# Project Report

## FooBooLoo

_By Andy Tran_

## Table of Contents

## 1. Introduction

The project involved designing and implementing a custom FizzBuzz online game. The game allows users to create their own versions of the FizzBuzz game by defining rules that replace numbers with specific words. The application follows a 3-tier architecture comprising a ReactJS front end, a .NET 8 Web API back end, and a PostgreSQL database, all containerized with Docker.

## 2. Requirements

The requirements for the project included:

- Users should be able to create custom FizzBuzz-like games with unique names.
- The game should replace numbers based on the rules defined by the user.
- A game session should serve random numbers to the player, and the player should respond with the correct word(s) based on the rules.
- The game should last for a user-defined duration, and no duplicate random numbers should be served within a session.
- The application should display the score at the end of the session.
- The solution should be containerized using Docker and include unit tests for both the front end and back end.

## 3. Project Structure

The project is organized into the following directories:

- **backend/**: Contains the .NET 8 Web API project and related files.
  - **FooBooLooGameAPI/**: The main API project.
  - **FooBooLooGameAPI.Tests/**: Unit tests for the API.
- **frontend/**: Contains the React front-end project.
- **docker-compose.yml**: Docker Compose configuration for setting up the development environment.

## 4. System Design

The system was designed with a 3-tier architecture, separating concerns into the following layers:

1. **Presentation Layer (Front End):**

   Implemented with ReactJS and TypeScript, responsible for user interaction, displaying games, managing game sessions, and generating relevant feedbacks.

2. **Business Logic Layer (Back End):**

A .NET 8 Web API responsible for managing game logic, validating rules, generating random numbers, and verifying player answers. This backend implementation was completely built based on repository design pattern to ensure an improvement for maintainability, testability, and flexibility of the codebase.

Benefits of the Repository Design Pattern:

- Separation of Concerns:
    - Data Access Logic: The repository pattern abstracts the data access logic, keeping it separate from the business logic. This makes the codebase cleaner and easier to manage.
    - Business Logic: Services like **GameService** and **SessionService** focus on business logic without worrying about how data is fetched or persisted.
- Testability:
    - Mocking: By using interfaces for repositories (e.g., **IGameRepository**, **ISessionRepository**), you can easily mock these dependencies in unit tests. This is evident in the test classes where **Mock<IGameRepository>** and **Mock<ISessionRepository>** are used.
    - Isolation: Tests can be written to focus on the business logic without needing a real database, making tests faster and more reliable.
- Maintainability:
    - Single Responsibility: Each repository class has a single responsibility: to handle data access for a specific entity. This makes the code easier to understand and maintain.
    - Encapsulation: Changes to the data access logic (e.g., switching from one database to another) can be made in the repository layer without affecting the business logic.

Why Not Other Design Patterns?

- Active Record Pattern:
    - Coupling: The Active Record pattern combines data access and business logic in the same class, which can lead to tightly coupled code and make unit testing more difficult.
    - Complexity: For complex business logic, separating concerns as done in the Repository pattern is generally more manageable.
- Service Locator Pattern:
    - Hidden Dependencies: This pattern can lead to hidden dependencies and make the code harder to understand and maintain.
    - Testability: It can also make unit testing more challenging because dependencies are not explicitly passed to classes.

3. **Data Layer (Database):**

The PostgreSQL database consisted of 4 tables in total:

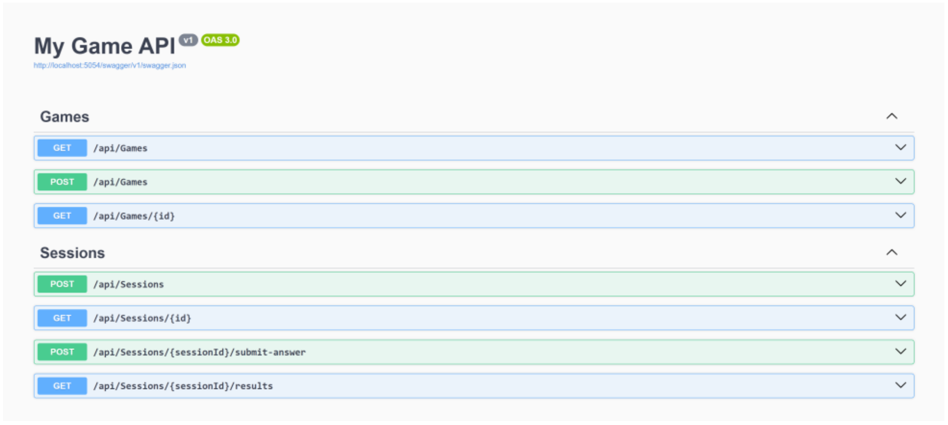
- Games table: stores all properties of each game instance such as name, author, range of numbers, and the time when it was created.
- Game rules table: collects all the rules that have been defined by users for any games existed in the system.
- Sessions table: saves all information regarding any specific sessions such as its duration, score, start time, and whether it is ended or not.
- Session numbers table: memorizes all the numbers that are served during any particular session and their answer states.

