Kulpreet Chilana
Dylan Joss
Yonglin Wu

6.170: Software Studio
Fall 2014
YouTube Jukebox: Real-Time Music Playlists for Everyone

Idea

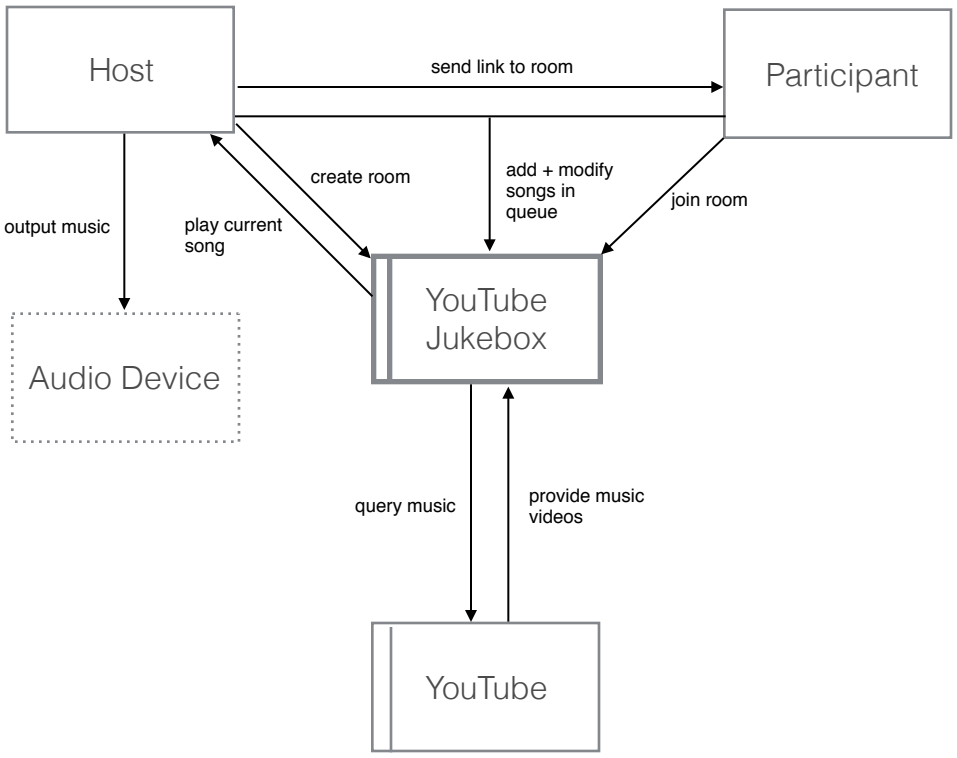
YouTube Jukebox is a web application that provides a simple way to collaboratively create music playlists in real-time using the many music videos available on YouTube. Once a piece of music finishes playing, it disappears from the playlist.

Purpose

The purposes of YouTube Jukebox are as follows:

1. **Allow users in close proximity of each other to all have control of what they're listening to.** Many times we find ourselves in a situation where only one person has direct control of what music is being played (e.g. DJ at a party, driver during a car ride). This application enables multiple users to decide what music should be played. Users can view what music is playing on a playlist and what music will be played, as well as add music.
2. **Allow friends to connect with each other's music tastes in real-time.** YouTube Jukebox provides a concrete way for friends to connect with each other by knowing what kind music each other is listening to when they are using the service.
3. **Make it easier to utilize YouTube as a service for listening to music.** YouTube is a general platform for hosting videos, but has also become one of the most popular websites for listening to music. YouTube Jukebox aims to highlight and build a platform on top of YouTube specifically for the purpose of accessing this large library of music

Context Diagram



Note: A host is a also a participant.

