VILNIAUS UNIVERSITETAS
Matematikos ir informatikos fakultetas

2nd Laboratory Work

itssoover
Design and Implementation

Ignas Časas
Mykolas Marius Budrys
Augustas Kniška

Matematikos ir informatikos fakultetas
Vilniaus universitetas
Lietuva

2025

Contents

1 Context

Our web-based platform offers a number of cognitive games that are designed to test and enhance skills such as memory, attention, reaction time, and problem-solving. Users can access the system directly through a browser without needing to install any software or mobile apps. The platform provides customizable game difficulties and tracks user performance over time. While engaging, the platform prioritizes measurable cognitive improvement over entertainment.

1.1 Technology Stack

- **Frontend:** Built with Blazor for interactive C#-based UI.
- **Backend:** .NET (C#) Web API for business logic, user authentication, and statistics tracking.
- **Database:** SQLite for lightweight storage of user data and game progress.

1.2 Content & Accessibility Boundaries

- Games focus on cognitive improvement, not entertainment or storytelling.
- No complex mechanics (e.g., multiplayer or 3D).
- Web-based only—no mobile apps, downloads, or third-party integrations.

1.3 Functionality & Monetization

- Single-player only; no social features.
- No in-game purchases or microtransactions.

2 Logical View

2.1 Class Diagram

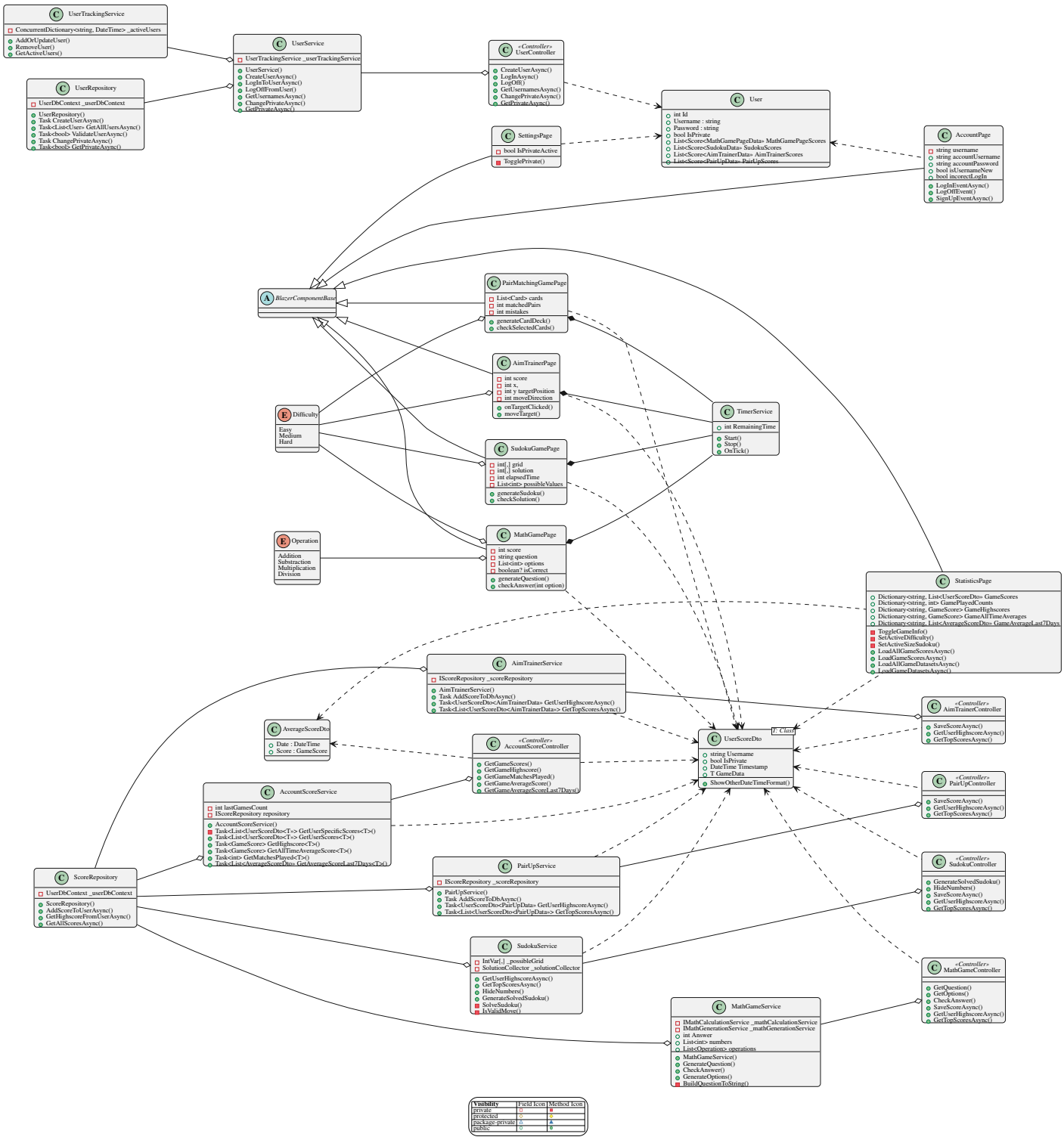

Figure 1: Class Diagram

Figure 1 illustrates the class structure of the application, dividing it into client-side and server-side components. On the client side, frontend pages inherit from BlazerComponentBase, including AccountPage and StatisticsPage for tracking cognitive progress, SettingsPage for managing privacy settings, and game-specific pages like MathGamePage, SudokuGamePage, PairMatchingGamePage, and AimTrainerPage, all of which utilize TimerService for countdowns. Most .cs files related to these pages are located in the Client/Pages folder, TimerService is located in Client/Services.

On the server side, controllers in the 'Server/Controllers' folder, including AccountScoreController, MathGameController, and others, expose API endpoints. Business logic is handled by services such as MathGameService, SudokuService, AimTrainerService, PairUpService, AccountScoreService, and UserService, all located in the 'Server/Services folder'. Repositories like ScoreRepository and UserRepository in the 'Server/Repositories' folder manage database interactions.

Shared resources include models from the 'Shared/Models' folder and Data Transfer Objects (DTOs) such as UserScoreDto<T> and AverageScoreDto, which facilitate communication between the frontend and backend.

2.2 Local User States

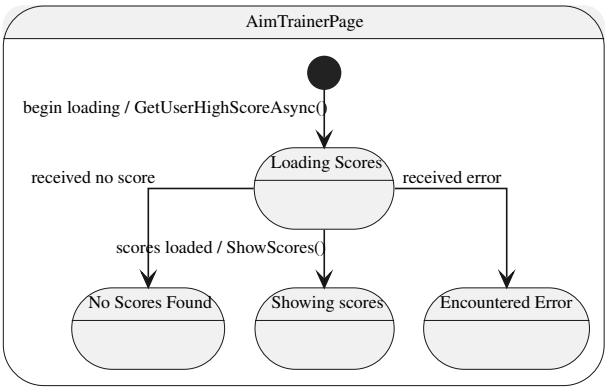

Figure 2: AimTrainerPage Score Fetching

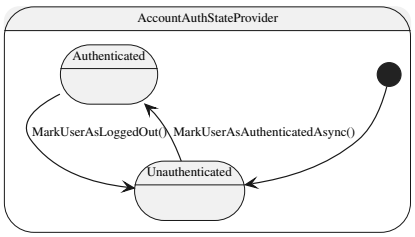

Figure 3: AccountAuthStateProvider State

Figure 2 outlines the asynchronous score retrieval process by the AimTrainerPage component. Other Page components in the Game Pages subsystem manage state in the same way. The state enters in a "Loading Scores" state where a background call (GetUserHighScoreAsync()) fetches data. Depending on the outcome, the state branches into one of three paths: it moves to "No Scores Found" if no score is returned, "Encountered Error" if an error occurs, or "Showing Scores" if the scores are successfully loaded. The User is shown different UI elements based on the AimTrainerPage state. This structure ensures that the application handles different scenarios gracefully while providing clear feedback to the user.

The AccountAuthStateProvider State diagram presented in Figure 3 defines the different possible states of user authentication kept by AccountAuthStateProvider. The component

begins in an "Unauthenticated" state. When an external call to mark the user as authenticated is made the component changes it's internal state to "Authenticated". If a MarkUserAsLoggedOut call is made and the AccountAuthStateProvider is in the "Authenticated" state the method then reverts the component back to the "Unauthenticated" state. The AccountAuthStateProvider component is user by the AccoutPage component. This model clearly delineates the transitions between being logged in and logged out, ensuring that the system maintains a secure and predictable user session management process.

