



Design Document

Team 12:

*Nancy Agarwal, Harshitha Janardan, Seth Maxwell,
Pranjali Raturi, Ian Ryan, Jack Sovich*

Index

Purpose	3
<ul style="list-style-type: none">• Functional Requirements• Non-Functional Requirements	
Design Outline	9
Design Issues	11
<ul style="list-style-type: none">• Functional Issues• Non-Functional Issues	
Design Details	18
<ul style="list-style-type: none">• Class Diagram• Sequence Diagrams• Navigation Map• UI-Mockups	

Purpose

The rise of new platforms such as YouTube have revolutionized how we consume media in Western culture. While online digital media is perhaps easiest to consume alone, many consumers desire to share media viewing with others.

streamshore is designed to fulfill this niche. It makes the viewing of online videos a shared experience similar to going to the movies together regardless of the distance between the viewers. It allows users to create public and private rooms where they can interact with others, simulating the experience of gathering together to watch videos they enjoy. The users will be able to communicate with each other through a live chat while watching a live stream of submitted videos. Users will be able to submit videos to a queue for a continuous viewing experience with others.

A similar platform, *watch2gether*, exists to serve the same purpose. However, this platform suffers from a poor video submission system. Namely, when a user submits a search query, the list of videos presented to choose from does not accurately reflect the same list returned when searching the same search query directly on YouTube. This makes finding a specific video difficult, and submitting a link to a video is more reliable than using the built-in search feature. Refining the user interface and improving the accuracy of the video search feature is the motivation for creating *streamshore*.

Requirements (Backlog)

Functional Requirements

1. Login, Account, and User Settings

As a user, I would like to...

- a. create an account on *streamshore*.
- b. log in to via my account credentials.
- c. recover my account password in case I forget it.
- d. verify my email address.
- e. choose a preset color scheme.
- f. change my password.
- g. become a premium user by attaining a set threshold of users in a room.

As a premium user, I would like to...

- h. have a special identifier.

2. Friends

As a user, I would like to...

- a. add other users as friends.
- b. view a list of my friends.
- c. see which viewing rooms my friends are in.
- d. set a nickname for every friend in my list.

3. Room Participation

As a user, I would like to...

- a. create a room to watch videos.
- b. easily join private rooms hosted by friends.
- c. see rooms I have marked as a “favorite”.
- d. invite others to the room I am currently in.

- e. add videos to a room's queue.
- f. see videos that are in the video queue.
- g. know how many people are online in a particular room and view a list.
- h. participate in a room as an anonymous user without logging in.
- i. downvote a video that is currently playing.
- j. downvote a video that is in the queue.
- k. play and pause the video locally and have it resume back to the room's video timestamp.
- l. locally adjust the video's volume.
- m. make a video fullscreen.
- n. view the videos in a theater mode.
- o. discover rooms that are marked public.
- p. search for rooms by title.
- q. submit videos from additional platforms other than YouTube (if time allows).

4. Room Setup

As a room manager, I would like to...

- a. set a name, URL, and description when creating a room.
- b. modify the description of my room.
- c. promote a user to be a room manager.
- d. push a video to the front of the queue.
- e. remove a video from the queue.
- f. restrict who can submit videos.
- g. host both private (hidden, link required) and public (easily discoverable) viewing rooms.
- h. host a friends-only viewing room.
- i. ban a user from a room.

- j. set parameters for the room's video queue, including who can submit, and whether the queue should be video-based or room-based.

5. Live Chat

As a user, I would like to...

- a. participate in a live chat with others in the room.
- b. toggle my view of the live chat or online users.
- c. directly mention someone in the live chat.
- d. direct message my friends privately (if time allows).
- e. see the history of direct messages (if time allows).
- f. be notified when I receive a direct message (if time allows).

As a room manager, I would like to...

- g. manage chat messages.
- h. mute a user.
- i. enable a live chat filter.
- j. disable the live chat in a room I have created.

6. General

As a user, I would like to...

- a. navigate easily to other pages on the site.
- b. create a playlist of videos from which I can choose the videos to be added to the room's queue.
- c. have a step-by-step guide on how to submit a video to the video queue.
- d. report a user to an administrator.
- e. report a room manager to an administrator.

As a premium user, I would like to...

- f. have my public rooms featured at the top of the homepage.

- g. have unlimited rooms (normal users limited to a certain amount).

7. Management

As a platform administrator, I would like to...

- a. view user reports.
- b. remove rooms.
- c. send announcement emails to all registered users.
- d. view a list of all active rooms.
- e. view a list of all registered users
- f. block the use of certain words in room names, URLs, and/or descriptions as well as user display names.

Non-Functional Requirements

Framework

We are building a web application with a separate frontend and backend. We intend to use Vue (JavaScript) as our framework for the frontend, and Phoenix (Elixir) as our framework for the backend. Utilizing a separate frontend and backend allows us to use languages with different goals (design being a priority for frontend, and performance being a priority for backend). In addition to Phoenix, we will use Ecto with MySQL to manage the database.

Security

Proper practices will be used to minimize risk of common vulnerabilities, such as SQL injection and man-in-the-middle attacks (e.g., form validation and HTTPS). Additionally, there will be a privileged user system, both site-wide (for administration), and per-room (for room management and moderation). Should a vulnerability be discovered, we will attempt to correct it within three days.

Usability

The interface will be created in a way that makes it easy to navigate and understand every feature. It will be easy to create a user account and video viewing rooms. We want to make it simple for the users to add videos from YouTube to a room's queue. Users should be able to participate, even if in a limited manner, without creating an account, as requiring account creation tends to be a barrier to entry for first-time users. Search will also be a primary feature, as having to leave the site to find videos takes users out of the experience - the search should not only be present, but accurately represent YouTube search results (something *watch2gether* greatly fails at).

Scalability

Our backend, Elixir, is deliberately chosen so that our application will scale sufficiently to handle any amount of users. Using Discord as a case study (<https://blog.discordapp.com/scaling-elixir-f9b8e1e7c29b>), it is evident that with proper design, an Elixir backend can scale to handle a vast amount of users concurrently. The system should be able to handle a workload of at least 10,000 concurrent requests with a chat delay of less than 1 second from message sent to message received, provided a standard, working internet connection. Given our intended division of tasks, the backend should be the limitation for concurrent users, not the frontend.

Hosting/Deployment

Both the frontend and backend should have a 24 hour availability. In order to accomplish this, a VPS (through AWS) will be utilized for deployment of the backend. The frontend will be hosted using GitHub pages.

Design Outline

Our project is a web application that allows users to create rooms where other people can join and collectively watch the videos queued in the room. Our web application uses the client-server model. This structure enables a single server to synchronize each user's activity and the video they're watching more easily. The client will send requests to the server, access data from the database in the server and respond back to the client.

Client

1. The client provides the users with an interface that allows them to interact with the system.
2. The client sends HTTP requests to the server through a JSON object.
3. The client will receive a JSON response from the server to display video viewing rooms, videos in a room's queue, and the messages in the live chat.

