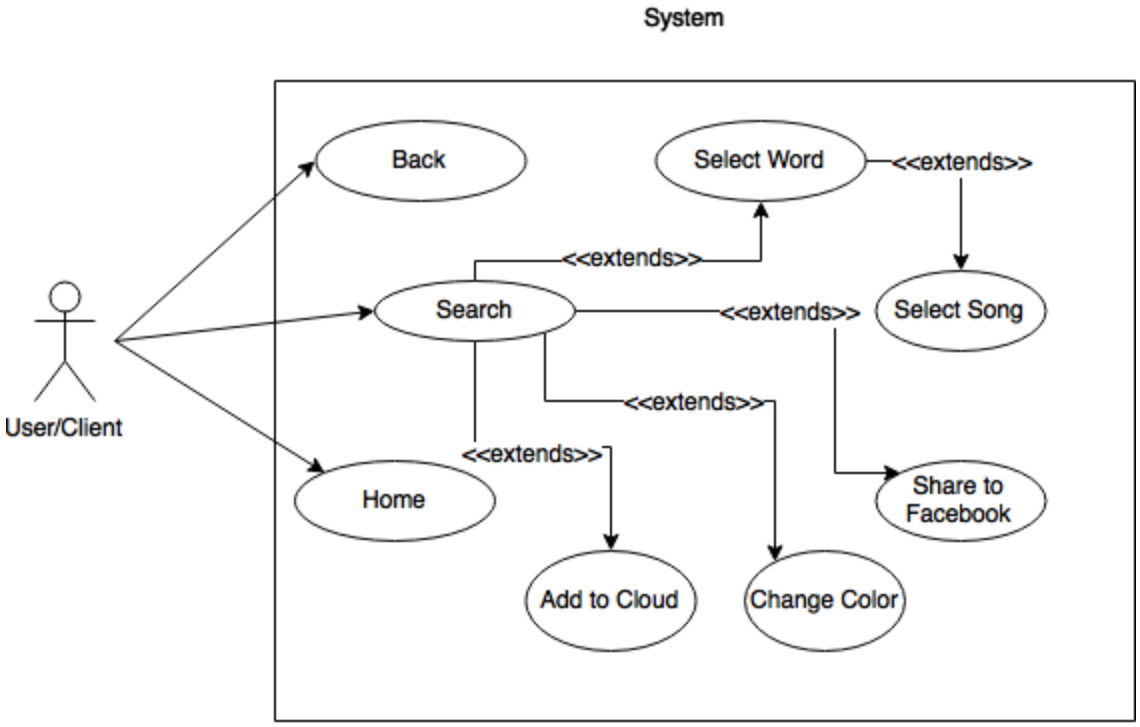
3.4.2 Backend Class Diagram



Backend Class Diagram explanation -

This diagram shows all of the different classes or modules that will be in the web server. There will be a module to receive requests from the front end and then a module that will handle sending data back to the frontend. This allows for two clear places that have the only interaction with the frontend. In addition to these modules, the web server will have a different module for all four major features of the system. The WordCloud module will be used for satisfying requirements 3.2.4 and 3.2.5. The Songs module will be used for satisfying requirement 3.2.8. The Lyrics module will be used for satisfying requirement 3.2.10, and lastly, the Suggestions module will be used to satisfy requirement 3.2.1.

3.5 UML: Use Case



Use Case Diagram Explanation -

The Use Case diagram is used to explain the different ways Actors, or entities are able to interact with the system. The User/Client should be able to interact with the system through these use cases made available through the web page interface. The search use case is when a word cloud should be generated and all use cases that extend search are reliant on a word cloud being generated before being able to access that use case. Each use case represented by an oval is an

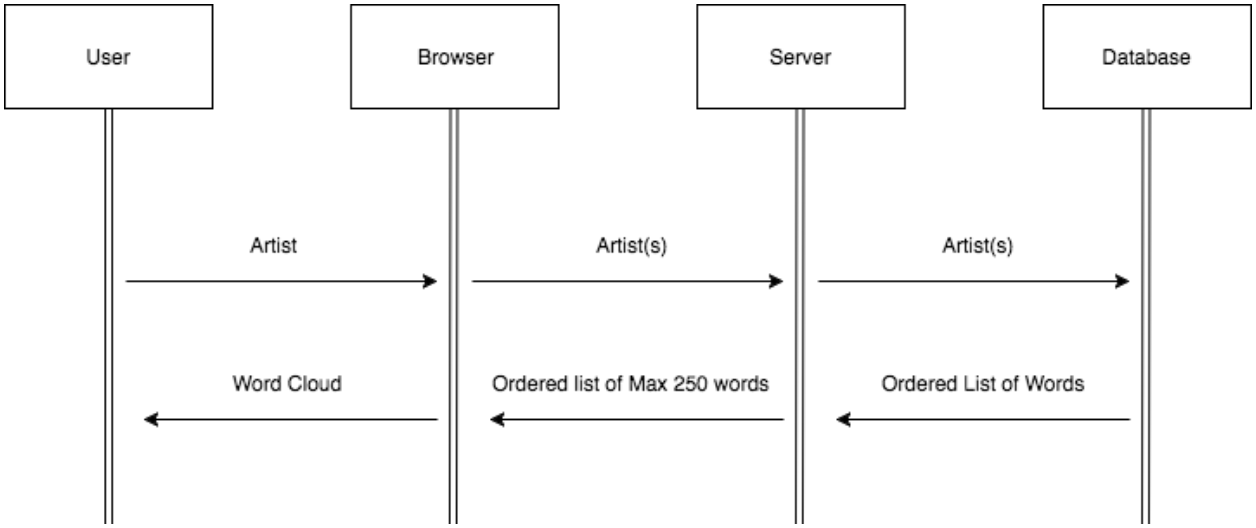
abstraction of the processes that handle the transition between states of the system from the perspective of the client (See 3.7 UML: State machine).

