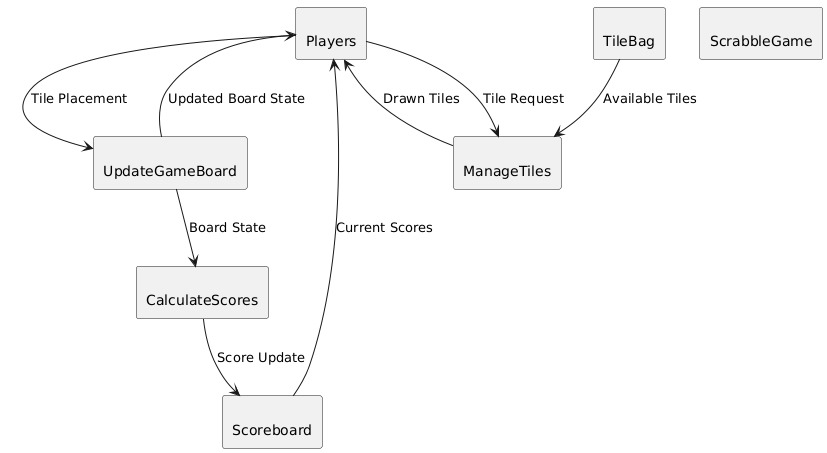
**HIGH LEVEL DESIGN FOR SCRABBLE GAME:**

**By team 1-d**

**Key components and their interactions:**

**Data flow diagram:**

****

**Request and response:**

**1. Players**

* **Interaction with ManageTiles**: Players send a tile request to the ManageTiles component to draw new tiles. They receive drawn tiles in response.
* **Interaction with UpdateGameBoard**: Players place tiles on the game board, which sends the updated board state back to them.

**2. ManageTiles**

* **Interaction with Players**: Manages the tile requests from players, fulfilling these requests by drawing tiles from the TileBag.
* **Interaction with TileBag**: Requests available tiles from the TileBag to meet the players’ tile requests.

**3. TileBag**

* Acts as a source of tiles for the ManageTiles component, providing available tiles as needed.

**4. UpdateGameBoard**

* **Interaction with Players**: Receives tile placements from players to update the board state.
* **Interaction with CalculateScores**: Sends the current board state to CalculateScores to determine players' scores based on the latest board configuration.

**5. CalculateScores**

* **Interaction with UpdateGameBoard**: Receives the board state to compute players' scores.
* **Interaction with Scoreboard**: Sends updated scores to the Scoreboard to keep track of the latest scores.

**6. Scoreboard**

* **Interaction with CalculateScores**: Receives the latest score updates and maintains a record of players' scores.

**7. ScrabbleGame**

* This component is not directly connected in this diagram but may serve as the main controller for the game, coordinating the interactions and managing the game flow.

**Non-Functional requirements:**

**Latency:**

1. **Tile Placement Latency**
   * **Requirement**: The system should update the game board state within 500 milliseconds of a player's tile placement to ensure a smooth and real-time gameplay experience.
   * **Justification**: Quick updates are essential for maintaining game flow, especially in multiplayer games where players expect immediate feedback on their actions.
2. **Score Calculation Latency**
   * **Requirement**: The score calculation process should take no more than 300 milliseconds after tile placement, ensuring that the scoreboard reflects the latest scores in near real-time.
   * **Justification**: Players need to see their scores update quickly to stay engaged and make strategic decisions based on the latest game state.
3. **Tile Draw Latency**
   * **Requirement**: The latency for drawing tiles from the TileBag and updating the player's tile rack should be less than 200 milliseconds.
   * **Justification**: Minimizing tile draw latency enhances the game’s responsiveness, providing a seamless experience as players quickly receive their new tiles and can proceed with the game.
4. **End-to-End Latency**
   * **Requirement**: The total latency from a player’s action (such as placing a tile) to all related updates (board state, scores, tile management) across components should not exceed 1 second.
   * **Justification**: This ensures that the entire game system feels responsive, even in a multi-component setup, and provides players with timely feedback.
5. **Network Latency (for Online Multiplayer)**
   * **Requirement**: The system should be optimized to handle network latency up to 100 milliseconds between clients and the server, without significant performance degradation.
   * **Justification**: In online multiplayer games, network latency can impact the perceived responsiveness. By accommodating network latency within reasonable limits, the system can deliver a consistent experience for players with varying internet speeds.
6. **Latency for Error Notifications and Feedback**
   * **Requirement**: Error messages or notifications related to invalid actions (e.g., invalid tile placement or an invalid word) should be displayed within 200 milliseconds of the action.
   * **Justification**: Providing immediate feedback on errors helps players correct their moves without waiting, leading to a smoother game experience.

