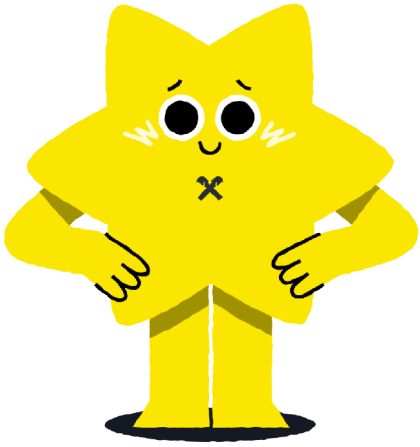


Challenge: Build the Future of Financial Literacy & Inclusion
Project title: RFMoneyMatters

Team: FOUREACH
04/05/2025
Tiranë, Albania
Version: 1.0

Table of Contents

1. Introduction

1.1 Problem Statement

Teenagers today lack engaging, age-appropriate tools to explore essential financial concepts like planning, budgeting, payment methods, and loans in a way that feels fun and relevant. As a result, they're neither motivated to practice smart money habits nor confident in making real-world financial decisions. This project seeks to fill that gap by delivering an interactive, personalized platform that turns financial learning into a playful, habit-forming experience for 12–17-year-olds.

By delivering a habit-forming, personalized platform grounded in youth-driven insights, RFMoneyMatters can become the go-to financial education app for teens. This not only drives high engagement and retention within a growing user segment, but also opens opportunities for partnerships with schools, financial institutions, and brands seeking to foster early financial literacy positioning RFMoneyMatters for strong market adoption and lasting social impact.

1.2 Objectives

The primary objective of this system is to deliver a safe, age-appropriate learning environment that guides users through curated lessons and quizzes, while enforcing content gating for those outside the ideal age range. In support of this, the system pursues the following specific goals:

- **Lesson Management**
  Provide an intuitive interface for organizing, browsing, and selecting modular lesson units.
- **Age-Based Content Gating**
  Automatically detect users under 12 or over 17 and display a tailored warning before granting access, ensuring compliance with intended audience guidelines.
- **Quiz & Challenge Delivery**
  Seamlessly present quizzes tied to each lesson, enforcing a question-by-question flow and preventing skips.
- **Progress Tracking & Reporting**
  Record each user's quiz responses, completion status, and performance metrics; expose dashboards for learners (to see their own progress) and for administrators/educators (to monitor performance).
- **Result Persistence**
  Safely store all results, enabling resume-where-you-left-off functionality and generating longitudinal progress reports.

1.3 Target Audience

- Adolescent Learners (Ages 12–17)
- Educators & Content Administrators
- Parents & Guardians

2. Requirements Analysis

2.1 Functional Requirements

User Management

- **Registration & Authentication:** Email/password + optional social/OAuth login.
- **Role Management:** Assign roles (Teen, Educator, Admin).
- **Age-Based Gating:** Detect users outside 12–17 and display a tailored warning before granting access.

Lessons & Quizzes Module

- **Lesson Catalog:** CRUD operations for lessons (title, type, content, difficulty).
- **Quiz Flow:** One‑question‑at‑a‑time, enforce answer before proceeding, no skipping.
- **Results Tracking:** Persist quiz results (score, completion flag, timestamp), allow resume-where-you-left-off.

Challenges & Rewards Module

- **Challenge Definitions:** Admins create challenges (title, category, date, description).
- **Gamification:** Award coins, badges, streak bonuses on completion.

Expense Tracking & Goals

- **Expenses:** Log entries (name, amount, date, category) per user.
- **Goals:** Users set targets, track ProgressPercentage, earn RewardCoins on completion.
- **Budget Simulation:** Interactive “What would you do with X lek?” exercises.

AI-Powered Virtual Assistant

- **Chat Interface:** Embedded widget.
- **Use Cases:** Instant FAQs, step-by-step budgeting help, quiz hints, goal-setting guidance.

Reporting & Analytics API

- **Progress Dashboards:** Learner view (personal stats) + Admin view (aggregate metrics).
- Enumeration of main modules (for administrators, users, virtual assistant, etc.).

2.2 Non-Functional Requirements

Performance:

- API response time < 200 ms under typical load.

