SFTB Project Report

A Visual Music Exploration Web Frontend for Music Streaming Services

***Robert Thurnher**, December 2011*

Summary

This project is in its essence taking the challenge of implementing a prototype app visualizing tunesBag data in a sonarflow way making use of Last.fm API solely via web technologies.
So, enabling to visually access a cloud-based music streaming service in innovative ways.
It is aimed to be as much as possible standards-based/compliant (as in "*things which got commonly coined under the buzzword term **HTML5***") with the web tech being used here.
That is, usage of any proprietary technologies such as Adobe Flash should be minimized.
As a matter of fact, Flash is only used as a fallback strategy within the audio playback component preferring HTML5 <audio> support whenever available and applicable.

Overview

More or less everything the app does is actually happening on the client side (browser).
The app is made with Node.js. Its code is entirely written in CoffeeScript.
Building (-> compiling, minifying, etc. deployable code) and dev running is done with Cake.
There's a small server-side component based on Express basically just serving the HTML/CSS/**JS** which makes up the actual app. Production code is hosted on Heroku.
Furthermore, Stylus is used as a CSS preprocessor and Eco for client-side templating.
Additionally, the app's code is structured as CommonJS modules being "stitched" together.
The app framework of choice here's Backbone.js keeping the app's structure itself sane and well designable. This generally provides some kind of MVC-styled container ready to use.
Similarity computation/visualization of artists is done via Last.fm API's artist.getSimilar.
The main component of the app's UI is built with SVG making use of the excellent Raphaël.js.

The audio playback component is based on the very useful SoundManager 2 library.
As already mentioned, this especially helps with making it more convenient to play sound in an as cross-browser-compatible way as possible.
Annotated code docs can be found at: sftb.herokuapp.com/docs/main.html
A basic technical howto respectively user manual can be found in the app's README.

Challenges

Main challenges of this project:

- Make it work as cross-browser-compatible as possible while still staying cutting edge.
- HTML5 <audio> support is quite fragmented and unstable across browser engines.
- Try to make it work performance-wise as efficient as possible.
- Should work on mobile devices with WebKit browsers as well.
- Lots of JS -> keep it sane, clean and manageable.

