Beats & Chats: Integrating Spotify and YouTube APIs with React & Firebase

Sri Shashank Katari
Master's Project in Information Technology
Computing Program, Applied Engineering and Sciences
University of New Hampshire

ABSTRACT

This paper presents the design and development of a socially interactive music player that combines modern web technologies with cloud-based services to deliver a dynamic and personalized user experience. The application uses a serverless backend model, where cloud APIs and Firebase Firestore handle all core data and playback functionality. This architecture removes the need for custom backend code, simplifying deployment while still enabling real-time interaction and dynamic content delivery.

To achieve this, the application leverages multiple tools: Spotify's Web API for fetching user-specific playlists, YouTube embeds for unrestricted playback access, and Firebase Firestore for implementing real-time chat functionality. The authentication flow was implemented using Spotify's OAuth 2.0 implicit grant, offering lightweight, token-based session handling directly within the browser. The frontend was built using React and Tailwind CSS to ensure responsive design, clean component structure, and a seamless user experience. The backend logic was kept lightweight, relying on external APIs and Firestore's cloud database to minimize infrastructure complexity. Evaluation was conducted through active usage and informal user testing. Features like Spotify login, playlist browsing, dynamic routing, and chat interactions were tested for performance and stability. The system demonstrated smooth navigation, accurate data retrieval, and real-time comment updates across sessions.

While the prototype met its functional goals, some limitations remain. Playback via YouTube lacks advanced controls or synchronization, and authentication relies solely on Spotify without a dedicated user system. Chat moderation is basic, and scalability concerns would need to be addressed in a production setting. However, the architecture lays a solid foundation for future enhancements such as group listening, expanded social features, and mobile deployment.

Keywords

Component-Based UI Design; API Integration; Event-Driven Architecture; Firebase Firestore; OAuth 2.0 Authentication; frontend development; React; Full-stack web application.

1. INTRODUCTION

This project is centered around the development of a music player web application that combines tools like React, the Spotify API, the YouTube API, and Firebase to support user-based playlist browsing, playback, and real-time discussion. The main focus of the study is on learning how these different technologies can be integrated into a functional system while gaining practical experience in full-stack development.

The music player prototype was designed to meet several feature-based goals. These included enabling users to log in with their Spotify accounts, browse their playlists, play full-length songs via YouTube, and participate in track-based discussions through real-time chat forums.

Through the development process, I aimed to improve my understanding of authentication flows, frontend frameworks, real-time NoSQL databases, and third-party API integrations. This learning-driven approach made the project valuable not just for the working prototype it produced, but also for the engineering and user experience skills it helped me develop through experimentation and iteration. I specifically chose to work with technologies I had limited experience with so I could explore them beyond surface-level usage. Instead of treating them as plug-and-play tools, I focused on learning how each one works, what tradeoffs they involve, and how they can be integrated effectively into a real-world use case such as a chat-based music player.

2. PROJECT OBJECTIVES

The overall goal of this study was to explore how different tools and services could be combined to build a web-based music player that supports playlist browsing, playback, and real-time discussion. To reach this goal, I planned a set of objectives that would allow me to apply and deepen my knowledge in web development, API usage, and user interaction design.

2.1 Understanding Third-Party APIs

The first objective was to learn how to work with third-party APIs to retrieve user-specific data. In particular, I wanted to understand how a music service API like Spotify could be used to access playlists and song information based on a user's login [1]. This objective focused on learning how to handle authorization flows and query external services using real user data.

2.2 Solving Playback Limitations

The second objective was to investigate how to offer music playback within the application, especially in situations where the primary data provider (Spotify) has usage restrictions. I planned to explore alternative solutions for playback using publicly accessible platforms [2]. This helped build critical thinking around evaluating fallback options and adapting application logic to changing constraints.

2.3 Creating Functional and Interactive Web Experiences

This objective focused on gaining hands-on experience with building a fully functional web application by integrating both frontend and backend concepts. I wanted to strengthen my ability to structure a frontend using reusable components and manage views using routing — concepts I explored through the use of React's component model and navigation patterns. I also aimed to implement UI logic with clean state management practices, using hooks and prop-based communication to support a dynamic user experience.

On the backend side, I worked with API integrations to retrieve user-specific data, which helped reinforce my understanding of how client-server communication works in full-stack systems. While I didn't build a custom backend, I had to simulate backend behavior in the frontend, particularly for token handling, error catching, and conditional rendering.