- Page render time < 1 s on mid-range mobile devices.

Security:

- Secure authentication (JWT + refresh tokens in HTTP-only cookies).
- Role-based authorization via ASP.NET Identity.
- Protections: CSRF tokens, input sanitization (XSS), parameterized ORM queries (SQL Injection).

Scalability & Availability:

- Stateless services behind a load balancer.
- Horizontal scaling of API and chat-bot.
- Automated failover for the database.

Maintainability & Observability:

- Centralized logging (e.g. Application Insights, ELK).
- Distributed tracing (e.g. OpenTelemetry).
- Health checks and alerting for uptime and error rates.

2.3 Constraints

Technical:

- Use .NET 8 for backend, EF Core for ORM, Next JS for frontend, and PostgreSQL.
- CI/CD via GitHub Actions, deployment to AWS (EC2/ECS) or Vercel/Railway.

Time:

- Hackathon deadline: 48 hours to MVP.

Resources:

- Team of four developers.
- Limited design assets—rely on open-source UI kits (e.g. Tailwind UI).

3. System Design and Architecture

3.1 System Architecture

3.1.1 Overview: Layered Architecture

RFMoneyMatters adopts a classic n-tier (multi-layer) architecture to promote separation of concerns, scalability, and maintainability. The system is divided into three primary layers:

- **Presentation Layer** (User Interface)

  Implements all customer-facing components: web pages, forms and API endpoints exposed to clients.

  Responsible for handling HTTP requests, rendering views, and basic input validation.

  Communicates exclusively with the Application Layer via well-defined service interfaces.

- **Application Layer** (Business Logic Layer)

  Encapsulates all core business rules and workflow logic: lesson management, quiz scoring, challenge assignment, progress calculations, and virtual assistant orchestration.

  Coordinates transactions and data flow between the Presentation Layer and Data Access Layer.

  Implements validation, authorization checks, and integration with third-party services (e.g., ChatGPT API).

- **Data Access Layer**

  Abstracts persistence details and provides repository interfaces for CRUD operations on the database.

  Utilizes Entity Framework Core for ORM, queries, and migrations.

  Manages connection pooling, query optimization, and implements caching where needed.

