# Mezzo - Internals

**Delaney Granizo-Mackenzie, dgranizo@ (PM)**
**Mike Mulshine, mulshine@**
**Anna Ren, aren@**
**Steven Tran, stran@**

Mezzo is based on a Django framework and hosted on Heroku. The architecture is standard Django, with a couple of modifications to enable interfacing with the Heroku architecture. The Procfile in the main folder details Heroku settings. We use a SQLite3 database that communicates with Heroku's PostgreSQL. The database is accessed via Dajaxice in Javascript and HTML. Most of the business logic is performed in Javascript; this includes the games and the communication with the database. We chose to run all our audio generation code in browser. This makes our application very lightweight and adaptable to different systems. The system should be built using the virtual environment included in the project folder. This can be accessed by calling "source venv/bin/activate/" while in the project folder. The code is currently being run on Heroku. We are currently in the range of usage where we don't have to pay anything for Heroku, but if our usage increases to the point that we have to start paying for server service, we may have to look at other options for better pricing. Heroku guides for launching and deploying are available online: https://devcenter.heroku.com/articles/getting-started-with-django

To run our website locally, collect static using 'python manage.py collectstatic' (responding 'yes' when queried) and then start the server with 'foreman start'. Both commands should be run in the virtual environment. The server can then be access at 0.0.0.0:5000. Changing the local port of the server is possible and documentation is available online. https://docs.djangoproject.com/en/dev/intro/tutorial01/

Unless otherwise described, all our architecture is vanilla django with python. /eartraining/urls.py dispatches requests. All our views are in pages/views.py. Our settings file in /eartraining/settings.py contains information on how our site is set up. We use static file serving and code to handle that is located in settings.py and urls.py.

**Database:** We used a SQLite database (generated automatically using Django models) to store our user data. In-game and other animations were coded using Javascript, and dajaxice was used to communicate between the databases and Javascript (for dynamic elements on our pages). The system can be built on any computer that has Django. Django takes care of most of the building automatically; the only thing the programmer must do is call "./manage.py collectstatic" to collect and put

into effect all of the static files and "./manage.py syncdb" to create/sync the database tables.

Django was used to connect the frontend of our website to the backend. Django templating was used for all of the HTML pages; the index, about, help, and other non-game pages are extensions of our "base.html" template, while the game pages are extensions of our "gamebase.html" template. The templates include basic structure for each page, like the menu bar at the top of the page and all of the imported stuff in the head part of the page that was common to all of the webpages, while the actual content of each page can be changed in that page's .html file. Links between pages were implemented using urls.py and views.py, as described by Django documentation. User authentication and registration are also taken care of in views.py.

The project uses a SQLite database generated by Django models. To create new database tables, "models.py" in the "pages" folder should be modified. To modify existing tables, "models.py" should be changed, the existing database should be deleted, and "./manage.py syncdb" should be run again. Currently, username and password data are in one table, while username, current and max interval levels, and current and max melody levels are in a second table.

In order to communicate between the database tables and Javascript, we use dajaxice. The python functions for this are in the "ajax.py" file in the "database" folder. The Javascript callback functions for dajaxice are located in "ajax-functions.js" in the "/static/js" folder. The HTML pages use Javascript to call the python functions in "ajax.py" using dajaxice, and the result is passed to the specified callback functions. This is how we pass user level information to and from the database (to increase levels and to generate music for levels).

**Music Technologies:** Vexflow, HTML5 Audio, and RiffWave.js allowed us to dynamically insert musical notation and produce clean-sounding tones without any non-browser code.

**Vexflow:** Vexflow is a music notation rendering API written exclusively in and for javascript. All of our code written using the Vexflow API can be found in static/js/music-staff.js and static/js/melodies.js. The functions drawStaff(), showMelody(), and melodyToNotes() in melodies.js, and the functions drawStaff(), showInterval(), and intervalToNotes in music-staff.js are all of the functions that include Vexflow code. See the Vexflow tutorial for help on using Vexflow: http://www.vexflow.com/docs/tutorial.html.

**HTML5 Audio:** HTML5 Audio is the standard audio element used by most modern websites to generate audio, but it primarily allows for playing pre-extant audio files. So, RiffWave.js encodes waveform/audio data into a format that can be understood by HTML5 audio. RiffWave uses Pulse Code Modulation and the RIFF [Resource Interchange File Format] to create a playable file.

**RiffWave:** Riffwave.js and HTML5 audio code can be found in playMelody() (melodies.js) and playIMelody() (intervals.js). In both of these functions, we generate the pure tone waveform data and modify with a gain envelope. This means we actually write data to an array where each value encodes the position of the speaker at that point in time. We can multiply these values to adjust the volume of the signals. The process of adjusting the characteristics of the sine wave over time is called adjusting the envelope. Riffwave.js code uses this array and encodes it in a format that HTML5 Audio can understand.

**Intervals:** All our behind the scenes music logic is coded using semitones. To generate intervals, we randomly select a number from 0 to 11, corresponding to the music notes A3 to A4 on the piano, and then randomly select from a class of numbers corresponding to intervals (newIntervalFromTest() in music-staff.js) which vary from test to test. So if we get a 3 and a 4 then we will have a C4 (middle C) to a E4, a major third; this is true because a major third is 4 semitones. We also calculate an offset value to determine whether the note should be represented as a flat from a note above, or a sharp from a note below. So if we get a 1 associated with a midi note 7, then we will represent an E4 as an F-flat4. A similar process is used to generate intervals between melody notes, but with less constraint on the interval classes.

**Frontend:** For the front-end design of the website, we used HTML5 and CSS3, with the help of Bootstrap and one extra library, animate.css (for the landing page animations). Bootstrap provided CSS and JS libraries to style our navbar, buttons, modal popups, and to provide a grid system for the physical page layout. HTML5's canvas elements were used to display the staves and notes for each choice, and CSS3's webkit animations were used in the navbar and landing page. We used templating to extend base html pages and produce modularity. The html content can be found in templates/*.