Additionally, I wanted to learn modern styling approaches by using utility-based design systems like Tailwind CSS and understand how layout, responsiveness, and UI consistency can be achieved without relying on heavy component libraries.

2.3.1 Enhancing Social Interaction

A key component of the application was designing features that enabled user interaction beyond passive playback. This included creating a commenting system where users could engage with one another under specific songs. I learned how real-time updates, content threading, and dynamic UI rendering contribute to a more interactive application.

2.3.2 Supporting Multiple Users

Another important learning goal was to make the application adaptable to different users. I wanted each user to experience the app with their own playlists, preferences, and comment history. This helped reinforce how user-specific sessions are managed and how to build conditional logic that dynamically responds to the logged-in user's context.

2.4 Database Design and Real-Time Data Handling

Finally, I focused on how to design and manage cloud-based NoSQL databases in a frontend-oriented application. I studied how Firestore enables hierarchical data structures like subcollections, supports real-time updates via listeners, and handles access control and dynamic queries without requiring a custom backend. This objective gave me insights into event-driven architectures and how scalable data models are built in client-managed apps.

These objectives were meant to give structure to the development process and guide my learning throughout the study. Each one focused on a specific area of growth, from understanding APIs to thinking about interaction design and user experience.

3. APPROACH, METHODS AND TOOLS

Throughout this project, I approached each stage of development as a way to study how different technologies work together to support a real-world application. Rather than starting with tools I was already comfortable with, I intentionally chose technologies I had limited experience with so I could learn something new through practice. My goal was to not just build a working music player, but to understand how authentication, frontend development, API integration, and real-time interaction come together in a single system. Each decision—what APIs to use, what libraries to adopt, and how to structure the user experience was guided by both the technical requirements of the project and my intention to grow as a developer.

3.1 Authentication and API Integration

3.1.1 Spotify OAuth and Token Management

A major part of the study involved user authentication and data access via the Spotify API. I studied the OAuth 2.0 implicit grant flow, which is often used in frontend applications that don't have a secure backend server to manage tokens [17]. I chose this method because it allowed for quick and direct token exchange through the browser. During implementation, I learned how to extract access tokens from the URL, store them in the browser's local storage, and use them to authorize further API requests. Working with Spotify's OAuth flow helped me understand how web applications handle login without storing user credentials. I learned how access tokens are passed through the URL, how to extract them securely on the frontend, and how to store them in the browser for later use in API calls. Instead of just following a typical implementation pattern, I came to understand the login system well enough to adjust it, handle edge cases like token expiration, and adapt it for multiple users. Additionally, I learned how to work with RESTful endpoints by studying Spotify's structured API design, including how different HTTP methods like GET and POST are used for resource access and modification. I gained experience interpreting JSON responses and understanding how data is organized and nested in third-party APIs. Working with endpoints taught me how to structure dynamic requests, handle rate limits, and manage authorization headers effectively in a real-world frontend integration.

3.1.2 YouTube Data API for Full Playback

To enable full music playback, I integrated the YouTube Data API v3. This was necessary because the Spotify API does not offer full-track playback unless the user has a Premium account, and in many cases, even 30-second previews were missing. I chose YouTube because of its wide music availability and its developer-friendly tools for searching and embedding videos [2][8]. I learned how to construct dynamic search queries based on track names and artist metadata, and how to filter and embed results using safe, responsive iframe containers. This also gave me experience working with search-based APIs, handling video embedding on the frontend, and managing fallback scenarios where one service doesn't meet the requirements of the application. Using YouTube allowed me to maintain a consistent playback experience for all users, regardless of their Spotify subscription status.