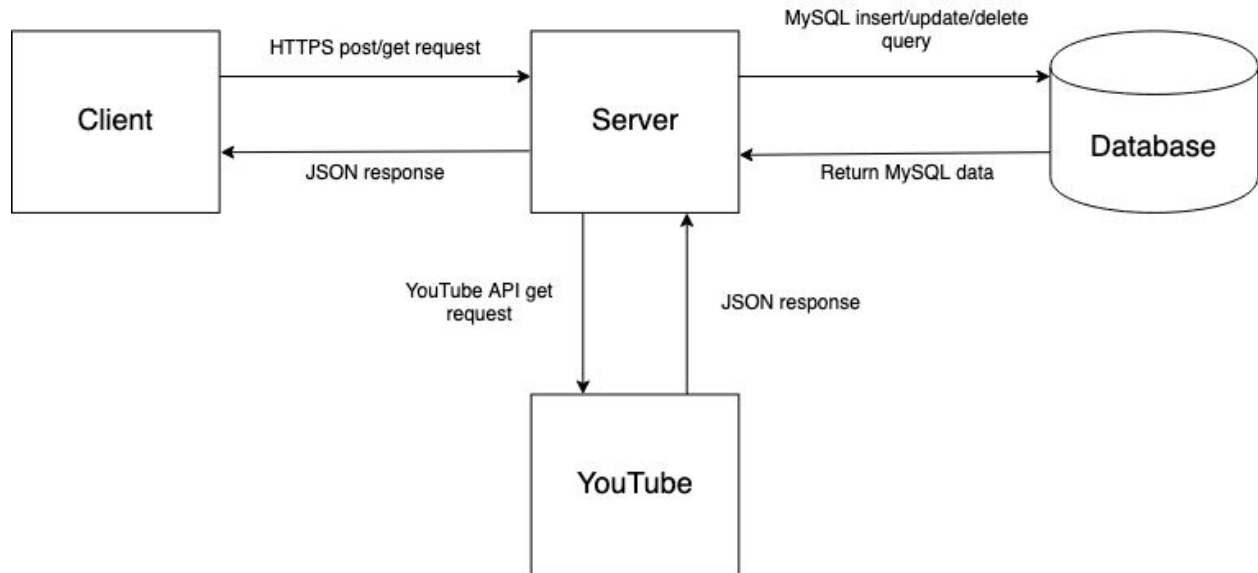
Server

1. The server will respond to client requests and transmit data between the client and database as requested.
2. The server will send chat messages between users in a room in real-time.
3. The server will retrieve videos from YouTube (or additional sources) by use of the service's API.
4. The server will transmit and synchronize the video feed among all clients in a room.

Database

1. The database will store user data (username, encrypted password, email, playlists, etc.), room data (name, settings, room managers), and direct message history.
2. The database will only send and receive data to/from the server.

High Level Overview of the System



Many web clients will connect to the server hosted in AWS. Clients will communicate with the server for user authentication, synchronizing the video feed, and fetching and sending live chat messages. To store user and room details, the server will communicate with the database to create, destroy, and update records. The client will also send YouTube search requests to the server, which will then communicate with YouTube through the YouTube API. The video data of selected videos will be stored in the database to avoid repeated API calls to YouTube.

Design Issues

Functional Issues

1. What information do we need for signing up ?

Option 1: Email, password

Option 2: Email, password, phone number

Option 3: Email, username, password

Decision: Option 3

Justification: For account creation, an email is necessary for account recovery.

In the case that a user forgets login credentials, an email can be sent to restore access to the account. A username is essential to being able to uniquely identify a user, and will serve as the uid. The password will provide additional security measures against unauthorized access to an account.

2. Do users have the option to view rooms without creating an account?

Option 1: Yes, users can access rooms without logging in

Option 2: Users must be logged in to view rooms

Decision: Option 1

Justification: Users should have the option to view public rooms because some users may not want to create an account but still wish to watch videos. This allows people to discover the platform before creating an account. While this will result in additional development effort, we believe from anecdotal evidence that requiring account creation will serve as a deterrence to new users.

3. How should users be notified when they are invited to a room?

Option 1: Email

Option 2: Site notification

Option 3: Email and site notification

Decision: *Option 3*

Justification: Accessibility is a large concern with room invitations, so regardless of whether the user views the notification on *streamshore* or email, joining the room should be instantaneous. Users should have the option to be notified of invitations even when they are not online, as joining a room through an invitation link may be the only reason some users will be accessing the site. This link should also be accessible on the site, as users should be able to join from anywhere on the site. The user will also have an option to turn off email notifications if they wish to reduce further notifications.

4. Should a user's profile be visible to others?

Option 1: Yes

Option 2: No

Decision: *Option 2*

Justification: There is no information available in the profile section that needs to be shared with other users. The profile serves primarily as an area for editing data associated with the user's account such as: the rooms a user owns, favorite videos, playlists, and account details. The nature of the profile section does not warrant the ability for it to be viewed by others, as it would detract from work towards more important features for minimal user benefit.

5. When should a requested video be removed from the queue?

Option 1: A video should only be removed when a room moderator removes it.

Option 2: A video will be removed from the queue when, at any time, more than 50% of users in a room vote to remove it.

Option 3: A video will be removed from the queue if more than 50% of users in a room vote to remove it, 60 seconds before the video is about to play.

Decision: Option 3

Justification: A room's audience should have the ability to self-moderate the videos that are played. In order to remove a video, users can downvote videos in the queue. Rather than constantly checking the percentage of users who have downvoted it to determine if a video should be removed, the downvote to room size ratio should be checked 60 seconds before the video will be shown, as this reduces the computational intensity of removing a video from the queue. After a video is removed, the next video in the queue will take its place.

6. What websites should we be able to play videos from?

Option 1: YouTube

Option 2: Twitch

Option 3: Netflix

Option 4: Any video streaming platform

Decision: Option 1

Justification: Paid platforms require an account to access their content, and streaming this content to users without paid accounts on those platforms may reveal legal issues. Additionally, since every user has a different account it would be difficult to sync the videos up. Another limitation is time: it will take additional man-hours to implement each source, hence we agreed it is

best to start with one. In order to satisfy the most users, it would be best to use a popular and free platform, which would be YouTube.

Note: We considered the possibility of Option 4, which would involve streaming a web client to all those in a room (the only could then navigate to any video on the web and share it). This is optimal from a user standpoint, but we concluded that the server resources required for this to be viable were too great of a drawback for the benefit offered.

7. Should we keep a chat history in rooms?

Option 1: Chat history for lifetime of the room

Option 2: Limited chat history (ex. 1 hour)

Option 3: No chat history

Decision: Option 3

Justification: Since the primary purpose of *streamshore* is real-time interaction, we do not believe users would greatly benefit from having chat history load when they join a room. Without the accompanying video side-by-side with the chat, old chat messages may not even make sense in many cases, and because of this, storing and transmitting chat history is not only unnecessary but may even detract from the experience in some cases. We will retain persistent direct messages, for the sake of scheduling or other uses.

Non-Functional Issues

1. What frontend language/framework should we use?

Option 1: React

Option 2: Angular

Option 3: Vue

Option 4: Ember

Decision: Option 3

Justification: Vue's structure allows for HTML, CSS, and JavaScript to be baked into every component. This allows for individual components to behave independently of each other. Additionally, it allows for direct control over which components can interact with others, and this approach makes it easy to create generic components that can be reused for multiple purposes. This structure reduces the overhead involved in the concurrent nature of the project.

2. What backend language/framework should we use?

Option 1: Spring Boot (Java)

Option 2: PHP

Option 3: Node.js (JavaScript)

Option 4: Django (Python)

Option 5: Phoenix (Elixir)

Option 6: Go

Decision: Option 5

Justification: While a backend like Node.js will work for the terms of the project as a class design, using Elixir will allow for the scalability of more users. This is due to more efficient performance in concurrency and parallelism, two aspects crucial to the processing of simultaneous chat messages in real-time (or as close as possible). Go would likely be a suitable framework for these reasons as well, but our main inspiration behind Elixir is Discord, a popular chat application that has the chat functionality of its backend written in Elixir.