The following are explanations for the specific use cases listed in the above diagram. The back use case represents when the user presses the back button to go to the previous page which can be viewed as the “Back” transitions in section 3.7 State Machine. The home use case represents when the user presses the home button represented as the “Home” transition in section 3.7 State machine. The search use case represents when a user selects an artist to generate a word cloud based on. The Change Color use case represents when a user selects radio buttons to alter the color of a generated word cloud. The Add to Cloud use case represents when a user chooses to add another artist to the already generated word cloud. The Share to Facebook use case represents when a user chooses to share a picture of a word cloud to facebook. The Select Word use case represents when a user chooses a word from a word cloud. The Select Song use case represents when a user selects a song from the song list page.

3.6 UML: Sequence

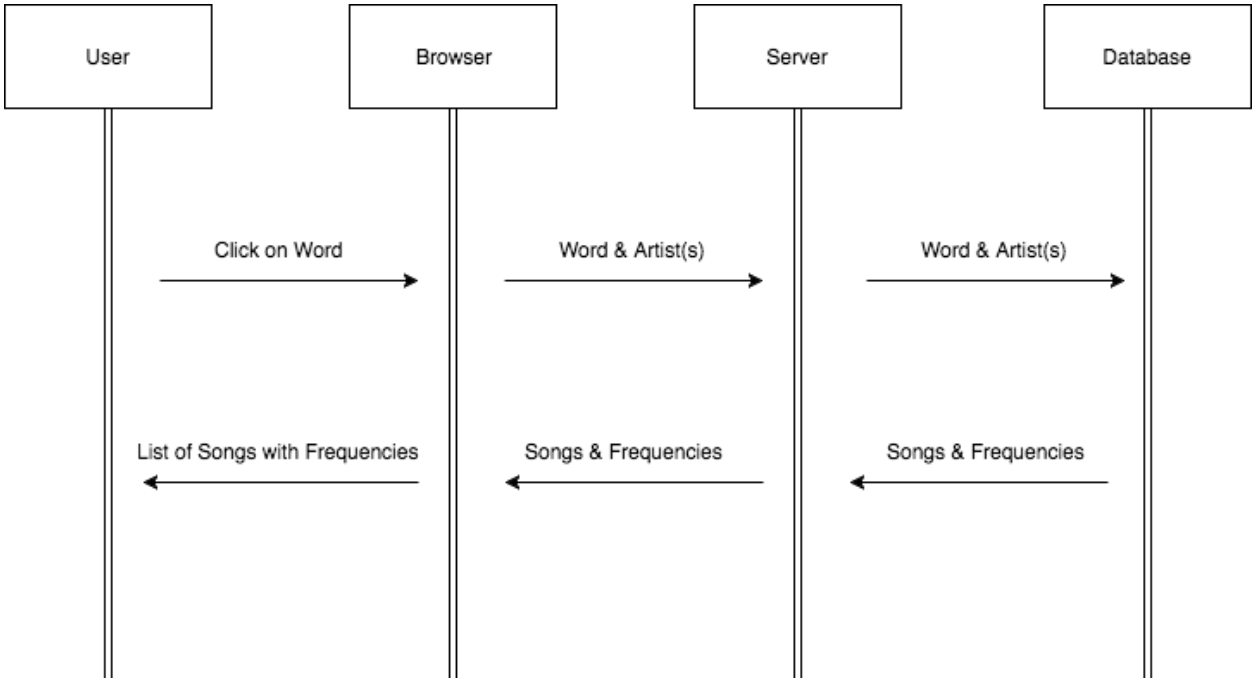
All sequence diagrams only display the data being transferred, but the specific methods and commands being used on the data are hidden. Each component represented by a rectangle is an abstraction of the processes used to get, manipulate, and send the data. Additionally, the diagrams use logical cohesion to group modules and common coupling to organize the communication between modules.