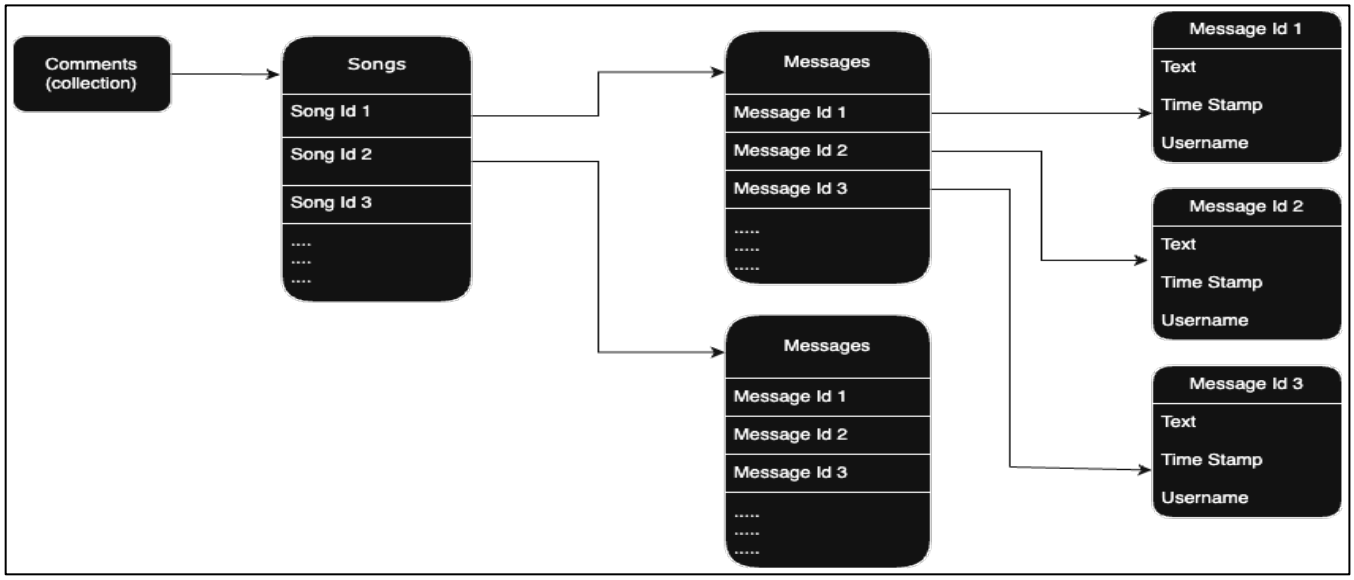

Figure 1 Firebase Firestore Chat Schema

3.2 Frontend Development

3.2.1 React for Component-Based Architecture

For frontend development, I chose React, an open-source JavaScript library developed by Meta [5], because of its component-based structure and declarative programming model. I considered other frameworks like Angular and Vue, but React's approach to state management and reactivity felt more intuitive and aligned better with the kind of UI updates I needed for this project. At the start, my understanding of React was basic, so I used this project as an opportunity to study it more deeply and apply it in a meaningful way. I worked with React's hooks such as useState and useEffect [16] to control how the interface responds to user actions, API data, and login sessions. I also explored React Router to manage navigation between views like the home screen, playlist library, and track details, while still keeping the experience within a single-page application [12]. Passing props between components and deciding when to lift or isolate state taught me to think more critically about how data should flow across the app. Working with React helped me focus on how to structure components properly, manage shared and local state, and organize the interface in a way that made the code easier to maintain and update.

3.2.2 Tailwind CSS for Interface Design

For styling the application, I chose Tailwind CSS, a utility-first framework [4] that encourages building designs directly in the markup using small, reusable utility classes. At first, I considered more traditional options like writing custom CSS or using Bootstrap. However, I found that Tailwind offered a more consistent and scalable approach for a project that required a lot of layout flexibility and visual experimentation. Using Tailwind pushed me to think more intentionally about spacing, typography, and alignment because I had to apply each design rule explicitly through class names. This not only made the layout more readable and easier to tweak, but also reduced the need to maintain separate CSS files. I was able to quickly build reusable layout patterns for playlist cards, chat boxes, and buttons without having to name and organize a large number of custom styles. It also helped me avoid some common CSS problems like naming conflicts and unpredictable cascading. By using Tailwind throughout the project, I gained confidence in designing interfaces that are both clean and responsive, and I developed a better understanding of modern frontend design principles.

3.3 Real-Time Interactivity

3.3.1 Firebase Firestore for Comments

For the real-time comment feature, I used Firebase Firestore, a cloud-based NoSQL database service offered by Google [3]. I chose Firestore because it supports real-time data synchronization out of the box, which made it a good fit for building per-song chat threads. Compared to traditional request-based databases, Firestore's ability to listen for updates using onSnapshot [13] helped me understand how event-driven systems work. I learned how to model data using subcollections, storing each song's comments under its unique ID and how to structure queries for efficient data loading [18] and updates. Figure 1 illustrates the structure of the Firestore database, how each song's comments are stored under the comments/songs/messages subcollection. This experience also taught me how to handle asynchronous operations, manage user-generated content, and build a lightweight system that supports multiple users without complex backend logic

