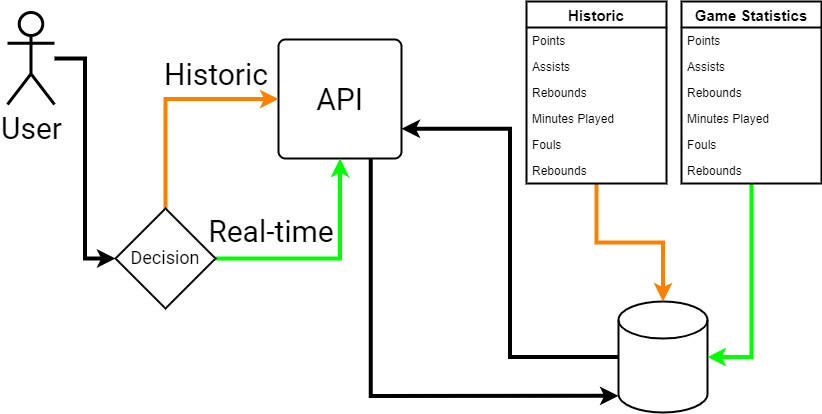
# **Real-Time In-Game Statistics Application (FitStat)**

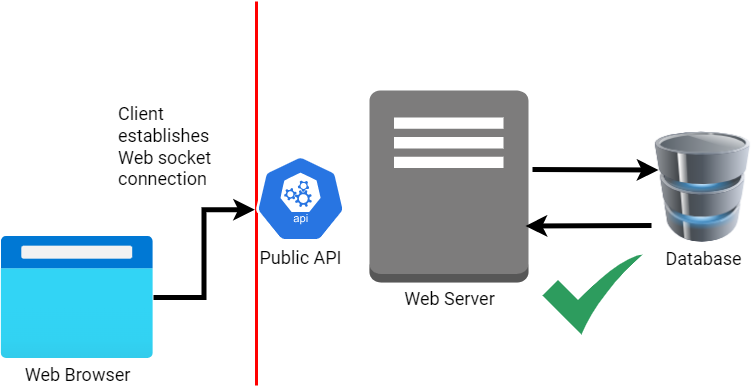
This document is a system design document for an application that serves real-time, in-game statistics. To address the concern of scalability, it is important to consider the workload of the application which should serve at most 100 users. Referring to the diagram below, the proposed application will need a REST API that can request and send data for historic games as well as in-progress games.



The REST API will use HTTP in executing the following process:

1. The client first establishes a Web socket connection. The connection should be a persisted connection which stays open for the duration of the session.
2. The API will read data with no ability to update or delete.
3. The web server will query the database and persist state to the user interface. The persisted state is updated in the database.
4. If changes occur in the database outside of the API, the database must notify the web server of the change. The web server will then access the new changes and send them to the UI in real-time.

**Diagram showing the relationship between the clients established connection, the API, and bi-directional communication between the server and database.**



## **Navigation**

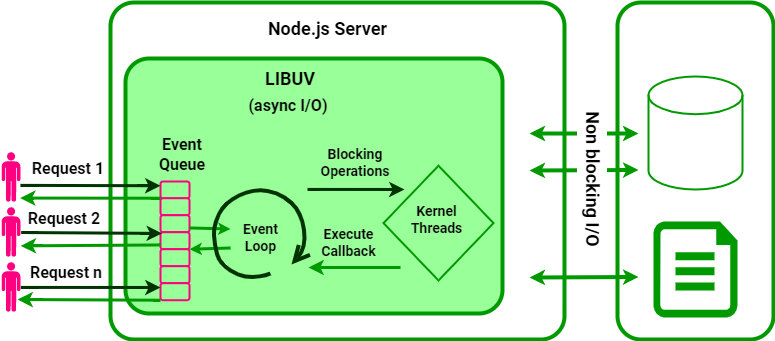
The applications API will be able to request two endpoints such as **/games in progress** and **/historic games**. These endpoints will control the navigation and rely on data from two database tables with respective names. A Model View Controller (MVC) architecture would be ideal – the view will render the options for navigation while the controller will facilitate the accessibility of different routes. The database can be programmed such that when a game in progress comes to an end, then the data of that table will be automatically pushed into the historic games table and dropped to create space for another real-time match.

## **Tools and Technologies**

The technologies to build this system are:

1. **Node.js** for managing the backend logic, requesting routes, increasing the system availability, creating event-driven features and non-blocking I/O of real-time events. Additionally, it allows the developer to use JavaScript for both the server-side and client-side.
2. **Socket.io**, a Node.js module which handles constant communication between a client and server, enabling the server to push real-time updates. It also enables real-time bidirectional event-based communication.
3. **MongoDB** database for secure and fast data retrieval.
4. A **JavaScript** **Frontend** **SPA framework** to persist state to the user interface – it must also have a feature for change detection.

**Diagram showing the non-blocking I/O process and how the server manages user requests in the event queue.**



## **Scaling**

Essential mechanisms to manage multiple users are load balancing and a content delivery network. If the application was to be accessed by 10,000 users simultaneously, it would experience a lapse in performance due to the overload of requests. The increase of users would require more resources – such as bandwidth and storage. Viable solutions include vertical partitioning, horizontal partitioning, or adding a web caching layer to offload read requests for games currently in progress.