Artist Cloud Creation Sequence



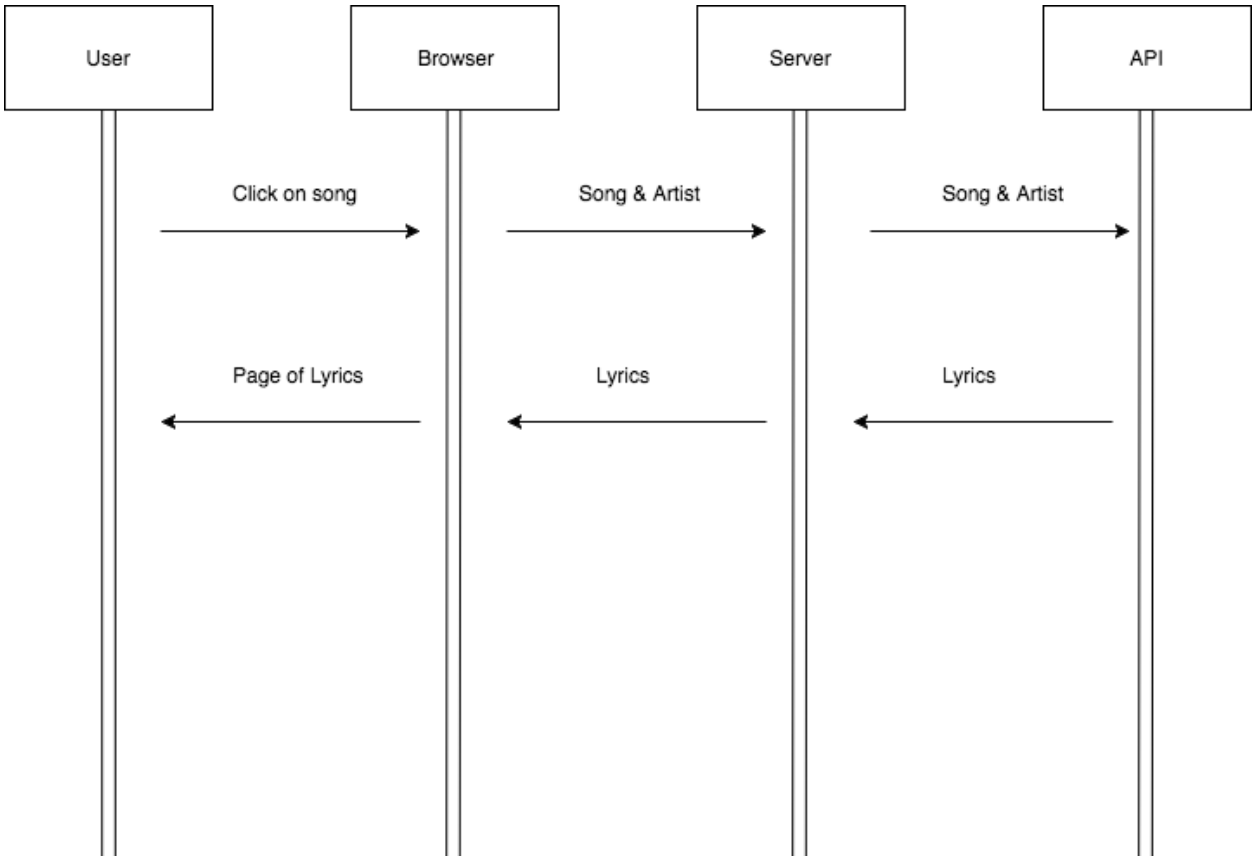
The first and most important sequence is when a user tries to create a word cloud with an artist or multiple artists. The user will enter an artist name and depending on whether the user clicks the Add to Cloud button or the Search button the Browser will send artists or an artist, respectively, to the web server. This satisfies requirement 3.2.2 and part of 3.2.5. The server will then get an ordered list of words with frequencies for each artist from the database. The server will calculate which words were used most and send the top 250 most prevalent words to the browser, which will display these words in the form of a word cloud to the user, with the artists' names that were used to create the word cloud as the title . This will satisfy functional requirements 3.2.3, 3.2.4 and the rest of 3.2.5 from the SRS document.