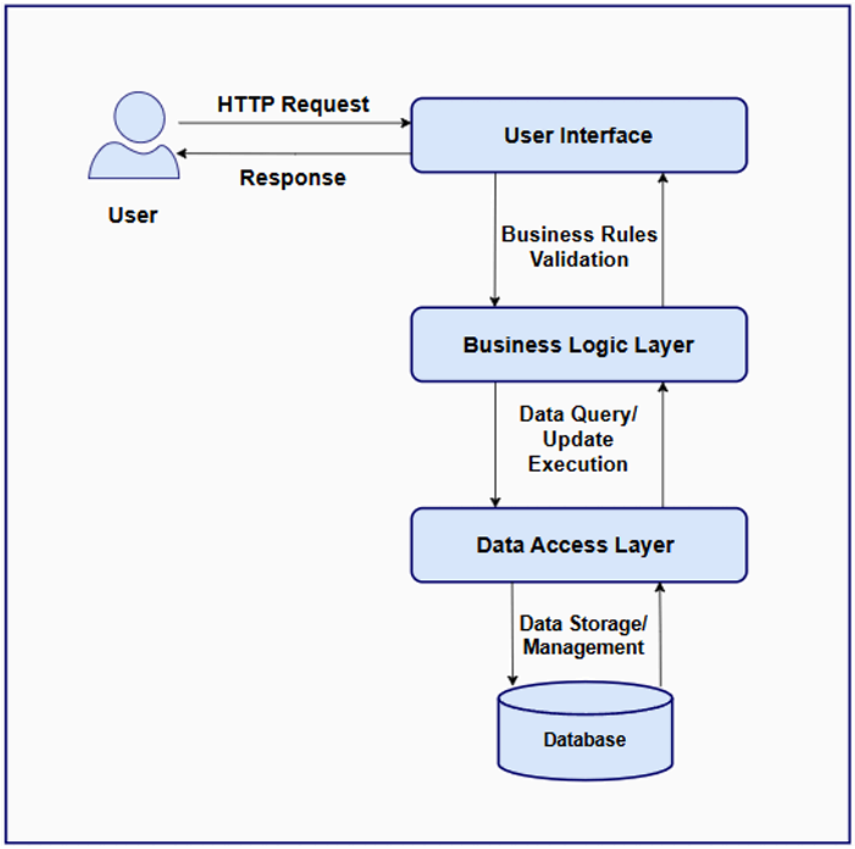


Figure 1. System Architecture Diagram

3.1.2 Data Flow and Communication

1. Client Request: User triggers an HTTP request (e.g., login, fetch lessons).
2. Presentation Layer: Controller or API endpoint receives the request, performs model binding and light validation.
3. Application Layer: Controller calls service classes which apply business rules, coordinate workflows, and call repository methods.
4. Data Access Layer: Repositories translate service calls into SQL queries/migrations against SQL Server.
5. Database: Executes queries, returns results back up through the layers.
6. Response: Processed data returns to the Presentation Layer, which serializes and sends the HTTP response back to the client.

This layered approach ensures each tier can be developed, tested, and scaled independently.