3. What should we use to host the backend?

Option 1: AWS

Option 2: Heroku

Option 3: Google Cloud Platform

Option 4: Self-hosted

Decision: Option 1

Justification: AWS has the ability to load balance, which will prove to be useful if *streamshore* experiences an influx of users. Additionally, AWS performs well with file transfers, which will be particularly useful for handling video files for a multitude of rooms. Options 2 and 3 have comparable performances, but the pricing for AWS is more compelling than the others. The scalability of all three of these options, however, will be crucial in transmitting a large amount of data concurrently during peak times, which will increase reliability of *streamshore* and improve user satisfaction.

4. Which database should we use?

Option 1: MySQL

Option 2: PostgreSQL

Option 3: MariaDB

Option 4: MongoDB

Option 5: Redis

Decision: Option 1

Justification: MySQL and PostgreSQL have the best integration with the backend we chose (Elixir with Phoenix), so it made the most sense to use one of the two. Our research indicated that MySQL was best for read-heavy applications, which we expect our application to be. We believe that user and room data will be read significantly more frequently than modified, so MySQL was the better fit for our use.

5. What standard should the API interactions follow?

Option 1: OpenAPI

Option 2: JSON API specification

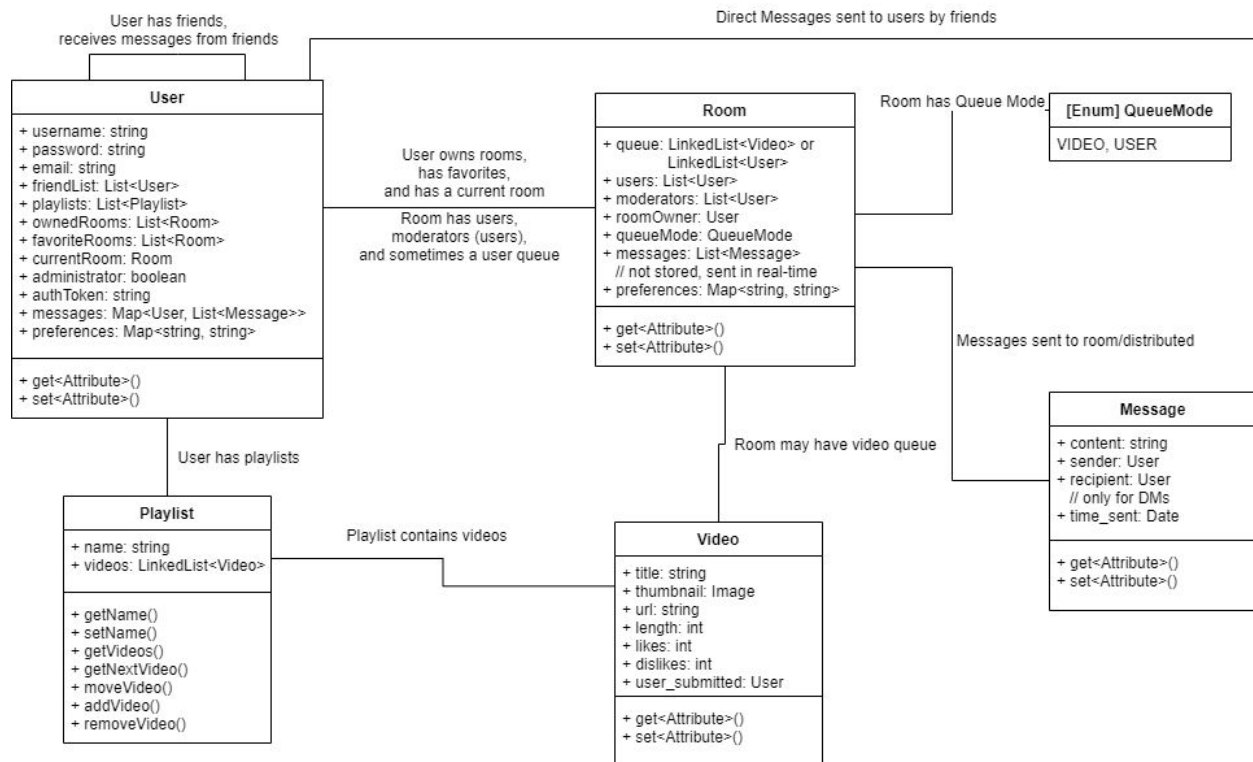
Option 3: JSON Schema

Decision: Option 2

Justification: The JSON API specification is a well-established standard for designing APIs. Its appeal is a consistent JSON request/response hierarchy. If additional video sources are added, using the JSON API specification will enable the same API request/response structure to be used. This would significantly reduce the overhead involved with integrating new video platforms.

Design Details

Class Design



The class structures represent the formatting of data structures throughout the system - data will be modeled to reflect these patterns for the frontend, backend, and database.

User

- Represents a user account.
- Contains the user's authentication details, friends, playlists, rooms, messages, and preferences.
- Created when a user signs up on the platform.

Room

- Represents a room, a viewing area, which users may join.
- Contains the room's queue and users, as well as the room preferences.
- Created when a user creates a room.

Video

- Represents a video submitted to a room (either in queue or playing).
- Contains video metadata, as well as the URL to load the video.
- Created when a video is referenced by room submission, playlist addition, or YouTube search.

Playlist

- Represents a user's playlist of videos.
- Contains a name and list of videos.
- Created when a user creates a playlist.

Message

- Represents a message sent in live chat or directly to a user.
- Contains message text and metadata.
- Created when a message is sent in either a room or a direct message.