Songs Containing a Selected Word Sequence



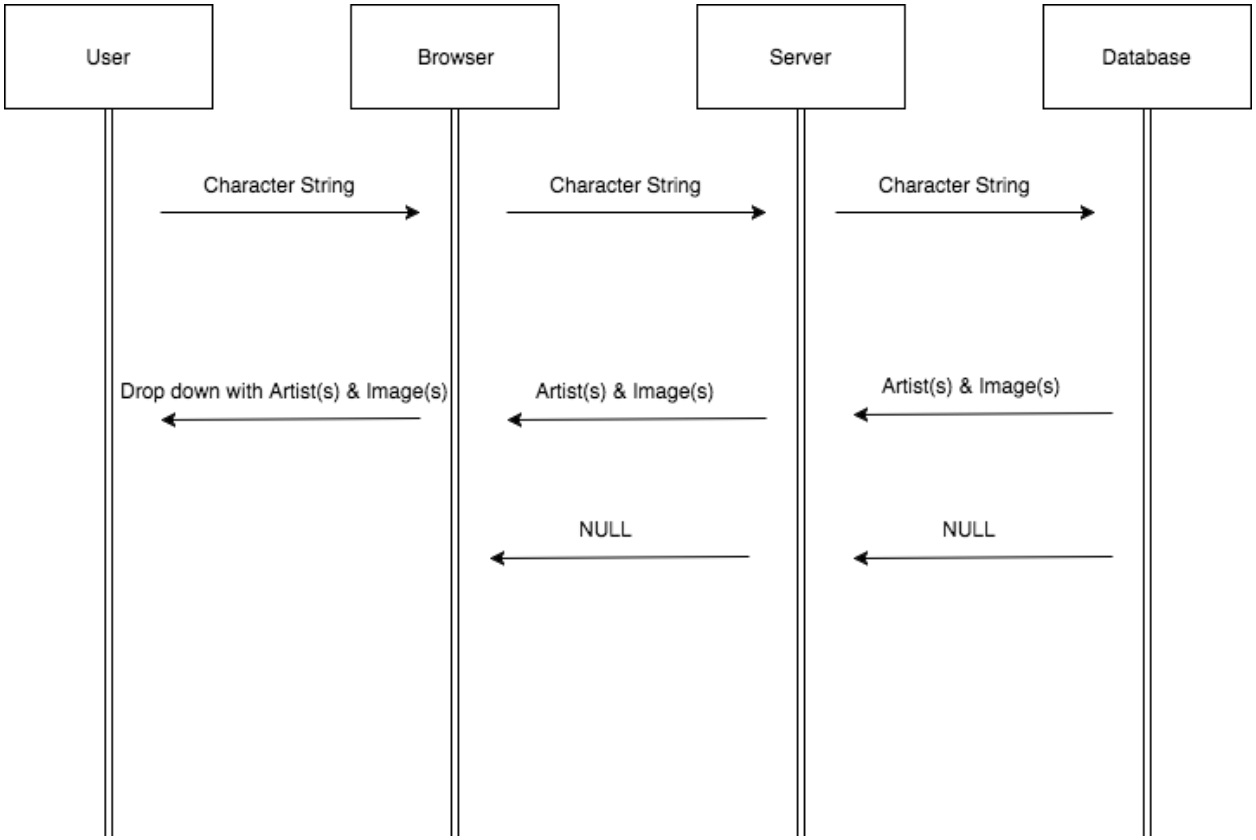
The next sequence occurs when the user clicks on a word within the word cloud. When this happens, the browser will send the selected word to the server, as well as the artist or artists names. The server will then get from the database each song by the artist(s) that contain the word and how many times the word appears in each song. The server will send this information to the browser, which will display the list of song, frequency pairs, as well as the selected word as a title, satisfying requirements 3.2.7 and 3.2.8.

Song Lyrics Sequence



The song lyrics sequence is initiated by the user clicking on a song in the song list page. The browser will send the song and artist to the server, which will use the received information to get the lyrics of the song from an outside API. The server will then send this to the browser, which will display the lyrics along with the song title and artist at the top. This sequence will satisfy requirements 3.2.9 and 3.2.10

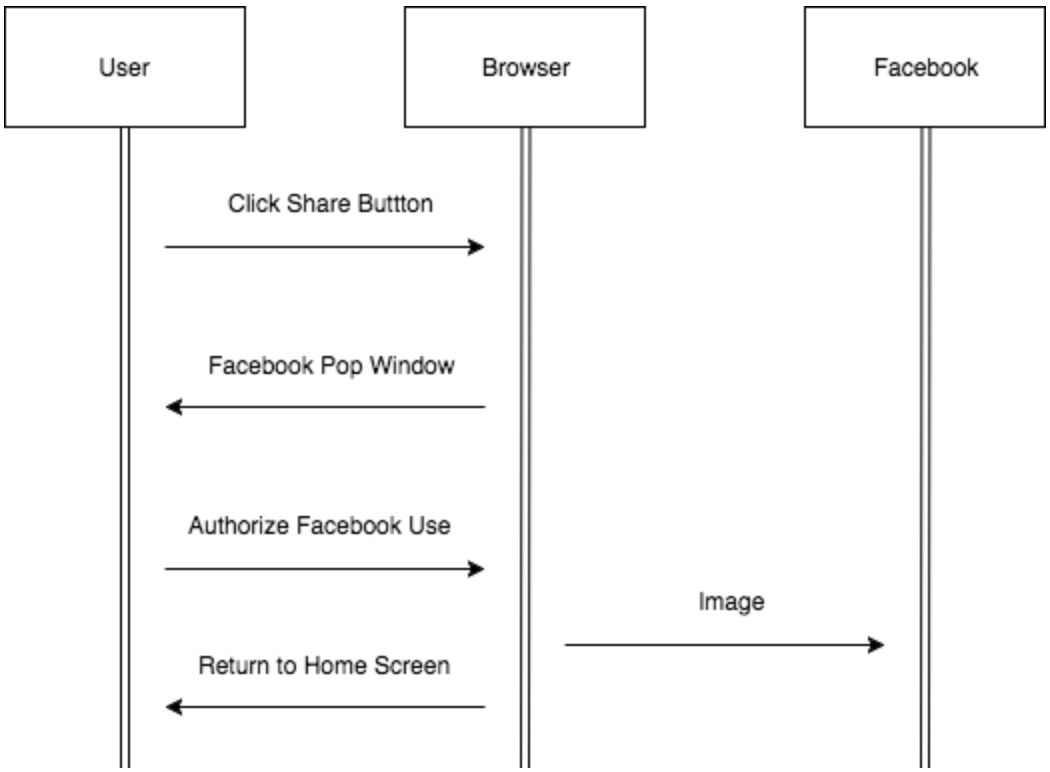
Auto Complete Sequence



The auto-complete sequence happens when the user is typing in the search bar. Every time a user changes the context of the search bar and the search bar contains more than three characters in it, the browser will send the current string in the search bar to the server. The server will then retrieve a list of possible artists whose names start with that string in alphabetical order. The server will then send the top five artists on the list, or less if the list is less than five artists long, along with their associated images to the browser. The browser will show the artist names and images to the user in a drop down menu below the search bar, satisfying requirement 3.2.1.

