Project Noize Further Development Guide

Design Specifications, Diagrams, and Lessons Learned
for the 2015-2016 Senior Projects Team

Introduction

Hey, fresh and hopeful computer science seniors! Welcome to the development guide for Project Noize from the 2014-2015 dev team. Here we will detail all of the decisions we made throughout our year, outline the work we completed, and give you some introductory steps to further development to hopefully make your lives a bit easier.

A bit of backstory: You guys are the third CSCI senior projects team to work with Radio 1190. We had to scrap what was given to us at the beginning of our year due to how amazingly incomplete and poorly documented that was. Fortunately, all parties learned from the frustrating experience, which is why this guide exists. Please consult this if you have any questions regarding the Noize Machine (what we have named the web application) or Project Noize as a whole (our general group/project name).

Remember: your job isn't to change the Noize Machine, so while we have documentation for how that works here, the prime focus is describing our workflow system. You need to know what we've built so you can build off of it/integrate it yourselves! But really don't change it.

Overview

Last year, we made the Noize Machine. As you know, Noize Machine is a stand alone web application. Python 2.7 handles the backend work: the watchdog that is constantly checking for updates to the music folder that has been mounted to the server and the commands to take that music and update SQLite databases are in that. There are two databases, one for music, and one for everything else (ads, legal IDs, sweepers). Python determines which database to associate each mp3 with depending on the name of the folder its in. We have eyed3 as a dependency in order to read the metadata of an mp3 file.

Additionally, Python handles all of the smart playlist generation. You probably won't have to touch any of the Python unless some song proximity protection algorithms need to be updated or a bug arises. To look at further commands for the backend, see the **Final Backend Admin Guide.pdf.**

AngularJS handles the middleground work. Playing the music is handled by SoundManager in the app/js/controllers.js folder. Index.html renders the actual web page, which relies heavily on the controllers.js file. You will have to edit these files if you want the human DJ view to mimic the Noize Machine view. The basic interface (as described to DJs) is in the **Final Use Guide.pdf.**

For a more detailed explanation of the backend and of the frontend controller controller.js, please see **backend.md** and **frontend.md**.

Dependencies

- eyeD3
    - http://eyed3.nicfit.net/installation.html
- SoundManager
    - Keep app/js/libraries/soundmanager2-jsmin.js where it is so the web page can see it.

Installation/Working on the Server

Production

Noize Machine is currently hosted on Radio 1190s server (SERVER ADDRESS). The current dependencies are already installed. If the system has to be moved to a new set of servers, only eyeD3 will have to be reinstalled, and it's better to follow the most up-to-date documentation on their website to do so.

Local

Work locally. Pull from https://github.com/edma8378/AutoDJ and check testingCrunkStep.md for good testing and pulling practices.

Each team member used a local HTTP server to work on features of the project, before merging with the dev server.

For linux machines we used apache2 to run the application locally. For me, it was as simple as the installation, and copying the app directory of the project to /var/www/html and pulling up localhost.

For setup of the backend for local use, see **testing.md.**

Dev Server

For setup of a development server**,** see **devServerSetup.md**.

Note: There is a Daemon on the Radio 1190 server that generates the playlists every night at 10pm and makes sure the watchDog is running. This process must always be running. If the radio station changes their server this daemon must be included. See backend.md for details.