QueueMode

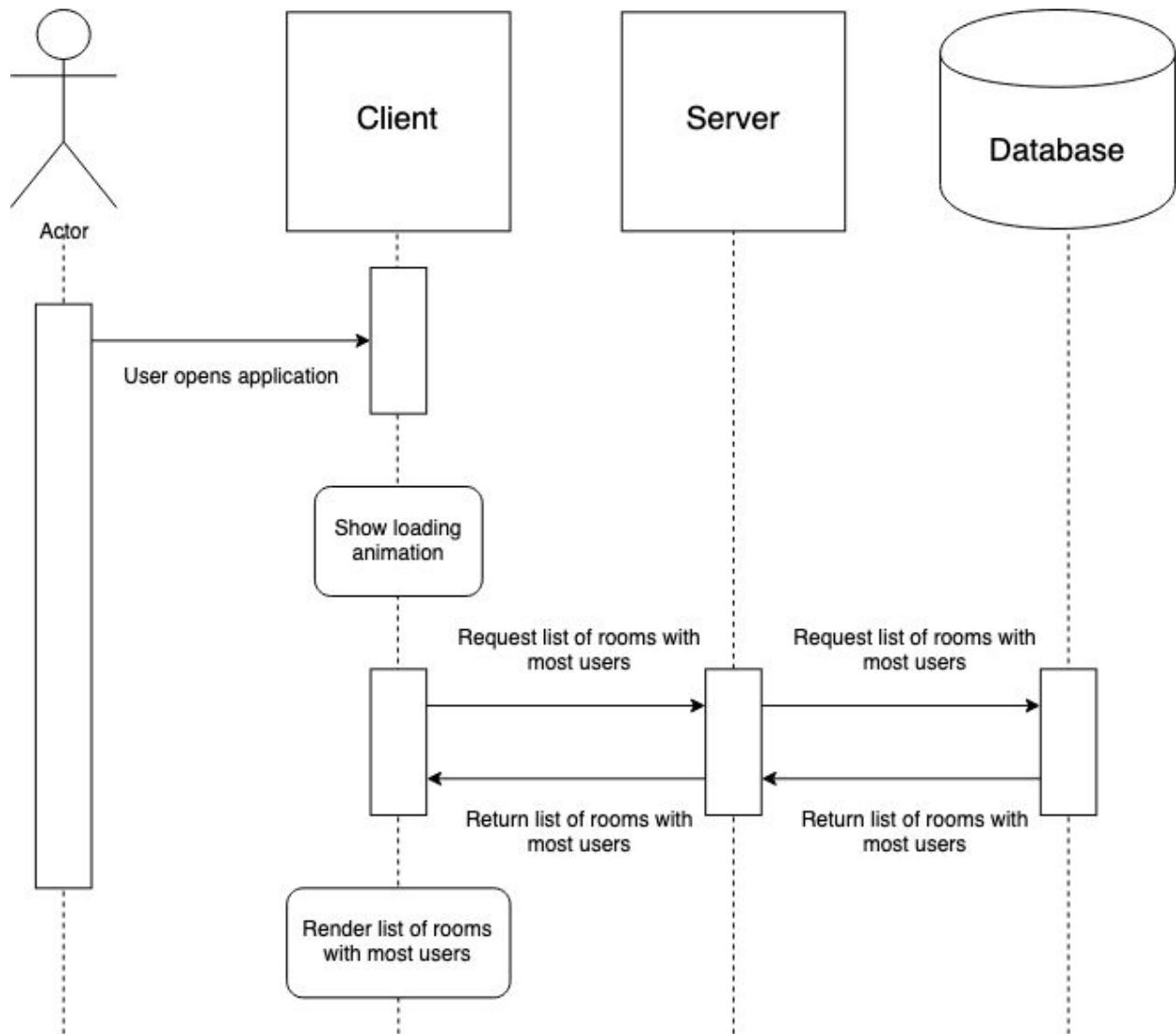
- Enum to specify the mode of the queue for a room.
- VIDEO means videos are submitted to a queue (next video in the queue plays next), USER means users are added to a queue (first video in the user's selected playlist plays next).

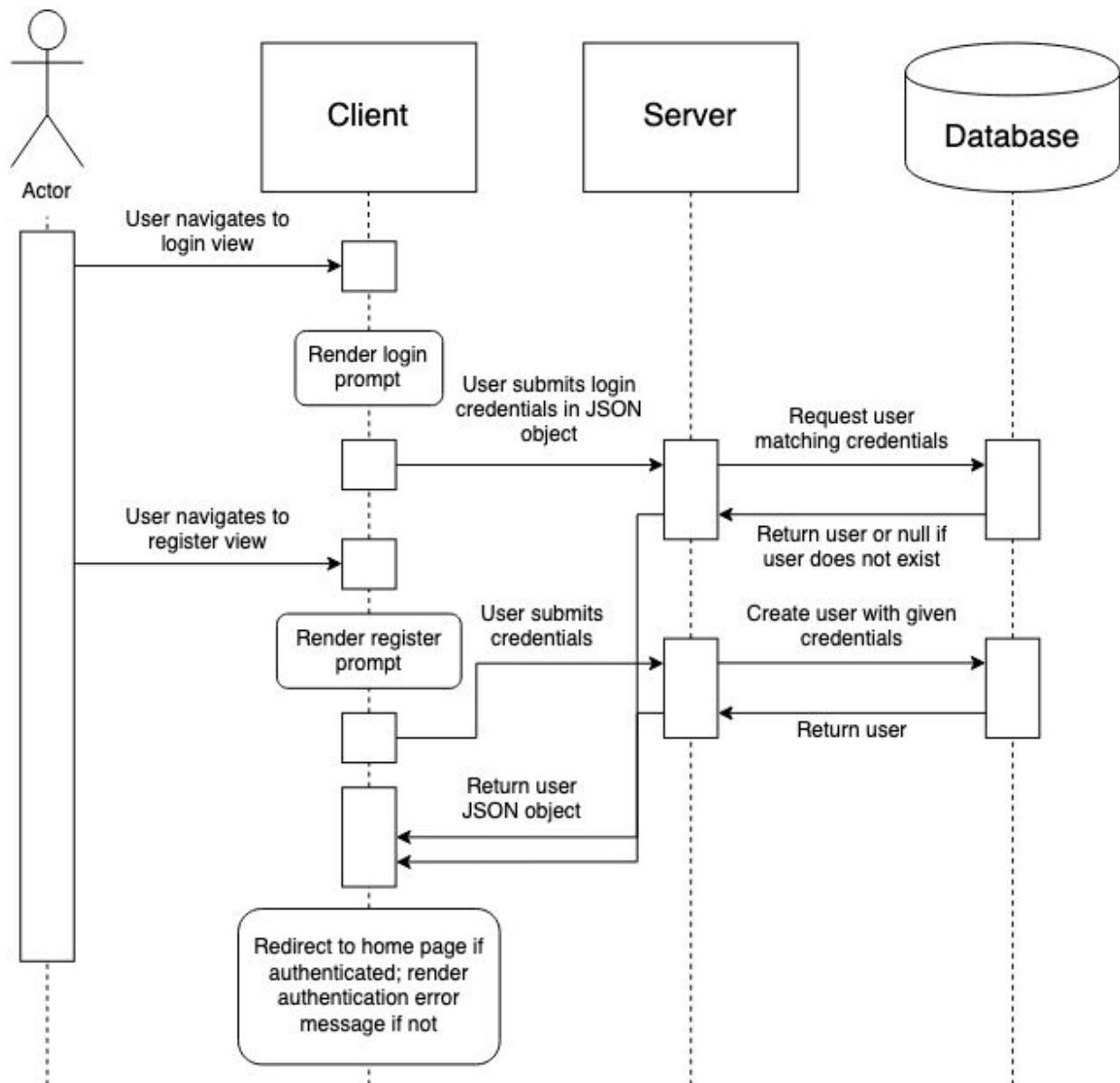
Class Interactions

- *Users* will be attached to friends, which are other *users*.
- *Users* may receive *messages* from friends, which maps another *user* to a list of *messages*.
- *Users* will be attached to *playlists*, which are named lists of *videos*.
- *Users* will be attached to owned *rooms*.
- *Users* will be attached to favorite *rooms*.
- *Users* will be attached to their current *room*.
- *Users* may belong to a queue in a *room*.
- *Videos* may be attached to a *playlist* or a queue in a *room*.
- *Playlists* will belong to *users*.
- *Playlists* will be attached to *videos*.
- *Messages* may be sent by a *user* to a *room* (not saved, but distributed to all users), or to a friend/*user* (saved).
- *Rooms* will have a *QueueMode*.
- *Rooms* may have a queue of *videos* or *users*.
- *Rooms* will be attached to current *users*.
- *Rooms* will be attached to the owner and moderator *users*.