Concepts

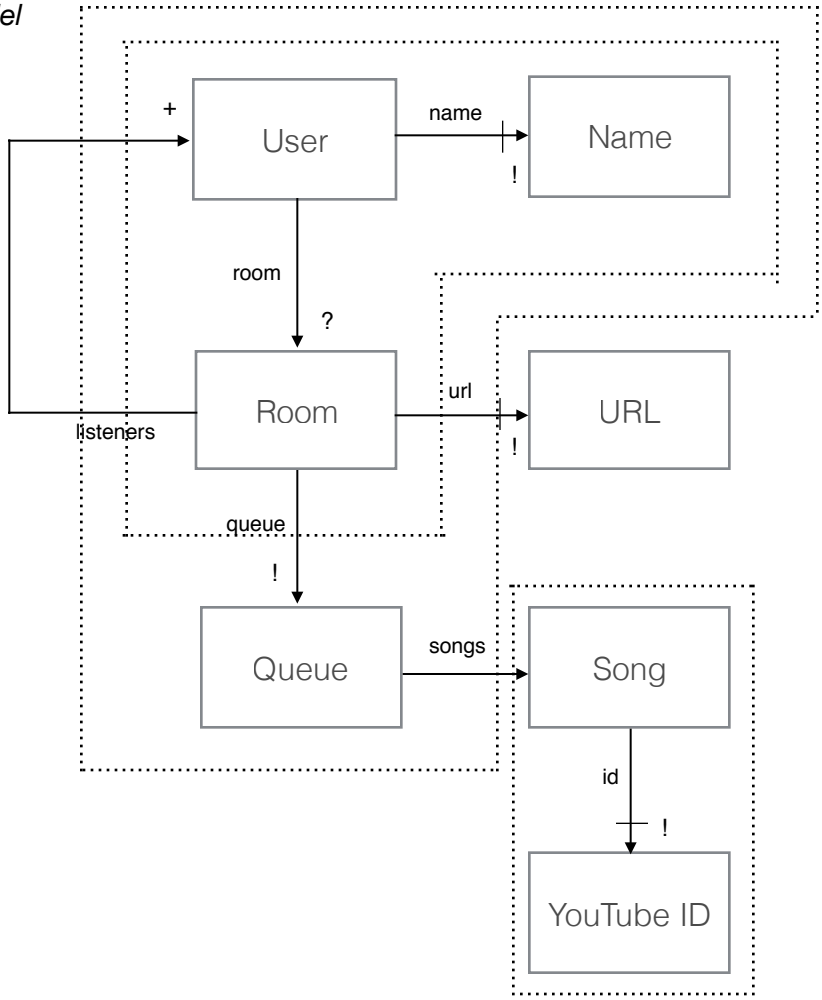
Room - a collection consisting of exactly one host, zero or more listeners, and exactly one queue. Is associated with a particular URL within the application. Representative of a group of users sitting in the same physical space. (Purpose 1,2)

Queue - the ordered sequence of music to be played. Once a song finishes playing, it is removed from the queue. Interchangeable with ‘playlist’ throughout this document. Users can search YouTube to add music videos to the queue and it will be played on the host’s device. (Purpose 2, 3)

Host - the person who creates a room and the only one who plays the music. May view the queue, add songs to the queue, or delete songs from the queue. A host must register an account before creating a room. (Purpose 1)

Participant - anyone with the link to a room shared by a host. May view the queue, add songs to the queue, or delete songs from the queue. A participant need not register an account before joining a room. (Purpose 1, 2, 3)

Data Model



API Documentation

API Endpoints:

List YouTube results from search query
POST /search
Request parameters:

- query: (String) query string to be sent to youtube

Response:

- success(200): returns a list of YouTube json objects
- error(500); returns error message: “Unable to fetch results from YouTube.”

Add a user to a room
POST /rooms/:roomId/users
Request parameters:
- name: name of the user to add to the room

Response:
- success(200): returns a list of listeners in the room
- error(500) returns error message: “There was an error joining the room.”
- error(404): returns error message: 'The room requested was not found.'

Delete a user from a room
DELETE /rooms/:roomId/users/:name
No request parameters
Response:
- success(200): returns a message: [User] just left room [roomId]
- error(500) returns error message: “There was an error leaving the room.”
- error(404): returns error message: 'The room requested was not found.'

Get list of songs from the queue of a room
GET /rooms/:room/queue/songs
No request parameters
Response:
- success(200): returns the list of songs in the queue
- error(404): returns error message: 'The room requested was not found.'

Add a song to the queue of a room
POST '/rooms/:room/queue/songs'
Request parameters:
- song: name of the song

Response:
- success(200): returns the list of songs in the queue
- error(404): returns error message: 'The room requested was not found.'
- error(500): returns error message: 'There was an error adding song to the queue.'

Pop off the first song from the queue of a room
DELETE '/rooms/:room/queue/songs'
No request parameters
Response:
- success(200): returns the list of remaining songs in the room
- error(404): returns error message: 'The room requested was not found.'
- error(500): returns error message: 'There was an error popping song off the queue.'

Design Challenges (Part 1)

Issue: Should access to rooms be restricted somehow? If so, how?

Possible Solutions

- A host can choose to share the link with whoever he wants. A person with the link can view and edit the queue. This has the advantage of not having to check user credentials when someone tries to access the room link. The downside is that an adversary with the link can maliciously modify the queue.
- Only specified user accounts can access the room. This has the advantage of security. The downside is the need to check credentials whenever someone tries to access the room, and requiring everyone who wants to use the service to register an account.

Resolution: we chose the first option as security is not a primary focus of this project, and the consequences of the malicious adversary having access to the room are minimal.

Issue: Should the focus of the application primarily be allowing users in the same room to create playlists or allowing users to remotely listen to the same music?

Possible Solutions

- The application is primarily for users in the same room to contribute to deciding what music to listen to. This ties in well with the "queue" concept of YouTube Jukebox, as users are listening to the same music, and the music will be taken off the queue once it is played. The downside is that users must be in the same physical proximity to take full advantage of the service.
- The application is primarily for users to remotely listen to the same music queue. This has the downside of having to deal with the complexity of users listening to different music at the same time, and the concept of music disappearing after it's been played doesn't work well with the remote focus as users do not know what others are listening to. There may be weird situations such as some music suddenly disappear while a user is still listening to it, or multiple pieces of music disappear at the same time because multiple users are listening at the same time. This is more suitable for a static playlist that doesn't change when music is played, not for a real-time music queue for YouTube Jukebox.