Testing

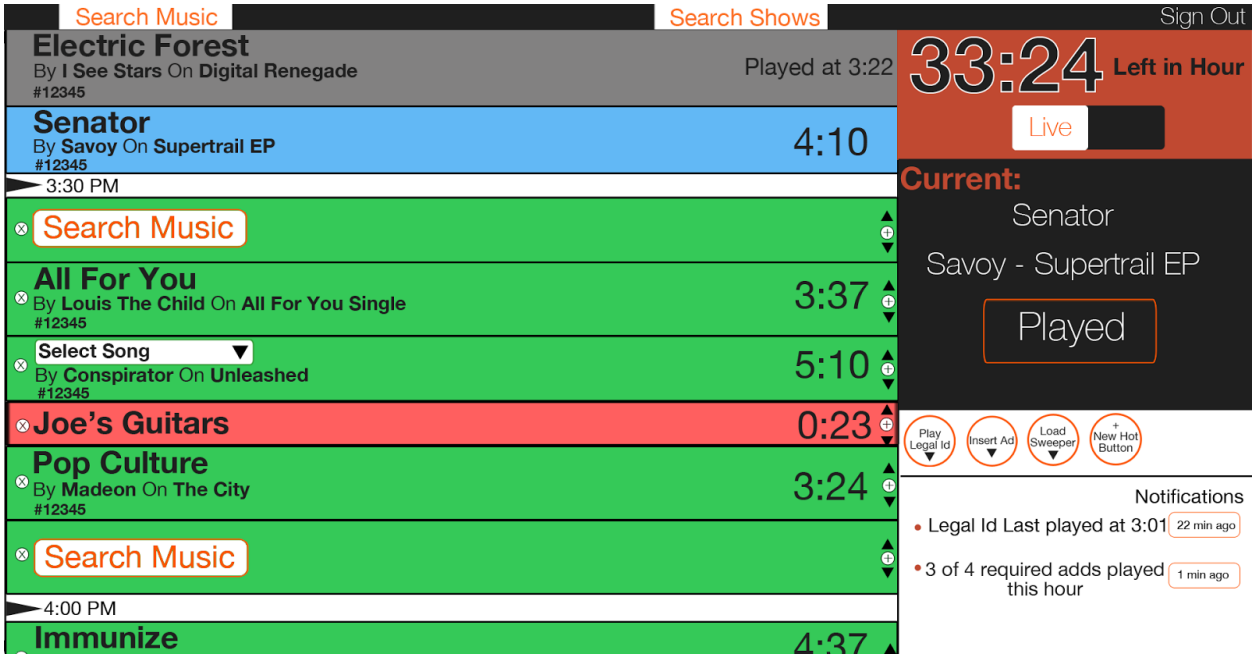
To check that everything is working properly with Noize Machine, perform the steps in **frontEndChecks.md**.

If you are working locally, perform the setup steps in **testing.md**, and then perform the front end checks.

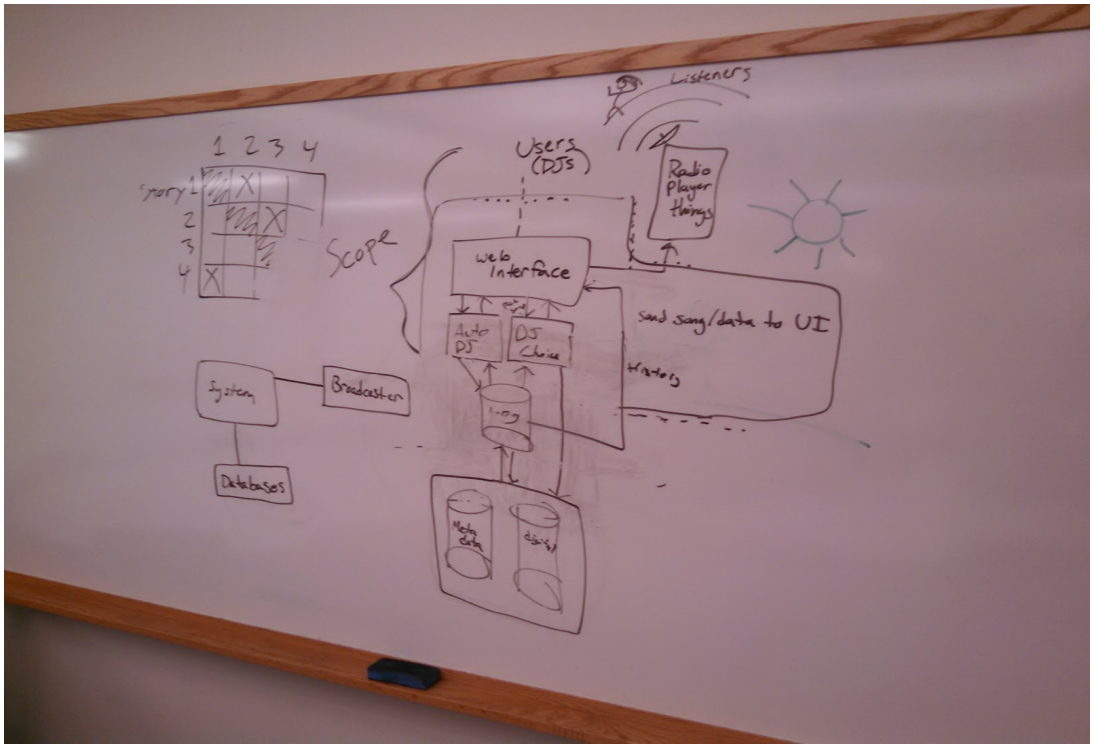
The class will require testing and documentation of those tests. We initially tried to use protractor tests, but they seemed more trouble than they are worth. If you want to try to implement a front end test suite or want to test the back end functionality see testing.md.