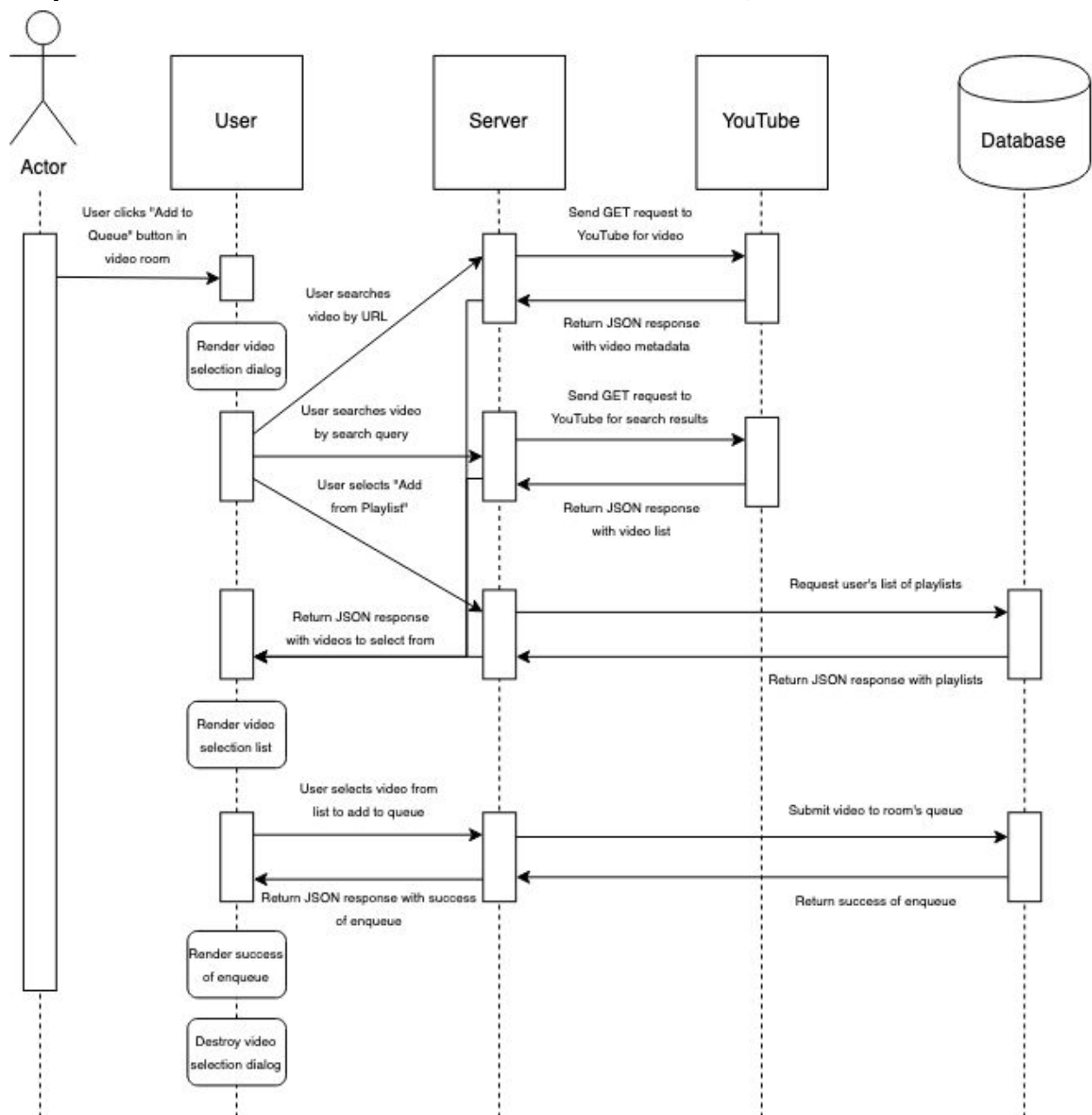
- **Sequence Diagrams**

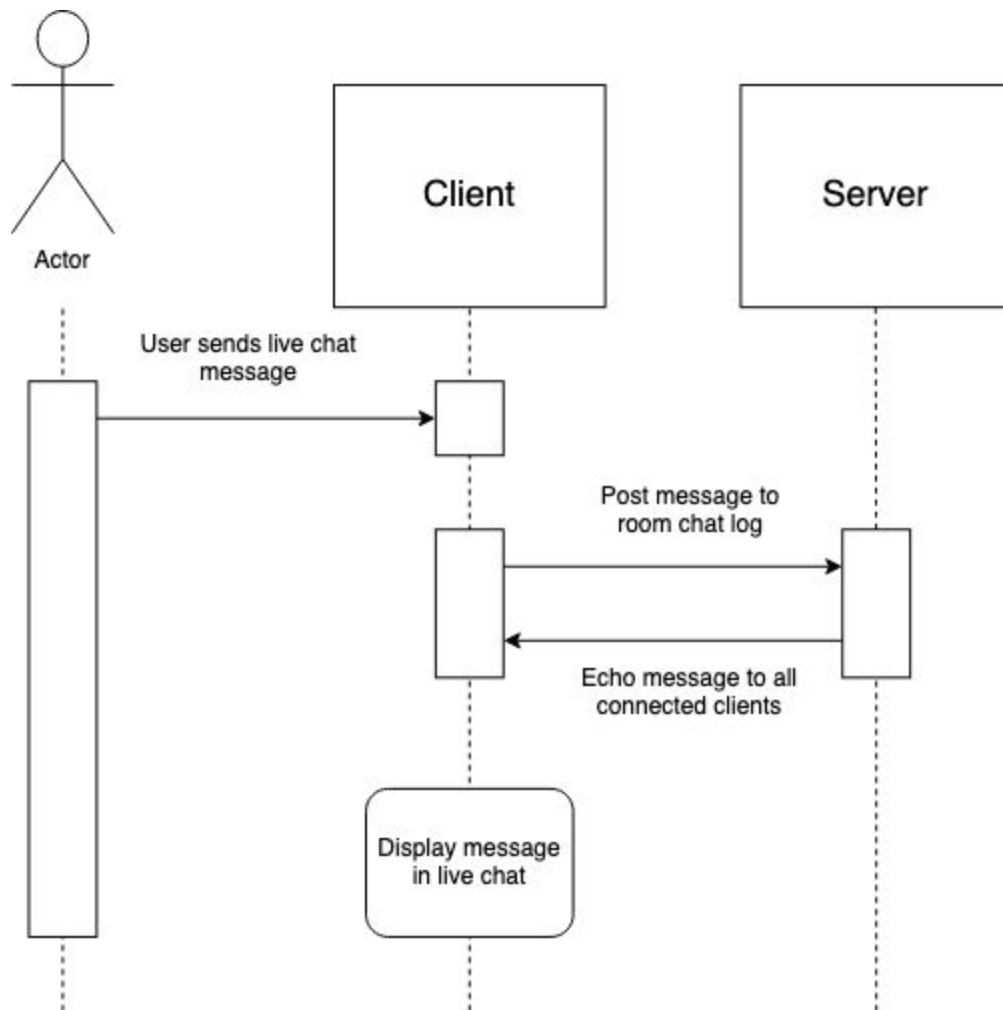
Sequence of Events When User Loads Home Page:



Sequence of Events When User Authenticates:

