MythBusters Software Design Document

Batuhan Efe Alkı¸s, Doruk Demirci, G¨okc¸en Enli, Bug˘ra I¸sın, Zeynep Yavuz

July 23, 2025

**Updated Software Design Document**

MythBusters Team July 23, 2025

# Contents

1. [System Overview](#_bookmark0) 2
2. [System Context](#_bookmark1) 2
3. [Key Features and Functionality](#_bookmark2) 2
4. [Assumptions and Dependencies](#_bookmark3) 2
5. [Architectural Design](#_bookmark4) 3
6. [Component Design](#_bookmark5) 3
7. [Data Design](#_bookmark6) 3
8. [Design Patterns](#_bookmark7) 4
9. [Implementation Notes](#_bookmark8) 4
10. [User Interface Design](#_bookmark9) 5
11. [External Interfaces](#_bookmark10) 5
12. [Performance Considerations](#_bookmark11) 5
13. [Error Handling and Logging](#_bookmark12) 6
14. [Design for Testability](#_bookmark13) 6
15. [Deployment and Installation Design](#_bookmark14) 6
16. [Change Log](#_bookmark15) 6
17. [Future Work / Open Issues](#_bookmark16) 6

1

# System Overview

The MythBusters application is a gamified web platform aimed at educating users about health myths through interactive gameplay. The system offers multiple mini-games, including a Car Racing Game, a Balloon Popping Game, and Hangman, each designed to test the user's knowledge in a fun and engaging way. A key aspect of the system is its ability to adapt the gameplay experience to the user's preference by offering different difficulty levels. This is achieved by dynamically altering game rules and parameters, such as scoring multipliers and opponent behavior. The initial entry point for any user is the profile creation process, which personalizes their experience and tracks their progress across the different games.

# System Context

**User Profile Creation**: This use case is initiated through a REST API endpoint. When a new user registers, their input is collected, validated, and used to construct a Profile entity. The backend service utilizes the **Builder** design pattern to create the data transfer object (DTO), assigns default avatars for the games, securely hashes the password, and stores the new user's data in the database.

**Dynamic Game Configuration**: This use case occurs on the frontend when a user selects a game and a difficulty level (e.g., "Easy," "Hard"). The application uses the **Factory** pattern to create a specific "strategy" object corresponding to the chosen difficulty. This strategy object, an implementation of the **Strategy** pattern, encapsulates the algorithms and parameters for that difficulty (e.g., opponent speed, question difficulty, scoring rules). The core game component then uses this strategy object to configure its behavior without being tightly coupled to the specific rules of any single difficulty level. This allows for flexible and easily extensible game mechanics.

# Key Features and Functionality

**User Management:**

* Collect user input (username, email, password) from a REST client.
* Validate input fields using annotations like @NotBlank, @Email, @Size.
* Use the **Builder** Pattern to construct the ProfileCreateDTO.
* Automatically assign default avatars for Race, Balloon, and Hangman games.
* Set a default profile image.
* Hash the password securely with PasswordEncoder.
* Save the completed profile entity to the database.

**Gameplay Mechanics:**

* Offer multiple, distinct game types (Car Race, Balloon Pop, Hangman).
* Allow users to select a difficulty level (e.g., Easy, Normal, Hard) before starting a game.
* Dynamically adjust game rules, scoring, and AI behavior based on the selected difficulty.
* Utilize the **Strategy** pattern to encapsulate the algorithms for each difficulty level, making them interchangeable.
* Employ the **Factory** pattern to simplify the creation of the correct game strategy object based on user input, decoupling the game screen from the concrete strategy implementations.

# Assumptions and Dependencies

**Backend:**

* The application is built using the Spring Boot framework.
* Spring Security is used for password encoding and authentication.
* Lombok is used for boilerplate code generation (e.g., for the Builder pattern).
* The Jakarta Validation API is used for input validation.
* A PostgreSQL relational database is used for data persistence.
* Default avatars with specific IDs are assumed to exist in the database.