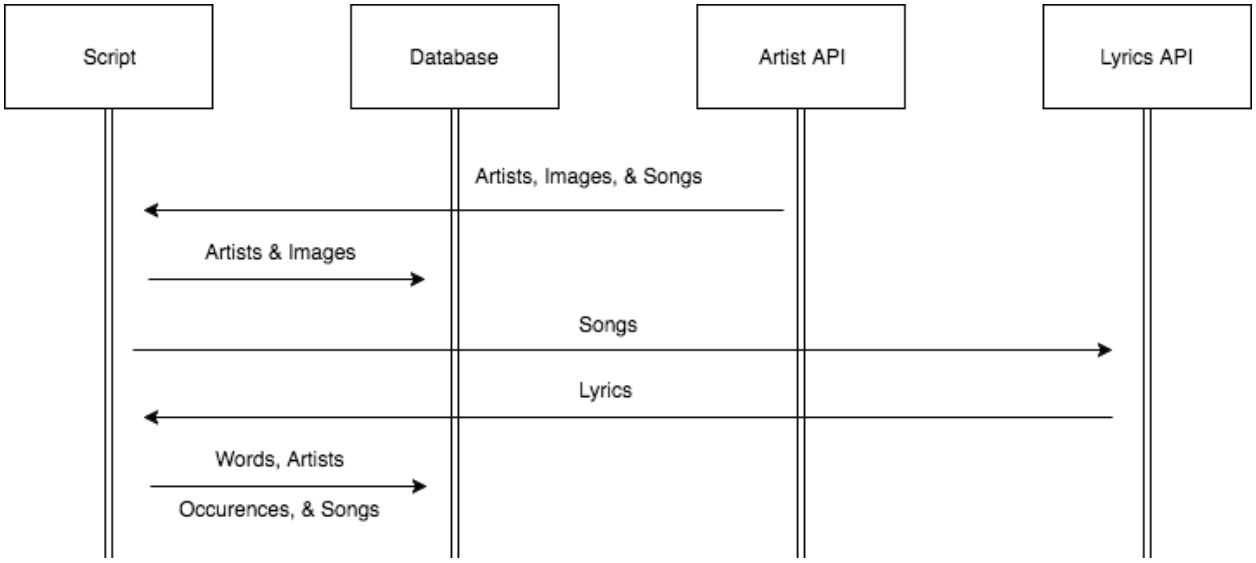
If no artist name starting with the string in the search bar exists, the server will return NULL to the browser and nothing will be displayed to the user.

Share to Facebook Sequence



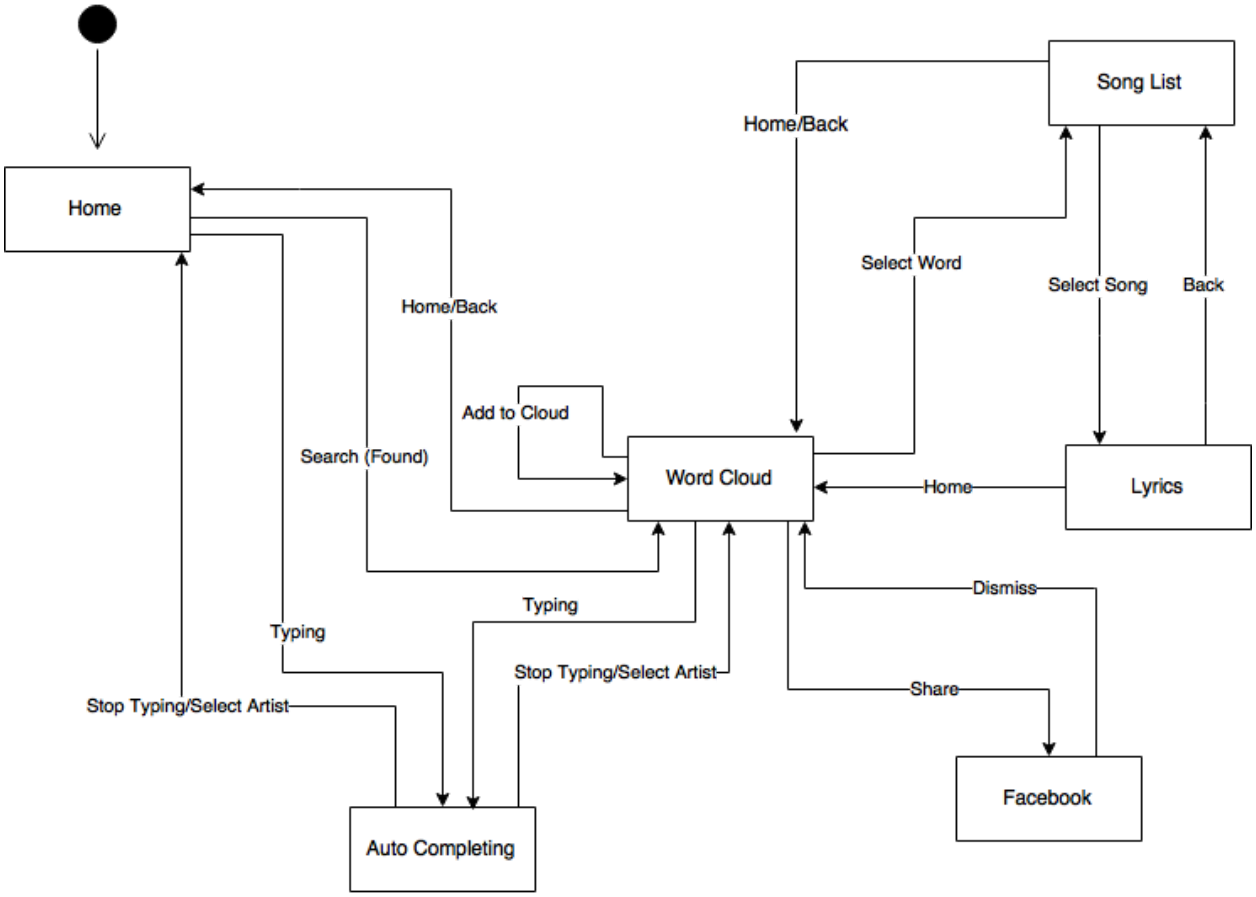
The share to Facebook sequence is initiated by the user clicking the share button on the home screen while a word cloud is present. The browser will then show a new window, which will allow the user to authorize our system to access the user's Facebook and the system will share the word cloud to the user's Facebook timeline, satisfying requirement 3.2.6.