2.3 Game States

These state diagrams are made to represent the internal state kept by components from the GamePage when the game is being played.

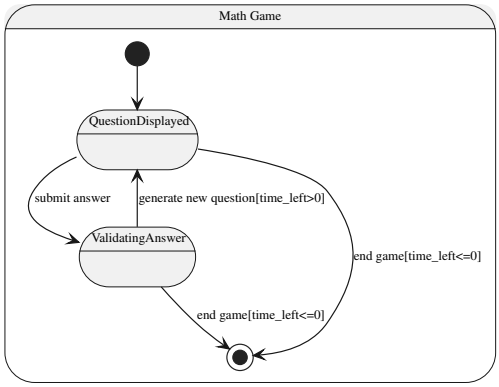

Figure 4: MathGamePage State Representation

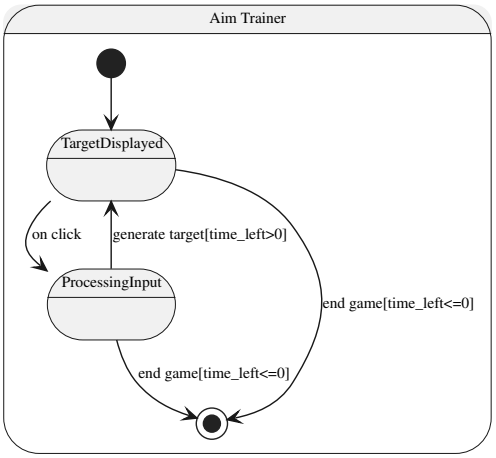

Figure 5: AimTrainerPage State Representation

In Figure 5 we model our *Aim Training* game. The game starts by displaying a target, and upon the user's click, it transitions to a state where the input is processed. If there is remaining time (i.e., [time_left>0]), a new target is generated, looping back to the display state; otherwise, the game ends. The *ProcessingInput* state handles updating both the failure count (if the target was missed) and the score.

Figure 4 represents our arithmetic based *Math Game*. It begins by displaying an equation to the user. When an answer is submitted, the system transitions to a validation state. The validation state is responsible for both making an asynchronous call to the back-end for answer validation and increasing the kept score if the answer was valid. If time remains, a new question is generated and the process repeats; if not, the game terminates. Modeling our game in this way highlights the cyclic gameplay loop that emphasizes both continuous engagement and time-bound gameplay, ensuring the game session concludes when the allotted time is exhausted.

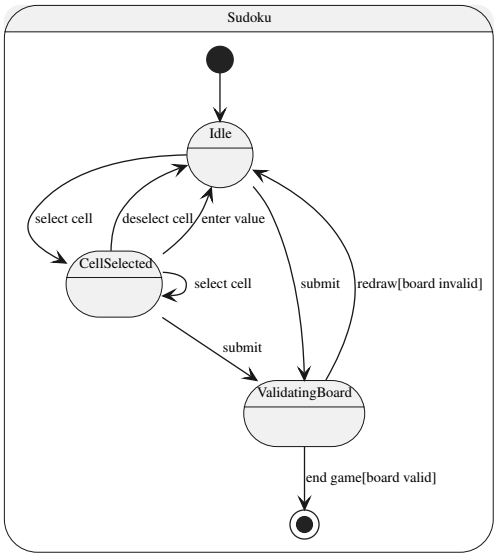

Figure 6: SudokuPage State Representation

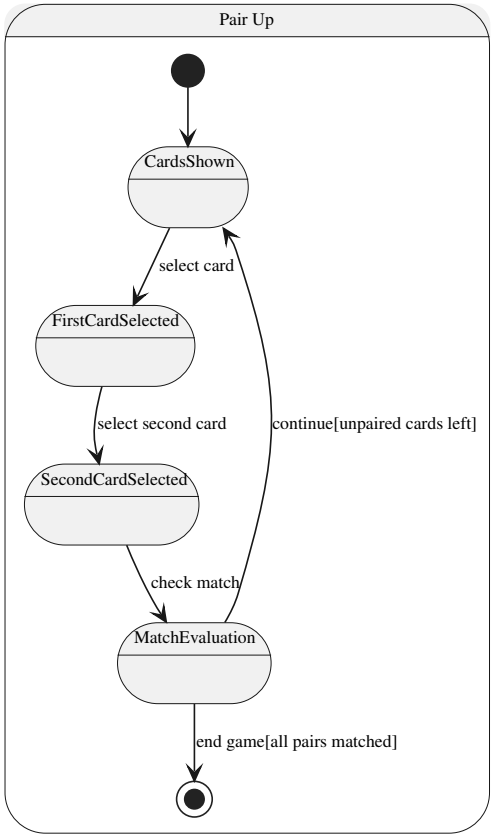

Figure 7: PairUpPage State Representation

Figure 6 (The *Sudoku* state diagram) captures the interactive elements of playing our *Sudoku* game. Initially, the board is idle, waiting for user interaction. When a cell is selected, the state changes, allowing the user to either deselect the cell, select another cell, input a value, or submit the board for validation. Upon submission, the system checks the board (by an asynchronous call to our API): if the board is invalid, it redraws and returns to the idle state; if the board is valid, the game ends. These states capture the essential user interactions and decision points necessary for the logic puzzle.

The *Pair Up* diagram (Figure 7) details the states of our memory-based matching game. The game starts with all cards being shown to the user. The user selects his first card (which the user cannot unselect), then his second card, prompting the system to evaluate whether the selected pair matches. If there are still unpaired cards remaining, the game cycles back to display the cards; if all pairs are successfully matched, the game ends. This design effectively captures the iterative nature of this game while clearly defining the conditions for continuation and termination.

2.4 Information Model

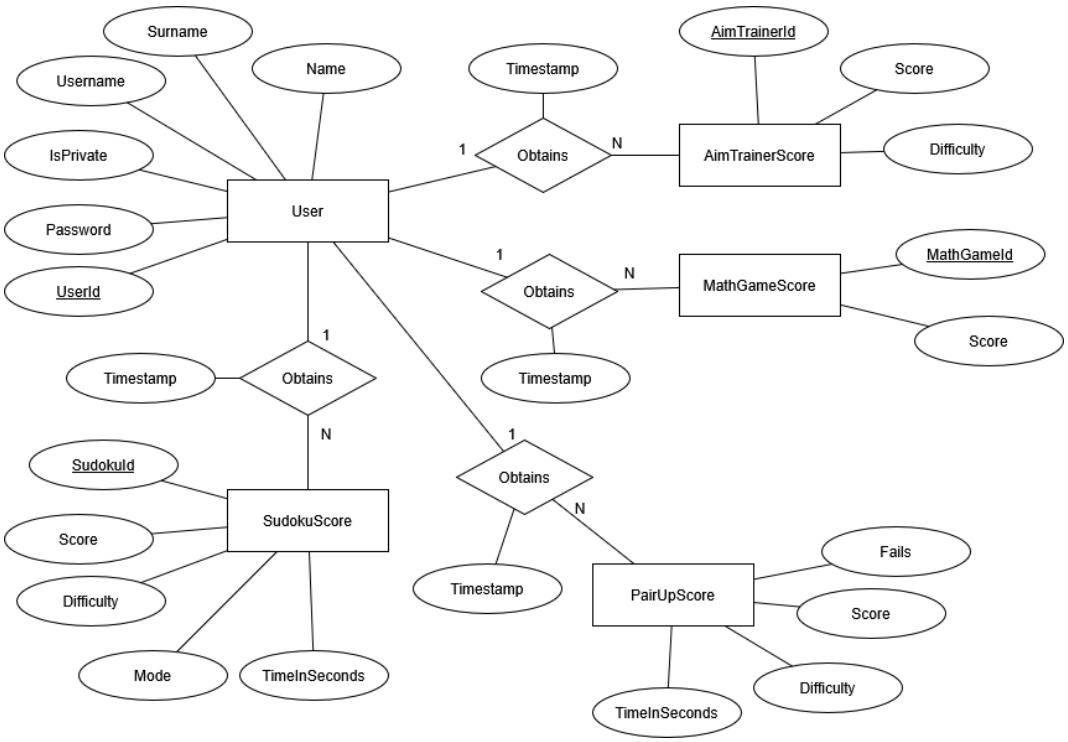

Figure 8: ER Diagram

In Figure 8, the database structure of the system is displayed using an Entity-Relationship (ER) diagram. The diagram represents the entities (tables), their attributes, and the relationships between them. The database consists of five tables, each representing a key component of the system: User, AimTrainerScore, MathGameScore, PairUpScore, SudokuScore. Each score table has a direct relationship with the User table. Since a user can play multiple games and record multiple scores, the relationships are one-to-many. The Timestamp attribute is recorded directly within each score table.