3.2 Database Modeling

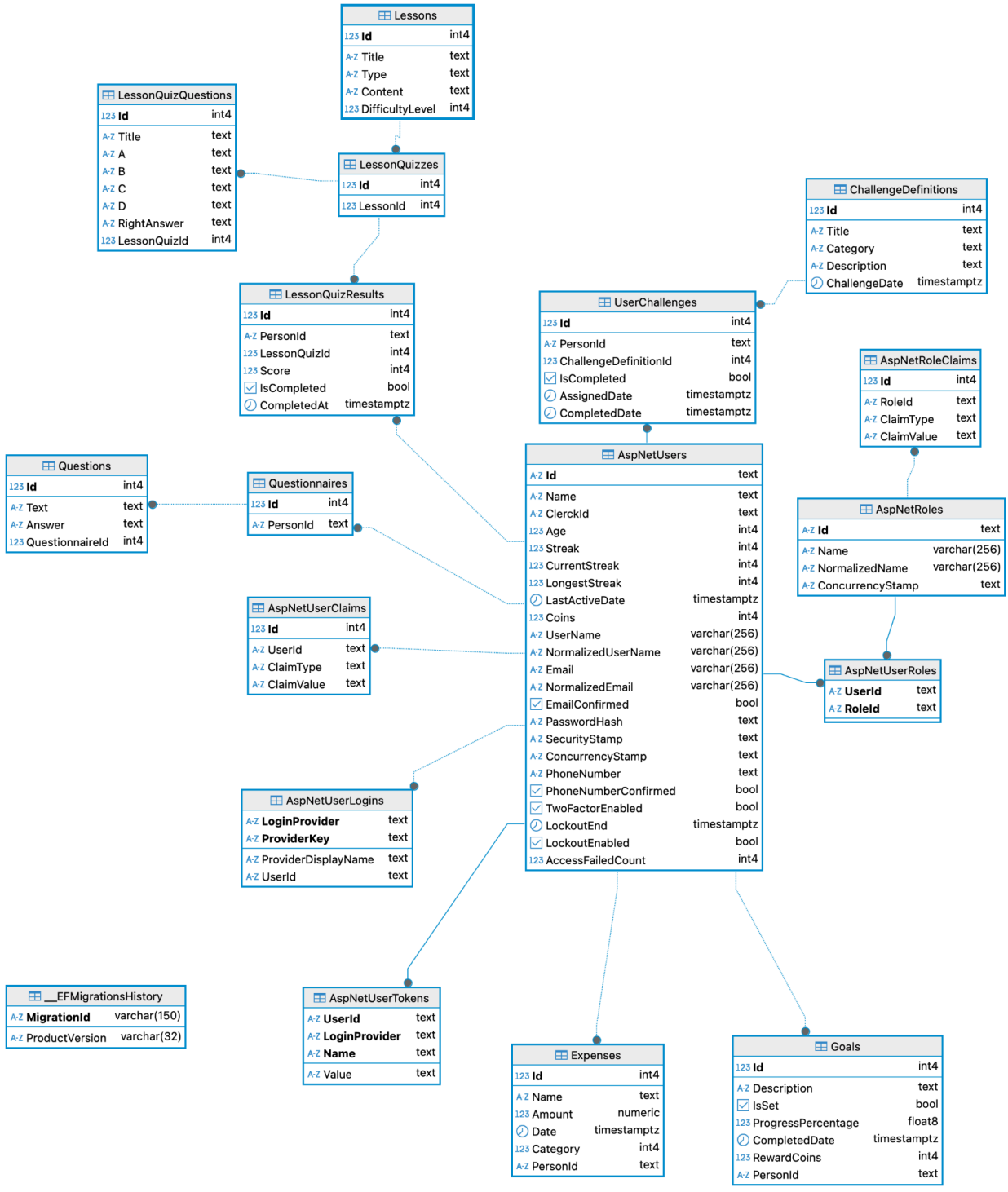


Figure 2. System Database Diagram

The ER diagram in Figure 2 illustrates the core data model for RFMoneyMatters:

- **AspNetUsers**: central user records, extended with custom fields like Streak, Coins, and LastActiveDate. Connected 1-to-many to claims, logins, tokens, and roles, ensuring each user's identity, permissions, and profile data are kept together.

- **Lessons → LessonQuizzes → LessonQuizQuestions → LessonQuizResults**:
  A Lesson groups multiple quizzes (LessonQuizzes).
  Each quiz contains several LessonQuizQuestions (with answer options A–D and the correct answer).
  Users' attempts are stored in LessonQuizResults, linking back to both the user and the quiz and capturing score, completion status, and timestamp.

- **ChallengeDefinitions → UserChallenges**:
  ChallengeDefinitions define each challenge's title, category, description, and scheduled date.
  UserChallenges tracks which users have been assigned which challenges, their completion status, and dates.

- **Expenses** and **Goals**:
  Expenses record individual spending entries (name, amount, date, category) per user.
  Goals capture financial targets, progress percentage, completion dates, and reward coins, also tied back to the user.

- Standard **AspNetRoles**, **AspNetUserRoles**, and **AspNetRoleClaims** tables manage role-based access.

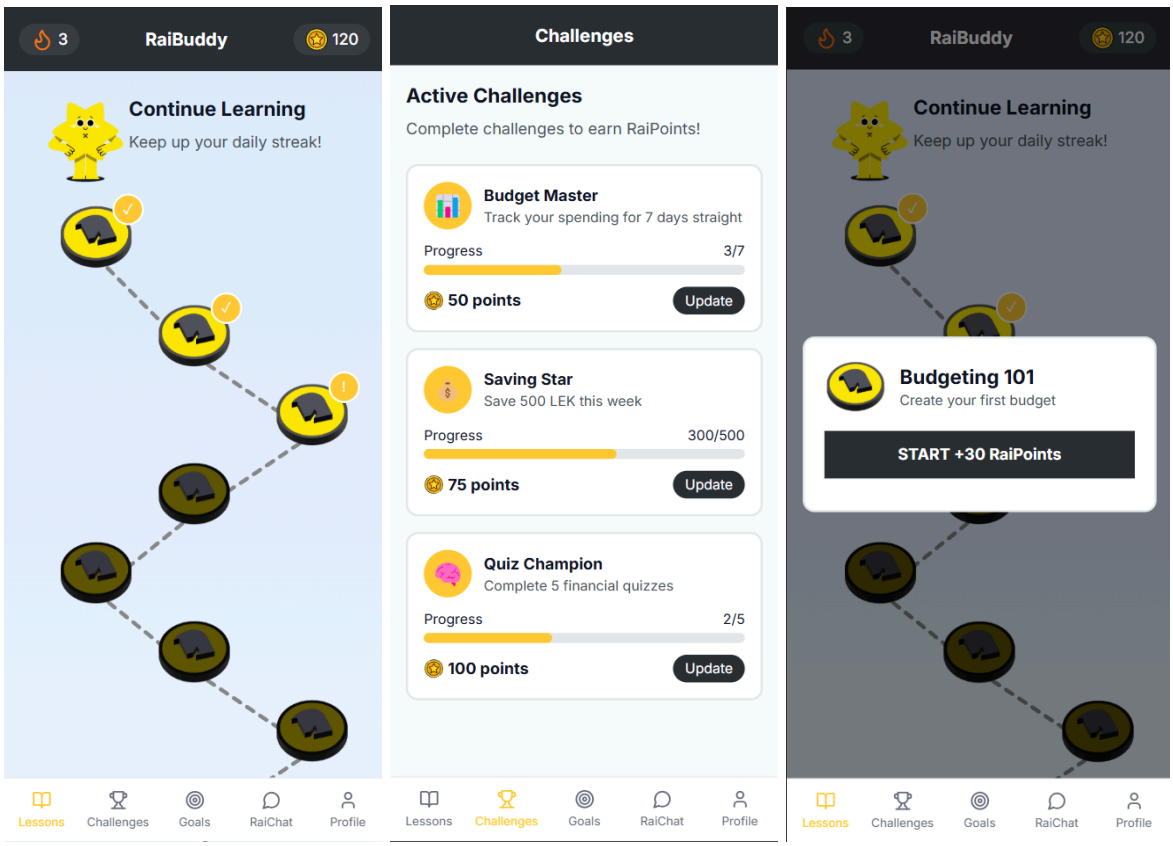
All relationships use foreign-key constraints to enforce referential integrity. For example, deleting a user will automatically cascade to their quiz results and challenges (but not to historical expenses or goals). This structure cleanly separates authentication, learning content, gamified challenges, and personal finance data, making queries and future extensions straightforward.