**Strategies for Managing Latency**

* **Caching**: Use caching mechanisms for frequently accessed data (like tile details and current board state) to reduce retrieval time.
* **Efficient Data Structures**: Use optimized data structures for game state and tile management to process and update information more quickly.
* **Load Balancing**: For online multiplayer, consider using load balancers and optimized routing to minimize delays, especially under high player load.
* **Edge Computing**: Use edge servers to serve players in geographically diverse locations, reducing network latency for global players.

**Security Considerations:**

Security is essential, especially for a system with sensitive player data, user accounts, and potential multiplayer interactions. Here are the key security requirements and practices:

* **Authentication & Authorization**:
  + **JWT (JSON Web Tokens)**: Use JWT for user authentication to ensure secure, stateless authentication for players.
  + **Role-Based Access Control (RBAC)**: Define roles (e.g., player, admin) to restrict access to certain features, ensuring that only authorized users can perform sensitive actions.
* **Data Encryption**:
  + **Encryption in Transit**: Use HTTPS to secure all data transferred between the client, server, and database.
  + **Encryption at Rest**: Enable Azure’s built-in encryption for MongoDB storage to protect data when it’s not actively used.
* **Data Validation & Sanitization**:
  + Use server-side validation to prevent malicious inputs and SQL injections.
  + Use libraries like validator in Node.js for input sanitization, particularly for data submitted by users.
* **Rate Limiting and DDoS Protection**:
  + Implement rate limiting to restrict the number of requests from a single IP, protecting against abuse.
  + Use Azure’s DDoS Protection to shield against distributed denial-of-service attacks, which can overwhelm the server with traffic.
* **Logging and Monitoring**:
  + Implement logging to track user actions and system events, with real-time monitoring for security incidents.
  + Use Azure Monitor or Azure Security Center to detect and respond to suspicious activity.
* **Regular Security Audits**:
  + Conduct regular vulnerability assessments, especially for dependencies in the MERN stack (MongoDB, Express, React, Node.js).
  + Use tools like OWASP ZAP or Snyk to scan the codebase for known vulnerabilities.

**Technology Stack:**

The MERN stack (MongoDB, Express.js, React, Node.js) combined with Azure services provides a robust environment for building and deploying web applications. Here’s how each component fits in with Azure:

* **MongoDB (Database)**:
  + **Azure Cosmos DB for MongoDB**: Use Azure’s Cosmos DB with MongoDB API compatibility, which provides a fully managed, scalable, and globally distributed database solution.
  + **Alternative**: Host MongoDB on Azure VM if you need more direct control over configurations.
* **Express.js (Backend Framework)**:
  + Run Express.js on **Azure App Service** or deploy it on an **Azure Virtual Machine (VM)**. Azure App Service provides managed hosting with options for auto-scaling and load balancing.
  + Integrate **Azure Key Vault** to securely store secrets, such as database credentials or API keys, used in Express.js.
* **React (Frontend Framework)**:
  + **Static Web Apps**: Host the React frontend using Azure Static Web Apps, which integrates well with GitHub for CI/CD (Continuous Integration/Continuous Deployment).
  + **CDN**: Utilize Azure CDN (Content Delivery Network) to serve static assets, ensuring faster load times by caching resources closer to the user.
* **Node.js (Runtime)**:
  + Deploy Node.js on **Azure App Service** for seamless integration with other Azure services, or use **Azure Functions** for serverless execution, which can reduce costs by only charging for executed code.
  + Use **Application Insights** for monitoring Node.js applications to capture insights into application performance and user interactions.
* **DevOps and CI/CD**:
  + Use **Azure DevOps** or **GitHub Actions** for continuous integration and deployment pipelines, ensuring that code is tested and deployed automatically.
  + Set up environments for staging and production to test new features before releasing them live.

**COGS (Cost of Goods Sold):**

* App service, Azure Functions, Azure cost management and billing: $0.00
* Azure Cosmos DB: $13.84 ($5.84 for 100 requests/ second and $8.00 for 32GB Transactional storage)