3.4 Code Organization and Development Workflow

3.4.1 Project Structure and Deployment Readiness

In organizing the codebase, I followed a clear separation of logic: pages were used for route-level views, components for reusable UI sections, and utils for managing API requests and helper functions. This structure helped keep the code clean and modular, which made it easier to debug, test, and extend as new features were added. The project followed a modular file structure that separated views, components, and utility functions for clarity and scalability (Figure 2). This structure supported a clean development process and made the application easier to extend. Developing and testing the application locally gave me full control over the setup and allowed me to iterate without external deployment delays. Once the review process is complete, I plan to deploy the application using Firebase Hosting or a platform like Vercel [14] to make it publicly accessible and easy to maintain.

```
// getUserPlaylists in spotify.js
export const getUserPlaylists = (token) => {
  return
axios.get('https://api.spotify.com/v1/me/playlists', {
    headers: { Authorization: `Bearer ${token}` },
  });
};
// getTracksByPlaylistId in spotify.js
export const getTracksByPlaylistId = (token,
                                      playlistId) => {
  return
axios.get(`https://api.spotify.com/v1/playlists/${play
          listId}/tracks`, {
    headers: { Authorization: `Bearer ${token}` },
  });
};
```

Listing 2 Spotify API calls for playlists and track retrieval

4. RESULTS AND ARTIFACTS

The project resulted in a working web-based music player that supports login with a Spotify account, playlist browsing, full-track playback through YouTube, and real-time song-specific chat functionality. Each major feature aligned with the study's objectives and served as evidence of what was learned and implemented throughout the development process.

After logging in, users are able to view their personal Spotify playlists, select any playlist, and browse the songs within it. The application uses the Spotify Web API to retrieve up-to-date playlist and track data based on the currently logged-in user.

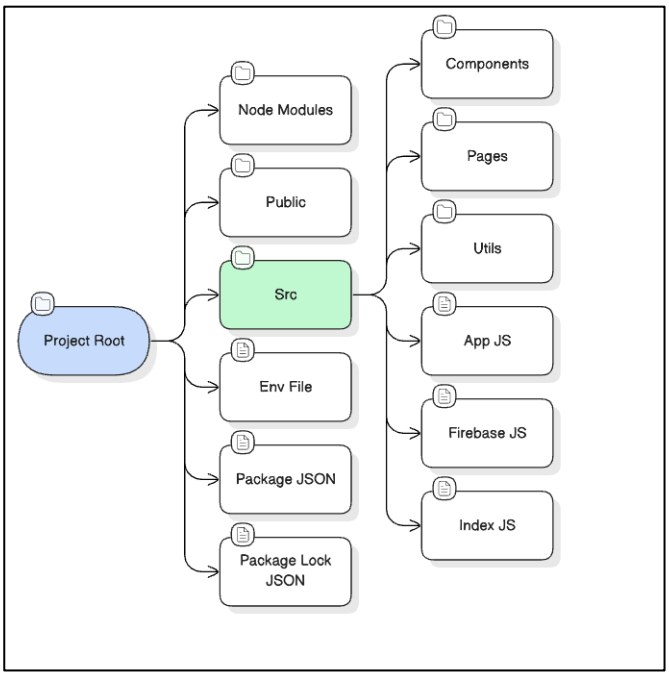

Figure 2 Project folder and file structure

This enables a dynamic and personalized experience instead of relying on a static or hardcoded interface. The real-time nature of this integration ensures that any playlist changes made in Spotify are reflected immediately in the application interface.

This functionality is powered by the API logic shown in Listing 1, where playlist and track details are fetched using the Spotify access token [11]. A combination of Spotify and YouTube APIs enabled user-specific playlist retrieval, metadata extraction, and playback mapping. These APIs ensured a dynamic, personalized experience. For detailed endpoint usage, refer to Table 1.

Playback was implemented using the YouTube Data API. Each track in a playlist is matched to a playable YouTube video, which is embedded directly in the interface. This solution ensured that most songs are available to all users, regardless of their Spotify subscription level. The YouTube player opens in a clean, fixed-position container on the screen and includes a close button for easy control.