Database Creation Sequence



The database creation sequence will happen at the beginning of the creation of the system, because the whole system relies on the data kept in our database. First we will write a script to scrape a third party API(Genius.com) for all its artists, associated artist images, and songs for each artist. The script will then enter each artist name and image into the database. After that, it will call another third party API(Lyrics.com) to get the lyrics to each song retrieved from the previous API. Using the fetched lyrics, the script will then enter into the database each word found for each artist, omitting common words as stated in requirement 3.2.4, how many times the word appears for each artist, and the songs it appears in for each artist. This will be necessary in satisfying requirements 3.2.1, 3.2.2, 3.2.4, 3.2.5,3.2.8, and 3.3.

3.7 UML: State machine



Explanation of State Machine -

The finite state machine view of the system has a definitive start as expressed by the large black circle in top left of the diagram, however the system has no definitive ending suggested by the lack of an open circle denoting the end to the system operation. For example, the “Word Cloud” state represents the screen where the user is viewing a word cloud and the arrows originating from that state represent the different paths that the user can take to reach another state. Each arrow represents an action that can be made by the user such as dismiss which means dismissing a new window generated by the system.

This diagram is an example of a simple machine used to abstract the system. The diagram abstracts how the front end should function and labels the key processes that should be done relative to user input. This abstraction will allow the team to visualize a high level view of the front end aspect of the system, and be able to tackle each state as its own subproblem. Each module was decided to be based on logical cohesion because each process within each module is related by functionality. Since this representation is skewed toward the front end, the coupling between modules is common coupling as each module will be working with shared data. In other

words the front end modules operate on the same data received by the server. In terms of information hiding, the state machine encapsulates various processes that take place during each state. For example the “Facebook” state encapsulates creating a pop up window, using the Facebook API to log the user in, and populating a Facebook post with an image of a word cloud.

The system begins at the “Home” state which represents what the user will see when they first access the system through a web browser. This was chosen to be a state to satisfy requirements 3.1.1.1 and 3.2.1 in the SRS. From this state the user can type into the search bar to transition to the “Auto Completing” state and search to either transition to an “Error” artist not found state or the “Word Cloud” state.

The “Auto Completing” state represents the state in which the system is attempting to auto complete what the user is typing into the artist search bar. This state is meant to satisfy requirements 3.2.1.3 REQ-4 through 3.2.1.3 REQ-7. This state can only be accessed by typing into the artist search bar and only returned from by not continuing to type.

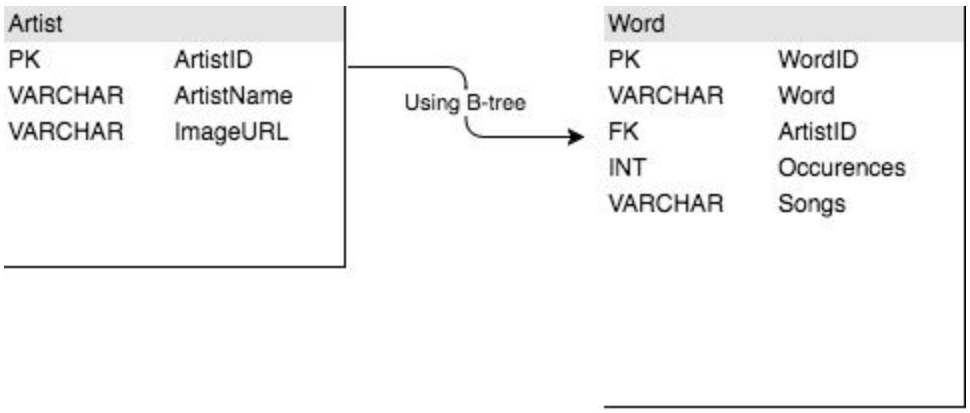
The “Word Cloud” state represents the state in which the system has generated a word cloud and the cloud is displayed on screen. This state is meant to satisfy requirements 3.2.3 and 3.2.4. This state can be accessed from all other states within the finite state machine diagram because it represents the main functionality and requirement for the system.

