Playlist App Report

Eugene Kolo I Braxton Brewton
eugene@kolobyte.com I braxtonbrewton@gmail.com
November 2015

Web

Frontend

We utilized HTML, CSS, and Javascript to handle client side operations. Our HTML and CSS present a responsive layout for the user with the help of Twitter's Bootstrap. Our Javascript code primarily used jQuery libraries to perform AJAX requests to the backend. These requests enabled our web app to respond real-time to user input.

Backend

We utilized Spark Java to create a RESTful API. The API can be used by any frontend, and features can be easily added to it. Our API always returns JSON and is documented at the bottom of this report.

We have 5 routes:

```
GET /api/getTop8
POST /api/addPlaylist
POST /api/addPlaylists
POST /api/getAutocomplete
POST /api/suggestPlaylist
```

Requests and Responses

The web app performs requests to 5 different backend routes.

Each time the **suggest playlists page** is loaded, a `GET /api/getTop8` request is made to the backend. The backend processes the request, and returns a map containing the top 8 most popular playlists. The map is then displayed on the browser in a table sorted by popularity.

When a typeable key is entered into the **song search bar**, an event handler is triggered that sends the thus far

typed text to the frontend `autocomplete()` function. The `autocomplete()` function performs a `POST /api/getAutocomplete` with the song search bar text. The backend then returns the songs in the database that prefix with it.

Upon clicking or pressing enter on any of the four rows of the autocomplete table found in the Suggest Playlists Page, the corresponding row's song and artist are passed to the `suggestPlaylist()` function. `suggestPlaylist()` submits a `POST /api/suggestPlaylist` with the given artist and song. A map representing a playlist that contains the selected song is returned.

A user may create a playlist on the **add playlists page**. A user can select songs by entering them through the song search bar and selecting them from the autocomplete table. Once the user has finalized their playlist, they must input a popularity value associated with it. The user must then click the **upload list button** to upload all songs from the **Songs to add table** to the `PlaylistDB` database as one playlist. Clicking the button triggers an event handler that performs a `POST /api/addPlaylist` with the given songs, artists, and popularity an associative array. The backend then stores the playlist into the database.

Alternatively, a user may upload up to 128 playlists at a time to the database by uploading a text file containing playlists. Clicking on the **Upload file button** the user triggers a `POST /api/addPlaylists` request containing the file. The file is then parsed by the backend and stores each playlist into the database.

Algorithms and data structures

Summary

1. Add new playlist(s) to database
2. Autocomplete to top 4 songs
    - List songs by popularity based on playlist's popularities that have the song.
3. Suggest most popular playlist with a specific song.
4. List top 8 most popular playlists

Bulk of the work is done when you add a playlist.
Special thought is put into keeping (2) highly responsive, as users are more understanding with adding a 100 playlists taking a longer time. Non-responsive autocomplete is awful. Nonetheless both actions run on the order of milliseconds.

Memory is considered lastly, as applications these days take up 200MB+ and nobody bats an eye. A playlist app can easily use 10MB and be fine. Nonetheless, the memory usage is minimal due to the usage of references and only allocating at the app launch.

Definitions and data structures used

Playlist database

`PlaylistDB` is a Priority Queue, implemented as a binary min heap. The starting database will be built in `O(L)` time (`L` inserts at `O(1)` time each). The key in the PriorityQueue is playlist popularity. The `PlaylistDB` is limited to 1024 playlists, and removes the least popular playlist on addition of any new playlists.

`L` = number of playlists at start
`N` = number of playlists currently in the DB

```
Functions:
Insert - O(logN)
Find-Min - O(1)
Delete-Min - O(logN)
Create - O(L)
```

Top 8 cache

The top 8 cache, `Top8` is implemented as an 8 element linked list. Updated when `PlaylistDB` is updated. It stores the top 8 most popular playlists based on popularity. Due to the small size, all operations are constant. When a playlist is added, a playlist may also be removed due to being replaced. The `Top8` cache is updated based on the result of that replacement.

```
Functions:
Add - O(1)
Remove - O(1)
Peek - O(1)
```

Autocomplete database

The **autocomplete database**, `AutocompleteDB` is a PatriciaTrie (a space-optimized Trie). A trie is a tree where each node is a letter. At app launch, all songs are added to it, taking `O(KM)` time. Searching the trie takes `O(M)`. The memory taken by the trie is on the order of `O(KM)`.

`K` = number of total songs
`M` = length of longest song

```
Functions:
Find-Prefix-Map - O(M), performs a tree traversal, where each node is a letter.
Create - O(KM)

Algorithms:
Get-Top-4 - O(M), essentially a K-selection algorithm
```

Song database
The **song database**, is a series of Maps that point to songs in the `PlaylistDB`. The song database is indexed by a song's title and author, and points to a **song object** stored in a `Playlist` in the `PlaylistDB`.

```
Functions:
Add-Song - O(1)
Find-Song - O(1)
Set-Pop - O(1)
Set-Best - O(1)
Get-Best - O(1)
```

Initialization

During app start up a `SongDB` is created that stores all songs, with their titles, and authors. Additionally the song titles and authors are pushed into an `AutocompleteDB`. `PlaylistDB` is initially empty.