Table 1 API endpoints description

Endpoint	Method	Source API	Functionality
`/v1/me/playlists`	GET	Spotify API	Fetches current user's playlists
`/v1/playlists/{playlist_id}`	GET	Spotify API	Retrieves track list of a selected playlist
`YouTube Search API`	GET	YouTube API	Finds full-track playable video

Another core feature of the system is the real-time comment interface, which allows users to engage in threaded discussions under each song. When a user submits a comment, it is stored in Firebase Firestore under a collection keyed by the song's Spotify ID. Each comment is saved within a comments subcollection and includes relevant metadata such as the user's display name, timestamp, and message content. Firestore's real-time listeners ensure that newly posted messages appear instantly for all users viewing the same track, supporting dynamic conversation without the need for manual page reloads.

Listing 2 demonstrates the function used to add these comments. It dynamically initializes the comment thread structure if it doesn't already exist and attaches server-generated timestamps to ensure consistency. The interface also includes a "Show more comments" toggle when threads grow beyond two messages, keeping the UI focused and manageable.

In addition to basic commenting, users can interact further by liking messages with a thumbs-up icon, replying to specific comments (which are displayed in a threaded format), and reporting inappropriate content. Reported comments are stored in a separate reports collection within Firestore for moderation. All these features are implemented using Firestore subcollections and conditional document structures, which keeps the system efficient and scalable. This flexible serverless backend design supports rich, real-time social features without the need for complex infrastructure.

```
// src/components/ChatBox.jsx
const sendMessage = async () => {
  if (!message.trim()) return;
  await addDoc(collection(db, 'comments', songId,
               'messages'), {
    text: message,
    user: currentUserName,
    userId: localStorage.getItem(
            'spotify_display_name') || 'Anonymous',
    timestamp: serverTimestamp(),
    parentMessageId: replyingTo ? replyingTo.id: null,
    likes: []
  });

  setMessage('');
  setReplyingTo(null);
};
```

Listing 2 Firebase Comment submission functionality

The application's user interface was built using React, leveraging its component-based architecture to build reusable, isolated modules that can manage their own state and lifecycle. Styling was achieved through Tailwind CSS, a utility-first framework that allowed for rapid and consistent design implementation across the app without writing custom CSS files. The layout was crafted to be responsive and mobile-friendly, using Tailwind's grid, flexbox, and spacing utilities.

The React components heavily utilize hooks such as useState for local state management and useEffect for handling side effects, like API calls and real-time listeners. For example, the ChatBox component uses useEffect to subscribe to Firebase Firestore updates, ensuring that new comments appear in real time without page refreshes. Conditional rendering and controlled inputs are used throughout to ensure seamless user interactions, such as displaying input fields for replies, toggling visibility of comment threads, and reacting to like/report actions.

The project follows a modular folder structure that separates concerns across components, views, and utilities. The components directory contains key feature-specific elements such as ChatBox.jsx, which manages real-time threaded comments, reply logic, and moderation actions like likes and reports using Firestore; Library.jsx, responsible for displaying user-specific Spotify playlists; and Player.jsx, which handles YouTube video embedding tied to individual songs. The pages directory organizes top-level views including Home.jsx, the animated landing screen with navigation, and PlaylistDetails.jsx, which displays track listings and associated YouTube players for the selected playlist. Utility functions are housed in the utils directory, with auth.js managing OAuth token extraction and local storage, spotify.js containing Axios-based API logic for playlist and track data, and youtube.js responsible for crafting search queries and rendering embedded playback. This structure promotes maintainability, clarity, and scalable component reuse throughout the application.

React Router handles client-side navigation, allowing seamless transitions between routes like Home and Library without full-page reloads. Local storage is used to preserve user session data like Spotify tokens and usernames across refreshes.

Overall, the frontend architecture balances clarity, scalability, and interactivity. It not only meets the functional goals of the project but also adheres to modern web development standards, making it well-suited for future enhancements and long-term maintainability.

5. DISCUSSION AND EVALUATION

5.1 Evaluation through Usage

This prototype was evaluated primarily through active usage and testing by the developer and test users. Spotify login, playlist browsing, YouTube playback, and per-song chat features were all tested for functionality and stability. The interface performed reliably, with smooth page transitions and consistent real-time updates via Firebase.

Instead of formal unit testing, the evaluation emphasized user experience and responsiveness. The real-time comment threads, dynamic playlist rendering, and embedded YouTube videos were tested in multiple sessions using different user accounts. Overall, the system delivered on its goal of creating an interactive music experience with minimal friction.

5.2 Limitations of the Current Prototype

While the prototype succeeded in integrating several modern web technologies, it has known limitations that were not addressed due to time and resource constraints:

- Playback Control: Embedding YouTube videos offers basic playback but lacks precise control, syncing across users, or playlist queuing.