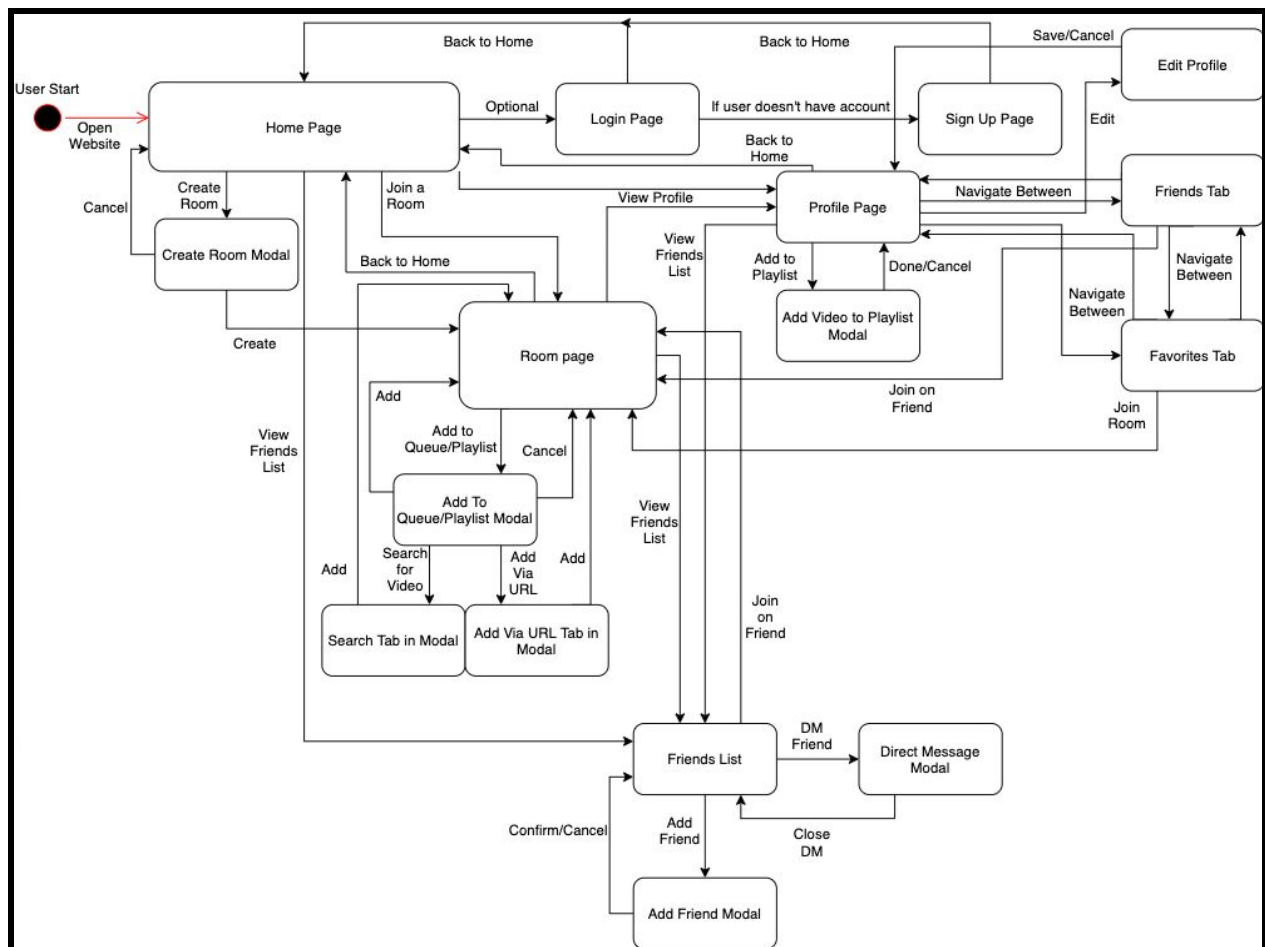
Sequence of Events When User Submits Video to Queue:



Sequence of Events When User Submits Message to Live Chat:

Navigation Map

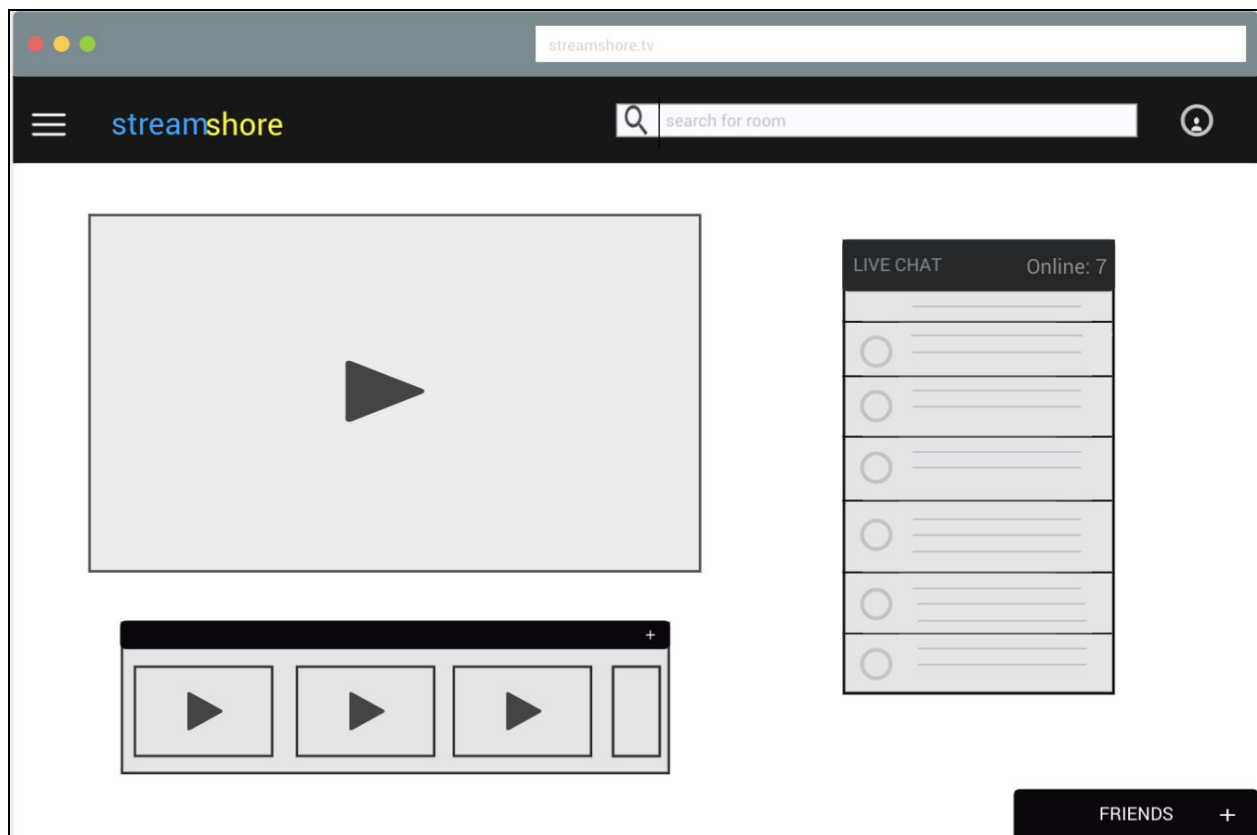
The navigation map is designed to show the flow of the application. Users are able to use the site without having to login, but they will have some limited features such as no friends list, favorited rooms, profile, etc. However, they will be able to join a room and add videos to the queue as a user with an account would. Additionally, we will have a collapsible navigation bar on the left hand side of the application that will allow users to navigate to video viewing rooms, their profile, the home page, and view any playlists that they may have created. There will also be a collapsible friends tab in the bottom right hand corner of the screen the the user can interact with friends with. The arrows show the paths to the screen or modal associated with each user interaction.