Our web backend uses Spark Java, communicates via HTTP and JSON. REST practices are attempted.

Adding playlist

The playlist database, can have an `Insert` of a new playlist, `P`, done in `O(logN)`. Due to lack of space, (1024 playlists already exist), a `Delete-Min` might occur. During the process of adding a playlist to database, the `SongDB` and `Top8` are updated as well.

Update song data

The playlist, `P`, will contain `S` songs and 1 popularity value, `V`.

Each song in the `SongDB` will `Set-Pop` `O(S)` times in `O(1)`, and potentially perform a `Set-Best` in `O(1)`, if `P`'s popularity is greater than the song's previous best playlist's popularity. This allows for efficient lookup when suggesting the best playlist that contains a specific song.

The amount to `Set-Pop` to is determined based on if a playlist was added and removed, due to being

replaced.

This results in each song's popularity increased in `O(S)` time.

```
for s in S: // O(S)
    SongDB[s].Set-Pop(V) // O(1)
    if (SongDB[s].Get-Best().Pop < P.Pop):
        SongDB[s].Set-Best(P) // O(1)
```

Updating top 8 playlists

The top 8 playlists are stored in a simple list cache. The least popular playlist in the top 8 playlists is located on the end. When a new playlist is added to the database, its popularity is compared to the least popular playlist in the top 8. If the new playlist's popularity is more, it replaces the least popular one. Due to the small size, and efficient removal, all operations are O(1).

```
if P.Get-Pop() > Top8[0].Get-Pop():
    Top8.Replace(0, P)
    Sort(Top8)
```

Autocompleting song

Songs are autocompleted using the `AutocompleteDB`, a Patricia Trie. Autocompleting is accomplished by sending a `POST /api/getAutocomplete` with an entered prefix. The Patricia Trie is then searched for that prefix. The top 4 songs that have the correct prefix are returned.

Because the number of songs returned is so small (song list is only 4000 elements), on average <4 songs, a simple sort is done to find the top 4 songs based on their popularity stored with them in their Java bean.

Contributions

Eugene worked on *AI Gore Rhythm's* Java data structures, algorithms, and the web backend. This is his first Java project from beginning to end. He additionally helped out with the frontend when necessary.

Braxton worked on *AI Gore Rhythm's* web frontend. He worked on the event handlers, requests to the backend, and the responsive layout.

Appendix

API Documentation

```
POST /api/addPlaylists
*   Gets fileData and parses out the individual playlists and adds them to the databa
se
*
*   @note: Updating the playlistDB also updates Song's popularities
*
*   @req: JSON of <fileName> assosicated with <fileData>
*       {<fileName>: <fileData>}
*   @res: 200 if successful
```

```
POST /api/addPlaylist
*   Gets a list of song titles and popularity, and adds them as a playlist to the dat
abase
*
*   @note: Updating the playlistDB also updates Song's popularities
*   @req: JSON of {"songList":[{
*                                  "title":"Obsession Confession",
*                                  "author":"Slash"
*                                },
*                                {
*                                  "title":"Obsesion",a
*                                  "author":"Roberto Pulido"
*                                }
*                              ],
*                  "popularity":"80"
*                  }
*   @res: 200 if successful
```

```
GET /api/getTop8
*   Returns Top 8 playlists based on popularity.
*
*   @req: blank
*   @res: JSON map of top 8 playlists song list sepated by ##
*       {"0":{"songList":[{
*                           "title":"Apple",
*                           "author":"Joe",
*                           "popularity":"80"
*                          },
*                          {
*                           "title":"Orange",
*                           "author":"Snoop",
*                           "popularity":"25"
*                          }
*                        ],
*               "popularity": "78"
*             },
*       {"1":{"songList":[{
*                           "title":"Cat",
*                           "author":"Janice",
*                           "popularity":"85"
*                          },
*                          {
*                           "title":"Dog",
*                           "author": "Jim",
*                           "popularity":"35"
*                          }
*                        ],
*               "popularity": "64"
*             },
*        ...
*       }
*
```

```
POST /api/getAutocomplete
*   Gets a string and searches the database for the most popular matches
*
*   @note: Case insensitive.
*
*   @req: JSON of "song" matched to partial completion
*       {"song": "obses"}
*   @res: JSON with top 4 most popular autocompleted songs
*       {"0": {
*               "title": "Obsesion",
*               "author": "Roberto Pulido"
*              },
*         "1":
*              {
*               "title":"Obsession Confession",
*               "author": "Slash"
*              }
*          ....
*        }
*
```

```
POST /api/suggestPlaylist
*   Gets a song title and suggests the most popular playlist that has it
*
*   @req: JSON of "song" matches to <songTitle>
*       { "song": {
*                   "title":"Obsesionado",
*                   "author":"German Montero"
*                 }
*       }
*
*   @res: JSON with most popular playlist that has the song
*       {"mostPopular": {"songList":[{
*                           "title":"Cat",
*                           "author":"Janice",
*                           "popularity":"85"
*                          },
*                          {
*                           "title":"Dog",
*                           "author": "Jim",
*                           "popularity":"35"
*                          }
*                          ...
*                       ],
*                      "popularity": "64"
*        }
```