CS3216 Group 9 AY19/20

Milestone 0

The Problem

At social gatherings and parties, it's hard for hosts or a designated DJ to know the preferences of the crowd. The DJ in question will also need to prepare a playlist in advance, or man the playback to ensure the party does not stop. With Juke Monster, music at social gatherings is now crowdsourced to your party guests. Guests now have the power to suggest songs and vote on them, taking away the need to have a DJ at all.

Milestone 1

Juke Monster

Juke Monster is a jukebox application intended for social gatherings and events, for hosts to grant DJ rights to everyone in attendance.

- Log into your Spotify account, select a playback device and open a room.
- Share the invite link via message or QR
- Sit back and relax as guests add songs into the queue and vote for their jam

Mobile Cloud - The Perfect Platform

Juke Monster was conceived to exploit the mobile cloud platform.

Given the ubiquity of mobile phone ownership, a mobile app is significantly more accessible than conventional desktop applications. Guests are more likely than not to own and bring a mobile phone!

In addition, most are reluctant to download a native application just to partake in an activity that is not likely to last more than a few hours. By using a browser based mobile application, participation is encouraged.

Juke Monster leverages the popularity of music streaming today by using Spotify as the platform of choice. Guests simply need to own a Spotify account to browse and add music to the room, and a Spotify Premium subscription is only required for the host to stream music on his playback device. No actual ownership of music is required.

Lastly, thanks to recent web technologies like service workers and websockets, users experience a blazing fast interface that updates songs and votes live. In the event of a disconnect, the user is able to perform a restricted set of actions, and any changes made would be transmitted on reconnecting. This leads to an overall experience rivalling native applications.