- Authentication Constraints: The OAuth 2.0 implicit grant flow was suitable for this project but lacks long-term support for refreshing tokens or secure backend storage.
- No Native User System: The app does not implement its own user login. All identification is based on Spotify display names, which limits moderation and personalization.
- Chat Moderation: Comment reporting is implemented, but there are no features like user blocking, profanity filters, or admin dashboards.
- Scalability Considerations: The current Firestore structure supports moderate use, but a production-scale system would need indexing, rate-limiting, and quota monitoring.

5.3 Possibilities for Future Enhancement

The current structure provides a strong foundation for continued development beyond the academic setting. If expanded, the app could support:

- Real-time Group Listening: Add synchronized playback so multiple users can listen together, like a "music party" session.
- Expanded Chat Features: Include tagged mentions, emoji reactions, and notification systems to drive conversation.
- User Accounts and Friend Lists: Implement a dedicated user system with profile pages and friend-based content filtering.
- Admin & Moderation Tools: Introduce dashboards to manage reported comments, users, or song discussions.
- Mobile Responsiveness and Deployment: Complete responsive design and host on Firebase or Vercel for wider reach.

Together, these improvements would move the app from a personal study prototype to a potential social platform for collaborative music exploration.

6. REFLECTION AND FORWARD PATH

This project helped me move beyond classroom exercises into a more realistic and iterative development process. I gained practical, hands-on experience with full-stack web technologies including API integration, authentication, and real-time databases. React's component architecture and state logic taught me scalable frontend design principles, while Firebase's event-driven data model introduced me to the challenges and advantages of real-time systems. Authentication via OAuth flows, playlist personalization, and real-time user interactions all contributed to a well-rounded technical foundation of concepts I had previously only encountered in theory. The real-time chat feature, Spotify authentication, and playlist integration pushed me to explore cloud-based data models, authentication protocols, and responsive UI design. Through this project, I not only strengthened my technical skills but also developed a greater appreciation for user experience and iterative problem-solving.

Much like project-based learning in advanced software development courses, this hands-on approach allowed me to learn by deepening my understanding through direct engagement rather than passive observation. Overall, this prototype demonstrated the feasibility and impact of combining interactive media with social features, and it lays a strong foundation for future enhancements and continued learning in full-stack web development.

7. REFERENCES

[1] Spotify for Developers. (2024). Spotify Web API. Retrieved from https://developer.spotify.com/documentation/eb-api/

[2] Google Developers. (2023). YouTube Data API v3. Retrieved from https://developers.google.com/youtube/v3

[3] Firebase. (2024). Cloud Firestore Documentation. Retrieved from https://firebase.google.com/docs/firestore

[4] Tailwind CSS. (2024). Tailwind CSS Documentation. Retrieved from https://tailwindcss.com/docs

[5] Meta (React Team). (2024). React – A JavaScript library for building user interfaces. Retrieved from https://reactjs.org/

[7] Stack Overflow Community. (2023). Handling OAuth token flow in React applications. Retrieved from https://stackoverflow.com/questions/tagged/oauth+reactjs

[8] W3C. (2011). Embedding external content (iframes). Retrieved from https://www.w3.org/TR/html5/the-iframe-element.html

[10] OpenJS Foundation. (2023). JavaScript Event Loop Explained. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

[11] Axios. (2023). Axios – Promise-based HTTP client for the browser and Node.js. Retrieved from https://axios-http.com/docs/intro

[12] React Router. (2024). Declarative Routing for React Apps. Retrieved from https://reactrouter.com/en/main/start/tutorial

[13] Firebase. (2023). Realtime Updates with onSnapshot. Retrieved from https://firebase.google.com/docs/firestore/query-data/listen

[15] Meta (React Team). (2024). React useState and useEffect Hooks. Retrieved from https://react.dev/learn/state-a-component

[16] Google Developers. (2023). Understanding OAuth 2.0 for Web Applications. Retrieved from https://developers.google.com/identity/protocols/oauth2

[17] Firebase. (2024). Security Rules for Firestore. Retrieved from https://firebase.google.com/docs/firestore/security/get-started

[18] DigitalOcean. (n.d.). An Introduction to Server-Side Rendering with React. DigitalOcean Community Tutorials. Retrieved May 8, 2025, from https://www.digitalocean.com/community/tutorials/react-server-side-rendering