**Frontend:**

* The front end is a single-page application (SPA), built using React, Emotion and Material UI.
* The **Strategy** and **Factory** patterns are implemented in TypeScript to manage game logic on the client side.
* The game components are designed to be decoupled from the specific game rule implementations, interacting with them only through a common strategy interface.

# Architectural Design

## Architecture Style

Layered (MVC)

## System Architecture Diagram

Controller → Service → Repository → Database

## Rationale

Ensures separation of concerns, makes testing and maintenance easier, and clearly defines responsibilities.

# Component Design

* + **Controller:** Handles HTTP POST /api/profiles, receives and validates JSON request body.
  + **DTO Layer:** Uses ProfileCreateDTO with an inner static Builder class for safe and readable construction.
  + **Service Layer:** Contains business logic for profile creation.
  + **Repository Layer:** Interacts with the database using ProfileRepository and

AvatarRepository.

# Data Design

## Data Entities

Profile (id, username, email, passwordHash, profilePhoto, raceGameAvatarID, bal- lonGameAvatarID, hangGameAvatarID)

## Data Validation

* + @NotBlank for username, email, and password
  + @Email for email field
  + @Size(min = 6) for password

## Data Flow

User → Controller → DTO → Service → Entity → Repository → DB

# Design Patterns

## Builder Pattern

**Location:** ProfileCreateDTO class

**Purpose:** Allows incremental construction of immutable DTOs from incoming re- quests.

1

Profile Create DTO dto = new Profile Create DTO . Builder ()

. username ( create DTO . getUsername ())

. email( create DTO . getEmail ())

. password ( create DTO . getPassword ())

. build ();

2

3

4

5

Listing 1: Builder Usage

### Benefits:

* Prevents telescoping constructors
* Enhances code readability
* Ensures only valid objects are built (internal validation)

**Strategy Pattern**

* **Location**: Frontend: CarRaceGameScreen.tsx, BalloonGameScreen.tsx, HangmanGameScreen.tsx and their corresponding strategy implementation files (e.g., CarRaceGameStrategy.ts).
* **Purpose**: To define a family of algorithms (in this case, different sets of game rules) and encapsulate each one, making them interchangeable. This allows the game's difficulty and behavior to be altered dynamically based on the user's selection, without changing the core logic of the game component itself. The component simply relies on the public interface of the strategy object to get the rules it needs.

**Benefits:**

* Flexibility: Easily add new difficulty levels (e.g., "Insane") by just creating a new strategy class.
* Separation of Concerns: The game's core logic is cleanly separated from the rules that define each difficulty.
* Simplified Maintenance: To adjust the balance of a specific difficulty, you only need to modify its corresponding strategy file.

**Factory Pattern**

* **Location**: Frontend: CarRaceStrategyFactory.ts, BalloonStrategyFactory.ts, HangmanStrategyFactory.ts.
* **Purpose**: To create objects without exposing the instantiation logic to the client. In this project, the Factory's role is to select and create the correct concrete strategy object based on a string input (e.g., "Easy", "Hard"). This decouples the game screen component from the concrete strategy classes, meaning the game screen doesn't need to know how to construct each type of strategy.

**Benefits:**

* Decoupling: The game screen does not need to import or know about EasyRaceStrategy or HardRaceStrategy. It only knows the factory.
* Centralized Control: The logic for which strategy to create is centralized in one place. If a strategy's constructor changes, only the factory needs to be updated.
* Simplified Client Code: The game screen's code is cleaner and simpler as it doesn't contain if/else or switch statements for object creation.

# Implementation Notes

During the user registration process, once the validated ProfileCreateDTO is constructed using the Builder Pattern, the backend service proceeds to construct the Profile entity, enriching it with default values. These include the default profile photo and three different avatar assignments for each game.