The “Facebook” state represents the state in which the user chooses to share an image of a generated word cloud to their Facebook timeline. This state is meant to satisfy requirement 3.2.6. This state opens a new window that the user can use to log in and post to Facebook which will be then be dismissed and take the user back to the “Word Cloud” state.

The “Song List” state represents the state in which the user selects a word from a word cloud and is taken to a list of songs by the specified artist containing that word ordered by number of occurrences. This state is meant to satisfy requirements 3.2.7 and 3.2.8.

The “Lyrics” state represents the state in which the user selects a song from the “Song List” state and is displayed the lyrics to the selected song. This state is meant to satisfy requirements 3.2.9 and 3.2.10.

3.8 Database Schema



Database Schema Explanation -
The database schema represents the internal structure of the SQL tables that will make up the bulk of information requests. The database is comprised of two tables an artist table and a word table.

The artist table is made up three fields: ArtistID primary key, ArtistName, and ImageURL. The primary key is an auto-incremented integer used for identifying the artist, artist name is a variable character that defines the artist's name, and the ImageURL is a variable character that specifies the artist name.

The word table is used to track information about words. The fields in the table are the WordID, Word, ArtistID, Occurrences, and Songs. The WordID is the primary key integer used for identifying songs. Note that there are multiple entries with the same word in this table. The word is a varchar specifying what the word is. The artist ID is an integer foreign key from the artist's column. The foreign key is indexed using b-trees for speedy categorization and look-up in O(log(n)). Occurrences is an unsigned integer that specifies the number of times the artist uses that word in their work. Songs is a varchar comma separated values(CSV) specifying a list of songs in which the word appears in among the artist’s work, and the frequency it shows up for each song.

The database is optimized around indexing words as well as finding artists.

The reason the team decided to implement a database was to satisfy requirement 3.3, the system shall respond to any user input within one second. This database will allow the system to run more reliably and theoretically more quickly.

4. Design Evaluation

4.1 Abstraction

Throughout the document there are a few different levels of abstraction that are used to create the diagrams. The software architecture, component, communication, sequence, top level data flow, and first level decomposition diagrams look at the system as a whole. They are very abstract in order to show high level processes and communication that are the foundation of the system. The class, use case, state, and second level decomposition diagrams use less abstraction and describe a lower level than the whole application. They separate the large main components of the system into smaller and more specific modules and subproblems.

4.2 Modularity

We used logical cohesion, a module whose elements perform similar activities and in which the activities to be executed are chosen from outside the module. We also used common coupling where modules are bound together by global data structures. Our architecture is modeled around model view controller(mvc). This pattern is to divide the program (or module) into three main modules: The *Model*, the *View* and the *Controller*. The *Model* handles the data of the program interacting with the database. The *View* is the module whose task is to display data to the user. It does not have any functionality in itself, other than what is needed to transform and lay out the data as needed by the display format. For our views in the MVC, we are respecting encapsulation by avoiding tight coupling between views. The *Controller* is the core module which makes all the decisions. It has all the relevant functionality of the program and interacts with the Model and the View, commanding them and passing data between them as needed.

4.3 Information Hiding

This design adequately uses information hiding to meet quality standards. Frontend and backend modules operate independently from each other. Only the minimum amount of information needed to meet requirements is shared between modules. Furthermore, the user only has access to the information needed to meet requirements, with most aspects of the design being hidden.

4.4 Simplicity

This design is more complex than other designs which could fulfil the same purpose, but the simplicity was sacrificed in return for performance gains needed to fully fulfil the stakeholders' requirements. Specifically, it did not seem feasible to meet the one second performance requirement (SRS 3.3) with other, simpler designs which were considered. (See Appendix) Even so, the intra-modularity simplicity of this design is very good. Pre-existing

frameworks will be used to minimize the number of lines and complexity of code needed to implement each component. The inter-modularity complexity also meets standards, with frameworks being used to keep frontend-backend communication very simple.

4.5 Hierarchy

This design has a clear overall hierarchy starting with the user and ending with the database. The user uses their browser to request information. The request passes from the front end browser, to the backend server, which passes the request to the database. Data then flows from the database to the user in the reverse of that ordering. Outside this structure, the design uses a chaos hierarchy design, without any excessively strict hierarchal requirements. The team judged that this project was not complex enough to require anything more strict.

4.6 Fog Index

The calculated fog index for this document is 11.38 which is below the target of 12.00 (lower score is better). The fog index calculates the complexity of the document and this metric was determined to be beneficial in measuring the quality of documentation.

