Multiplayer Trivia Project
CS343 – Group 17

1. Group Members & Roles

SU Number	Name & Surname	Role	Primary Responsibilities
26978725	Ashton Weir	WebScraper, Frontend	WebScraper, frontend pages, documentation.
26947110	Gregory Gebers	Backend, Database	Database, authorisation, backend.
27205762	Katja Wood	Frontend, Documentation	UI and Report.
26989395	Andrew Cottrell	Backend, Database	Database, backend, gameplay loop.
27251489	Benjamin Conolly	Frontend	Frontend pages, documentation.

2. Introduction

2.1 Project Overview

The main objective of this project is to develop a **real-time**, **multiplayer trivia web application** that delivers an engaging competitive experience. Authenticated users can **host or join matches**, choosing categories and difficulty levels. Each match has **4 rounds** of **7 questions** from categories like General Knowledge, Science, Entertainment, Politics, Geography, and Sports. Matches support **multiple players** with real-time scoring, countdown timers, and post-match summaries. Users can **register, customize profiles, view match history, and track progress** via daily and weekly leaderboards.

Administrators can manage users and questions, including editing, deleting, filtering, and removing players from active matches. All questions are collected with a custom web scraper and stored in a Supabase database to ensure reliability and data integrity.

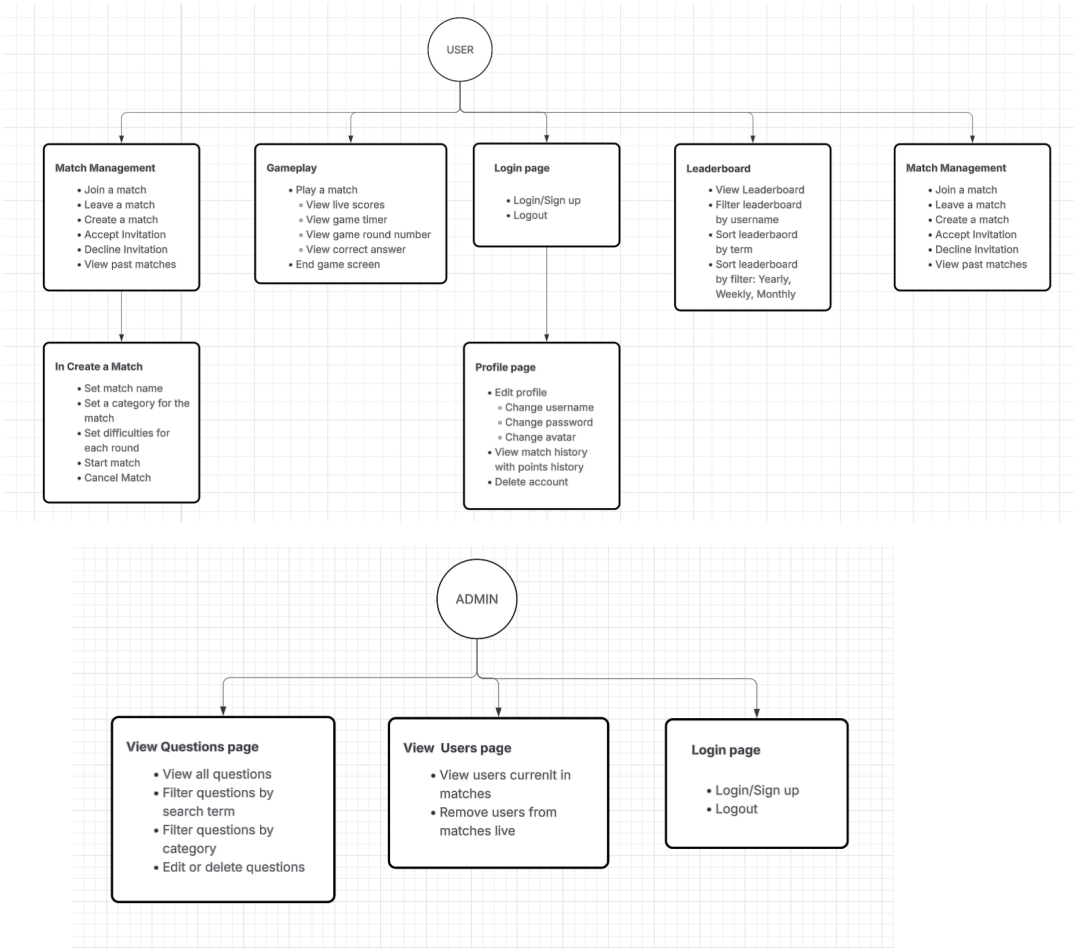
2.2 Motivation

The **motivation** behind the **technical stack** chosen and the **game design** was driven by a combination of project requirements, group skill constraints, and ease of development.

