Cos 333: ASKDj final report

Anqi Dong, Ben Stallworth, Rahji Abdurehman, Tom Kelly

12 May 2014

1 Process

Although this project felt like it started somewhat behind, in part due to us not making substantial progress over spring break, we ended up with a working product that had a tight, clear set of intended features. As we had anticipated, the server side of the project was largely finished before the front-end, although minor changes had to be made to the back-end from time to time in order to make the front-end's operation more natural. As Professor Kernighan also anticipated, dealing with audio is not an easy task, and we thus had to postpone our ambitions of combining music from off-device sources: other users' devices, as well as Spotify, YouTube, Google Play Music, Grooveshark, and other cloud music providers.

The workflow for the iOS app was developed by all of us the first week back after the break. Shortly afterwards, the intended server APIs, accessible through HTTP requests, was developed. We originally wanted to plan out the database schemata at this time as well, but later decided that it would make more sense to work on the database structure as we developed the server code.

Due to changing needs and as we gained a better understanding of what was easier and what was significantly harder to do in iOS, some server calls were added or modified to aid client development. The most notable changes included disabling Django's CSRF (cross-site request forging) mechanisms, and treating post request bodies as JSON objects instead of the traditional POST structure of equal sign-delimited key-value pairs (as iOS didn't provide serialization options for traditional POST requests outside of actual web forms).

2 Interface

We structured our application to run off a server-client relation. The client played back music, issued user requests surrounding the state of the room the user was in, and reported the state of the server back to the user. The server stored the state of each room and thus served as the "oracle", reporting the reference state of a room to the client. It accepted requests from potentially many users, updated the state of rooms based on the requests, and returned errors if the requests were invalid or would result in an indeterminate server state. The server also has the capability to negotiate P2P connections between clients (say, to stream music) and to download remote shared resources, although we did not need either of these functionalities in the current iteration of our application. The main relation here is that the client provides updates to the server state, and the server checks for inconsistent input. The server may create errors due to invalid input, and the client must check for all errors generated by the server, both intentionally or unintentionally.

Communication between the server and client was based off of RESTful transactions using HTTP with either parameterized GET requests or POST requests with a JSON-formatted body. All responses given by the server were in JSON. We made an effort to conform to standards, such as using POST requests only for operations that mutate server state, and returning appropriate error codes for invalid requests. However, as we did not intend for third-party clients to access the server directly, there was no real need for standards compliance of the server beyond what was necessary for iOS (and potential future clients) to readily communicate with it.

3 Server

3.1 Architecture

We chose to use a session-based system backed by HTTP cookies to track users. This approach was an interesting challenge, because although we didn't ever ask the users to make accounts, we still bound music and room information to a particular user session, as specified by the cookie, and this meant that the user never had to specify his/her room after joining one. This system caused some trouble, however, in that it generated a large number of old users in the session store in the database—users who joined a room, improperly left, and repeated this process many times. We decided that we needed to periodically clean the session store by removing stale sessions, and this approach in theory works well.

We transmit metadata about a song to the server. In our case, this was the `MPMediaItemPropertyPersistentID`, but in other applications, this could also be something like a Spotify URL. Due to bandwidth and DMCA considerations, at no point does our server interact with actual music files.

3.2 Django

We originally thought about using Node.js, but eventually decided to use Django under the assumption that group members were more comfortable with Python and that Python had better namespacing facilities than JavaScript. Later, a small hacky Node.js server was created to deliver static responses when the server specs were still being worked out, and we wondered momentarily if choosing Python/Django was necessary. In the end, though, the main task of our web server was to manage requests and to store them in a database, and Django provided good facilities to do these things.

The most annoying part about using Django was allowing for agile development strategies surrounding the database. During development, we had to continually update and augment the database schemata. Unfortunately, Django 1.6 and below only handles adding new models as tables to the database, and doesn't automatically notice if rows have been added or modified, or if constraints (such as primary keys, foreign key relations, and uniqueness specifications) have been added to the database model. We ended up using South to perform these database schema "migrations" for us, but due to how South worked, the workflow we ended up using was:

1. Create a local database (say, in SQLite) and sync up its schema to mirror the current model being used in the AWS RDS database. South has commands to do this; generally, they work.

2. Run Django management commands to setup the South migration history in the local database.
3. Every time the database schemata (Django models) are changed, run the Django management command to generate a new migration file.
4. Apply the migration file locally.
5. Push the migration file to the AWS server and have it be applied there.

This meant that we had to maintain a full test database locally, which was a significant hassle and made the migration history annoying to version control. Upon checking Google for alternate opinions and conferring with classmates via Piazza, we came to the conclusion that we couldn't readily do much better.