5. Appendix

During the generation of this document, the team discussed two system designs that were created during the feasibility stage.

The first design proposed involved a server client architecture where the server handling the client requests would be constantly calling the third party APIs in order to obtain data requested by the client. This would be making several API calls for every request made by any client and organizing that data to send back, however this design proposed did not require a database or scraper of any kind. The idea would be for the server to receive a request from the client, then gather the necessary data, organize that data, and send the data to the client without storing any of that data.

The second design proposed is the design described and outlined by the design document. This design requires a scrapper and database to store relevant data that users may be able to request. The server would have direct access to the database and the scraper will be run in advance of the system operation and periodically to ensure updated data. This idea would be for the server to have already stored and easily accessible data to send the client when a request was sent to prevent the need for multiple API calls per client request.

As the team transitioned to the preliminary design phase of generating the design document, the team discussed the pros and cons of both designs proposed in the feasibility stage. The first design was agreed to be beneficial in terms of not needing a database for a relatively unknown amount of data and for an easier path to building an operational system. However, the first design was also agreed to be theoretically slower in terms of performance compared to the

second design, and generally dependant on third party APIs being available in order to be functional. The second design was agreed to be theoretically much faster, less reliant on third party APIs, and less taxing on the server hardware due to less data manipulation needing to be done in real time as opposed to the proposed upfront data manipulation done by the scraper and database. The second design was however deemed to be more complex in terms of setup and maintenance and risky in terms of the amount of storage needed for the system's database. After much deliberation the team decided that the second design was more feasible despite the increased complexity because it would be more certain to meet the performance requirements dictated by the stakeholder.

After deciding on a design the team created the diagram in section 2.1 Architecture Diagram to solidify the system architecture for the design. The team chose a graphical representation of the ADL to make the architecture easier to understand and easier to visualize while the team developed more detailed design diagrams. Another reason for the graphical ADL is to allow the team to easily compare and contrast the implemented system with the designed system. This will help determine any architectural drift and erosion which can be utilized during the testing phase to adjust test cases.

Once an architecture for the system was agreed upon, the team approached the detailed design diagrams from an adaptive design standpoint. The team worked on and created the diagrams and justifications for diagrams that were most easily understood. The data flow diagram and finite state machine diagrams were the first diagrams to be discussed and developed because the team already had existing interpretations of those diagrams derived from discussions during previous phases of the project. The team would discuss and determine the next diagram to complete based on how easily it was understood by the team as a whole. This approach allowed the team to leverage their understanding of the previously completed diagrams to develop the more complex or difficult to understand diagrams such as the communication diagram. After discussing the communication diagram with a CP, the team used the already developed sequence diagrams to aid in the creation of the communication diagram. The design strategy also aided in design continuity between diagrams because the inspiration and understanding of diagrams were derived from previously developed diagrams.

After developing diagrams, team members were tasked with justifying design decisions based on satisfying the requirements and adhering the design principles outlined in lecture 7. Designs were adjusted to meet the new requirements from team 17's design document and to better represent the design principles for software design.

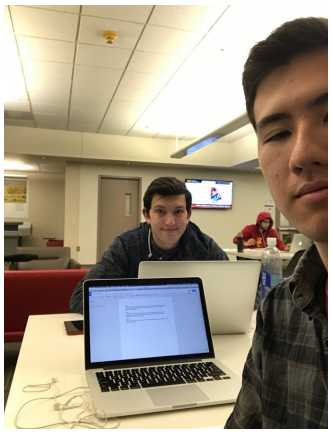
5.1 Team Meeting/Asana Notes

2/9/17
Team met at VKC 4pm-6:30pm to begin working on the design document where we worked out the data flow diagram, software architecture diagram and the finite state machine diagram. No picture taken.

2/10/17
The team met at VKC 3pm-7pm worked out the data flow diagram up to the first level decomposition, the database schema, the class diagram for the front end, the use case diagram, and lot of sequence diagrams.

2/11/17

Kyle and Andrew met at Leavy 10am-12am to write explanations of sequence diagrams, finite state machine, and use case diagram. The other three team members were assigned worked to complete on their own, Alec Schule Data Flow Diagram, Alec Fong Professionalism and Architectural style, Noah Bergman Component Diagram and Class Diagram for front end.



2/12/17

The team excluding Alec Schule met at VKC 7pm-8:20pm to continue work on the design document. The focus of this meeting was to continue writing the explanations for each diagram and to discuss the metrics that would be used to evaluate the design. The team then went to SAL 8:25pm-9:30pm for office hours to obtain clarification for certain aspects of the design document.

2/13/17

Alec Fong, Kyle, and Andrew met at VKC at 6:30pm-9:00pm to continue working on the design document. The focus of the meeting was to further develop the explanations and justification for design decisions. Noah and Alec Schule were assigned work to do outside of the meeting.

2/15/17

All team member excluding Noah Bergman met to finish the document and visit office hours. The team finished the design evaluation section and the appendix.

The team delayed implementing Asana for tasks due to the high frequency of meetings, however the team did utilize the task management system toward the completion of the design document.