3 Development View

3.1 Package Diagram

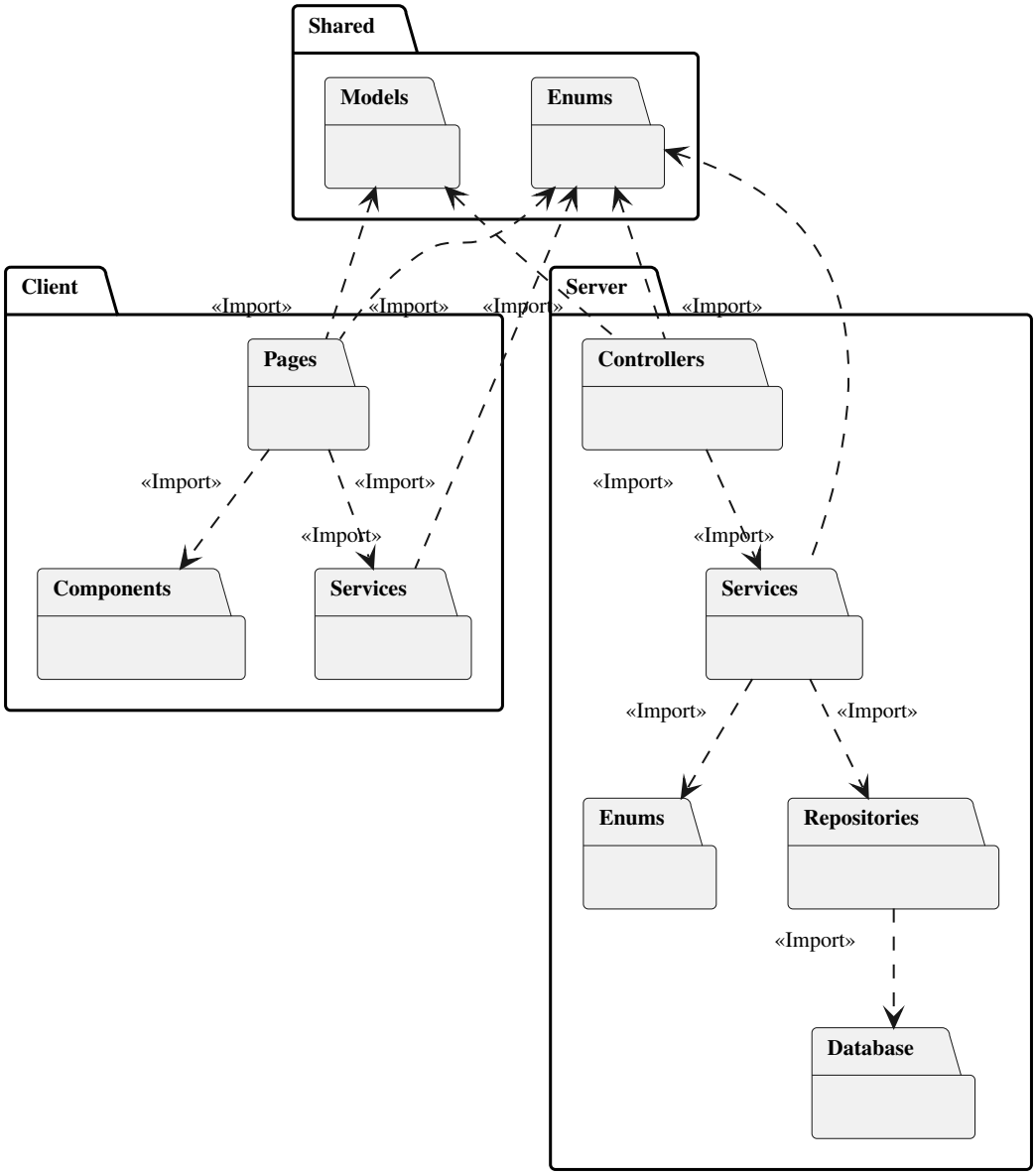

Figure 9: Package Diagram

Figure 9 provides an overview of how the codebase has been organized and divided into packages. This diagram includes some parts of the system that were not mentioned in the class diagram, such as Shared/Enums, Client/Components, and Server/Database. This decision was made in order to simplify the class diagram while maintining the key logical elements.

The diagram is divided into Client, Server, and Shared folders. The Shared package highlights common elements like data models and enumerations that are used across both the

frontend and backend, ensuring consistency and reducing redundancy. The arrows indicate dependencies, illustrating how functionality is physically imported from one package to another

3.2 Component Diagrams

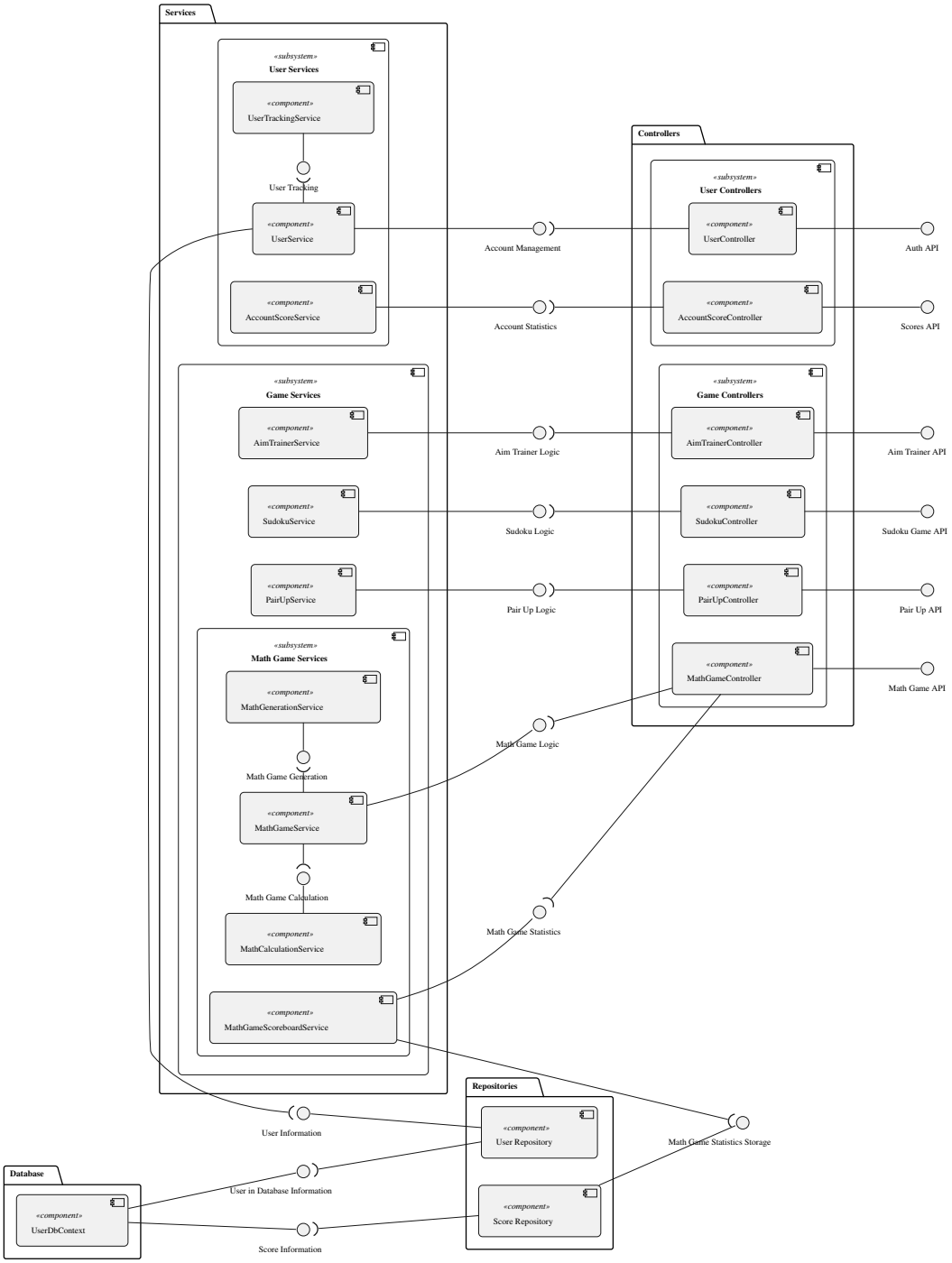

Figure 10: Server Component Diagram

Figure 10 illustrates the architecture of components from the Server package. Controllers provide API endpoints for frontend services, with each controller communicating with its corresponding service layer through well-defined interfaces. Services encapsulate the core business logic including game mechanics, user tracking, and account management. These services interact with other necessary services or repositories as needed. Data access is handled through repositories that abstract database operations.