3.3 Using AWS

Setting up AWS for deployment was one of the most frustrating parts of this project. We chose to use the Elastic Beanstalk framework to try to eliminate the difficulty of installing and maintaining our own database software, in order to make our use of AWS more like a platform-as-a-service (PaaS), analogous to what Heroku would offer. Unfortunately, Elastic Beanstalk was very hard to use, being poorly documented and having a weak online community answering questions about it. In many cases, it took a lot of flailing around on Google and piecing together incomplete comments from multiple sources to hack things together so that they would work more or less as intended on the server.

We used Rahji's AWS account because he had free credit, but due to limited face-to-face communication over break, setting up credentials for others on this AWS account was difficult enough that we needed to meet together after the break to figure it out. Frustratingly, Amazon somehow assumed that we would only need help spinning up a new instance of our server code each time, and actually didn't provide *any* documentation about how to connect and push to an existing environment (server and API for pushing changes). It took us a while to figure out how to set up exactly *one* environment, then to push changes to it from multiple computers.

It turned out to be fortunate that we were using "free" AWS credit, for two reasons: it is easy to accidentally spin up an extra EC2 or RDS database server instance, and be charged by it. Furthermore, pushing to Elastic Beanstalk somehow makes a lot of requests to the Simple Storage Service (S3), and the number of requests blew way past the 2000 free monthly request quota.

Elastic Beanstalk, while convenient for getting off the ground, may be too restricting for some applications. For example, we tried to set up a `cron` table in the server to perform database cleanup tasks. (We originally tried to call database cleanup operations when certain API calls were executed, but that led to nasty race conditions on the server.) However, Elastic Beanstalk doesn't specify where the application we deploy actually gets installed in the server's file system, and we suspect it even mangles the path so that it is not always the same. Because we didn't know the absolute path for where our Django application was installed, or even if it would have a permanent path, we were unable to get `cron` to invoke the correct commands.

We then contemplated using the Celery tool (`http://www.celeryproject.org/`) to perform task scheduling instead, but that looked even harder than hacking together a cron table, so we shelved this idea for the time being. (It is probably easy to start a second EC2 server to run this task, but given that our server is not receiving actual hits, we couldn't justify this additional cost.) The issue is that Elastic Beanstalk hides the server setup abstractions from the user, consequently making the process of incorporating lower-level tasks like cron jobs and custom server configuration options rather obtuse.

After getting everything set up properly, pushing server updates to Elastic Beanstalk was very straightforward. Even so, we will reconsider Elastic Beanstalk as an "easy-to-use" service in a future application, especially if it requires server-side customization.

3.4 Scaling and concurrency

For a project of this scope, one does not expect to need to deal with much scaling or concurrency issues. However, because it is not uncommon to see a user with a few thousand songs in their music library, we have to allow for a potentially large number of songs belonging to each user. Also, it is in our best interest to design our system to be able to deal with many users interacting with a given room at one time.

Most of dealing with concurrency was a simple matter of structuring the database carefully. For example, we originally tried to store the number of votes for a song as a field inside the song table. This means that we need to increment the vote count for a vote from any user, which easily leads to blocking issues. By making each vote a unique row based on the votee and the song voted for, we instead use the database as a journal and shift the burden of handling the concurrency onto the database.

It turned out that the structure of database queries sometimes has a pronounced effect on performance. For example, when listing the songs in a room's community pool, we output information about the number of votes on each song, and whether the requesting user has voted for the song. We must poll the rows in the votes table in order to do so, and we thought we would optimize by checking whether votes existed for each given song or not, using the query `song.votes_set.exists()`. However, this query ended up hitting the server each time we serialized a song object into JSON, and made the pool listing request take a minute for a few thousand songs. This was unacceptable, and after some profiling, we noticed this performance hit, and instead cached the list of songs with votes inside a calling object, instead passing this list into the song object serialization so that it could do a quick lookup in a set instead of querying the entire database.

4 Client

4.1 Interface

Because our app is intended to have a tight set of intended use-cases, we tried to make the user interface as streamlined as possible. When entering the app, users join a room (perhaps through creation), and use discrete panes to explore the room through three different view: the queue, the request list, and the pool.

The pool is simply a list of all the songs currently available to be played in the room. At the moment, only the DJ can add songs to the pool since we were unable to successfully play music off of a separate device. The DJ adds songs by clicking on a plus sign icon in the top right, which brings up a music-picker controller generated for developers by Apple. Using the Apple-generated controller brings the big advantage of giving the user a comfortable and familiar experience to quickly navigate through their library of songs. Unfortunately, it also brings the disadvantage that it is very difficult to customize the controller. For this reason, the controller does contrast with the rest of the app in some moments. In addition, there are a few limitations of the controller, such as being unable to deselect songs once chosen, that we were unable to find workarounds for.