## 5. Implementation

### 5.1 Back-End Implementation

The back end was implemented using .NET 8 Web API. Key components included:

- **Entities (Database Models):**
    - `Games`: Stores the game name, author, and rules.
    - `Sessions`: Tracks active game sessions, player information, and scores.
    - `SessionNumbers`: Logs numbers served in a session to ensure no duplicates and track the status of their respective answer.
    - `GameRules`: Contains all the game rules that have been set by users.
- **Controllers:**
    - `POST /api/games`: Create a new game.
    - `GET /api/games`: Retrieve all games.
    - `GET /api/games/{gameId}`: Get a specific game by its ID.
    - `POST /api/sessions`: Start a new game session.
    - `GET /api/sessions/{sessionId}`: Get a specific session by its ID.
    - `POST /api/sessions/{sessionId}/submit-answer`: Submit an answer and validate it against the game rules.
    - `GET /api/sessions/{sessionId}/results`: Get the result of a specific session by its session ID.



http://localhost:5054/swagger/v1/swagger.json

Games

GET /api/Games

POST /api/Games

GET /api/Games/{id}

Sessions

POST /api/Sessions

GET /api/Sessions/{id}

POST /api/Sessions/{sessionId}/submit-answer

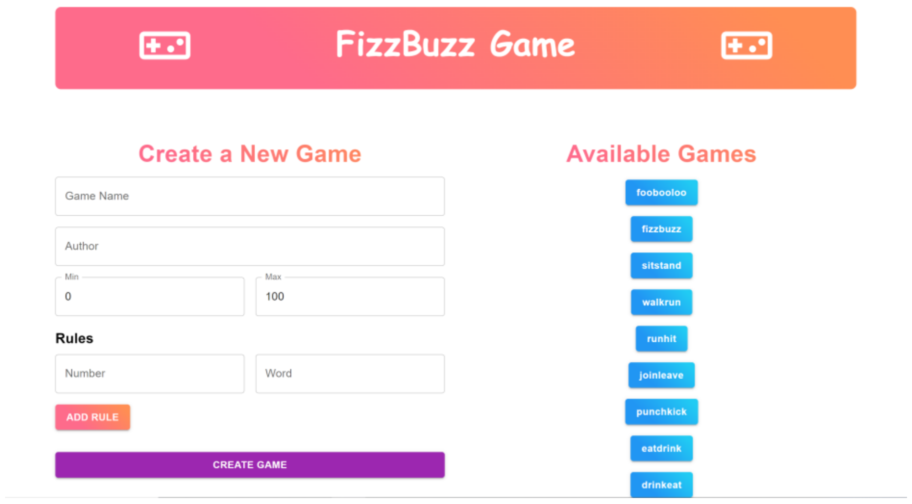
GET /api/Sessions/{sessionId}/results

- **Services:**
    - Random number generation to avoid duplicates within a session.
    - Answer validation based on the custom rules defined by the user.
    - Persistence of game and session data in the PostgreSQL database.
- **Repositories:**
    - Behave as a bridge to let the game API connect and interact with the PostgresSQL database.
- **DTOs (Data Transfer Objects):**
    - Be responsible for handling the differences in data structures for API requests and responses.
- **Mappers:**
    - Convert between DTOs and entities.
- **Containerization:**
    - Dockerfile was created to containerize backend service including the .NET API and PostgreSQL database.