The following outlines the mapping between components from a server component diagram and corresponding classes in a class diagram, organized by their respective subsystems or folders:

1. **Services**
    a. **User Services Subsystem**
        - **Component:** `UserTrackingService`
            - **Equivalent Class:** `UserTrackingService`
        - **Component:** `UserService`
            - **Equivalent Class:** `UserServices`
        - **Component:** `AccountScoreService`
            - **Equivalent Class:** `accountScoreService`

    b. **Game Services Subsystem**
        - **Component:** `AimTrainerService`
            - **Equivalent Class:** `AimTrainerService`
        - **Component:** `SudokuService`
            - **Equivalent Class:** `SudokuService`
        - **Component:** `PairUpService`
            - **Equivalent Class:** `PairUpService`

    c. **Math Game Services Subsystem**
        - **Component:** `MathGameServices`
            - **Equivalent Class:** `MathGameService`
        - **Component:** `MathGenerationService`
            - **Equivalent Class:** (Not in Class Diagram)
        - **Component:** `MathCalculationService`
            - **Equivalent Class:** (Not in Class Diagram)
        - **Component:** `MathGameScoreboardService`
            - **Equivalent Class:** (Not in Class Diagram)

2. **Controllers**
    a. **User Controllers Subsystem**

- **Component:** `UserController`
  - **Equivalent Class:** `UserController`
- **Component:** `AccountScoreController`
  - **Equivalent Class:** `AccountScoreController`

b. Game Controllers Subsystem

- **Component:** `AimTrainerController`
  - **Equivalent Class:** `AimTrainerController`
- **Component:** `SudokuController`
  - **Equivalent Class:** `SudokuController`
- **Component:** `PairUpController`
  - **Equivalent Class:** `PairUpController`
- **Component:** `MathGameController`
  - **Equivalent Class:** `MathGameController`

3. Repositories Folder

- **Component:** `User Repository`
  - **Equivalent Class:** `UserRepository`
- **Component:** `Score Repository`
  - **Equivalent Class:** `ScoreRepository`

4. Database Folder

- **Component:** `UserDbContext`
  - **Equivalent Class:** (Not in Class Diagram)

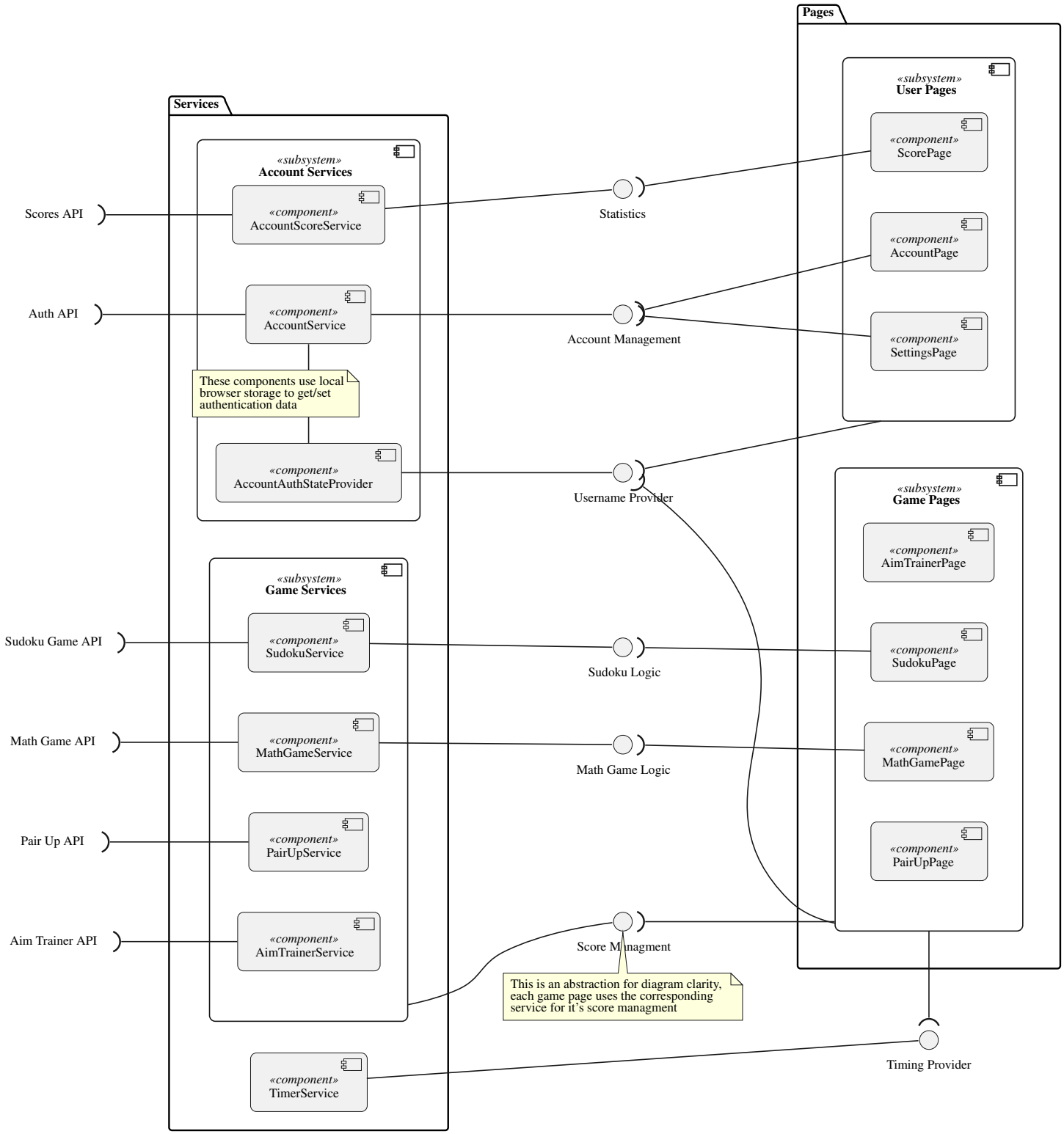

Figure 11: Client Package Component Diagram

Figure 11 visualizes the components inside the Client package. The pages are divided into two subsytems: User Pages - for handling client-side authentication and account

managment related views, Game Pages - for pages containing game implementations. The Page components interface with components from both the Account Services and Game Services subsystems. The service components are responsible for storing game or account related data locally, substituting parts of game logic implementation or making HTTP requests to the Web API. Hence they require HTTP API Endpoints for their functionality.

The following outlines the mapping between components from a client component diagram and corresponding classes in a class diagram:

1. **Services**
    a. **Timer Service**
        - **Component:** `TimerService`
            - **Equivalent Class:** `TimerService`

    b. **Account Services Subsystem**
        - **Component:** `AccountScoreService`
            - **Equivalent Class:** `AccountScoreService`
        - **Component:** `AccountService`
            - **Equivalent Class:** (Not in Class Diagram)
        - **Component:** `AccountAuthStateProvider`
            - **Equivalent Class:** (Not in Class Diagram)

    c. **Game Services Subsystem**
        - **Component:** `SudokuService`
            - **Equivalent Class:** `SudokuService`
        - **Component:** `MathGameService`
            - **Equivalent Class:** `MathGameService`
        - **Component:** `PairUpService`
            - **Equivalent Class:** `PairUpService`
        - **Component:** `AimTrainerService`
            - **Equivalent Class:** `AimTrainerService`

2. **Pages**
    a. **User Pages Subsystem**
        - **Component:** `ScorePage`
            - **Equivalent Class:** `StatisticsPage`
        - **Component:** `AccountPage`
            - **Equivalent Class:** `AccountPage`
        - **Component:** `SettingsPage`
            - **Equivalent Class:** `SettingsPage`

b. Game Pages Subsystem

- **Component:** `AimTrainerPage`
  - **Equivalent Class:** `AimTrainerPage`
- **Component:** `SudokuPage`
  - **Equivalent Class:** `SudokuPage`
- **Component:** `MathGamePage`
  - **Equivalent Class:** `MathGamePage`
- **Component:** `PairUpPage`
  - **Equivalent Class:** `PairMatchingGamePage`