Mockups & Diagrams

Intended Mockup for human DJ to-do playlist



DMM Mockup



Lessons Learned

AKA: How to Make Your Lives Easier

- Pick an end goal and stick to it. Our semester got a whole lot better when we focused on the one thing instead of trying to attack a bunch. Then we were able to add on from there. For example, focus on building a web app that manages their music and is searchable then build from there.
- Front End test suites are for suckers (Front end tests can be useless). Manual tests and documenting them will make your lives easier.
- Do not try to add to DJ Pro, start from scratch.
- You don't necessarily need to use AngularJS as they can be two apps tied together, or you can plug Noize-Machine into your app somewhere. That decision is up to you.
- Django isn't a bad thing to do it in if you want to. It needs to be updated on their server, but that is about it.
- First task should be trying to copy over DJ Pro locally. It took me about a week. I started a Django project from scratch and went from there. Set up a psycopg2 database like theres is. It should show you what's going on in the settings.py file within djpro.
- The old DJPro resides in /var/www/django on the radio1190.colorado.edu server. SSH is your friend.
- The username, name of database and password are all based on how the psycopg2 database was set up. It's a little tricky on Windows, but it can be done.
- Set up a development server. We used a raspberry pi, and added a ton of music to it in order to test our app with a large amount of songs. Also, a dev server makes it easier for you to not screw stuff up.
- Don't change anything in our code on their server. You should know that but just to double check. If a change needs to be made and it should be updated, pull it from our Git onto your local machine, change it, push to Git. Test extensively. The radio station should always have our app working.
- Our github was laid out as such: We all developed on local feature branches (i.e. Adding error checking functionality was on a checkFunctionality branch). Then we pushed that to github and did a pull request into a "dev" branch. The dev branch was pulled down on to our raspberry pi in order to test the app. When it was tested enough (after a couple weeks) dev was pushed into master, which was getting pulled onto their server.
- Learning simple UNIX administration is a good idea. We had a person with extensive knowledge working on the dev server, but everyone should have an idea.
- If you want to make sure you're not wasting time setting up everything on everything's computers, make the dev server an actual dev server. That would kind of suck though because everyone would probably be developing in vim, and people hate that

- Don’t try to merge yours with ours till the end(ish). Also, when you do, copy and paste ours into yours and don’t replace where ours is on the server so that it is always up and running. Then, they can start using yours when it is tested (they had us use it overnight live on the radio station, so yours would be some intense user testing).
- Recommended roles: UI people, Controller people(i.e. middleware), backend people, dev server guy, Oddjob person. We had the people spread out over controller, UI and odd jobs, but there was a main person on both controller and UI to go to.
- Don’t get hacked, that kind of sucked.
- To be cheesy, have fun. You guys get to work with music, and who doesn’t love music. Plus, you’re graduating in a year, so there’s that.