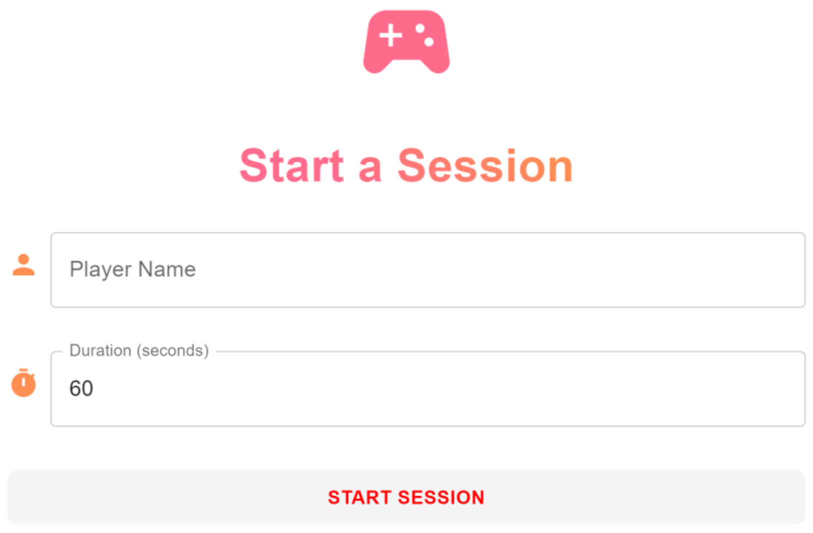
## 5.2 Front-End Implementation

The front end was implemented using ReactJS with TypeScript. Key components included:

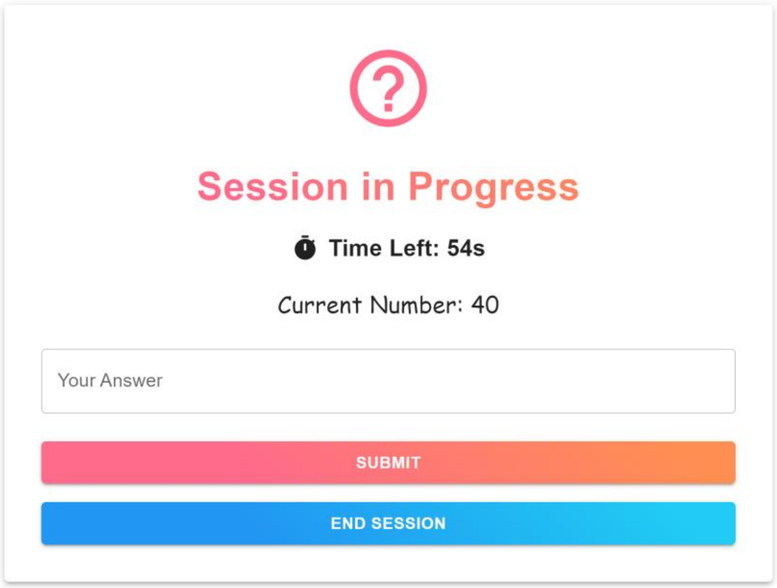
- **Home Component:**
    - Displays the home screen to the users where they can see the form to create a new game and the list of available games that they can choose to play.



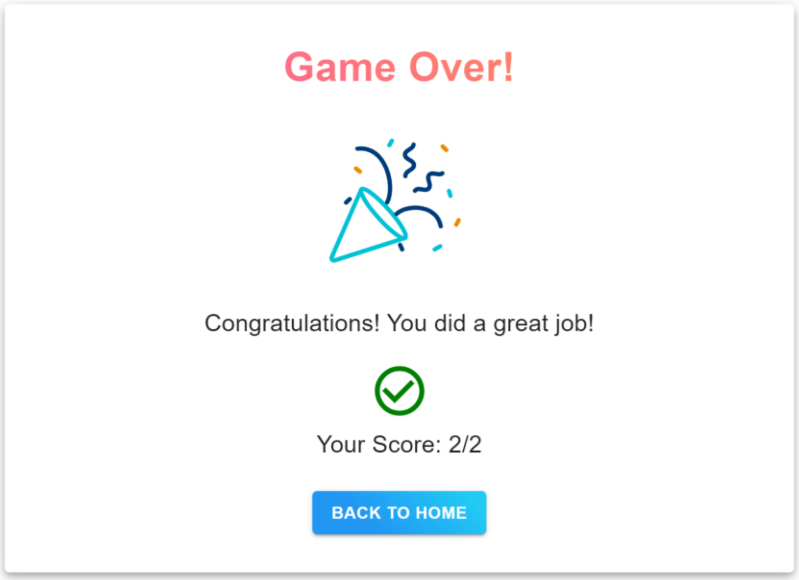
- **Game Component:**
    - Allows users to begin a session by entering their names and duration that they want to play the game.



- **Session Component:**
  - Showing up an in-progress session where the user answers the given question within the pre-defined duration. Alternatively, the user can choose to end the session early by clicking the button "END SESSION".