4 Process View

4.1 Sequence Diagrams

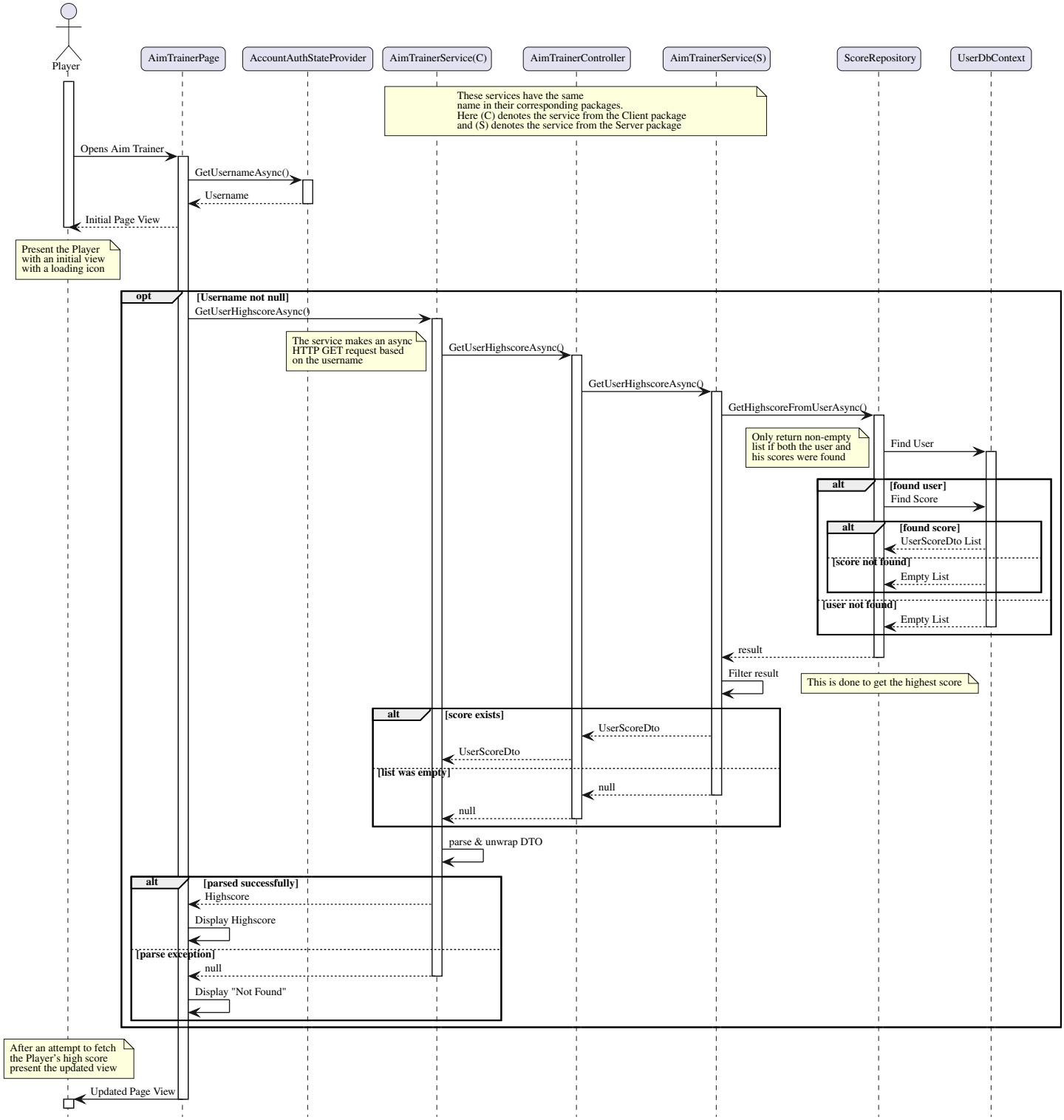

Figure 12: *Aim Trainer* Page Loading Sequence

In order to visualize the way in which score fetching works for all of our games we will do a case analysis on the *AimTrainerPage* component. Figure 12 illustrates the process flow when a player opens the Aim Trainer page. Initially, the page immediately requests the username from the authentication provider component. After this, an initial page is shown to the user with an icon to indicate the fact that scores are being loaded. Once the username is retrieved and confirmed not to be null, the page initiates an asynchronous call to the client-side service to fetch the player's high score. The client service makes an HTTP GET request to an endpoint provided by the Aim Trainer controller, which, in turn, invokes the server-side service. The server service interacts with the Score Repository and User Database Context to locate the corresponding user and their scores. Depending on whether the user and score exist, the repository returns either a list of scores or an empty list. The server service then filters the result to extract the highest score, and this filtered result is passed back up the chain and sent back as JSON (if a score was found) and an empty response otherwise. After parsing and unwrapping the data on the client side, the Aim Trainer page updates its display, showing either the retrieved high score or a "Not Found" message if no score was found. Finally, the updated view is presented to the player. This is the way all of our games implement the score fetching functionality.

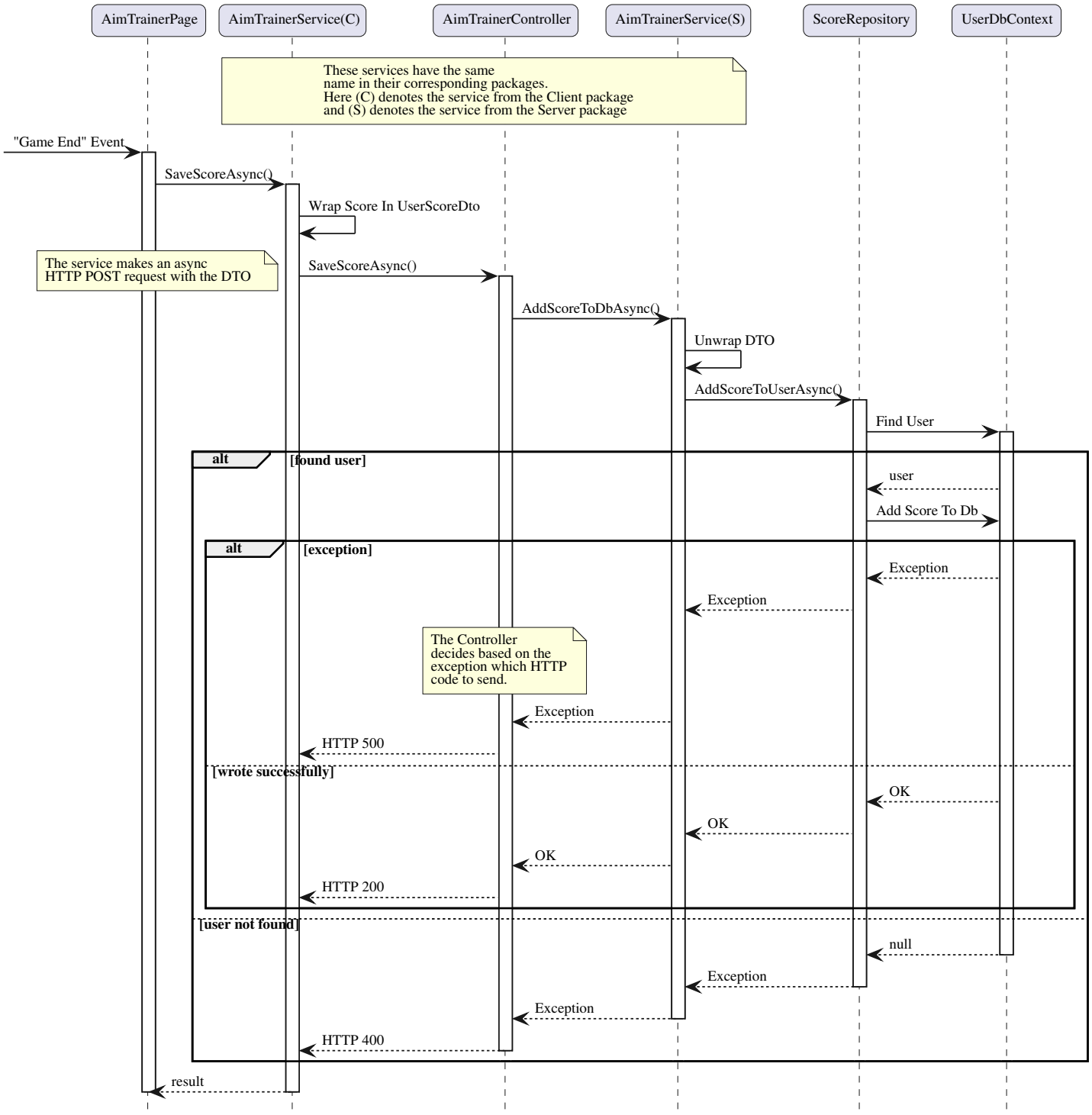

Figure 13: Aim Trainer Score Setting Sequence

This sequence diagram (Figure 13) is an example of the process initiated by finishing a play session in one our games **WHILE AUTHENTICATED** (otherwise no such process occurs on the client side). Once again we analyze only the AimTrainerPage, however other pages operate in a similar way. Once the game ends, the page calls the client-side service