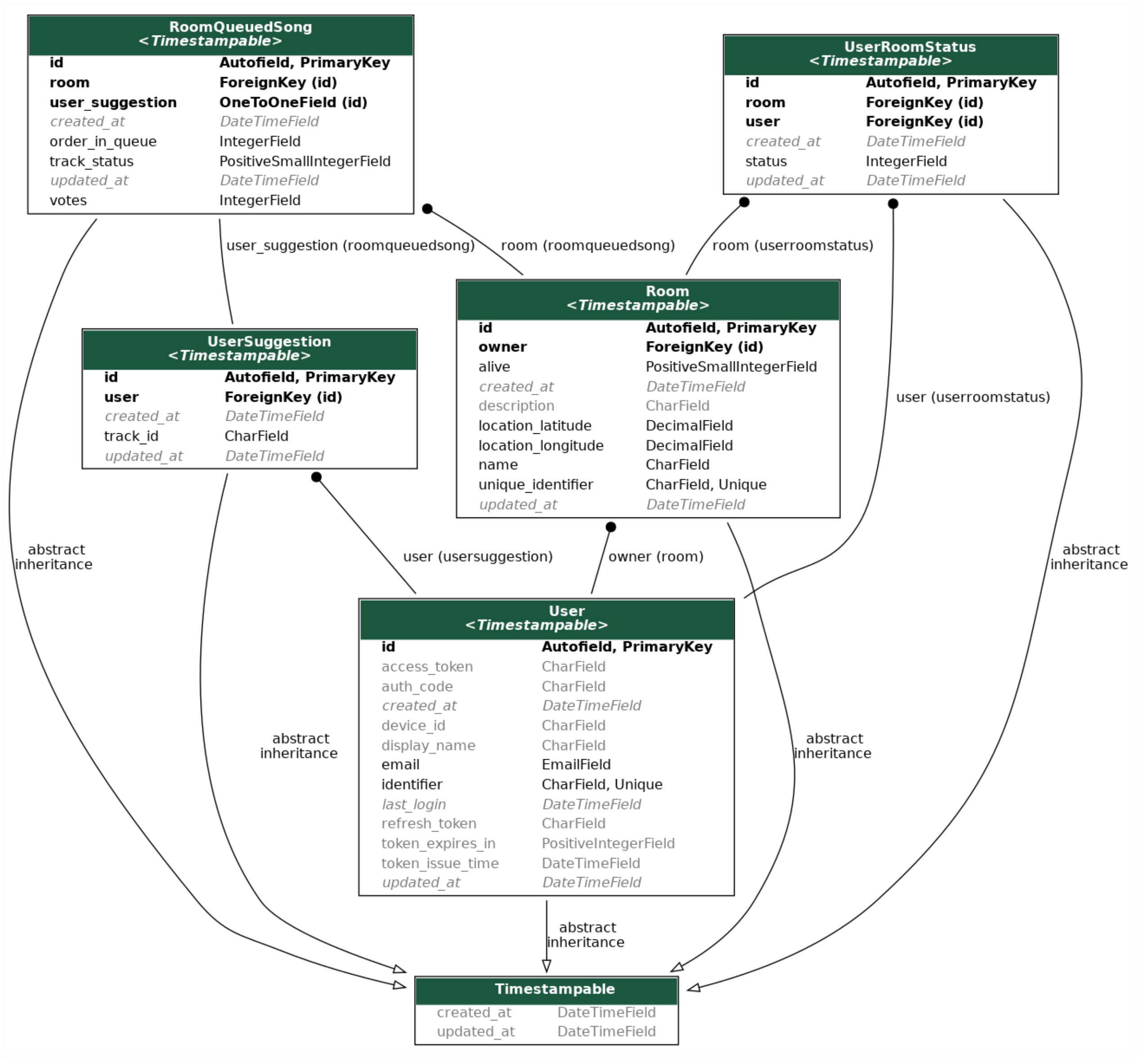
Milestone 2

- We need to tailor the user experience for two pools of users: event hosts and event attendees.
- It only takes 3 button clicks for hosts to turn their spotify premium enabled device into a jukebox.
- Attendees only need to click a hyperlink to see the song queue where they can vote and nominate songs.
- Both user flows are hassle free without installation.

We foresee that small-scale marketing in the form of posters around campuses would help the idea to gain publicity. Traction and widespread adoption would take place through overlapping social circles. When attending a gathering where juke monster is used, new people get exposed and would use it again in their next one. Being a good conversation starter and a novel idea, it could easily spread by word of mouth.

Milestone 3

The following ER Diagram was generated with the help of django extensions[1] and pydot[2].



RoomQueuedSong <*Timestampable*>	
id	**Autofield, PrimaryKey**
room	**ForeignKey (id)**
user_suggestion	**OneToOneField (id)**
created_at	*DateTimeField*
order_in_queue	IntegerField
track_status	PositiveSmallIntegerField
updated_at	*DateTimeField*
votes	IntegerField

UserRoomStatus <*Timestampable*>	
id	**Autofield, PrimaryKey**
room	**ForeignKey (id)**
user	**ForeignKey (id)**
created_at	*DateTimeField*
status	IntegerField
updated_at	*DateTimeField*

Room <*Timestampable*>	
id	**Autofield, PrimaryKey**
owner	**ForeignKey (id)**
alive	PositiveSmallIntegerField
created_at	*DateTimeField*
description	CharField
location_latitude	DecimalField
location_longitude	DecimalField
name	CharField
unique_identifier	CharField, Unique
updated_at	*DateTimeField*

UserSuggestion <*Timestampable*>	
id	**Autofield, PrimaryKey**
user	**ForeignKey (id)**
created_at	*DateTimeField*
track_id	CharField
updated_at	*DateTimeField*

User <*Timestampable*>	
id	**Autofield, PrimaryKey**
access_token	CharField
auth_code	CharField
created_at	*DateTimeField*
device_id	CharField
display_name	CharField
email	EmailField
identifier	CharField, Unique
last_login	*DateTimeField*
refresh_token	CharField
token_expires_in	PositiveIntegerField
token_issue_time	DateTimeField
updated_at	*DateTimeField*

Timestampable	
created_at	DateTimeField
updated_at	DateTimeField

There are a few things to note with regard to the schema. Firstly, we separated primary keys into functional and logical. The functional primary key was the primary key actually implemented, and used by the Django ORM to determine uniqueness of objects. Django uses an auto-incrementing number field (id in all of the tables), allowing it to uniquely identify each object regardless of whether another key was specified. The logical primary key was the field which we viewed as the logical identifier of each object (unique_identifier in Room and identifier in User). These were instead set to unique for greater flexibility, and we would be assured that the table would not be made invalid due to erroneous updates, as we were not modifying the primary key directly.