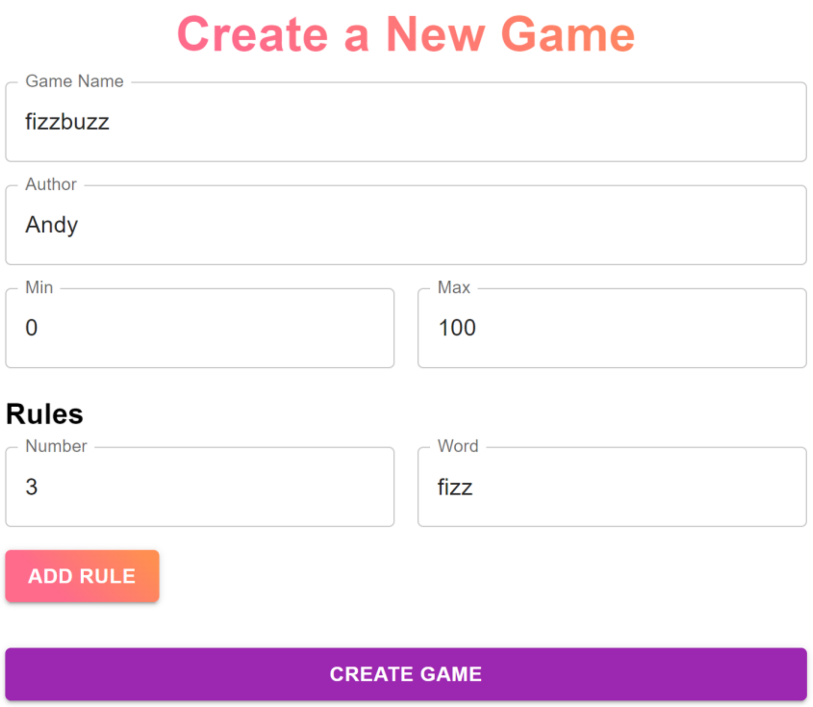
- **Result Component:**
  - Generates the result of the most recent session after it is ended.


Game Over!

Congratulations! You did a great job!

Your Score: 2/2

- **GameForm Component:**
    - Allows users to create a new game by specifying the game name, author, range of numbers, and custom rules.


Create a New Game

Game Name: fizzbuzz

Author: Andy

Min: 0

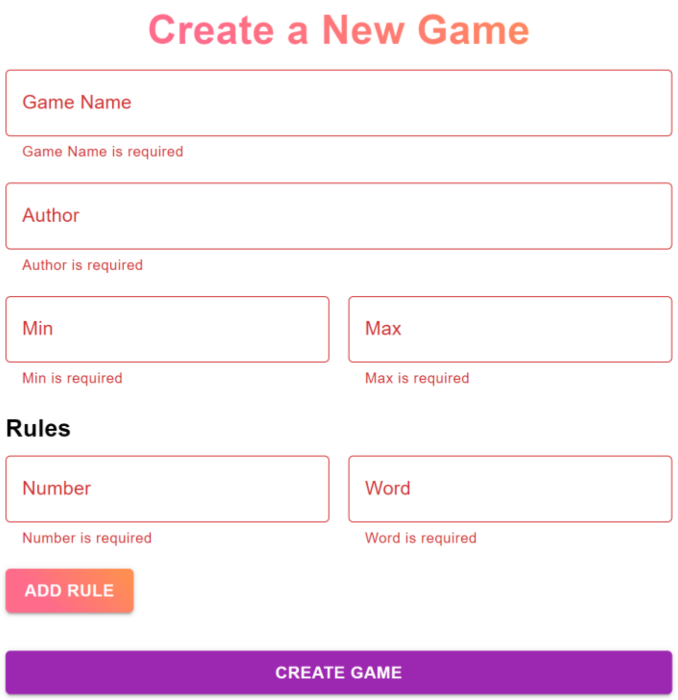
Max: 100

Rules

Number: 3

Word: fizz

    - Validates users' inputs to prevent invalid or inappropriate requests and align with the requirements.