to save the player's score asynchronously. The client service wraps the score within a DTO and forwards it via an HTTP POST request to an API endpoint provided by the Aim Trainer controller. The controller then uses the server-side service component to process the request. The server-side service unwraps the DTO and uses an interface provided by the Score Repository, which attempts to find the corresponding user within the given database context. If the user is found, the repository attempts to add the score to the database; if an exception occurs during this process it is caught by the controller and the controller decides the appropriate HTTP error code to return (HTTP 500 for DB exceptions, or HTTP 400 when a user is not found, indicating that a deceptive request might have been sent). If the user is found, and the insertion process is successful the controller returns a HTTP 200 OK code. Finally, the result is propagated back through the client service to the Aim Trainer page, updating the UI accordingly.

4.2 Activity Diagram

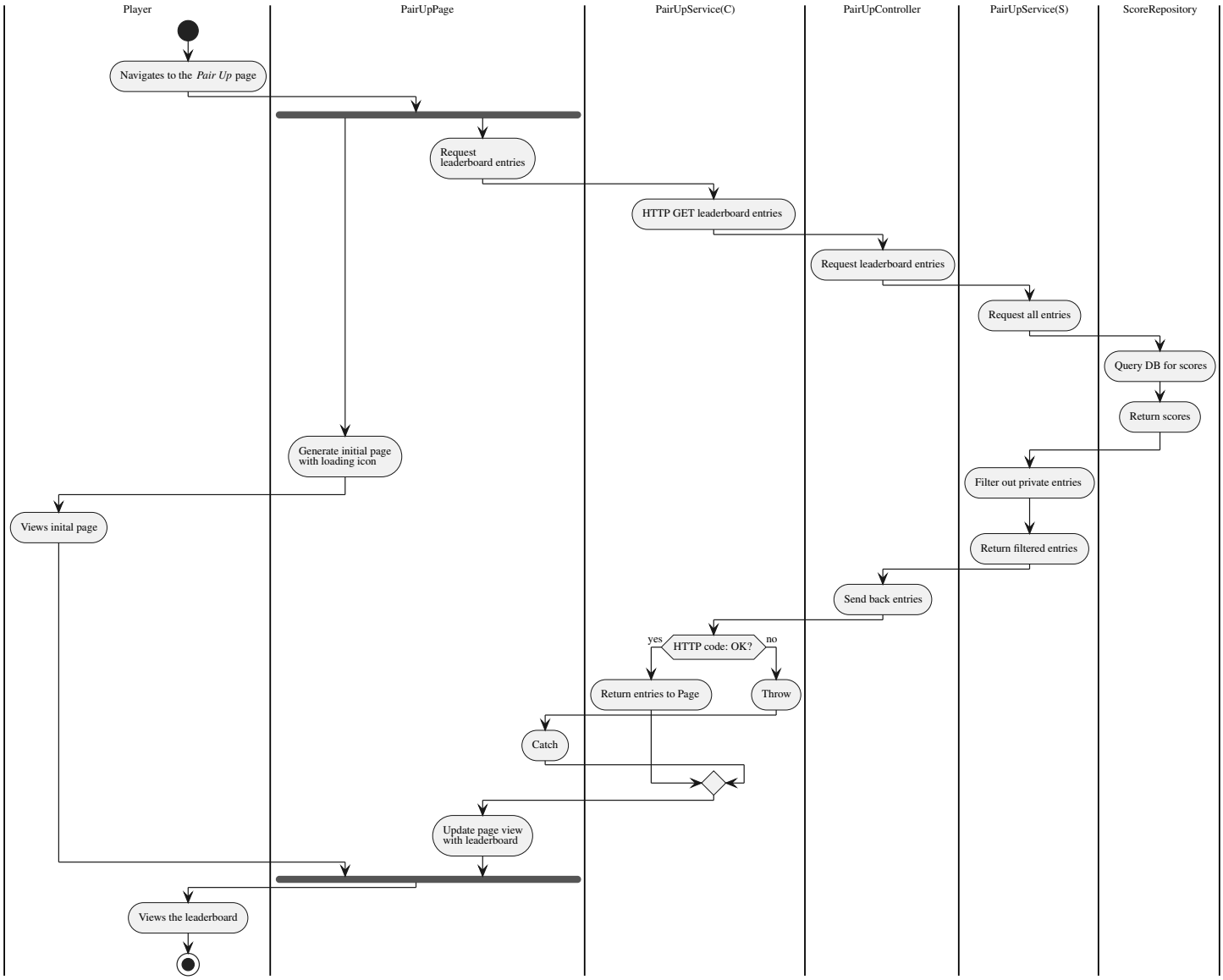

Figure 14: Leaderboard Loading Diagram

To show the way in which leaderboard scores get loaded by the components in the Game Pages subsystem we show an example of how the PairUpPage component executes this functionality (Fig. 14. This process is initiated by a Player navigating to the Pair Up game page. The PairUpPage component then handles showing the Player the needed loading icons and making an asynchronous call chain to fetch the leaderboards from the API in parallel. The HTTP request is sent by the PairUpService(C) (C - indicating the client side service) component. The PairUpController component handles receiving the request and passing it along to the PairUpService(S) (S - indicating the server side service). PairUpService(S) then receives the leaderboard entries by using the ScoreRepository component and filters out the

entries of players which have chosen to hide their high scores. It then returns the entries to the controller which sends a response to the initial HTTP request. The PairUpService(C) parses the response and throws an exception if it did not receive a valid response. The PairUpPage component then handles logic for updating the final page view.