3.3 User Interface Design

3.3.1 Design Principles

- **Simplicity**
  We strip away visual clutter and surface only the most relevant controls and data points. Every screen follows a clear hierarchy of information, with concise labels, generous white space, and a limited palette of accent colors, to help teens focus on tasks without distraction.
- **Consistency**
  Reusable components (buttons, cards, form fields) share uniform styling, behavior, and spacing across the app. This predictable interface reduces the learning curve, as once a user masters one pattern, say, a modal dialog or tabbed panel, they'll recognize it everywhere.
- **Responsiveness**
  Built on Tailwind CSS, our layout adapts fluidly to different screen sizes. Grid classes and flex utilities ensure that cards reflow, navigation collapses into a hamburger menu on mobile, and touch targets remain large enough for comfortable tapping.
- **Accessibility**
  Color contrasts meet WCAG AA standards, form fields are labeled for screen readers, and keyboard focus states are clearly visible. We include skip-links for quick navigation, ARIA attributes on custom widgets, and ensure all interactive elements are reachable without a mouse.

3.3.2 Mockups / Wireframes



120

Question 1

What is a budget?

- ✗ A list of things you want to buy
- ✓ **A plan for how you will spend your money**
- A type of bank account
- A way to borrow money

Not quite right. Let's learn from this!

A budget is a plan that helps you manage your money by tracking income and expenses.

Got it

RaiChat

Hi there! I'm **RaiBuddy**, your financial education buddy! 😎...
08:20 AM

Hi, what is saving?
08:22 AM

Hey! Great question! 😊 **Saving** is putting aside a portion of your money instead of spending it all right away. Think of it like building a treasure chest for things you really want or need in the future! 💰
Here's a quick breakdown of why saving is super important:
Future Goals: Want that new PS5 or a cool pair of sneakers? Saving helps you reach those goals without going into debt.
Emergency Fund: Life can be unpredictable! Having savings means you're prepared for surprise expenses (like a

Ask RaiBuddy a question...

Lessons | Challenges | Goals | RaiChat | Profile

Expense Tracker

Summary

Total Expenses:	**3,000 LEK**
Needs:	0 LEK (0%)
Wants:	3,000 LEK (100%)

Needs: 0% Wants: 100%

Recent Expenses

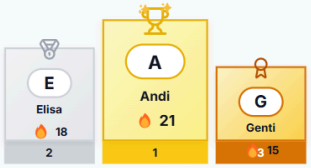
W Movies — Want — **3,000 LEK** — May 4

Leaderboard

Streaks | RaiPoints

Top Players in Albania

Your rank: #4



All Players

A	Andi — Tirana	21 days	
E	Elisa — Durrës	18 days	
G	Genti — Vlorë	15 days	
4 A	Alban — Tirana	14 days	You
5 B	Bora — Shkodër	12 days	

3.4 Artificial Intelligence Integration

Our platform includes a built-in chat feature powered by OpenAI's APIs, allowing users to ask anything they need and receive instant, accurate answers. This section describes how we integrate a conversational AI assistant.