The pool itself is listed in alphabetical order by song title. Depending on whether the user is the DJ or an audience member, clicking on a button in the right of the cell will have different effects. If the user is the DJ, there is a plus button that will add that song to the queue. If the user is an audience member, clicking the song will upvote the song. The app uses the server to keep track of the up votes of this member, and if the user taps the song a second time, the app down votes the song, canceling out the up vote. The app relays this upvote information to the user by greying out the upvote total if the user has already upvoted the song.

The request list is the collection of all songs that have been requested by an audience member. In other words, it is the list of all songs with at least one up vote that has not been cancelled. For both the DJ and the audience members we sort this list by the number of up votes for each song. Once again, the specific view for each song, along with the result of tapping the view, depends on the mode of the user. The DJ can add songs to the queue, while audience members can up vote the songs.

Finally, the queue shows the users what is being played at the moment and what is upcoming. This view was by far the hardest to implement. The majority of the work with Apple's music player API was done here. Unfortunately, the music player is one of the worst modules Apple gives to developers, and so dealing with audio playback was a bit of a nightmare. In some cases, the music player would send notifications concerning song and playback state changes in the wrong order. The music player was also notoriously fickle about its state when queried. We first attempted to set our "play/pause" button so that it queried the music player in order to figure out what view to show and action to perform. It turned out that these were calls were so unreliable that our button was almost never receiving the correct state. The queue view also has basic music playback controls for the DJ, and tapping the little play button next to a given song plays that song, placing it at the front of the queue in the process.

In order to provide high-quality playback and guard against issues such as a slowly responding server or a bad network connection, the DJ's queue is handled locally on the DJ's device, while audience members' queues are updated only from the server. The queue is the only type of state data where a client, rather than the server, is authoritative.

4.2 Design language

The color and design scheme for the app were an attempt to marry the simplicity of our first, "vanilla" implementation with an enticing finish so that the app will attract members for qualities beyond its functionality. Our initial app design, the one presented at the demo, was based primarily on the template classes that Xcode gives as a way to ease into iOS development. Therefore, this design was primarily monochromatic and focused on functionality over ease of use to the user. Our redesign attempted to introduce a sense of colour into the app. This helped to unify the various screens we present and also gave the app a much more professional appearance. We also focused on clarifying functionality on-screen in the redesign. Users in the second version are given much more feedback and much clearer hints as to what happens when they tap on a specific point on the screen and things of that nature. Our goal is to make our app fulfill the user's expectations so completely that a new user will be able to use the app in much the same way as an experienced pro. To this end, we ended up keeping many of the basic system-provided templates as base designs and customizing off of them, hoping to capitalize on the familiarity that users already have with their own devices. We believe that the result is an app that looks professional and polished, but is also easy and enjoyable to use.

5 Future additions

5.1 Geolocation

Our server code currently has provisions for accepting and validating latitude and longitude information when creating a room, and will store these values to the database when such information is in the HTTP request. When listing rooms, one can also specify the geographical centre of the search as a latitude-longitude coordinate. However, the server is not actually performing calculations based on these location points, and thus we have not bothered to call geolocation APIs in the client.

There is some finesse inherent in actually implementing a scalable geographical distance query on a database. To find the distance between two pairs of latitude-longitude points, we usually use the Haversine formula, which requires the evaluation of five trigonometric functions and a square root for each distance calculation. Although this formula is reasonably efficient for small datasets, it becomes unreasonable if we use it indiscriminately to perform a query over a large set of points. Packages such as GeoDjango abstract the process of performing these distance calculations, but they seem like overkill for our applications. We also considered rolling our own nearby queries for the database, but that would definitely require much carefulness and time.

5.2 Multi-source music playback

Currently, our iOS client plays music by inserting songs into the built-in iPod ("Music" app) playlist, and triggers playback on that playlist. This allows us to display album artwork and support control center and lock screen controls for the app in a way the user would expect. We surveyed song playback apps on the Apple App Store, and most apps that play music strictly from the user's local library also take this approach. However, this `iPodMusicPlayer` is actually already very brittle, and this resource sharing means that our app has immense trouble recovering if the user exits and begins playing music using another app. (However, the user shouldn't have a reasonable expectation of the music queue being preserved if he/she disrupts playback by going into a different music app anyway.)

To play music from other sources, such as a stream from another device or using the Spotify API, we will not be able to use these high-level provisions of the operating system, and will have to use lower-level audio APIs. For streaming, we don't even know if iOS has a good way of pulling a song file off a device and transmitting it to another device, especially if the audio file has DRM attached to it. Getting these sources to work will require some tough decisions, a large amount of app rearchitecturing, and the consideration of many edge cases and pathologies.

5.3 Refactoring

Because we didn't know what we were doing when we started this project, we sometimes don't follow good software engineering practices. For example, in both server and client code, certain swaths of code (such as generating and parsing HTTP requests) become boilerplate and are copy-pasted around. We imagine that we could use function decorators in Python and a class inheritance hierarchy in Objective-C to clean up this redundancy, but such a operation is too risky to perform now.