[1] "django-extensions" https://django-extensions.readthedocs.io/en/latest/. Accessed 27 Sep. 2019.
[2] "pydot" https://pypi.org/project/pydot/. Accessed 27 Sep. 2019.

In addition, the schema was written and implemented with the design once use forever philosophy in mind. As a result, some fields (e.g. location_latitude in Room) and tables (UserSuggestion) were not utilised for the scope of our project. These additional fields would enable a greater ease of enhancement of existing features and implementation of further features.

Milestone 4

Login

Initiates the authorization flow

Request Method + Relative URL	`GET /authorize/new`

Request Params	`{`
	  `redirect_to` *the link to redirect to on frontend*
	`}`

Response Values	*Redirects to Spotify API*

Initiates the exchange of Spotify authentication code for access and refresh token

Request Method + Relative URL	`GET /authorize/done`

Request Params	`{`
	  `code` `(optional),` - *auth code provided by Spotify if ok.*
	  `error` `(optional),` - *error provided by Spotify if user did not approve.*
	  `state` `(optional)` - *should contain the path to redirect to after sign in.*
	`}`

Response Values	**`200 OK`**
	`{`
	  `access_token,`
	  `refresh_token,`
	  `spotify_access_token,`
	  `spotify_refresh_token`
	`}`

Refreshes an expired set of tokens

Request Method + Relative URL	`POST /authorize/refresh`

Request body	**`(needs authentication)`**
	`{`
	  `access_token,`
	  `refresh_token,`
	  `spotify_access_token` `(optional),`
	  `spotify_refresh_token` `(optional),`
	`}`

Response Values	**`200 OK`**
	`{`
	  `access_token`
	  `refresh_token`
	  `spotify_access_token` *if provided in request*
	  `spotify_refresh_token` - *if provided in request*
	`}`

User

Updates the device that the host of a room is currently using

Request Method + Relative URL	`PUT /users/device`

Request body	**`(needs authentication)`**
	`{`
	  `device_id`
	`}`

Response Values	**`200 OK`**
	`{`
	  `device_id`
	`}`

Room

Gets information about a specific room. Notably, it checks if the requester (if authenticated) is the host of a room.

Request Method + Relative URL	`GET /rooms/<room_id>`
Request Params	**`(optional authentication)`**

Response Values	**`200 OK`**
	`{`
	  `name,` **-** *name as provided by user*
	  `unique_identifier,` *- auto generated unique 5-letter alphanumeric identifier*
	  `location_latitude,` *- unused currently, for future geolocation features*
	  `location_longitude,`
	  `alive,` *- room's alive state*
	  `description` *- description as provided by user*
	  `isHost` *- whether the requesting user is the creator*
	`}`

Creates a new room with the values input by user

Request Method + Relative URL	`POST /rooms/`

Request Params	**`(requires authentication)`**
	`{`
	  `name,` *- name of the room*
	  `description,` *- description*
	  `device_id` *- playback device*
	`}`

Response Values	**`201 CREATED`**
	`{`
	  `name,` **-** *name as provided by user*
	  `unique_identifier,` *- auto generated unique 5-letter*

	alphanumeric identifier
	  `location_latitude,` *- unused currently, for future geolocation features*
	  `location_longitude,`
	  `alive,` *- room's alive state*
	  `description` *- description as provided by user*
	}

Deletes a room

Request Method + Relative URL	`DELETE /rooms/<id>`
Request Params	**`(needs authentication) -`** *only the owner of the room is able to delete the room*
Response Values	**`204 NO CONTENT`**

Milestone 5

Query 1

ORM Query

```
user.access_token = token_data['access_token']
user.refresh_token = token_data['refresh_token']
user.token_issue_time = datetime.now()
user.token_expires_in = token_data['expires_in']
user.save()
```

SQL Query

```
UPDATE user
SET updated_at = '<date>', access_token = '<access token>',
refresh_token = <refresh token>, token_expires_in = <duration>,
token_issue_time = '<issue datetime>', identifier = '<user id>'
WHERE user.id = <primary key id>
```

This query is called whenever an existing user logs in. On log in, we receive authentication tokens from Spotify, and our backend stores these tokens for later use. As seen from the ORM query, we simply set the parameters and save the object. This is translated to a single SQL update statement setting the new values on the row with ID corresponding to the object.