The implementation of this logic in the service layer ensures clean separation between data transfer and persistence logic. Below is the full code block in ProfileService class responsible for setting up the default profile configuration:

1

// Step 1: Create and populate the Profile entity

Profile profile = new Profile (); profile . setUsername ( create DTO . getUsername ()); profile . setEmail( create DTO . getEmail ()); profile . setPassword Hash ( hashed Password );

// Step 2: Assign default profile photo profile . setProfile Photo (" https :// www . w 3 schools. com / howto / img\_avatar. png

");

// Step 3: Assign default avatars for each game type profile . setRace Game Avatar (

avatarRepository . find ById (1)

. orElse Throw (() -> new IllegalArgumentException (" Default race avatar b u l u n a m a d "))

);

profile . setBaloon Game Avatar ( avatarRepository . find ById (3)

. orElse Throw (() -> new IllegalArgumentException (" Default baloon avatar b u l u n a m a d "))

);

profile . setHangman Game Avatar (

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

avatarRepository . find ById (5)

. orElse Throw (() -> new IllegalArgumentException (" Default hangman avatar b u l u n a m a d "))

);

20

21

22

Listing 2: Default Profile Initialization and AvatarAssignment  
This code ensures:

* Every user is initialized with a valid profile image and playable avatars.
* The system avoids null values in the Profile entity for these fields.
* Runtime failures are handled gracefully using orElseThrow(), which provides mean- ingful error messages if defaults are missing.
* **Listing: Strategy Pattern Usage in CarRaceGameScreen.tsx**

// The game component uses a strategy object to get its rules.

// It doesn't know or care if it's an 'Easy' or 'Hard' strategy.

// Get a strategy object from the factory

const strategy = CarRaceStrategyFactory.createStrategy(difficulty);

// Use the strategy's interface to retrieve game parameters

const scoreMultiplier = strategy.getPlayerScoreMultiplier();

const computerMoveInterval = strategy.getComputerMoveInterval(mode);

const computerAccuracy = strategy.getComputerCorrectChance();

// These values are then used in the game's logic

const handleAnswer = (selected: string) => {

// ...

if (isCorrect) {

// The score calculation uses the value from the strategy

setScore(s => s + 10 \* scoreMultiplier + diff \* 2);

}

// ...

};

* **Listing: Factory Pattern Implementation (CarRaceStrategyFactory.ts)**

// This is the implementation inside the Factory file.

import { CarRaceStrategy, EasyRaceStrategy, NormalRaceStrategy, HardRaceStrategy } from './CarRaceGameStrategy';

export class CarRaceStrategyFactory {

public static createStrategy(difficulty: string): CarRaceStrategy {

switch (difficulty) {

case 'Easy':

return new EasyRaceStrategy();

case 'Normal':

return new NormalRaceStrategy();

case 'Hard':

return new HardRaceStrategy();

default:

// Default to a standard strategy if the input is unknown

return new NormalRaceStrategy();

