# MythBusters Project Software Architecture Overview

### **1. High-Level Architecture Design**

A **Client-Server Architecture** is the most suitable model for this project. It will be implemented with a modern **Single-Page Application (SPA)** on the frontend and a Spring-BootBackend. This approach aligns perfectly with the specified technology stack (React and Spring Boot) and provides a clear separation of concerns.

#### **Architectural Diagram (Structured Description)**

This model can be broken down into three primary layers:

1. Client-Side (Frontend)

* **Framework:** A Single-Page Application (SPA) built using **React** and **TypeScript**.
* **UI Library:** **Material UI (MUI)** for a consistent and professional component-based design.
* **Responsibilities:**
  + **Presentation Logic:** Renders all user interfaces, including the main dashboard, game screens, flashcards, profile pages, and leaderboards.
  + **User Interaction:** Manages all real-time user input and game interactions (e.g., answering questions, controlling game characters).
  + **State Management:** Holds the temporary state of the application, such as the current game's status or user session information.
  + **API Communication:** Makes asynchronous HTTP requests (REST API calls) to the backend to fetch data, authenticate users, and send results.

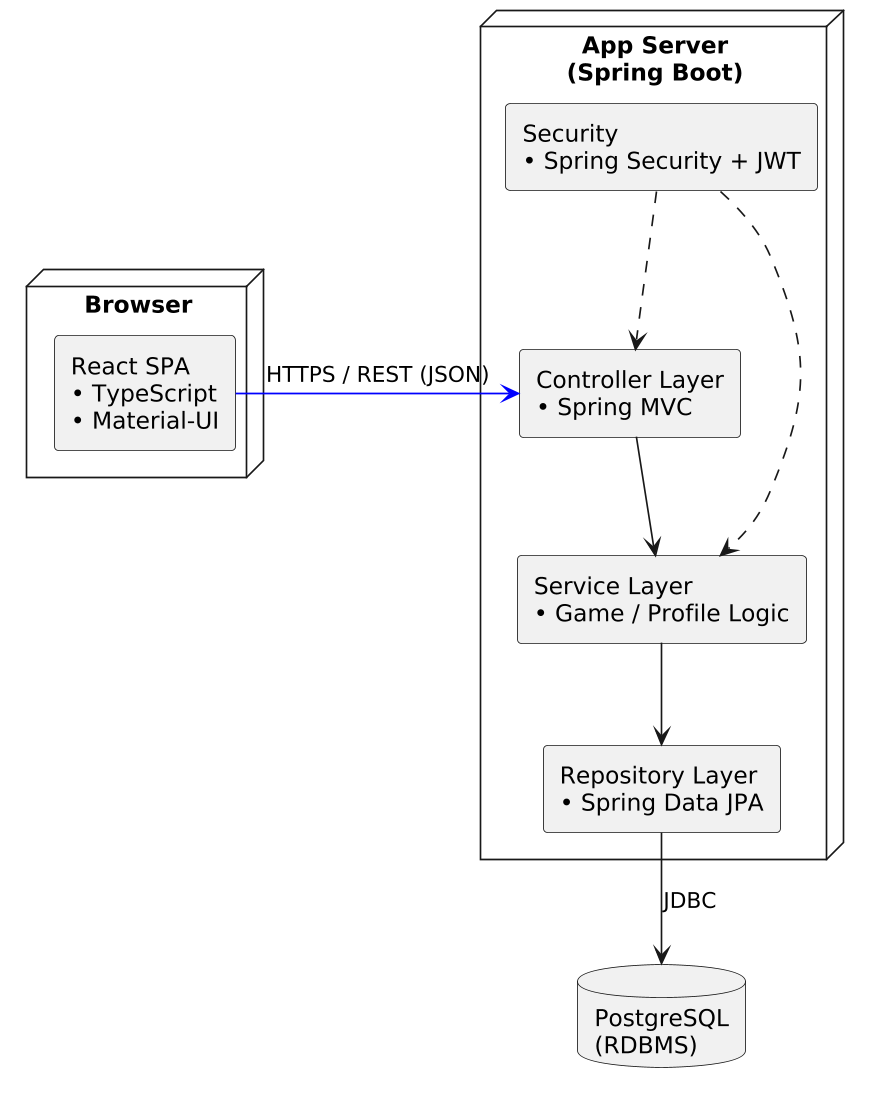
2. **Server-Side (Backend)**

* **Framework:** A **Java-based Application** built using the **Spring Boot** framework.
* **Responsibilities:**
  + **API Endpoints (Controller Layer):** Exposes a RESTful API for the client to consume. Endpoints will exist for user authentication, game data, scoring, profile management, and leaderboards.
  + **Business Logic (Service Layer):** Contains the core application logic. This includes validating game results, calculating scores, managing user currency, processing avatar purchases, and compiling leaderboard data.
  + **Data Access (Repository Layer):** Manages all interactions with the database, abstracting the database operations from the business logic.
  + **Security:** Handles user authentication, session management, authorization to protect user data and secure API endpoints.

3. Persistence Layer (Database)

* **System:** A PostgreSQL relational database.
* **Responsibilities:**
  + **Data Storage:** Persistently stores all critical application data.
  + **Key Data Schemas (Tables):**
    - Profile: Stores user account information (e.g., username, hashed password, profile details, avatar ID, profile photo).
    - Leaderboard: Records individual game scores linked to users.
    - Avatars: Contains information about available avatars and their costs.

**2. Diagram or Structured Description that shows major architectural components (e.g., client-server, layered architecture, microservices, etc.)**

****

### **3. How the Architecture Facilitates the Use Cases**

This client-server architecture directly supports the identified use cases in a scalable and maintainable way:

* **For Gameplay (Use Case 1):** The interactive and visually rich game experience is handled entirely by the **React frontend**, providing a fast and responsive user interface without needing to constantly communicate with the server. When a game is completed, the client sends a single, secure API call to the **Spring Boot backend**. The backend's **Service Layer** then validates the score, updates the user's points and currency, and records the transaction in the **PostgreSQL database**. This ensures that the core logic is secure and data integrity is maintained.
* **For Buying Avatars (Avatar Customization) (Use Case 2):** The frontend fetches the user's currency and available avatars from the backend. When a user purchases an avatar, the client sends a request to a specific endpoint (e.g., /api/profile/purchase-avatar). The backend handles this as a single, atomic transaction: it verifies the user has sufficient funds, deducts the currency, and updates the user's profile in the database. This prevents data inconsistencies.
* **For Viewing Leaderboards (Use Case 3):** The frontend simply requests the leaderboard data from the backend. All the complex work of querying the database, aggregating scores from the Leaderboard table, and ranking users is performed efficiently on the **server-side**. The backend then sends a clean, sorted list of data to the client for display. This minimizes the amount of data transferred and the processing load on the user's device, ensuring the leaderboard loads quickly.