Query 2

ORM Query

```
Room.objects.values('owner').filter(alive = 1, unique_identifier =
room_id)
```

SQL Query

```
SELECT room.owner_id
FROM room
WHERE (room.alive = 1 AND room.unique_identifier = <room id>)
```

The above query is called when our app needs to find out the owner/creator of a particular room in the database. Note that there is a flag alive = 1, used to denote if a room was previously deleted by the owner (1 for alive and 0 for deleted). This lazy deletion is implemented in our current iteration of our app to open ourselves to the ability to offer history-related features. For this particular query, we want to ignore deleted rooms as if they did not exist, and thus filter by alive=1. We also scope the result of the query to a single attribute, represented as the object itself in ORM and the primary key of the object in SQL.

Query 3

ORM Query

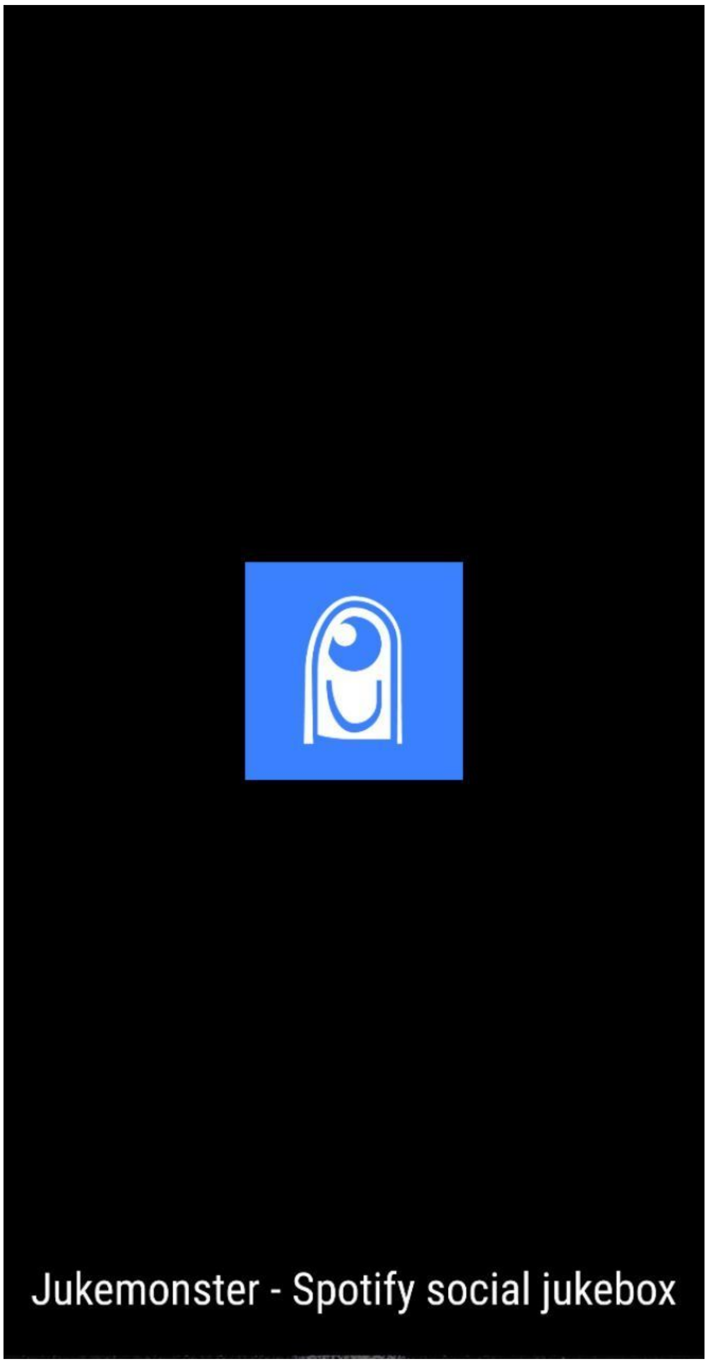
```
cls.objects.filter(identifier__in=user_ids).values('identifier',
'access_token', 'refresh_token', 'device_id')
```