}

}

}

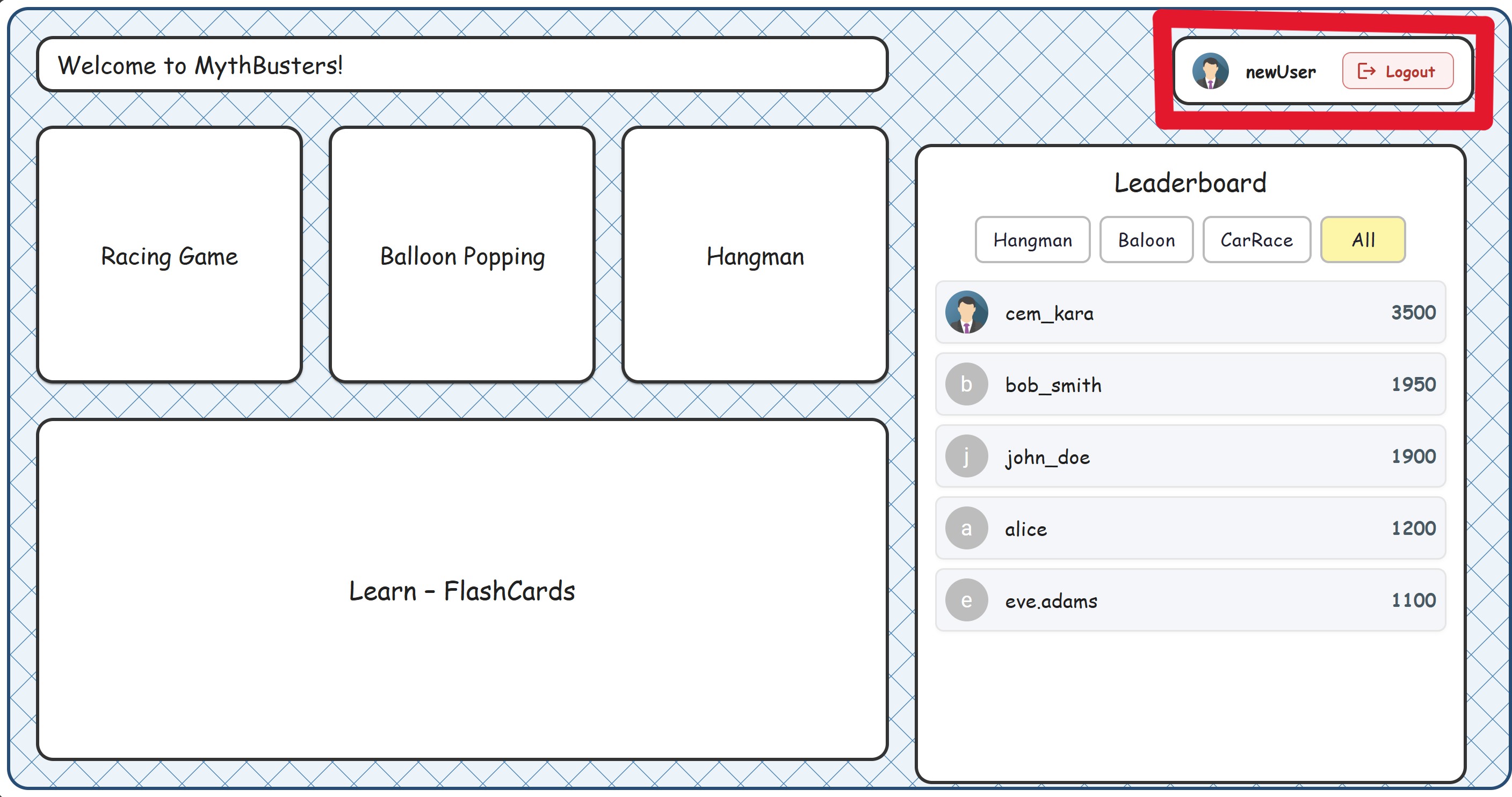
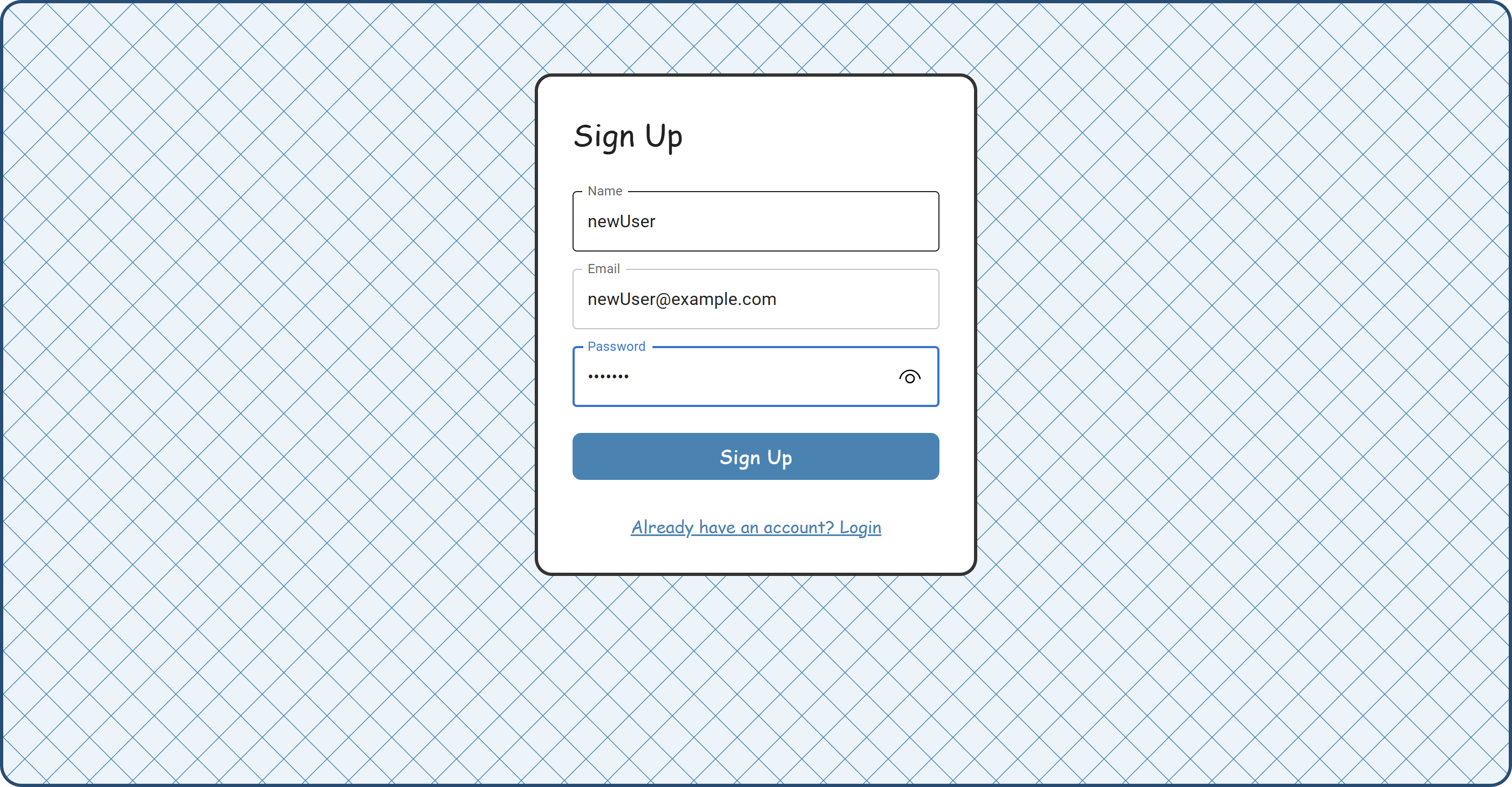