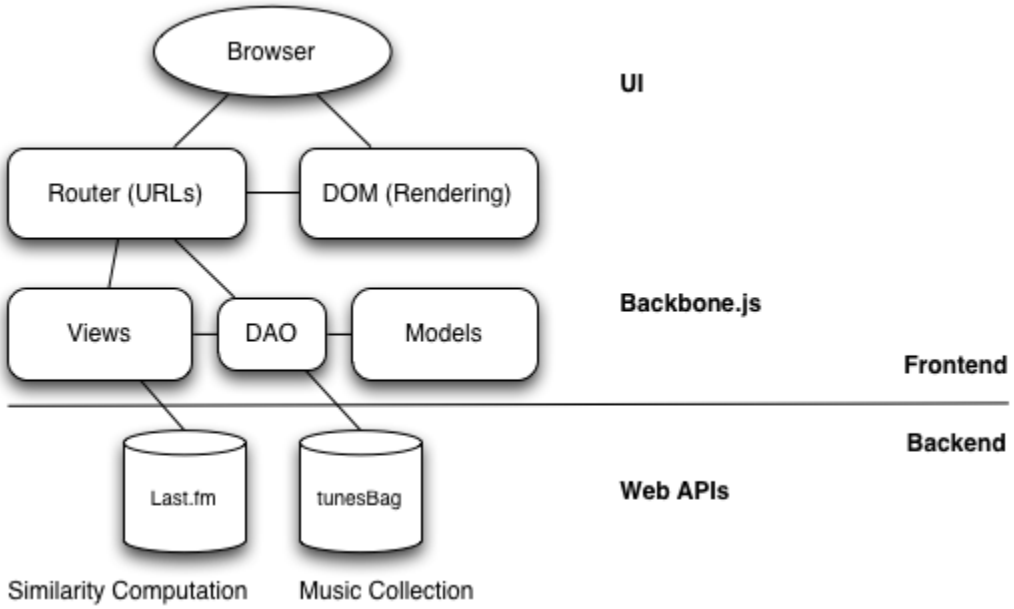
Solution

The overall decision was made to mainly target modern WebKit browsers (Chrome, Safari).
Of course, best performance can be seen in Chrome with its top-grade JS engine V8.
Furthermore, modern instances of the Gecko engine (Firefox) were made to work as well.
The biggest challenge with fully supporting the latter was mainly its lack of MP3 support within HTML5 <audio> (it's supporting Ogg Vorbis instead due to licencing issues).
Solution here is to use SoundManager 2 library which has fallbacks to audio playback via Flash built-in with a convenient JS API on top (see above).
Another interesting detail which turned out is that HTML5 <audio> support in Mobile Safari is pretty picky about correct content type headers set in HTTP responses being used.
This temporarily was an issue with the tunesBag streaming MP3 responses solved through a bit of according API parameters tweaking.
Unfortunately, only more recent versions (3.0+) of Android's default browser support SVG.
Generally, the app runs pretty nicely on the iPad (resolution optimized to fit) on iOS 5.
Additionally, related HTML meta tags specifically targetting mobile devices were used.
Plus, Google's Mobile Bookmark Bubble library is applied to improve the UX on iOS devices.
To visually communicate when the app's working/loading things an activity indicator is shown.
To keep the performance of the app on an adequate level several measures were taken.
For example, it's attempted to make as few web API requests as possible and only at times

when it's necessary. Consequently, as much data as possible is cached for some respectively reasonable amounts of time on the client. HTML5 Local Storage is used for that shimmed by a nice little library called Kizzy which includes fallbacks for incapable browsers (read IE).
The main means to keep the code and general app structure in order is the aforementioned combination of CoffeeScript, Backbone.js with jQuery and CommonJS modules.

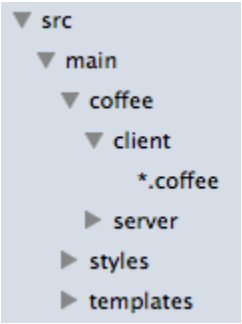
Architecture

Here's a diagram of the overall software design of the app:



So basically the app connects to tunesBag and Last.fm data (for music collection and similarity computation) via JSONP with their (more or less) RESTful web service APIs.
The data is processed and managed by a DAO (Data Access Object pattern) which puts the data into convenient model abstraction objects.
Moreover, the DAO then hands these models to views which are controlled by the main router object (~ controller) generally handling URLs onhashchange (HTML5 history.pushState being not reliable enough yet cross-browser-wise). App resources are cached via HTML5 AppCache.
The whole UI is within the browser's DOM (Document Object Model) on a single HTML page.
In a nutshell, the design follows MVC (Model View Controller pattern) all laid out on the client.
For minifying and optimizing the JS code deployment-ready Google Closure Compiler is used.

Here's an overview of the app's source code file layout:



Implementation

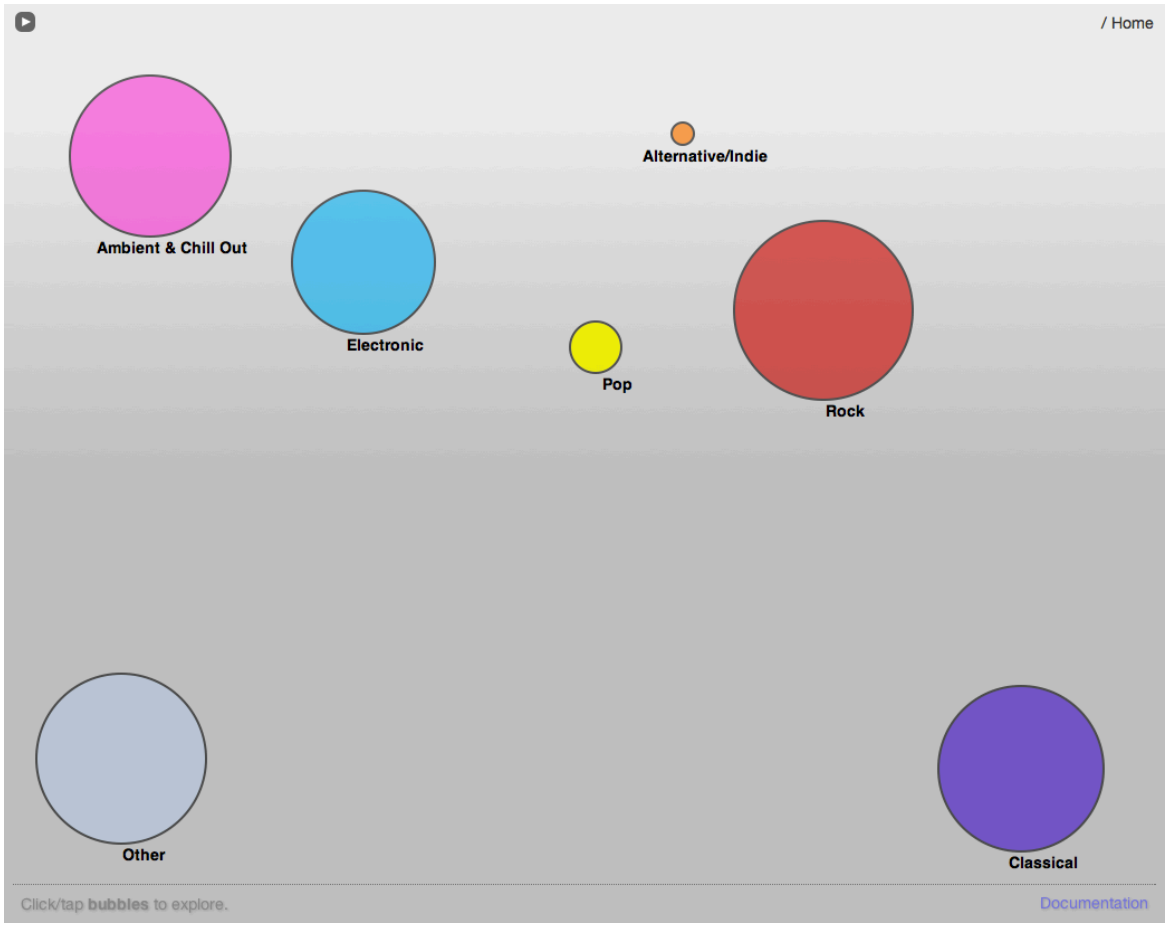
There's some general configuration given defining the colors and center coordinates of genre bubbles as well as sub genres subsumed within. This data is used for matching with the data fetched from tunesBag defining the music collection coming from a user account specifically registered for this prototype. Genres data is cached on the client for max. 5 mins.

The algorithm used for displaying the artists according to their respective relative similarities is Spring Graph. In detail, an implementation of it in the context of Google Caja project is used which has been slightly adapted to work with SVG as required by this app.

Spring graph weights data is cached for max. 30 mins after respective initial computation. Especially with many artists within one genre this computation can take a relatively long time as every artist has to be compared to each other (each comparison taking one Last.fm web API request). So if there are, say, 20 artists inside a genre 20 API calls have to be processed.