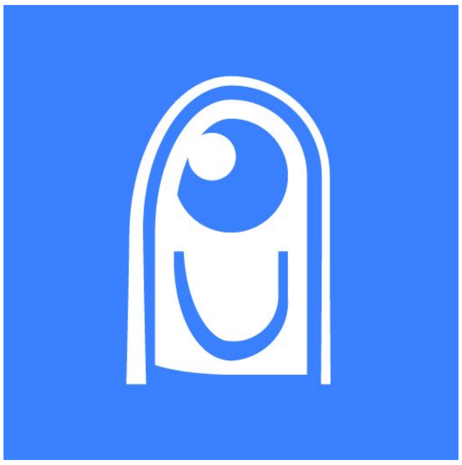
SQL Query

```
SELECT user.identifier, user.access_token, user.refresh_token,
user.device_id
FROM user
WHERE user.identifier IN (<User ID 1>, <User ID 2>, <...>)
```

This query returns a list of (user ID, access token, refresh token, device ID), given a list of user IDs. Primarily used by our playback scheduler, this query retrieves the information required for us to make the API call to Spotify to play the song on the user’s device. In ORM, we filter the objects down to those with identifiers found inside our input set, then scope the results to only the interested attributes. This translates to a fairly standard SQL SELECT statement that filters the rows by membership of the attribute user.identifier in our input set.

Milestone 6





Our logo is the Juke Monster itself. It is a stylized and anthropomorphized interpretation of the traditional jukebox design. The right half is more geometric and serious while the left half is more fun and wild, resulting in an asymmetry that creates tension and interest. Coupled with our unusual product name, we hope the icon attracts attention and invites people to try our product.

Milestone 7

Our UI design is simple. It has a home/landing page that lets you join or host a room, the host page is a simple form, and the room page, while the most complex, should be familiar to anyone who has used a music app. It consists of the currently playing song with a background that enhances the album art on one half of the screen, and a queue of songs that users can upvote or downvote to influence the song order on the other half. A floating action button brings up a modal to search Spotify for tracks to add to the queue. Toolbar buttons let users leave the room, invite others to the room, and for hosts, change their Spotify Connect playback device. Our UI design is the best possible because it uses familiar patterns in other music apps, is responsive to mobile, tablet and desktop platforms to fit a variety of uses, and is beautiful with album art displayed prominently.

After reading about Object-Oriented CSS, we thought it was the best methodology to adopt as Ionic framework itself follows the same general ideas, but goes further with heavy usage of CSS variables. We also felt more familiar with it than the others as it is used in Bootstrap, a popular framework that all of us have used before. The dark iOS theme supplied by Ionic framework was used generally, with tweaks here and there to make our design less generic. To implement these styling tweaks, we did not use inline styles but rather added new CSS classes to the `theme/variables.css` file. To separate structure and skin, we have some generic classes that can be applied to any element, like `.transparent`. To separate container and content, we did not use location-dependent styles, but inverted the relationship, e.g. `h1.display-1` instead of `.display-1 h1`. We also did not use IDs to style content.

Milestone 8

Best practices for the adoption of HTTPS:

1. Choose and ensure that the SSL/TLS protocols are up to date to make sure that they are free from past vulnerabilities. TLS 1.3 is the latest version.
2. Ensure enforcement of TLS. When there is mixed content or usage of third-party JavaScript libraries, even though the main page is served with TLS, the other assets must also be transmitted with TLS.
3. While TLS ensures protection of data, for sensitive information, it might be good to disable caching, which can be set in the HTTP header.

Certificate pinning is the practice of storing and only expecting a certificate from a host - rather than trusting any certificate issued by a certifying authority. To achieve this, it is either pinned on the client on development (not really possible for our case, since our server would be constantly updating its certificate), or pinned on the first connection, then compared with on subsequent connections and requests.