- **Frontend:** React.js with CSS provides **component-based architecture**.
- **Backend:** Node.js with Express.js enables straightforward **server-side logic**, simplifying communication with the database.
- **Database:** Supabase was selected for its **ACID compliance**, supporting reliable storage and relational data modeling for users, matches, questions, and scores.
- **Authentication:** Token-based authentication ensures safe user sessions.
- **Real-time Communication:** Socket.IO handles live game state synchronization, score updates, and countdown timers, which are key for multiplayer responsiveness.

3. Use Case & User Stories

3.1 Use Case Diagram



3.2 User Stories

As a **player**, I want to:

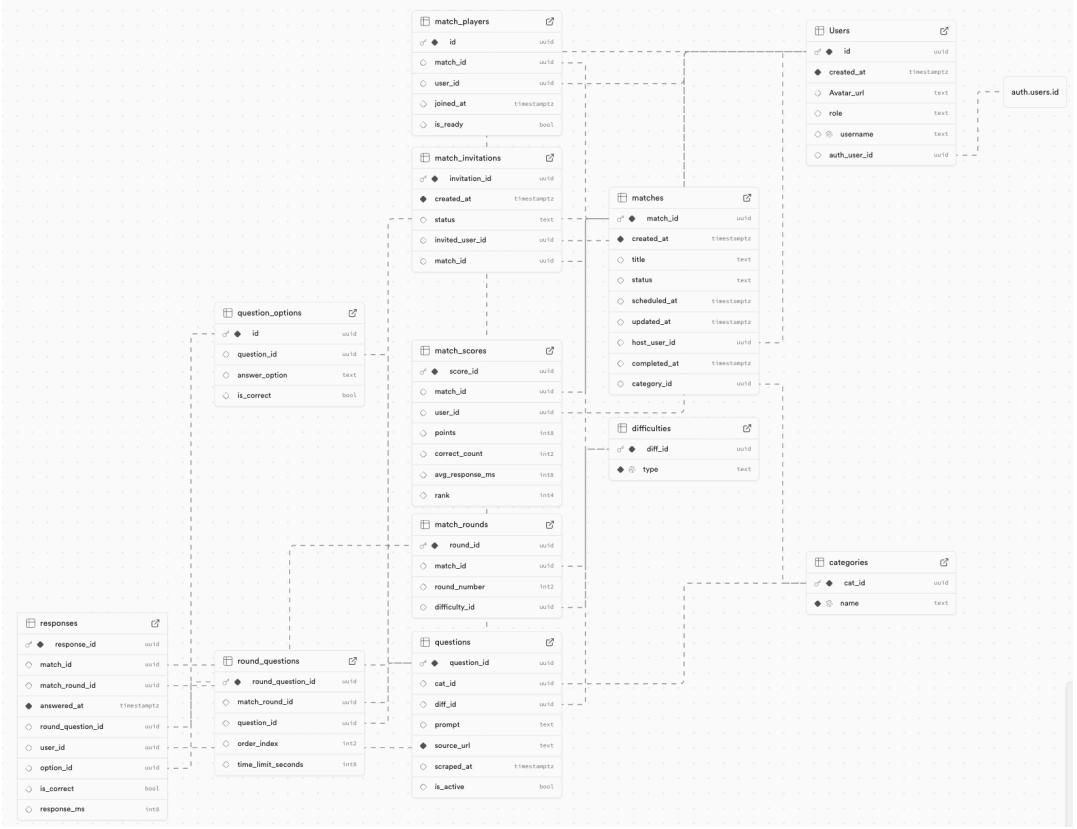
- Register, create a profile with a username, email and password, and edit my username, password, and avatar.
- Create trivia games, select categories and difficulty levels, and invite other players.
- Join invited games and accept or reject match invites.
- View leaderboards with ascending/descending sorting.
- Access match history and highscores, filter by category, and search past matches.

As an **admin**, I want to:

- Manage users in current games and be able to remove users from matches.
- See all questions in the question bank.
- Sort the questions by a search term or filter by category.
- Edit or delete questions in the question bank.

