Jacob Evelyn
CPSC 183
Final Project - MashupMachine
December 17, 2011

Implementation

I first got the idea for the MashupMachine website app in class, when we were talking about remix culture towards the beginning of the year. I thought about possible implementations, and after doing some research, several technologies sprung up that I knew I'd want to use.

Here's how it works:

1. A MashupMachine client tells the main MashupMachine server that it wants a song clip, at a certain tempo.
2. The MashupMachine server fetches a random song from another website that hosts lots of song clips for legal use. To speed things up later on, the MashupMachine server could host its own database of these songs to cut this fetching time dramatically, but for now this is good enough and it also lets us be broad in our implementation without having to create too much architecture.
3. The MashupMachine server, having a random song, then finds the tempo of that song, as well as the downbeat of the start of a major grouping of measures. It then cuts this grouping of the song, and time-stretches it so that it matches the desired tempo.
4. The MashupMachine server feeds this new, trimmed and stretched song clip back to the client. The client waits until the next major downbeat, and then plays the song on top of other songs already playing. This cycle continues indefinitely.

Notes: some of this is subject to changing (as mentioned in step two). It would probably be faster to have the server make its own local mashup and simply stream the data out, but that eliminates a lot of the possibility of individualization.

First, I'd need a way of identifying the beats of songs—a simple trick for humans but a relatively approximate, and computationally complex task for humans. Then I discovered The Echo Nest, a company that runs complex sound- and language-processing algorithms with the goal of identifying everything it can about music, from what artists are popular right now based on Internet textual data, to what the rigidity of a song's tempo is and how jazzy that might make it sound based on the song's actual sound. The Echo Nest then makes this data available to software developers via their own API, and this incredibly useful tool can be found behind hundreds of popular music apps and websites. The Echo Nest's detailed data on every single beat of a song would let MashupMachine know where best to take song clips, and how best to adjust the tempo of a song to match the tempo of the mashup being generated.

Just as important, I'd need a website to get music from. My first thought was the Free Music

Archive, an ever-growing repository for music licensed under Creative Commons (and some other similar licenses). Several things make the FMA immediately attractive:

1. When uploading songs to the FMA, artists have the option of indicating what they approve of people doing with their music. Permission to play publicly? Permission to remix? Permission to use for commercial purposes? This way I could filter my program's searches and only take songs I know the artists are okay with MashupMachine using.
2. It has its own API to use in accessing its database of songs and song data.
3. It continues to expand. This extends the possibility of infinite music generation and is to me a lot more exciting than drawing from a limited catalog.
4. It's free! Hence the name.
5. It's featured on The Echo Nest's website as an affiliate that The Echo Nest recommends working with.

And yet, unfortunately, the FMA doesn't live up to the hype. It keeps its song files in hard-to-get locations that make it difficult to access the raw song data. (Instead, where you think you're going to the song URL you're usually redirected to the URL of the song *page* on the FMA's website. They have a lot of security measures that make their files hard to get to, which is a real shame.

I ended up deciding to use (at least for the time being) EMI, another partner of The Echo Nest. EMI lets developers access 30- and 60-second preview clips of many of their songs, which makes things difficult for getting beat data for the song (more on this later) but it makes the size of the song downloads MashupMachine does considerably smaller.

The next major component is actually playing songs at the same time. Fortunately, the working specifications for various components of HTML5 are being implemented by browsers (especially Chrome, Firefox, and Safari), and among these is the new Web Audio API, which specifies native advanced audio behavior in the browser for the first time. Among the features of the Web Audio API is the ability to play and synchronize multiple streams of music, and to have precise audio timing for buffered sound playback. Which is exactly what I need!

The last big aspect is the matter of trimming and, more importantly, time-stretching song clips for easy insertion into the mashup. Because the HTML5 Web Audio API provides the ability to seek through songs and to adjust the playback rate of buffered sound files, this aspect is the least important for my creating a technical proof-of-concept because I could at the very least have the Web Audio API handle things. I would say, however, that this is also probably the *most* important aspect to handle well for the overall quality of the mashup experience, for several reasons. Having the Web Audio API skip to specific points in a song through JavaScript is an unnecessary seek operation and, worse, means we'll be slowing down our program by downloading entire songs of which we only intend to use a small clip. In addition, changing the playback rate in client-side JavaScript like this leads to some serious clicking and skipping of the audio as the sound card now has to seek to non-contiguous points of sound data.