For our use case, this is not too important and hence we only use Certificate Authorities (CAs) to issue and update our certificates. While security is indeed important, in our app we are not storing any personal information of the user (as it is data obtained from Spotify), and we are mostly a "bridge" between the user and Spotify (hence any security we implement would have to comply with Spotify). As such, normal CAs are sufficient, and there is no need to practice certificate pinning.

Milestone 9

For the guests, the room state in kept in synchronization with websockets. In event of disconnect, the latest seen room state (containing song queue and vote counts) is kept in local storage. An offline banner is also displayed.

We allow the offline user to pass votes on songs he already sees which would be passed when connected to the internet again. All changes to the playlist and now playing would be made known too.

The nature of crowdsourced jukebox is online first because playlist state like now playing changes over a few minutes. So our offline voting mechanism is there to fill the gap of intermittent disconnect due to bad internet.

If the host gets disconnected from jukemonster, put his spotify device is still online, the music would continue streaming according to the queue set up guests. Spotify servers can continue cuing the now playing track to the playback track.

Milestone 10:

JWT is an authentication mechanism that uses signed and encoded information about the user and his related claims. This information is created, signed and issued by the authenticating authority (in our case it is our own backend server). To authenticate a request, the server/authenticating authority essentially just needs to:

1. Verify the signature of the token
2. Retrieve the encoded information (user, claims)

As such, the validity of each JWT token does not (or should not) need to rely on any server-stored information about of the given token. For this reason, JWT is said to be 'stateless' - there is no need for the authenticating authority to look up state about issued tokens as all relevant state should be encoded within the JWT token. This is great for scalability, as there is no need for a memory store to look up and ensure a token's validity.

However, being stateless poses several problems. As it relies on the content of the claims to find out if a user is permitted to perform operations, relying on fully JWT-based authentication might result in large token authentication. For instance, rather than a database lookup to see if a user is allowed to perform certain actions based on information of the user's account, a proper JWT would instead store all these information within the token.

In the case of our application, there is a need to find out if the user is allowed to perform some actions on a room or not, and this is dictated by their relationship with the room. As new rooms are constantly being created, it is not possible to have a way to encode all the rooms that a person can access within the JWT token in a constant-space manner.

Moreover, some state-based authentication would still need to rely on server-based memory to perform. For instance, an unexpired JWT token is still valid even after the user 'logs out', and the solution to sessions would still have to rely on a memory store of the 'logged in' state of the user.

Session-based authentication, on the other hand, is 'stateful' - authenticating authorities would need to store and look up sessions. While this is less scalable, it allows us to perform actions like sign in & out, as well as verify the state of the user with relation to their application usage, as mentioned above.

For our application, logged-in state is not that important. As such, we chose JWT as our authentication strategy. The permissions for most operations like room entry and room information do not rely on the state of the user in the app in our app usage, and any user is able to enter rooms and nominate songs. As such, these operations are stateless and 'safe' with JWT. For operations where users need to be in the room like song nomination and voting, WebSockets are required, hence a session is implied. As for more complicated permissions like the room ownership, we had to perform a database lookup to find owner information. As our app is already heavy on scheduled DB access to retrieve required information (like Spotify access token) to play and queue songs, the stateless nature of JWT over session authentication helps to let us run our server more reliably.

Milestone 11

Backend

Django (selected)
Pros
- Everyone in the team knows Python - important for speed of development
- Relatively skinny framework, so not bloated with unnecessary files and easy-to-understand codebase.
- (Less) opinionated structure enables us to more freely structure our app, as opposed to the other MVC frameworks, which may not really suit our app.
- Django REST framework provides an easy way to transform DB info into models into json (through serializers) and define endpoints.
-

Cons
- Not a lot of helpers out of the box - a lot of seemingly common functions like model initialisation, template rendering methods need to be written verbosely
- Third party libraries are poorly documented. This cost us a lot of unforeseen development time in the end.

Given the need to use our app to write scheduling logic, we decided to proceed with Django for the team's familiarity with Python. Furthermore, given our use case of the app and how unlike normal MVC it is, we also thought that the speedup from Rails would be negligible for our case.