5 Physical View

5.1 Deployment

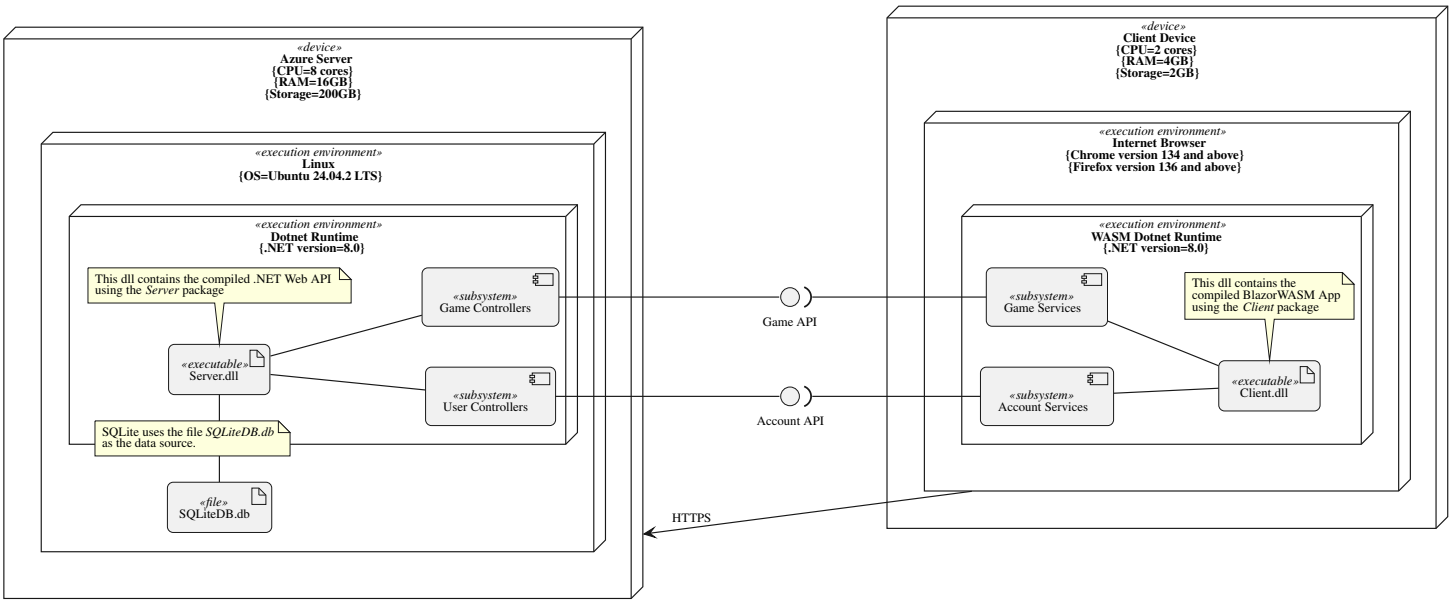

Figure 15: Deployment Diagram

Figure 15 illustrates the deployment of the web application, which consists of a client-side Blazor WebAssembly (WASM) front-end app accessed from a browser and a back-end Web API hosted on an Azure server. The client browser runs the provided .NET WASM runtime, running Client.dll as an executable. The backend operates in a Linux (Ubuntu) environment within Azure, where the .NET runtime hosts the Server.dll executable. The Server.dll contains an entry point for the Web API, the Client.dll contains an entry point for the Blazor App. Data is persisted using an SQLite database (SQLite is built into the Server.dll). Communication between the client and server occurs over HTTPS, ensuring secure data transmission. This design enables a lightweight client while leveraging cloud infrastructure for back-end processing and storage of user and game data.

5.2 CI/CD

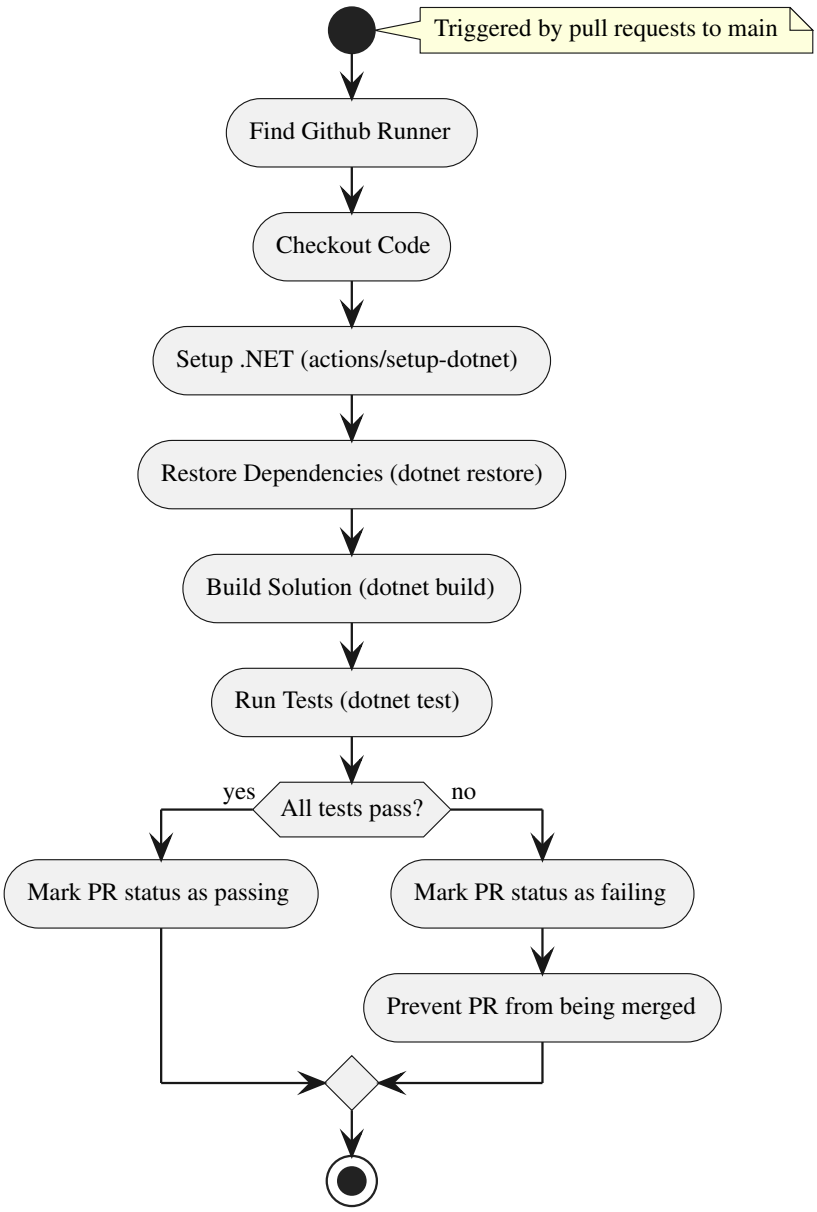

Figure 16: CI Pipeline

Figure 16 focuses on our pull request (PR) validation workflow. Triggered by any pull request targeting the main branch, this pipeline starts by setting up a linux environment (latest Ubuntu) and executing a series of steps: checking out the code, setting up the .NET environment, restoring dependencies, building the solution, and running tests. The critical decision point here assesses whether all tests pass. If they do, the PR status is marked as passing, signaling readiness for review and potential merge. If any tests fail, the pipeline marks the PR as failing and prevents it from being merged, thus upholding code quality and integrity before changes are integrated into the main branch.

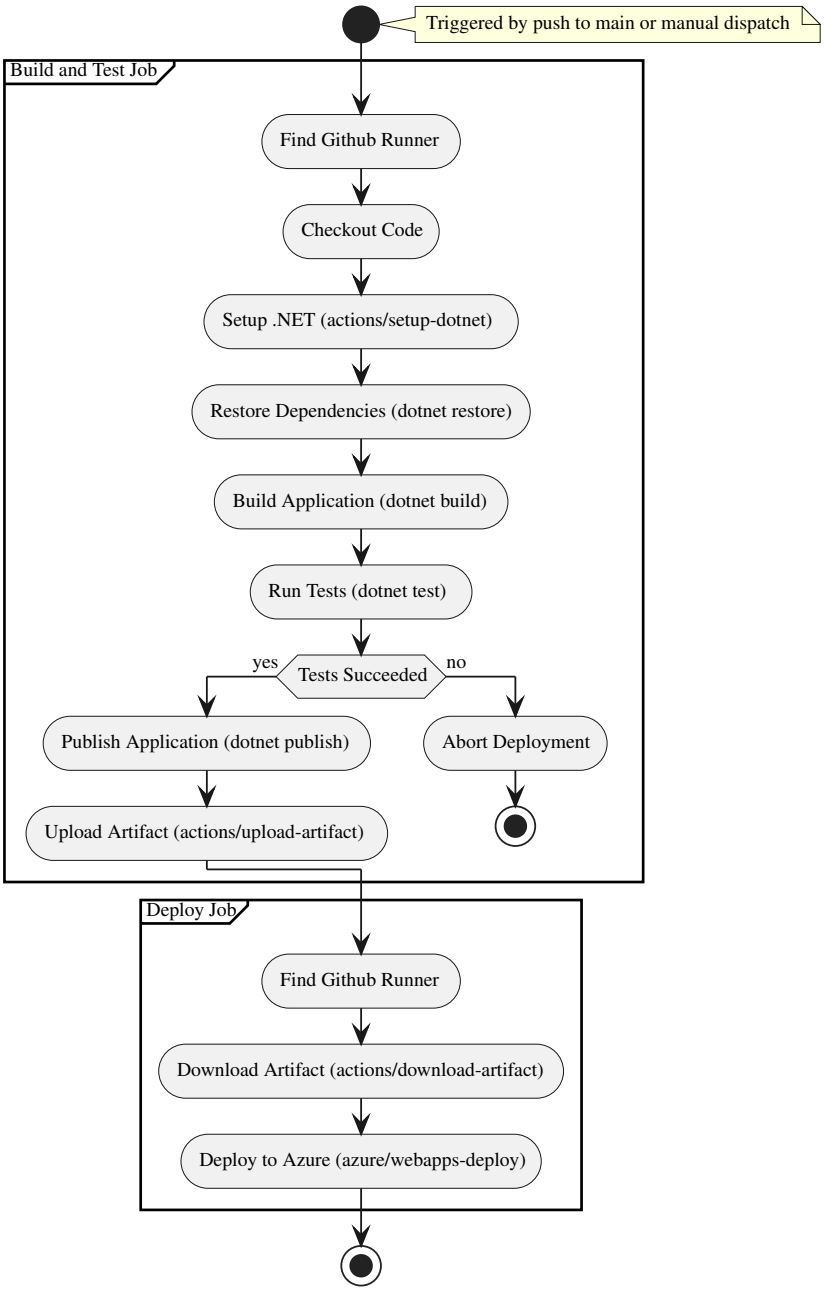

Figure 17: CD Pipeline

Figure 17 illustrates our CD pipeline using GitHub Actions. The whole process is excecuted on a linux(latest Ubuntu) environment. The process begins when a push to the main branch or a manual trigger starts the pipeline. Within the "Build and Test Job" partition, the pipeline checks out the repository, sets up the .NET environment, restores dependencies, and builds the application. After running tests, the workflow evaluates the outcome: if the tests succeed, it proceeds to publish the application and upload the resulting artifact; if any test fails, the deployment is aborted immediately. Following a successful build and test phase,

the "Deploy Job" partition retrieves the artifact from the previous step and deploys it to Azure, ensuring that only verified code is promoted to production.