Create a New Game

Game Name

Game Name is required

Author

Author is required

Min

Min is required

Max

Max is required

Rules

Number

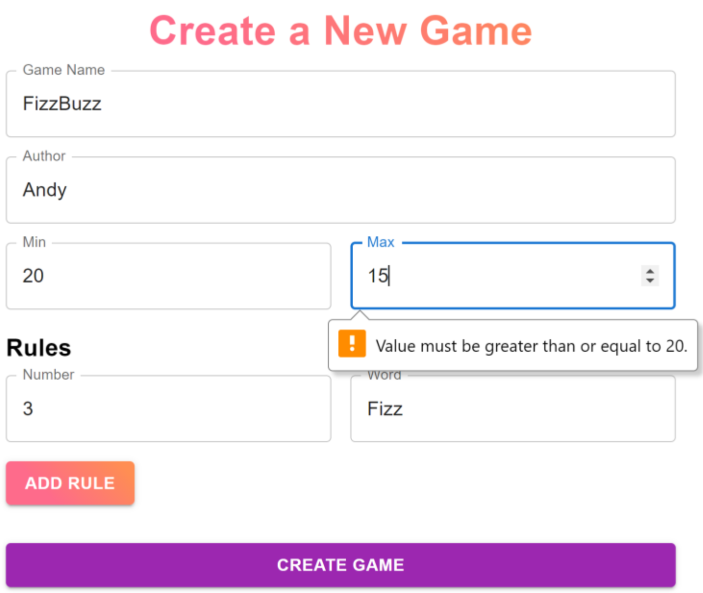
Number is required

Word

Word is required

ADD RULE

CREATE GAME



Create a New Game

Game Name: FizzBuzz

Author: Andy

Min: 20

Max: 15

Value must be greater than or equal to 20.

Rules

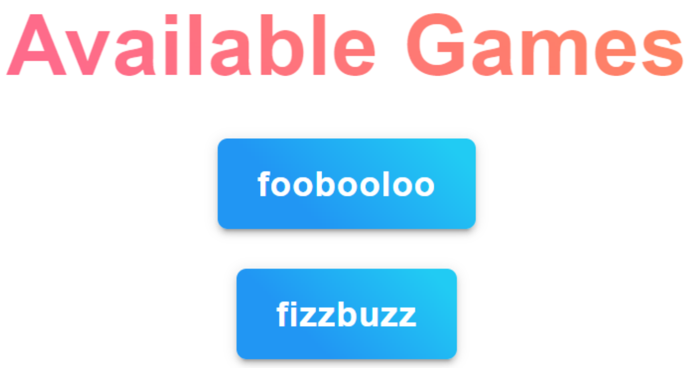
Number: 3

Word: Fizz

ADD RULE

CREATE GAME

- **GameList Component:**
  - Displays a list of all available games, allowing users to select and start a game session.

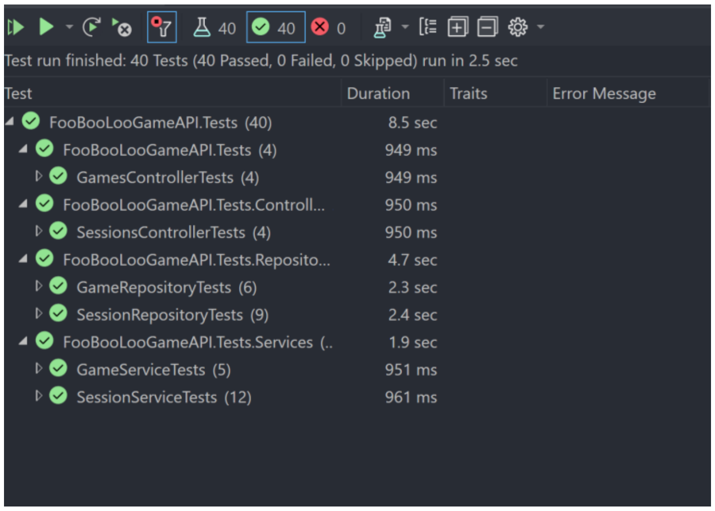


- **State Management:**
    - Redux was used to manage the state across components, including the list of games, current session data, and player scores.
- **API Integration:**
    - Axios was used for making HTTP requests to the back-end API to create games, start sessions, and submit answers.
- **Routing:**
    - React Router was used to manage navigation between different pages, including the home page, game creation page, and session page.
- **Containerization:**
    - A Dockerfile was created to containerize the React application.