Ruby on Rails
Pros
- We have an experienced Rails developer in the team
- Helpers and 'magic' out of the box can help speed of development
- Clear, opinionated structure makes it easy for the team to understand
- Existing rich library (gems) that allow easy extension

Cons
- Gems are pretty badly maintained. But not a problem for a short term project like 3216 :)
- Quite bloated - a lot of unnecessary things are set up on initialisation (such as tasks, mailers) for the use case for our app, so codebase would be polluted and daunting to navigate.
- Ruby is a tricky language for newcomers to understand, so in the context of this team and project we cannot afford the time to learn it.

This was a close second choice, but given the time frame of our project, the last reason here was why we didn't end up choosing it.

Express

<u>Pros</u>

- Very rich existing libraries to support a lot of features

<u>Cons</u>

- Javascript is messy, would slow down development
- No one in our team used before

Express was a clear no for our team.

Frontend

React (selected)

As React was a common standard that the whole team knew and had experience with, it was a clear choice. Besides, its popularity also means that there is a rich set of libraries that can help support our desired features.

Angular

One of our team members (Subbash) has experience with Angular, but the rest of our team have never used it. We decided not to use Angular as it has a steeper learning curve, which would slow down development.

Ionic Framework (selected)

Ionic 4 is now using web components, which means it is embracing open web standards and is no longer tied to Angular. What initially attracted us was the huge amount of components that emulate the native look and feel, and what sealed the deal was React support, including hooks. Even though we cannot use native libraries for assignment 3, using Ionic means that we can easily extend the app with native capabilities in future.

Material UI

Material UI was another choice we considered, with a huge variety of components too. In the end, we did not pick it because all Material UI components have their styles defined inline, which we thought this was a strange decision that goes against open web standards, and would make code very messy with the mixing with logic and style.

Mobile site design principles

1. Minimising data input
    - Room page

        - Simple taps to vote songs
    - Share dialog
        - Uses native share functionality if available with Web Share API
        - Otherwise uses QR codes and social media sharing so that guests do not have to manually input
2. Intuitive navigation
    - Home, Host and Room page
        - No unnecessary screens that could complicate host and guest flows
    - Room page
        - Header contains most navigation links with leave/delete and invite
    - Authentication flow
        - Redirects back to where the user came from
3. Finger-friendly tap targets
    - Room page
        - Floating action button to add song
4. Minimised clutter
    - Home page
        - Two accented buttons for main user flows while displaying use case cards below
5. Easy authentication
    - Sign in page
        - Uses pre-existing spotify account for identification

Milestone 12 & 15

Common Workflows

1. Room Creation
    a. Host opens juke.monster
    b. Logs in with spotify premium account
    c. Selects name, description and device
    d. Room created
2. Room Share & Join
    a. Tap invite button within room to share native links, QR codes and hyperlinks to instant messaging apps
    b. Open the above links and login with spotify account to enter the room
    c. Or open juke.monster and key in the roomId
3. Song nomination and voting
    a. Search songs by title and artists
    b. Tap to add them
    c. Upvote, downvote or stay neutral to songs added by you or others
    d. Playlist will follow the real time priority queue