6 Use Case View

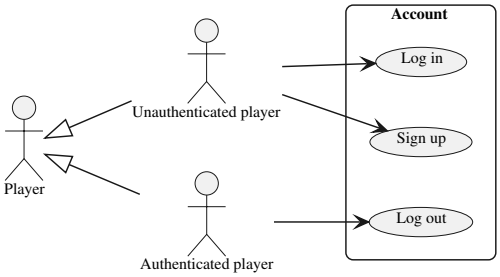

Figure 18: Account Use Case

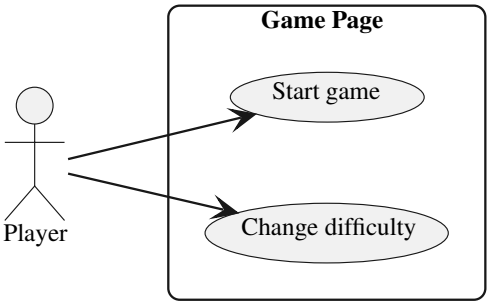

Figure 19: Game Page

Figure 18 shows how players interact with the Account module. Unauthenticated players can log in or sign up, while authenticated players can log out.

Figure 19 outlines two key actions: starting a game and changing the difficulty level. This design ensures that players have direct control over initiating gameplay and tailoring the challenge to their preferences.

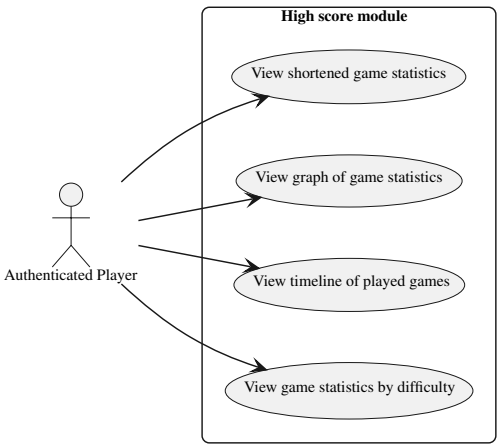

Figure 20: High Score Module

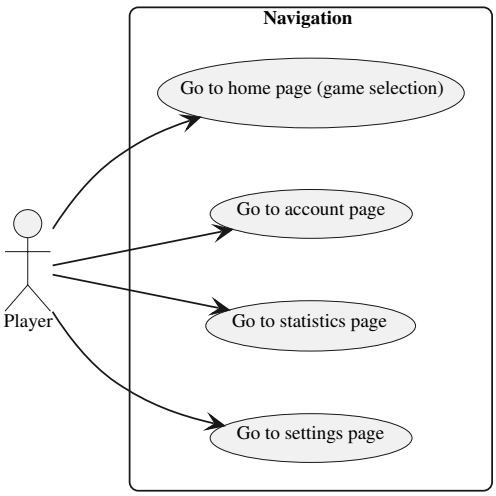

Figure 21: Page Navigation

Figure 20 details various ways for authenticated players to review game performance. Users can view short game statistics, detailed graphs, timelines of past games, and statistics filtered by difficulty. The module provides multiple perspectives to help players track and analyze their performance.

Figure 21 maps out the main paths for navigating through the website. Players can easily switch between the home page (which doubles as the game selection page), the account page, and the statistics page, ensuring a user-friendly and integrated browsing experience.

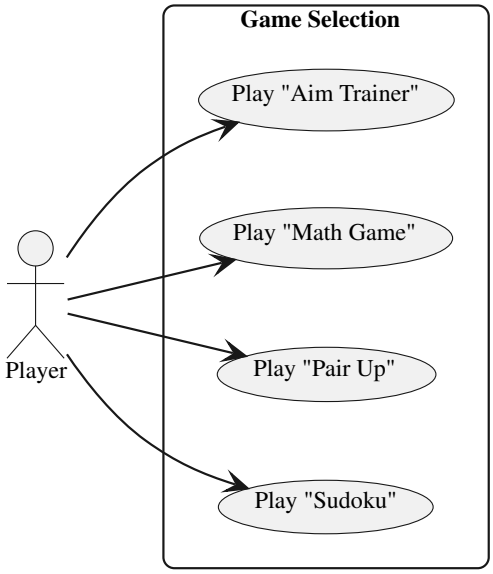

Figure 22: Game selection

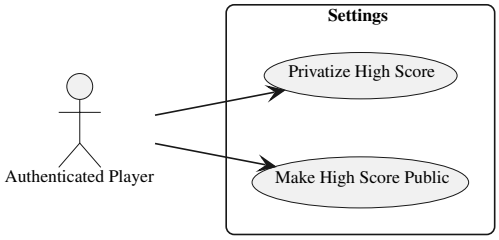

Figure 23: Account Settings

Figure 22 illustrates the available games that players can choose from. Players can select one of four games: Aim Trainer, Math Game, Pair Up, or Sudoku. This selection allows users to navigate to a game page based on their preferences.

Figure 23 shows the settings available to authenticated players. Users can choose to make their high scores public or keep them private. This feature provides control over personal game performance visibility, enhancing user privacy options.

7 Traceability

Use Cases	Coverage
Figure 18 Account Use Case	
Log in	Figure 3 - *Shows state transitions between authenticated and unauthenticated states, including login process*
Sign up	*Not explicitly covered in the diagrams*
Log out	Figure 3 - *Shows transition from authenticated to unauthenticated state during logout*
Figure 19 Game Page Use Case	
Start game	*Not explicitly covered in the diagrams*
Change Difficulty	*Not explicitly covered in the diagrams*
Figure 20 High Score Module Use Case	
View shortened game statistics	Figure 2 - *Handles asynchronous score retrieval and display process* Figure 11 - *Details frontend components for displaying statistics*
View graph of game statistics	Figure 2 - *Fetches score data for graphical representation* Figure 11 - *Shows UI components for graph rendering*
View timeline of played games	Figure 2 - *Retrieves historical score data for timeline display* Figure 11 - *Details components for timeline visualization*
View game statistics by difficulty	Figure 2 - *Handles filtered score data by difficulty level* Figure 11 - *Includes filter components for statistics display*

Figure 21 Page Navigation Use Case	
Go to home page (game selection)	Figure 1 - *Shows page relationships including HomePage component* Figure 11 - *Shows UI routing components*
Go to account page	Figure 1 - *Shows AccountPage component and relationships* Figure 11 - *Shows account page components*
Go to statistics page	Figure 1 - *Shows StatisticsPage component and relationships* Figure 11 - *Details statistics page UI components*
Go to settings page	Figure 1 - *Shows SettingsPage component and relationships* Figure 11 - *Shows settings page UI components*
Figure 22 Game Selection Use Case	
Play "Aim Trainer"	Figure 5 - *Details game states and transitions for the Aim Trainer game*
Play "Math Game"	Figure 4 - *Shows states and transitions for the Math Game*
Play "Pair Up"	Figure 7 - *Illustrates memory matching game states and logic flow*
Play "Sudoku"	Figure 6 - *Details puzzle game states and user interactions*
Figure 23 Account Settings Use Case	
Privatize High Score	*Not explicitly covered in the diagrams*
Make High Score Public	*Not explicitly covered in the diagrams*

Note: The current system architecture does not fully implement some user needs and requirements as indicated. Use cases marked with *"Not explicitly covered in the diagrams"* indicate architectural elements that may need additional documentation or implementation. All other requirements have corresponding architectural components as detailed above.