3.4.1 Virtual Assistant Integration

We use OpenAI's GPT endpoints via simple REST calls. Parameters like model choice, temperature, and token limits let us balance response quality and performance.

Conversation Flow

1. **User → Chat Widget:** User asks a question.
2. **Widget → Backend:** Metadata (user/session) is added.
3. **Backend → OpenAI:** A prompt bundle (system + context + user message) is sent.
4. **OpenAI → Backend:** AI returns a reply, which we lightly sanitize and log.
5. **Backend → User:** The response appears in the chat.

Key Use Cases

- **24/7 Support:** Instant answers to FAQs.
- **Guided Workflows:** Step-by-step help for tasks.

3.5 Design Patterns Used

Below are the primary design patterns that underpin our application's architecture:
Dependency injection - Our application relies on dependency injection (DI) as the backbone of module composition. At startup, we register each service interface alongside its concrete implementation in a centralized container. Controllers never instantiate services directly; instead, they declare their dependencies via interfaces, and the DI container supplies the appropriate implementations at runtime. This approach delivers several concrete benefits:

- Strong Contracts & Information Hiding
  By coding against interfaces, controllers and higher-level modules only know about the abstraction, not the implementation. This "contract" enforces clear boundaries between layers and prevents leakage of internal details.
- Improved Testability
  Since dependencies are injected, it's straightforward to swap real implementations for mocks or stubs in unit tests. Tests can exercise controller logic in isolation by providing lightweight fakes for downstream services.
- Simplified Cross-Cutting Concerns
  Logging, caching, authentication handlers, and other middleware can also be injected, allowing us to apply them uniformly without scattering code across controllers.

Repository Pattern
To keep business logic free of persistence details, we introduce a repository layer that encapsulates all data-access operations behind well-defined interfaces. Key advantages include:

- Clear Separation of Concerns
  The repository abstracts away database schemas, query syntax, and connection management.
- Consistency & Reusability
  Common data-access logic lives in one place, reducing duplication.
- Test Isolation
  Like services, repository interfaces can be mocked in tests, allowing business logic to be validated without a live database. Alternatively, in-memory implementations can simulate data scenarios for faster, deterministic testing.