I planned to perform these operations using the Echo Nest Remix API, a Python library (wrapped around the Python Echo Nest library creatively named pyechonest) that provides relatively fast and simple cutting, time-stretching, amplification, pitch-changing, song reversing, and other cool sound operations. Unfortunately (see below), this didn't work out quite as well as I had hoped, which indirectly became one of the major headaches of this project by forcing me to re-write pretty much all of my code. Since then, I've decided to ultimately try to use SoundTouch, a C++ sound-editing library that I would run through a custom module I write for a Node.js server.

"Well, this sounds hot, Jacob!" you say. "Can I see it?"

[Edit: Yes! Demo's here; it can take a little bit to get going so be patient. May only work in Chrome for now.]

Heh heh.

Unfortunately, the implementation isn't quite at the point of completion where I want it to be, though I've been working on it almost nonstop for several weeks now. Here's what I've done:

First, I built as much as I could in client-side JavaScript, which is (among the relevant technologies for this project) what I'm most comfortable using. I built a little interface, CSS'ed my page from horrifically ugly to moderately ugly, and then got busy making some data requests.

A big issue in doing everything from the client side is the restriction on cross-origin resource sharing that any readers of this (which is an unlikely assumption already) are probably unfamiliar with. Essentially, this "same origin policy" means that when you use the traditional technique for getting data from a server to a webpage (called an XMLHttpRequest—just go with it), instead of getting the data or the image or the song file you requested, you instead get a nice message saying that the "Access-Control-Allow-Origin" of the server isn't allowing you to do what you want. Now in HTML5 they're letting servers choose if they want to allow clients of certain domains to access certain content, but for better or for worse, neither The Echo Nest nor EMI have implemented that, so I was out of luck there.

Fortunately, there's a somewhat sneaky way around this via JavaScript called JSONP, which uses the really cool-sounding technique of script tag injection. Without going into details, JSONP lets you get textual data from certain servers that otherwise wouldn't give you squat. JSONP seemed like the answer.

Then I ran into OAuth. EMI, though generous enough to give developers 30-second song preview files of many of its songs, decided that it would be ludicrous to just put these preview files up on their server and be done with it. Today, music copyright issues are more delicate than

ever, and if EMI simply hosted all of these sample files for easy access, then anyone with enough determination to learn both The Echo Nest and EMI's APIs *and* enough time/interest for about four hours of coding could simply mass-download all two-thousand-or-so song preview clips. Which, you know, would… be bad. It might make people want to legally purchase the full songs from EMI, for instance. Or it might, of course, acclimatize people to preview clips to the point where nobody wants to listen to full three-minute marathon songs anymore, bringing the entire pop music industry to its knees… before it realizes it can now sell around seven or eight times as many songs as before. Either way, the powerful monopoly EMI has on these few specific song previews would surely be ruined, and EMI absolutely can't have that. So instead they protect their assets with OAuth.

OAuth is a technique with the rare quality of being fairly widely used (Twitter, Netflix, Foursquare, etc. all use OAuth to govern the interactions between developer apps and their services), and yet incredibly poorly documented. There are no OAuth tutorials at all. There are plenty of OAuth libraries in every programming language (mostly for the server-side implementation) which range from the simple to the obscene, but none of them are particularly easy to implement, and—most importantly—I really like to do things for myself when I can.