4. Data Modelling

4.1 Entity-Relationship (ER) Diagram



4.2 Database Schema

The database consists of the following key tables, demonstrating a normalized structure:

- **`Users`**: Stores user profile data, linked to Supabase Auth.
- **`Categories`** & **`Difficulties`**: Lookup tables for question categorization.
- **`Matches`**: Core table for each game session, linked to a host and category.
- **`Match_Rounds`**: Defines each round within a match and its difficulty.
- **`Questions`** & **`Question_Options`**: Store the trivia questions and their multiple-choice answers.
- **`Round_Questions`**: Links specific questions to a match round, defining the game's structure.
- **`Match_Players`** & **`Match_Invitations`**: Manage player membership and invitations.
- **`Responses`**: Records each player's answer to each question, including correctness and response time.
- **`Match_Scores`**: Aggregates the final score and stats for each player in a match.

4.3 Normalisation to Third Normal Form (3NF)

The database schema adheres to **Third Normal Form Normalisation** (**3NF**) to minimise redundancy as well as avoid update anomalies. The normalisation process follows the following stages:

- **1NF** (**First Normal Form**): All tables have **atomic column values** and **unique** rows, achieved by using **primary keys**. Therefore there are no repeating groups or arrays.
- **2NF** (**Second Normal Form**): All non-key columns are fully **functionally dependent** on the entire primary key. This can be seen in **`Match_Scores`**, where points depend on both **`match_id`** and **`user_id`**.
- **3NF** (**Third Normal Form**): There are **no transitive dependencies**. Non-key columns depend only on the primary key. A key example of 3NF compliance is the **`Responses`** table. Storing answer details directly in **`Round_Questions`** would have introduced transitive dependencies, since those details depend on the specific **`response_id`** and not on the round question itself.

5. Authentication

We used a **token-based authentication** system integrated with **Supabase Auth** to manage both the player and admin users access.

5.1 Authentication flow:

- **User Registration**
  - A new player creates an account with a username, email, and password.
  - Passwords are hashed using **bcrypt** and managed by **Supabase Auth**.
  - After a user registers, a new entry is added to the **`Users`** table linking to Supabase through the **`auth_user_id`** field.

- **User Login**
  - Users provide their email and password.
  - **Supabase Auth** verifies the credentials by comparing the hashed password stored in the system with the hash of the entered password.
  - When a user logs in, Supabase Auth creates a session, and our backend takes that session and sets the following two HTTP-only cookies on the logged in user's browser to allow for reloads.
    - **Access Token** (`sb_at`): Short-lived JWT for API requests.
    - **Refresh Token** (`sb_rt`): Long-lived token used for refreshing the aforementioned access token.
  - **Admin verification** is done with the **`role`** field in the **`Users`** table which determines whether the user has admin privileges.

- **Authenticated requests**
  - The client includes the access token in the **Authorization header** in all API requests, the server validates the token and **grants access** to protected endpoints based on the user's role and associated permissions.

5.2 Token Scheme

- The **access token** (`sb_at`) mentioned above, contains the user ID, email, role, and expiration and is short-lived. The **refresh token** (`sb_rt`) on the other hand is long-lived, and used to obtain a new access token without loggin in again.
- When the access token **expires**, the client sends a refresh token to the server to get a new access token.
- This provides seamless authentication and smooth user experience without the constant need to reenter login details.

5.3 Password Security

- Passwords are hashed with **bcrypt** by **Supabase Auth**, therefore meaning that plaintext passwords are never stored.
- During login, input passwords are hashed and compared to the stored hash and access is only granted if they match.

6. API design

6.1 REST API Principles

In this project, we developed our backend API with Node.js and Express to manage users, matches, and questions and structured it according to the REST API principles (Pautasso et al., 2016), which provide guidelines for scalable, maintainable, and predictable web services. These principles are evident in our code with the following aspects:

- **Uniform Interface:** Each resource (e.g., user, match, question) is uniquely identified by a URL and manipulated via standard HTTP methods (GET, POST, PUT, DELETE).
- **Client-Server Separation:** The client handles the user interface and interactions, while the server manages data access, security, and business logic.
- **Statelessness:** Each request contains all information needed; the server does not store session state between requests.
- **Caching:** Resources can be cached unless explicitly marked otherwise.
- **Layered Architecture:** The design supports multiple layers, enabling scalability and separation of concerns.

In addition to this the following endpoints were implemented. These endpoints act as entry points for the frontend to communicate with the backend, ensuring all interactions between users and the system are structured and secure:

- `/auth` – Registration, login, token refresh
- `/users` – Fetch or update profiles, view match history
- `/matches` – Create, join, leave matches; fetch match details
- `/questions` – Retrieve, edit, or delete questions

6.2 WebSockets

The application uses Socket.IO to enable real-time, two-way communication between the server and connected clients. This supports the live game mechanics such as:

- **Scores & Timers**: Updated instantly for all players.
- **Player Notifications**: Join/leave events and match readiness broadcasts.