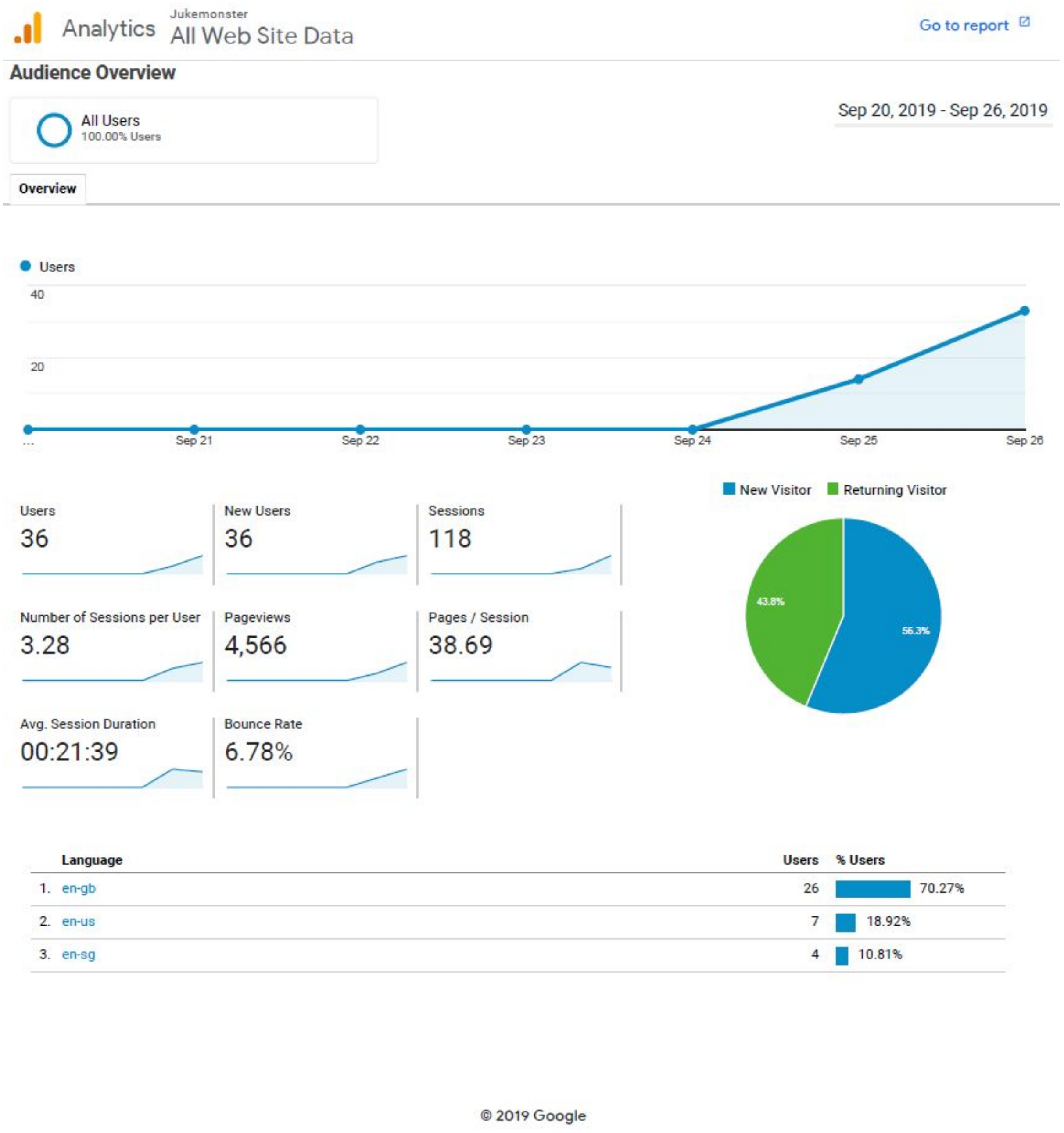
Why is above flow the most optimum?

Spotify accounts are popular among music listeners and guests usually have at least a free spotify account already. By using that to identify guests, they do not need to remember another username or password. We are also closely tied to Spotify as we need a user to be authenticated in order to search and play songs - an alternative would be to serve the songs straight from our server but that would be very heavy and expensive for us to implement.

Apart from the practical reasons that makes Spotify suitable for us, Spotify's social network also allows friends to share the current songs they are playing, which is aligned with our app's goals to enable the shared listening of songs within groups.

Instead of opening a website and keying in an alphanumeric roomId, we give the option of sharing links to room with native links to paste in instant messaging apps where they could have a mutual group. In case the gathering is large scale with multiple overlapping social circles, QR codes alleviates the need for having a mutual group.

Milestone 13



Analytics Jukemonster All Web Site Data

Go to report

Audience Overview

All Users
100.00% Users

Sep 20, 2019 - Sep 26, 2019

Overview

Users
36

New Users
36

Sessions
118

Number of Sessions per User
3.28

Pageviews
4,566

Pages / Session
38.69

Avg. Session Duration
00:21:39

Bounce Rate
6.78%

	Language	Users	% Users
1.	en-gb	26	70.27%
2.	en-us	7	18.92%
3.	en-sg	4	10.81%

© 2019 Google