## 6. Testing

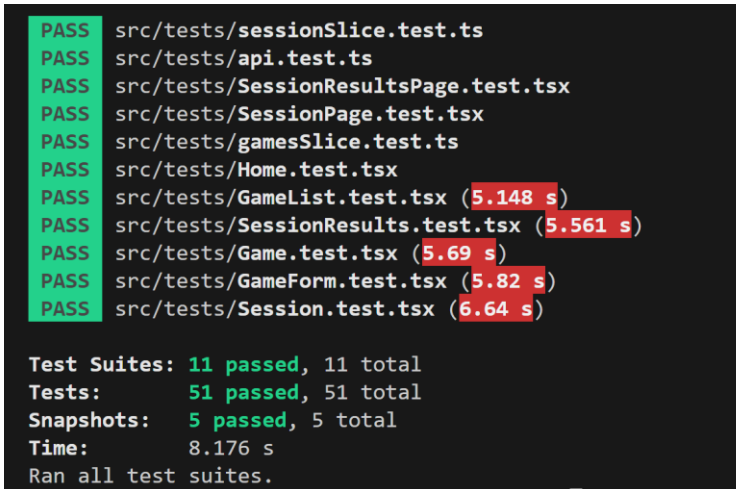
Unit tests were written for both the front end and back end to ensure the reliability of the application:

- **Back-End Testing:**
    - xUnit and Moq were used to test the .NET API's business logic and repository methods.
    - Tests covered scenarios such as game creation, session management, random number generation, and answer validation.



```
Test run finished: 40 Tests (40 Passed, 0 Failed, 0 Skipped) run in 2.5 sec
Test                                      Duration   Traits   Error Message
◢ FooBooLooGameAPI.Tests (40)              8.5 sec
  ◢ FooBooLooGameAPI.Tests (4)             949 ms
    ▷ GamesControllerTests (4)             949 ms
  ◢ FooBooLooGameAPI.Tests.Controll...     950 ms
    ▷ SessionsControllerTests (4)          950 ms
  ◢ FooBooLooGameAPI.Tests.Reposito...     4.7 sec
    ▷ GameRepositoryTests (6)              2.3 sec
    ▷ SessionRepositoryTests (9)           2.4 sec
  ◢ FooBooLooGameAPI.Tests.Services (..    1.9 sec
    ▷ GameServiceTests (5)                 951 ms
    ▷ SessionServiceTests (12)             961 ms
```

- **Front-End Testing:**
  - Jest and React Testing Library were used to test React components and their integration with Redux.
  - Tests ensured that components correctly displayed data, handled user input, and made appropriate API calls.



```
 PASS  src/tests/sessionSlice.test.ts
 PASS  src/tests/api.test.ts
 PASS  src/tests/SessionResultsPage.test.tsx
 PASS  src/tests/SessionPage.test.tsx
 PASS  src/tests/gamesSlice.test.ts
 PASS  src/tests/Home.test.tsx
 PASS  src/tests/GameList.test.tsx (5.148 s)
 PASS  src/tests/SessionResults.test.tsx (5.561 s)
 PASS  src/tests/Game.test.tsx (5.69 s)
 PASS  src/tests/GameForm.test.tsx (5.82 s)
 PASS  src/tests/Session.test.tsx (6.64 s)

Test Suites: 11 passed, 11 total
Tests:       51 passed, 51 total
Snapshots:   5 passed, 5 total
Time:        8.176 s
Ran all test suites.
```

## 7. Deployment

The application was fully containerized and can be deployed using Docker:

- **Docker Compose:** A `docker-compose.yml` file was provided to deploy the both the back-end and front-end implementation including PostgreSQL database, .NET API, and the React-TypeScript frontend framework.
- **Port Configuration:** The back-end API was configured to run on port 5000, and the front-end application was served on port 3000.

## 8. Conclusion

FooBooLoo is a comprehensive project that demonstrates the integration of a .NET 8 Web API with a React front-end, all managed through Docker for ease of deployment.

The project is well-structured and successfully met the requirements by providing a customizable FizzBuzz game that users can play online, with an attractive user interface and robust error handling. The solution was implemented using modern web technologies and best practices, including a clear separation of concerns, state management with Redux, and containerization with Docker. The application is fully tested and can be easily deployed, making it a scalable and maintainable solution.

## 9. Future Enhancements

Possible future enhancements could include:

- **User Authentication:** Implementing user authentication to allow players to save their game progress and view their past scores.
- **Leaderboard:** Adding a leaderboard feature to display top scores globally or within specific games.
- **Enhanced Game Rules:** Allowing more complex game rules, such as combining multiple conditions with logical operators.

This project demonstrates a comprehensive approach to full-stack web application development, showcasing skills in both front-end and back-end technologies, as well as system design, testing, and deployment.