UI Mockups

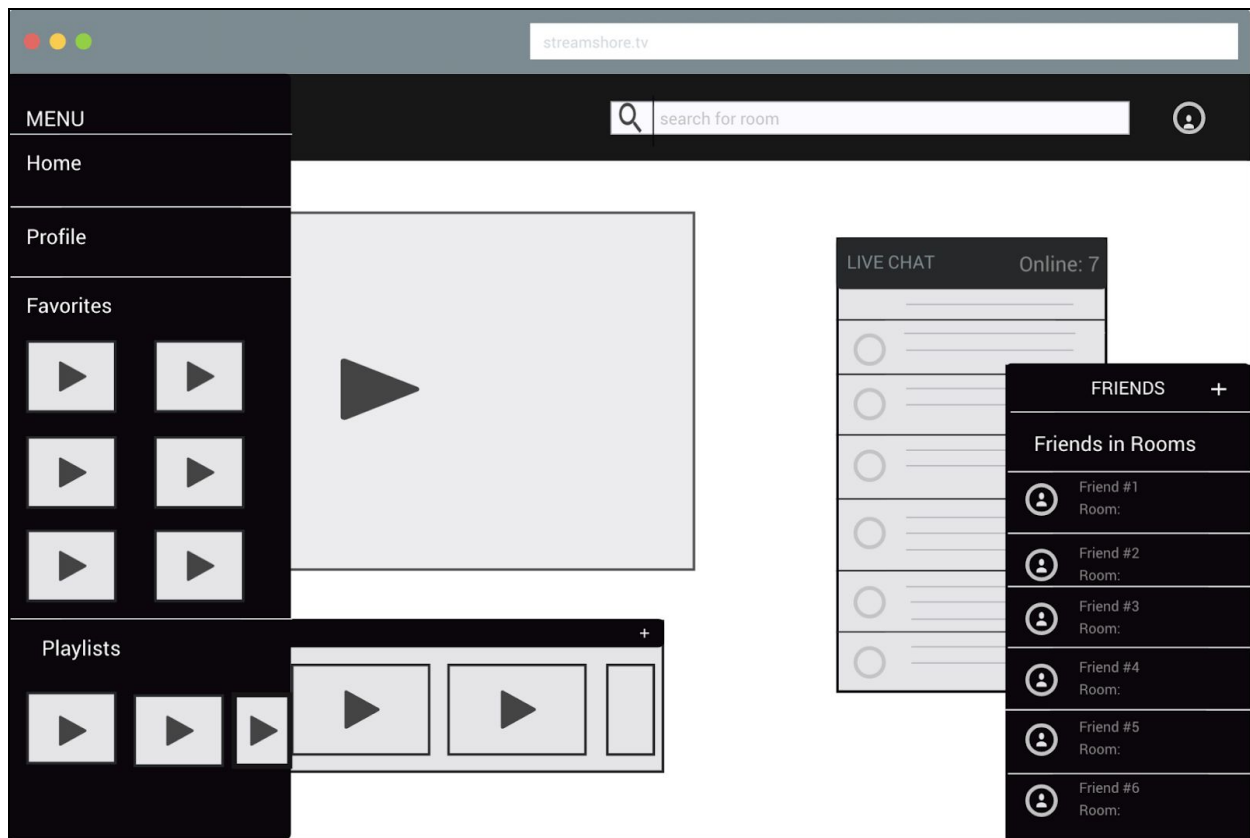
Video Player

We have designed a navigation bar at the top of each page that is always present. We have the video player for the room in the center of the screen. The live chat for the users is present to the right. The video queue is under the video player for easy access. The overall organization of the UI is clean and easy to access.



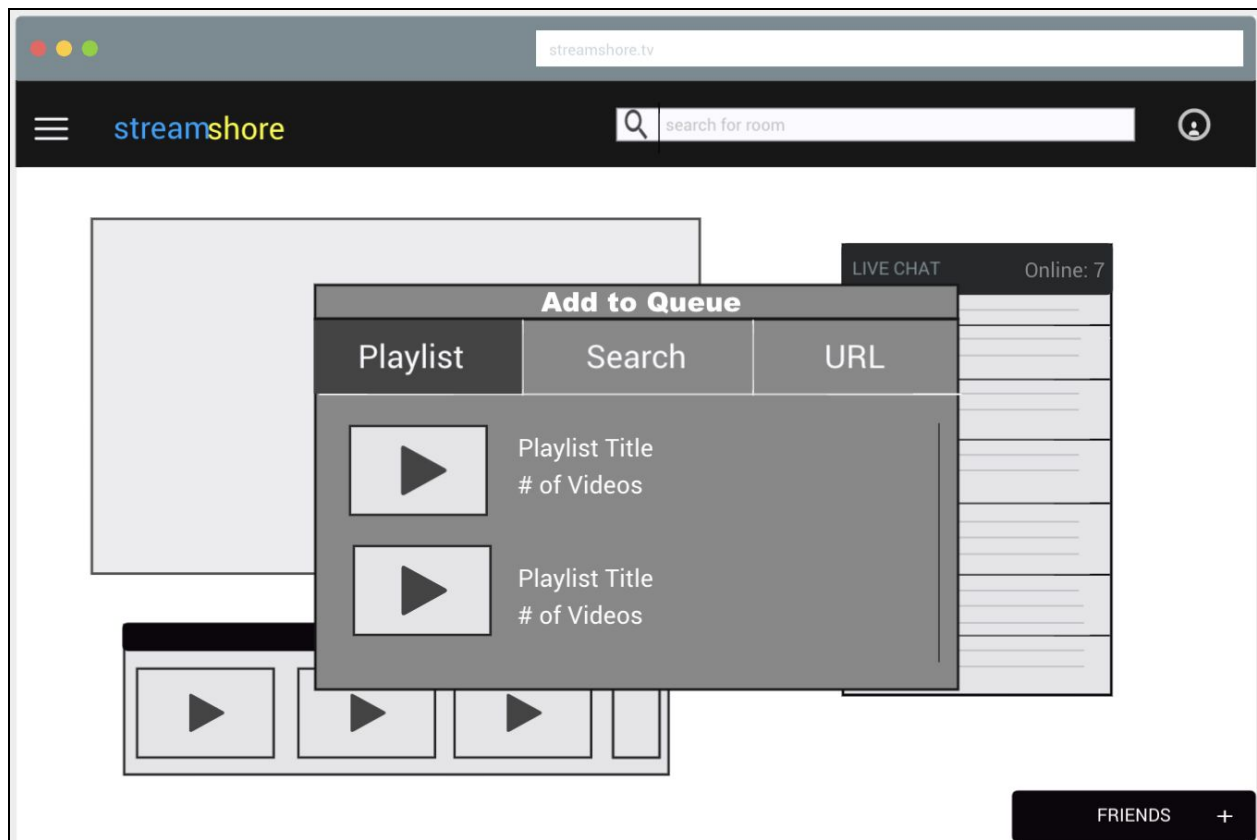
Menu Bar and Friends Tab

The menu bar has options for Home, Profile, Favorite Rooms and Playlists. This is to make it easier for the user to navigate through the important features of the website. The friends tab at the bottom of the screen is also a feature that is present throughout the website. It gives users an idea about the activity status of their friends. It is also the place where a user can add new friends.



