Cos 333: ASKDj internals

Anqi Dong, Ben Stallworth, Rahji Abdurehman, Tom Kelly

May 13, 2014

1 Server

We ran our server using the Elastic Beanstalk platform-as-a-service provider on Amazon Web Services. We used a Linux/WSGI/Python/PostgreSQL stack, using the Django framework to code the server application logic. Elastic Beanstalk (in theory) simplifies server code deployment into an act of pushing into a remote repository.

1.1 Server setup

The `server_django` directory contains all the necessary files for deployment to an Amazon Elastic Beanstalk environment instance. This code assumes that Python 2.7 is running on the server. Django also assumes that a WSGI application is accessible. (We used Elastic Beanstalk's Python 2.7 environment running on 64-bit Amazon Linux, which gives us these prerequisites for free.) The `requirements.txt` contains all the Python package dependencies for the project, and is parseable by `pip`. Important dependencies installed by `pip` from the dependencies file include Django 1.6 and psycopg2 (which provides bindings for Postgres).

The `05basic.config` file in the `.ebextensions` folder contains more configuration instructions for the server, including installing a package for Postgres administration using `yum`, and migrating the database to the latest version.

The `.ebextensions/08cronclean.config` file contains commands that should be invoked periodically to clear old entries from the database (something Django does not do manually). Due to substantial difficulties setting up a cron table in Elastic Beanstalk or running our own task scheduler, these cleanup tasks are only called upon each deployment for now.

1.2 Local setup

Python 2.7 (`http://www.python.org`) and Git (`http://git-scm.com`) are needed on the developer's local machine in order to run certain management and deployment scripts related to Django and Elastic Beanstalk. The South package is necessary to update and maintain the database schema corresponding to the code's Django models, and can be obtained using `pip`. A script (`migrate.sh`) is provided inside the server directory to simplify typing the command in Unix-based systems. You will need to maintain a local database (that is compatible with Django) and keep its schemata in sync with the remote database in order to generate these migrations.

In the submission's root directory, the `awspush.sh` script for Unix-based systems pushes the current Git commit of the `server_django` directory to the Elastic Beanstalk environment specified in `.elasticbeanstalk/config`. Detailed instructions for configuring the Git repository to push to Elastic Beanstalk are provided in `server_notes/setup_notes.txt`. Credentials (access key and secret key) for accessing the AWS API can be generated at `https://console.aws.amazon.com/iam/?#users`. After running setup, a `.elasticbeanstalk` folder will be created in the root directory, and certain git directives will be altered by the EB shell script.

A PostgreSQL server needs also to be created (such as one on Amazon RDS), and its connection details specified in `server_django/polyphony/settings.py`. Elastic Beanstalk provides facilities to automatically generate a database and store connection parameters as system environment variables. As it may be useful to have a separate local database for debugging purposes, we also include a sample local settings file at `server_notes/local_settings_demo.py` for importing settings locally.

1.3 Code organization

The Django setup largely follows the template created by the `django-admin.py startproject` command (using `polyphony` as the project name). The `polyphony` directory mainly contains configuration information, and the `settings.py` file tries to import a non-version controlled `local_settings.py` file in order to alter settings to make the server runnable on the local machine.

The actual web server code resides in the `apiserve` directory; this is the application's namespace. The database schemata that this Django application uses are defined in `models.py`. All routing is handled by `urls.py`, but response handling code is contained in three files: `room_manage.py` for room-related tasks, `diag_views.py` for testing and diagnostic views, and `views.py` for all other production tasks. We lifted and adapted the `JsonResponse` object from Django 1.7 and included an implementation in the `util` subpackage, in order to easily send JSON-formatted output back to clients.

A custom file is placed in `apiserve/management/commands` in order to define a `manage.py` routine that clears the database of rooms with no DJs.

2 iOS app

The iOS client was written and compiled in Xcode 5.1.1. The code currently has no dependencies on external libraries. All code and resources for the app can be found in the `Polyphony` directory, which is a buildable Xcode project.

2.1 Structure

The app is divided into three sections: The intro section, the DJ section, and the Listener section.

The first section contains the home screen (`MainScreenViewController.m`), the controller for creating a party (`StartAPartyViewController.m`), the list of parties (`PartyTableViewController.m`) and each party description (`PartyViewController.m`). The home screen is the initial view controller. It contains a title image and two buttons. The "Host" button performs a push segue to the `StartAPartyViewController`, which contains text fields to input a party's details. There is also a

button which triggers a segue to the DJ section of the app. The "Listen" button performs a push segue to the list of parties, which are populated into a `UITableViewController` using a server call. Clicking on one cell in the table segues to another `UITableViewController` with cells containing greater details about the party, and the option to join the party as a listener, taking the user to the third section.

The second section of the app is the DJ section. This section contains views for the queue, request list, and pool panes. The panes are organized in a `TabBarViewController` (see `DJTabBarViewController.m`). Each pane is itself embedded in a navigation controller to provide consistent views. The request and pool panes (`PartyPoolViewController` and `DJRequestsViewController`) are subclasses of `SongListViewController`, which is itself a subclass of Apple's `UITableViewController`. The queue pane (`DJPartyQueueViewController`) is a custom view controller that contains a `UIView` for the music playback buttons and a `UITableView` for the song list. The Pool and Request list are updated from the server. The queue is maintained locally. Songs are transferred to the queue from the pool and request panes by posting notifications, which the queue listens for. The add to pool controller (an controller from Apple) is called programmatically from the pool pane.

The third section of the app is the Listener section. This section is essentially the same as the DJ section, but with different functionality. The pool and request panes (`ListenerPartyPoolView-Controller` and `RequestsViewController`) are subclasses of the same `SongListViewController` as the pool and request panes of the DJ section. However, when tapped, the cells in these panes upvote the song rather than adding it to the queue. Upvotes (and cancellation of upvotes) are sent through a server request. The queue pane (`PartyQueueViewController`) is very similar to the DJ version, but the queue pane here does not allow for playback, and rather is read-only. Another difference is that it is updated from a server request, similar to the pool and request panes.

2.2 Requests and data models

We have abstracted out the code for server requests into various classes, with each one performing one request. In this way we can make server requests, both asynchronous and synchronous, simply by calling an initializer function. Sometimes we pass a handler function to perform after the request is complete. The arguments to these server requests are dictionaries that we serialize into JSONs. These calls are sprinkled throughout the code.

We used a few original data structures in our code, but they were generally very simple. We used the `PartySong` class as a lightweight encapsulation of a few properties of a song to pass to the server. In addition, we had two custom classes, `SongTableViewCell` and `DJSongTableViewCell`, to encapsulate the table cells we used to add buttons and make requests.

3 Testing tools

Most of the testing for this project was done ad-hoc, using either a browser or debugging tools in Xcode while running the iOS client. However, the `test_script` directory contains a few Python 2.7 scripts that perform tasks that are difficult to mechanize, such as making large HTTP POST requests. The `requests` package, version 2.2.1 or later, must be installed via `pip` for some of these scripts to work.