6.3 Integration

REST API principles handle initial data loading and administrative tasks while **WebSockets** take care of the real-time interactions like questions, answers, and scores. This hybrid approach ensures scalable, maintainable data management and a smooth multiplayer experience.

7. Design Patterns & Application Architecture

The project follows a **client-server architecture** with a clear separation between the frontend and backend. The **frontend**, built with React, provides the user interface, while the **backend**, developed with Node.js and Express, manages game logic, data handling, and real-time communication. This separation enhances modularity, and maintainability.

7.1 Client (Frontend Architecture)

The frontend is built with React using a component-based architecture and functional components with hooks. The **structure** can be described as follows:

- **Pages**: Major routes such as **`Home`**, **`Match`**, **`Leaderboard`**, and **`Admin`**.
- **UI Components**: Reusable elements like buttons, modals, and forms.
- **Containers**: Handle feature-specific logic and API calls, such as **`MatchContainer`** which contains all the elements that manage joining and starting games.

While **state management** is defined through the following:

- **Local State**: For UI-level interactions such as modals and form inputs.
- **Global State** (Context API / Redux): Maintains shared data like authentication status and match details.
- **WebSocket State**: Handles real-time updates such as score changes.

The frontend applies a **container-presentational design pattern** to separate business logic from visual components, ensuring better readability and reusability. It also makes extensive use of custom **React hooks** to encapsulate shared logic, such as authentication (**`useAuth`**) and real-time communication (**`useWebSocket`**).

7.2 Server (Backend Architecture)

The backend uses **Node.js/Express** and follows a **layered MVC structure**:

Routes → Controllers → Services → Models

- **Routes:** Define API endpoints (e.g., `/api/match`, `/api/users`) and direct requests to controllers.
- **Controllers:** Handle incoming requests, validate inputs, and delegate processing to services.
- **Services:** Contain the core application logic such as game flow, question handling, and leaderboard updates.
- **Models:** Define database schemas and manage data persistence.
- **WebSocket Layer:** Manages real-time communication for clients.

On the server side, the **singleton design pattern** is used to maintain a single shared database connection instance across the application, improving performance and consistency. The **observer design pattern** on the other hand, underpins the WebSocket event system, allowing the server to broadcast updates to all the connected clients.

8. Deployment & Operating Environment

- **Development Environment:** Local development using **Node.js** and **Vite**.
- **Deployment Environment:** Containerized with Docker; `docker-compose.yml` defines frontend and backend services. Running `docker compose up` builds and starts the full stack.
- **Hosting Platform:** Deployed on **Docker**.
- **Major Dependencies:** Frontend relies on **React**, **React Router**, **Vite**, **Socket.IO**, while the backend relies on **Node.js**, **Express**, **Socket.IO, Supabase JS Client**, **dotenv**.
- **Environment Variables:** Configuration keys include `SUPABASE_URL`, `SUPABASE_ANON_KEY`, `SUPABASE_SERVICE_ROLE_KEY`, and `PORT` for the backend server.

9. Extra Features & Known Issues

An **extra feature** we implemented was real-time **notifications** for when users are invited to a match, which allows them to either accept or decline the invite.

However, some known issues are that we were not able to resolve are as follows:

- No player status functionality.
- Admin removing users from games is untested.

11. Contribution Log

We used a Notion page to keep track of all our tasks, and had functionality that allowed us to assign tasks to different group members. In addition to this, all of our meetings had minutes written up for them, which allowed us to keep track of things discussed in these meetings, and all be on the same page.

SU Number	Name & Surname	Contribution
26978725	Ashton Weir	Webscraper, helped frontend pages, documentation.
26947110	Gregory Gebers	Database, authorisation, backend.
27205762	Katja Wood	Wireframe, frontend, CSS, documentation.
26989395	Andrew Cottrell	Database, backend, main gameplay loop.
27251489	Benjamin Conolly	Most frontend pages, documentation & usecase diagrams, backend for Admin pages.

12. AI Usage, External Code & References

12.1 AI usage

ChatGPT and DeepSeek were used as debugging tools in the development of our web application. Spelling and grammar in this report were also checked using AI tools.

12.2 References

GeeksforGeeks, 2025. REST API Introduction. 3 September. Available at: https://www.geeksforgeeks.org/node-js/rest-api-introduction/ [Accessed 19 October 2025].
Shvets, A., 2020. Dive Into Design Patterns. Independently published.
Britto, V., 2023. React Design Patterns: The Container/Presentational Pattern. Medium, 28 July. Available at: https://medium.com/@vitorbritto/react-design-patterns-the-container-presentational-pattern-775b91aa0c49 [Accessed 19 October 2025].