Functionality

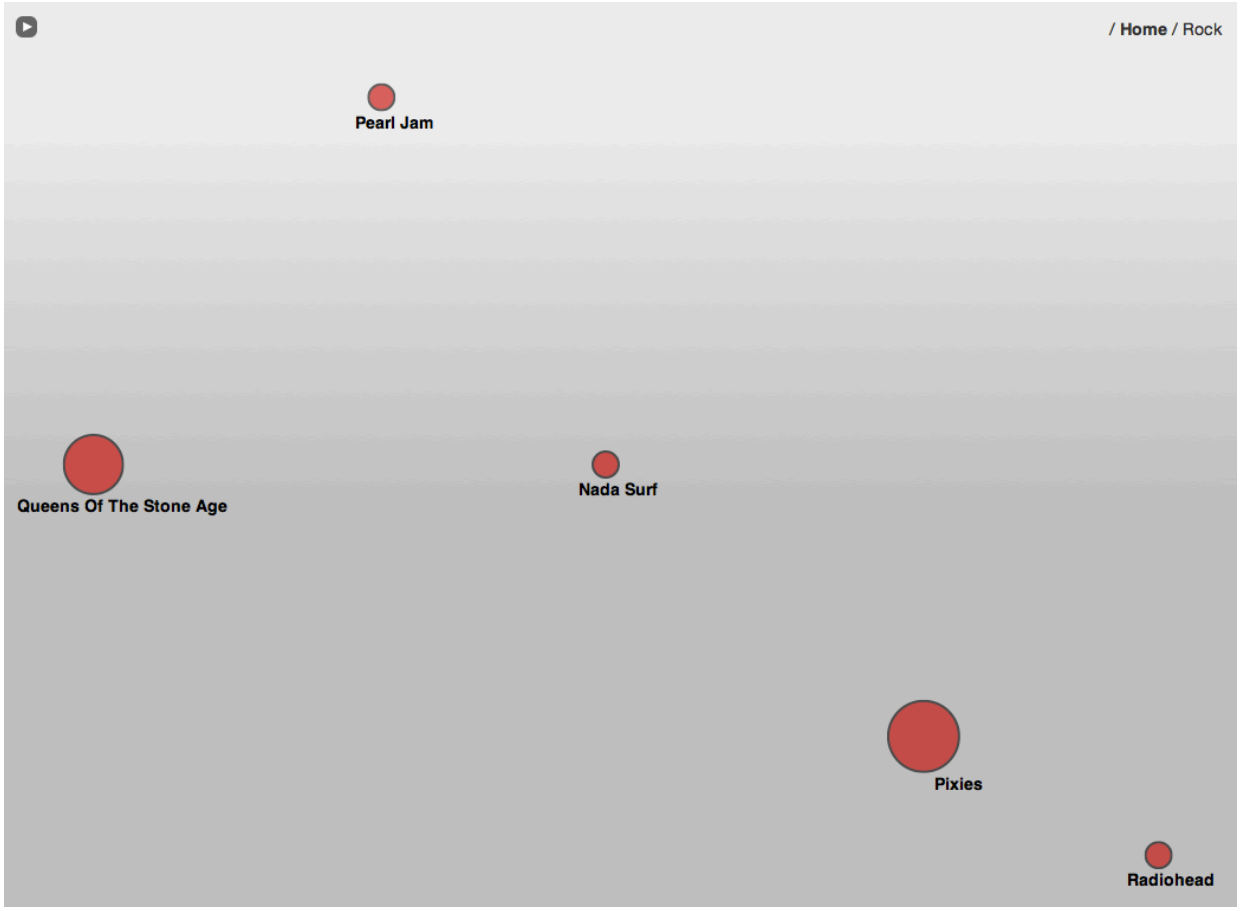
Here's a screenshot of the central home view:



So when entering the app on its home page screen the user is presented some bubbles representing genres of the music collection.

Each bubble's size is relative to the amount of tracks inside, colors and arrangement are controlled by configuration as mentioned above.

The user can click/tap on a genre bubble which will zoom into the genre view at hand, e.g.:



Immediately after zooming in an activity indicator is shown to inform about the ongoing similarity computation in the background.

Initially, the artist bubbles are arranged in an ellipsis around a center bubble.

When the similarity computation eventually finishes the activity indicator disappears and the spring graph animation runs for a certain amount of time (there are 50 steps of 100 ms duration each -> 5 s overall timespan) ending in a (hopefully) interesting bubbles arrangement (size indicates amount of tracks inside, white bubbles show that no similarity was found at all).

Now, the user can zoom in on an artist by clicking/tapping its bubble leading to an artist view.

Here the user is presented the tracks of the artist currently randomly arranged.

Clicking/tapping on a track bubble loads/plays/pauses tracks (loaded ones are “glowing”).

Additionally, this can be controlled via a simple audio player control at the top of the screen.

Navigation between views can be done via a breadcrumbs navigation control in the top right.

Here’s how playing a track in an artist view can look like:

Conclusion

All in all, the path taken has proven to be a viable one.
One of the areas which offer additional work is increasing the potentially large amount of time computing the artist similarities can take (maybe HTML5 Web Workers could be useful here).
Plus, the visualization/animation itself could be tweaked and iterated upon. E.g., scaling according to spring graph boundaries and damping (cooling) towards animation ending.
Furthermore, the simple audio player control of this prototype could be improved.
Additionally, track bubble views arrangement currently being more or less completely random should be more sophisticated in a real product, of course.
Another area which could use some further refinements is the build process and testing.
Finally, config data/parameters currently being readable from source needs further thought.
Nevertheless, it's shown to be possible creating an app as defined by the requirements laid out here solely relying on state-of-the-art web technologies.
So further possibilities to extend on this work is to either focus on improving a desktop version, making a version specifically tailored to mobile devices or for example a version targeting third-party integration within Spotify's desktop clients via their app development platform.