OAuth, in a simplified explanation, is a method for accessing restricted resources on remote servers. First, you have to obtain both a "consumer key" and a "shared secret" from the company hosting the service. Then, when you want to access a resource (like an mp3 from EMI, to pick a completely random example), you submit a typical HTTP GET request (this is easy to do), but this request has to have certain features. In it you have to a have a timestamp that is within the past ten minutes of the current time of the server you're accessing, along with a string of randomly-generated nonsense characters that won't repeat during any ten-minute time span. You also need a bunch of other parameters that specify everything from your consumer key to lots of details on the specific asset you want access to. Then you take that whole request, and put them in this special format where you alphabetize some things and put certain symbols in other places and decode the same symbols into letters in other places. The whole thing is so complicated and so badly documented that there's zero chance of getting it right your first two hundred tries or so. Finally, you take this whole request and use a special encryption algorithm that takes your "shared secret" as the key of sorts to the encryption, to generate your "signature." You then send a *different* request that has all of this info in it, including the signature, to the server, which uses the shared secret that it has stored on file for you to decrypt the signature and then see if everything in the signature matches the rest of the request. Finally, if it likes what you've given it, it gives you a time-limited URL for the song you originally wanted, so if you don't access that song right away you're toast. The worst part is definitely the fact that because every part of it is incredibly precise and weirdly formatted and encrypted, and all of your data can't be older than ten minutes, it's extremely difficult to test, a fact that's magnified a thousandfold by the lack of good documentation. Basically the only worthwhile documentation on OAuth in *the entire Internet* is on the Netflix website, which is never a good sign.

After spending two full days of self-loathing, I finally got OAuth to work when I ran into another

problem. Because I was doing everything client-side, I had no way of getting around the cross-origin restrictions to actually *get* the song files I wanted; JSON-P only works for certain formatted text information. I ended up hacking out something I'm calling *audio tag injection* that uses the same technique as JSON-P to get around these domain restrictions and get the song files as my code got the URLs from EMI through the use of the new HTML5 <audio> tag. I had, temporarily, a site that randomly got song clips and (albeit one-by-one) played them.

Unfortunately, by some cruel twist of fate, though <audio> and the Web Audio API are both products of HTML5, data from the former can't be ported to the latter, so I was stuck for how to do precise playback that only the Web Audio API could guarantee. There actually is a specification for the MediaElementAudioSourceNode interface, which links <audio> to the Web Audio API, but it's not currently supported yet by any browsers. Tough times on the cutting edge.

So this is when I busted out some Python. I figured that since I'd be using Python for editing individual files anyway, I might as well use it for proxy-serving me audio files from EMI, because server-side code (in this case Python) isn't bound to the pesky cross-origin restriction. I did some reading and hacked up a Python CGI script to accomplish this (my first real server-side coding ever), and finally (eureka!) was able to have songs loaded directly into Web Audio API buffers.

And then things went sour again. I tried installing the Echo Nest Remix API Python library, and ran into problems. I was using Python 3, and though the library said it works in Python 3, it clearly didn't. No problem then, I switched to Python 2. Not only did it still not work, but it also gave me a lot of trouble getting the basic CGI stuff to work. Hmm. I tried several different flavors of both Python 2 and 3, and every time was worse than the last. Which is the problem with having a programming language change so quickly that things that worked fine in Python 2.0.5 are broken by 2.0.8. (As a related aside, my final project for CPSC 185 last semester was in large part a bunch of Python scripts. Without knowingly installing any new versions of Python since then, I discovered this year that the scripts no longer run in either Python 2 or 3.) I mean, who decides to make a programming language that has two separately-evolving production versions?

With this, I gave up on Python entirely and turned back to JavaScript and to what I hoped would be a new friend, Node.js—JavaScript for the server.

I set about getting used to server-side JavaScript by porting most of my client-side JavaScript into Node. I thought this would be a trivial learning exercise but it ended up being a multi-day event because of the way Node handles things like files and streams. In fact, after about three days of non-stop coding I've ported everything I can to Node but it doesn't [as of 12/17/11] quite work right. I'm following documentation and tutorials perfectly, but somehow the files I download from EMI through node are always a little bit wonky (sometimes too big, sometimes too small, sometimes they play fine but claim to be zero bytes long, and sometimes they're exactly the right size but won't play at all). Hopefully I'll figure this out soon because I feel like this is the last

major hurdle (famous last words, I know) between where I am now and a basic working demo!

[Edit 12/18/11: Fixed the bug! Man, that one has been killing me for the past three days! Site now performs basic random song loading and playing. Get ready for a demo soon.]

I'll throw up a working website as soon as I can, and feel free to contact me (jacob.evelyn@yale.edu) if you have any questions!

Thanks for slogging through this lengthy read!

Jacob Evelyn