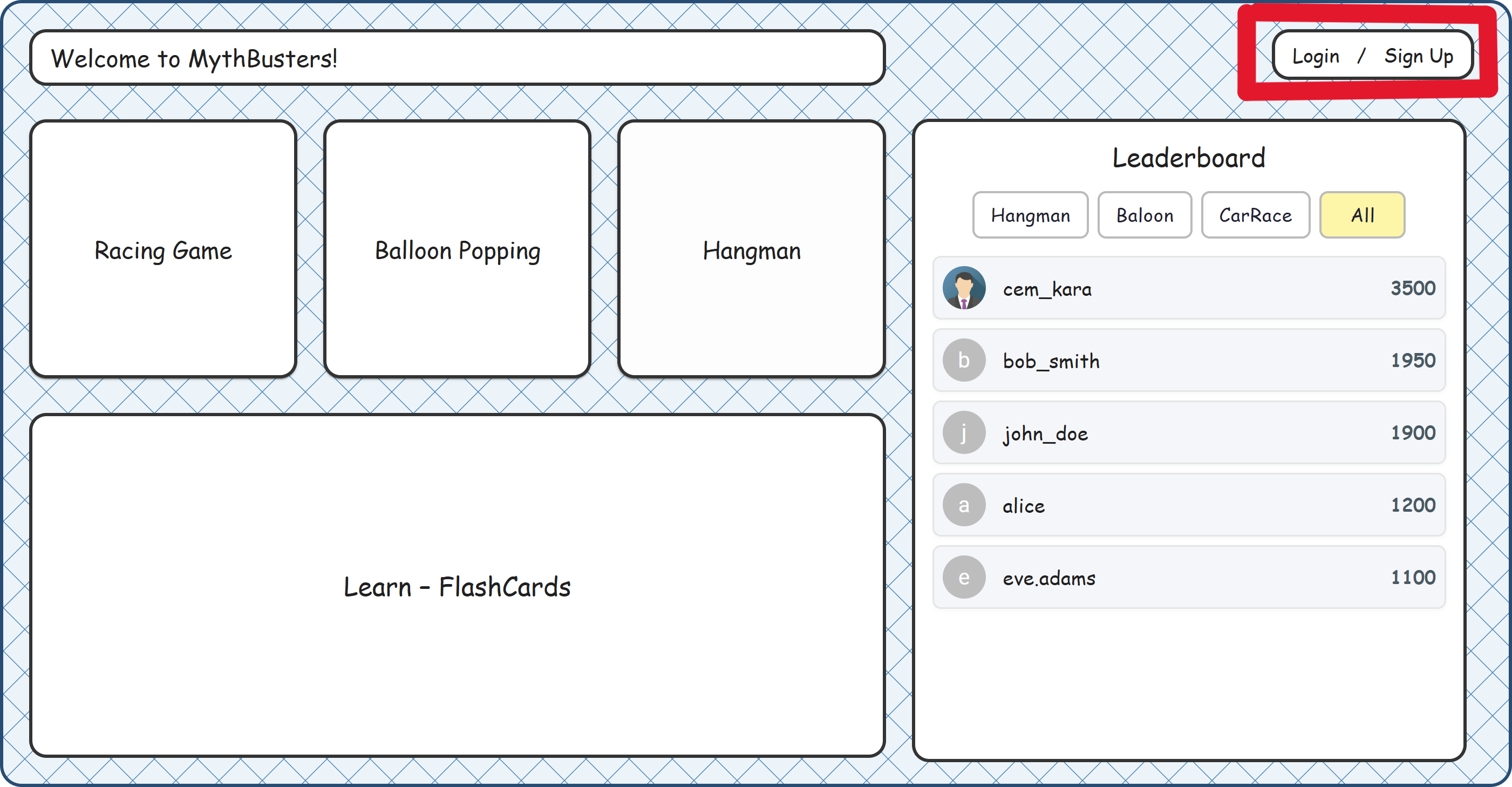
// This is how the Factory is used inside CarRaceGameScreen.tsx