Resolution: we chose the first option as we thought the disadvantages of the second option conflicted with the original intent of the application: real-time, collaborative music playlists.

Issue: Which user(s) should be able to output music from their computer, and who should be able to edit the queue?

Possible Solutions

- Any participant can output music from the room, any participant can edit the queue. The advantage is uniformity. The downside is synchronization: suppose one user finishes listening to a song before another. By our model, the finished song is dropped from the queue, interrupting the second user.
- Only the host can output music from the room, any participant can edit the queue. The advantage is avoiding the synchronization issues of the above solution, and distinguishing the host from other users by giving him more application privileges.

Resolution: We chose the second option as it avoided the synchronization issues of the first, and because our application is primarily for users in the same room. It doesn't make sense to have multiple people play the same music in the same physical location.

Issue: Should we require user accounts?

Possible Solutions

- Don't require accounts at all. The advantage is that anyone can start using the service right away, and sharing among a large group of people is more effective. The downside is that the host can't use the application to share the link. The host has to send the link through text or online messengers or physically telling participants the room link.
- Require accounts for hosting/creating rooms but not for joining rooms. The advantage is that it becomes easier to associate room links with hosts (the room link can just contain the host's username), and anyone with the link can immediately join rooms. The downside is not being able to share the link within the application.
- Require accounts for hosting/creating rooms as well as for joining rooms. The advantage is consistency: an account is required before a user can perform any action on the site. The downside is the startup barrier for users getting started with the service.

Resolution: We chose the second option. We want to combine the advantages of being able to share to a large group of people at the same time and being able to identify the host and give specific privileges to the host (e.g. playing music).

Design Challenges (Part 2)

Issue: Should the url of API include resource IDs such as roomId?

Possible Solutions

- Don't include resource IDs in url of API, include them in request parameters instead. In our application, there are rooms and each rooms contains multiple users, and users can join certain rooms. In this scheme, the API for adding a user to room is:

  POST /api/join, with request parameters roomId and username

  The downside of this is that it does not follow RESTful convention.

- Include resource IDs in url of APIs, using a more nested structures. In our application, there are rooms and each rooms contains multiple users, and users can join certain rooms. In this scheme, the API for adding a user to room is:

  POST api/rooms/:room_id/users with username in request body

  The downside of this is that it has longer urls (with long room_id embedded), and with a nested structure. But it follows RESFTful convention well.

Resolution: We chose the second option. Technically both solutions work, but the second option follows the RESTful convention better and would make more sense for the users who are familiar with REST APIs. Also urls are supposed should refer to resources (nouns), not actions (verbs), and option one violated that. And like Charles Liu said, “the combination of action and resource names and determines a route better than just a "verb", and keeps the API from growing uncontrollably”, which I agree completely.

Issue: HTTP GET request is not allowed to contain a message body
(Note: we don’t have this challenge anymore since we changed the API urls, but it is still interesting to discuss). In our POST requests, the parameters are put in request.body, but it doesn’t work for GET, so there needs to be a work around.

Possible Solutions:

- Changing the method from GET to POST. This is a fast solution that gets things done, but it is a ugly solution because our method is retrieving some objects from server, not modifying contents on the server, so we really should use GET instead of POST.
- Store the parameter in as query parameter. This is a better solution then the first since the GET method is preserved. Now the url look like api/queue/songs/?room=[roomId], and the router can parse the query parameter.

Resolution
Clearly second solution is the better. Eventually we did not user either of the solutions because we redesigned the urls. now the parameters is embedded in url, like api/rooms/[roomId]. It is much cleaner and intuitive.

Data Design

See data model design with contours in Data Model section.

Based on the above contours, we created our schema. An example of the user and room schema are as follows:

User

```
{name: ‘Jason’, email: ‘cool_chinese_guy@gmail.com’,
password:’$2a$10$X76OmIbMnB4sGrbFvNfho’, room: Room object}
```

Room

`{listeners: [Jason, Dylan], queue: [YouTube song object]}`

We justified our data design decisions by considering what information an object would need in order to complete a particular operation. For example, the operations involving a room (e.g. joining the room or adding a song to the queue) don't require knowledge of the host of the room, only the ID of the host (which we set as the ID of the room upon its creation). Therefore, the host of the room is not an attribute of a Room object. Conversely, Room object is an attribute of a host since we need to know the host's ID in order to associate that ID with the room when the room is created.

name - needed for display purposes (e.g. to display the who is hosting the room or who is listening in the room)

email - needed for host authentication

password - needed for host authentication

listeners - needed to display who is listening in the room

queue - needed to display the list of songs to be played for the room

We received comments from P3.1 regarding splitting up the User object into a 'Host' and 'Participant' object so as to reflect the different operations and information associated with each. Because we received this feedback late Monday night (after our API was already implemented), we didn't think this change would be feasible to complete before the Tuesday night deadline. We will consider this design change for P3.3.