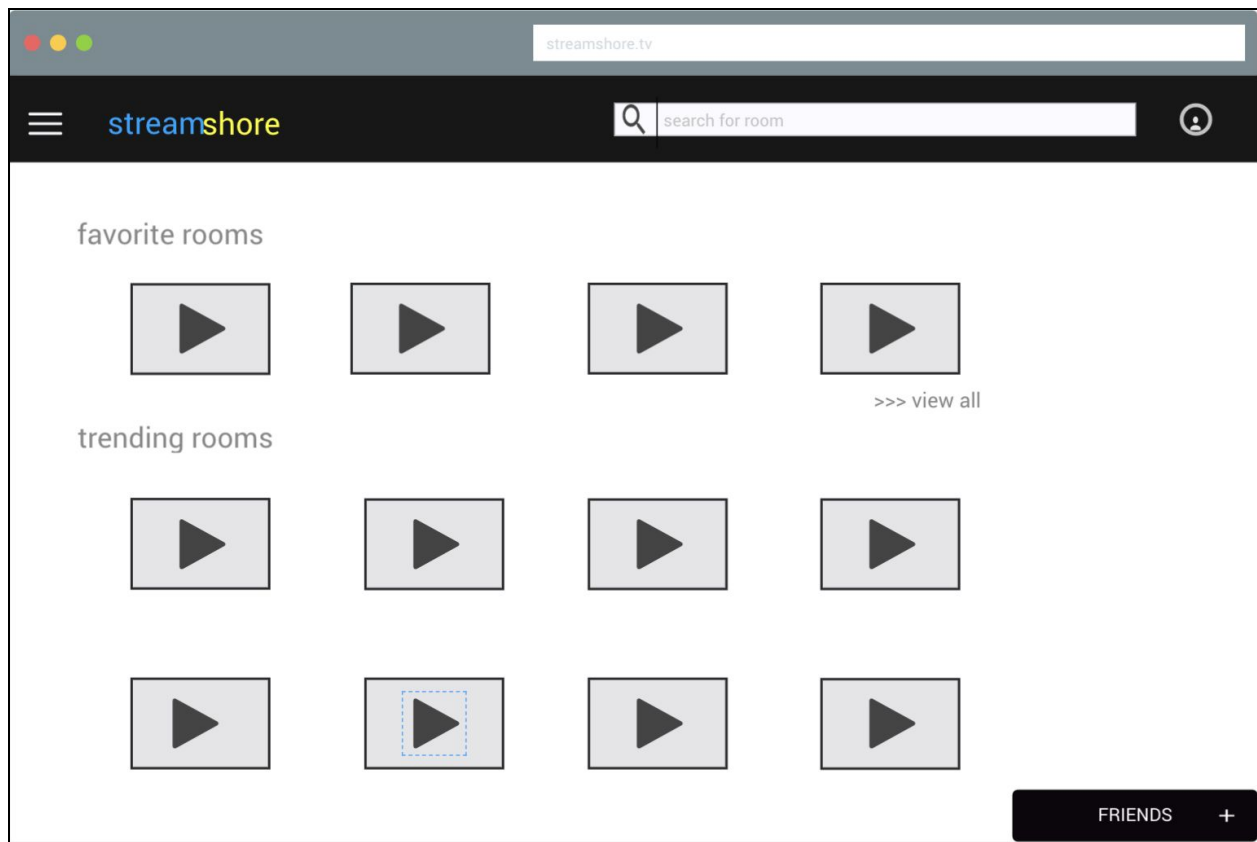
Queue Modal

This is the modal that pops up when the user clicks the plus button on the top right of the queue. The user is then able to select what video to add to the queue by choosing a video from their playlists, searching up the title of the video, or pasting the video's URL. Once the user has selected the video(s) to be queued and clicked to the side, the modal will not be displayed on the screen.



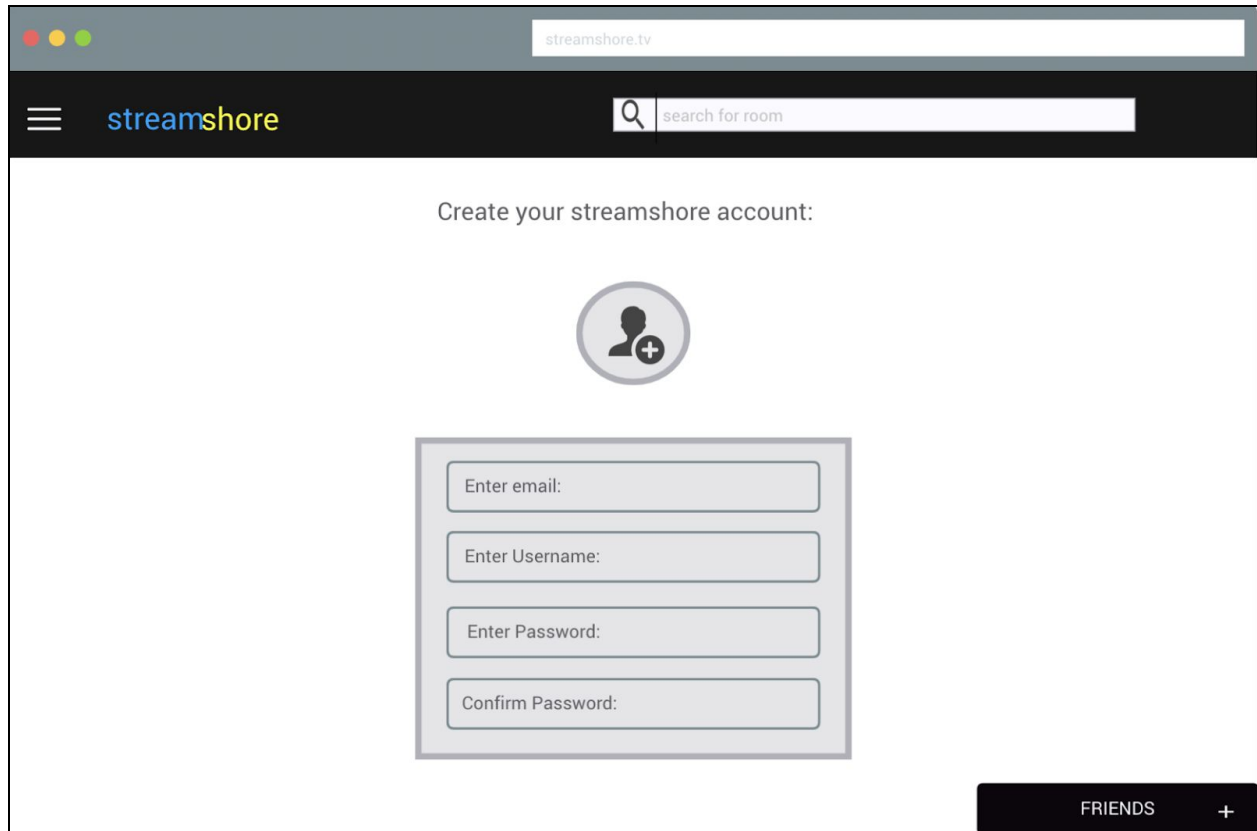
Website homepage

This is the page that is displayed when the user first comes on the site. It displays a list of the user's favorite rooms as well as the trending rooms throughout the website. The trending rooms are based off the traffic that public rooms get. The rooms are updated periodically to display the most popular rooms at any given time.



Profile Sign Up Page

This is the sign up page that is displayed to a user when they are trying to create a new account on streamshore. This page contains some mandatory information that each user needs to add to create an account.



The screenshot shows a web browser window with the address bar displaying "streamshore.tv". The page has a dark header with the "streamshore" logo on the left and a search bar on the right containing the text "search for room". The main content area is white and features the heading "Create your streamshore account:". Below the heading is a circular icon with a person silhouette and a plus sign. Underneath the icon is a light gray box containing four input fields, each with a label: "Enter email:", "Enter Username:", "Enter Password:", and "Confirm Password:". In the bottom right corner of the page, there is a dark button labeled "FRIENDS" with a plus sign next to it.

Profile

The profile page for a user has three main sections. The favorites section contains the user's favorite videos. The friends section contains a list of all the friends that the user has. The playlist section contains all the playlists that the user has created.