// The component simply calls the static method to get the object it needs.

const strategy = CarRaceStrategyFactory.createStrategy(difficulty);

# User Interface Design

Although this use case is primarily a JSON REST API, a minimal front-end interface is provided for demonstration.



(a) Before sign-up (b) Sign-up form (c) After sign-up

Figure 1: UI flow for the registration process

# External Interfaces

### Endpoint:

POST /api/profiles

### Sample JSON Input:

1

{

" username ": " john\_doe ", " email ": " john@ example . com ", " password ": " secret123 "

}

2

3

4

5

# Performance Considerations

* + Lightweight operation (single DB insert)
  + Three avatar fetch queries
  + No loops or external API calls

# Error Handling and Logging

* + Validation errors return 400 with standard Spring handling
  + Runtime exceptions (e.g., avatar not found) return 500

# Design for Testability

* + Builder allows test DTO creation without JSON
  + Service is testable in isolation

# Deployment and Installation Design

**Backend:**

* + Spring Boot JAR
  + application.properties for DB config

**Frontend:**

* + npm install to install necessary packages
  + npm start to start the frontend application

# Change Log

* + Initial version created on 30.06.2025
  + Focused exclusively on profile creation with Builder pattern
  + **23.07.2025:** Updated sections 1-4 to include frontend use cases for gameplay, detailing the use of Strategy and Factory patterns.

# Future Work / Open Issues

* + Add client-controlled avatar/profile image selection
  + Add profile editing
  + Add endpoint for available avatars