Feature Request: Next 20 Results

**January 20th 2018**

# **OBJECTIVE**

To give users the ability to explore beyond the first 20 search results.

# **BACKGROUND**

Currently, Jammming supports the ability to search the Spotify library and display the results of the search for a user to interact with. However, the search results that are displayed are limited to the first twenty results that are a match to the user’s query. Due to the fact that many songs share the same name and the large number of songs available for many artists, this limitation cripples the application’s ability to be used as the music discovery vehicle it is intended to be. Implementing this feature is critical to take this application from a novelty interface to a useful tool for music discovery .

# **TECHNICAL DESIGN**

Fortunately Spotify provides a us method for iterating through search results by adding an offset field to our query.

# **Spotify Module’s Next and Prev functions**

Four new functions will need to be added to our Spotify module along with two new global variables. Function **next()** and function **prev()** will first add / subtract 20 ( the current limit set for search results ) to / from a new global variable **offset**.

Our **search()** method will need to be modified to check the value of its **term** argument against a new global **searchTerm** variable to determine if it’s a new search and reset the **offset** if so. Then save the value of it’s **term** argument to our **searchTerm** variable before calling the API with both **searchTerm** replacing **term** in the query, and adding an offset field to the query, the value of which being our **offset** variable. The **searchTerm** variable is necessary to recall the value of our initial search term when our **next ()** / **prev()** functions are called. The two other functions **getOffset()** and **getSearchTerm()** return the values of our newly defined global variables for use in the other parts of our application.

# **App Component’s Next and Prev functions**

Two new functions will need to be added to our App.js file. **Next()** will call the **Spotify.next()** while **prev()** will call the **Spotify.prev()** function. Both methods then call the **App search()** method passing **Spotify.getSearchTerm()** as its argument.Both functions must be bound to the **App** instance in the constructor.

# **Passing Props**

The **App** component must pass it’s **next()** and **prev()**  as props to the **SearchResults** component, which is where we will render our new buttons that will allow our users to interact with the new feature, in addition the returned value of our **Spotify.getOffset()** function which will allow us to conditionally render the ‘prev’ button. The **SearchResults** component will in turn pass these same props a new **Buttons component**.

# **SearchResullts Component’s RenderButtons method**

It wouldn’t make sense to render navigation buttons if there aren’t search results to navigate so a new **renderButtons()** method will be created in the **SearchResults** component which will conditionally render our **Buttons** component if either the **searchResults** prop length is greater than zero ( indicating results are being displayed ) or if the **offset** prop is greater than zero ( to navigate back from beyond the limit of the Spotify library ).

# **Buttons Component**

A new component **Buttons** will be created. It requires two methods **handleClick()** and **renderPrev()** and will render two <button>s. The **handleClick()** method’s ‘direction’ parameter will accept arguments from the ‘next’ and ‘prev’ buttons, the value of which will determine whether it calls the **next()** or **prev()** method in the **App** component. The **renderPrev()** method will be called from within the **render()** method and will conditionally render the ‘prev’ button if the **offset** prop is greater than 19 ( indicating that the results have been offset beyond the first 20). This prevents sending a negative value as offset to the Spotify API ( which will cause an error ) and creates a logical display senario. The component will render the button(s) within a div with a class name of ‘buttons’ that can be targeted as such for styling.

# **Buttons**

Both buttons will be rendered with class names equal to their respective function to aid in targeting them for styling. Both buttons will also have an ‘onClick’ event attached. A click will trigger the **handleClick()** method bound to the **Button** instance and a value representing the intended purpose of the button (‘prev’ or ‘next’) which will allow the **handleClick()** method to determine which method to call. The button’s text should represent its function

**Styling**

A **button.css** file should be created and imported into the **Button** component for the purpose of styling the buttons. The buttons should be within a flex container and spaced evenly. The buttons should be semi-transparent and styled to resemble track forward / back buttons to indicate their function to non-english speaking users.

# **CAVEATS**

This feature provides no means for indexing results. A user might want to skip to page 5 of the results or back to the beginning of the results pages. In order to index results the application would need to make a query to the Spotify API with a limit parameter set to none and then store all of the results within the results array which could then be iterated. Due to the potential of a very large number of results to be returned, both of these steps are likely to introduce a considerable performance drain. In the interest of keeping this application lightweight and streamlined we have decided not to pursue this functionality at this time. If sufficient user feedback is received to warrant a change it we may elect to add a ‘Page One’ button in future iterations as this can be accomplished with minimal overhead. Also with the current design it is possible for a user to continue to click the ‘next’ button beyond the end of the Spotify library results. This will result continued calls to the API with an increasing offset value and while it does not result in an error, the user will have as many clicks back to return to the results. In the interest of separation of concerns we have elected not to implement any checks into the **Buttons** component since this behavior does not introduce any faults and the state is easily reset by repeating the search. This issue may also be addressed in future iterations.