3.6 UML Diagrams

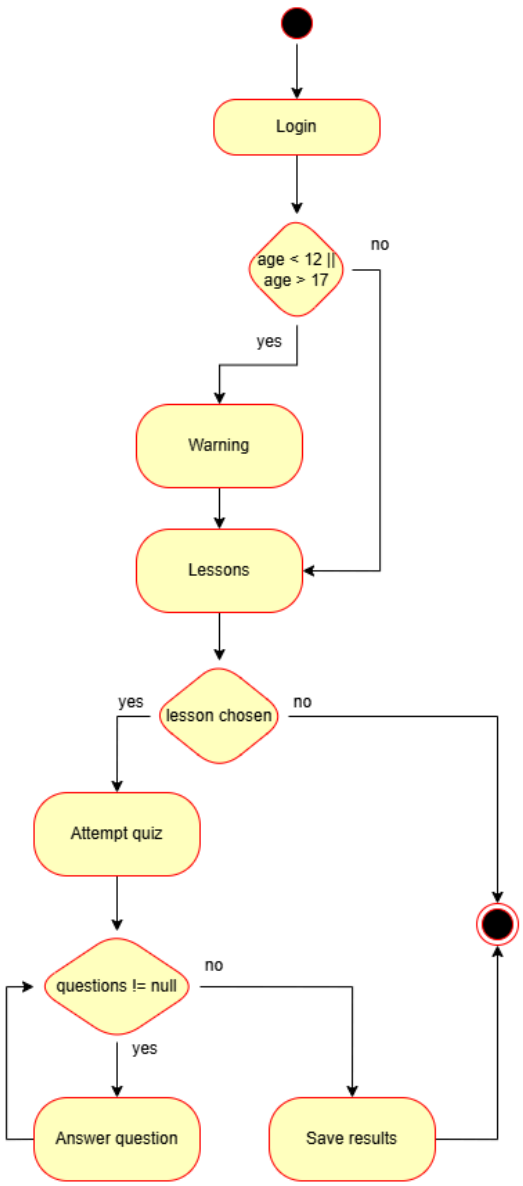


Figure 3. Data flow diagram for attempting a quiz

When a user logs in, the system immediately checks their age: if they are younger than 12 or older than 17, they must first acknowledge a warning before proceeding; otherwise they bypass the warning entirely. In either case, they are then presented with a list of lessons. If they choose not to select a lesson, the session simply ends. If they do pick a lesson, they enter its associated quiz, which delivers questions one at a time in a loop—each question must be answered before

the next is shown—until there are no questions left. At that point the system saves the user's results and the session concludes.

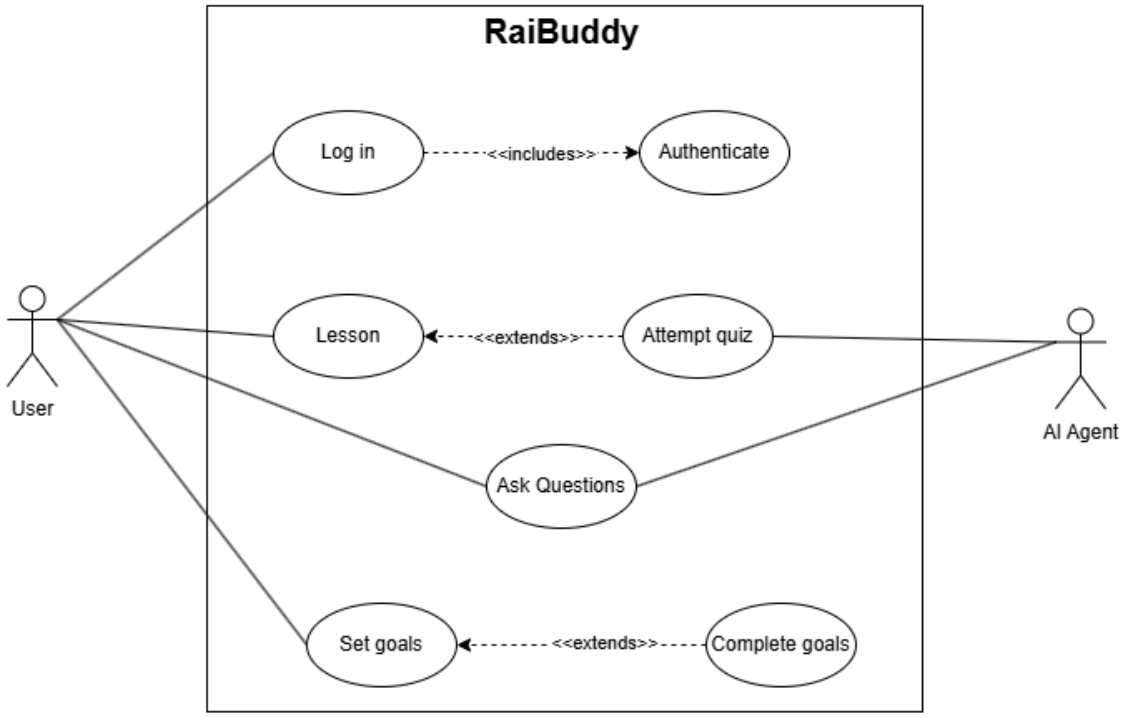


Figure 4. Use case diagram

The UML use-case diagram for RaiBuddy shows two primary actors—the User and the AI Agent—interacting with the system's core functions inside its boundary. The User begins by logging in (which includes an authentication step), then can browse and launch Lessons (with the "Attempt quiz" use case extending Lessons), ask Questions, and set personal learning goals (with "Complete goals" extending goal-setting). The AI Agent collaborates on the "Attempt quiz," "Ask Questions," and "Complete goals" use cases, meaning it grades quizzes, answers user queries, and helps drive goal fulfillment. The dashed «include» arrow into Authenticate indicates that every Log in action triggers authentication, while the «extend» arrows show that quizzes and goal completion are optional specializations of the lesson and goal-setting flows, respectively.

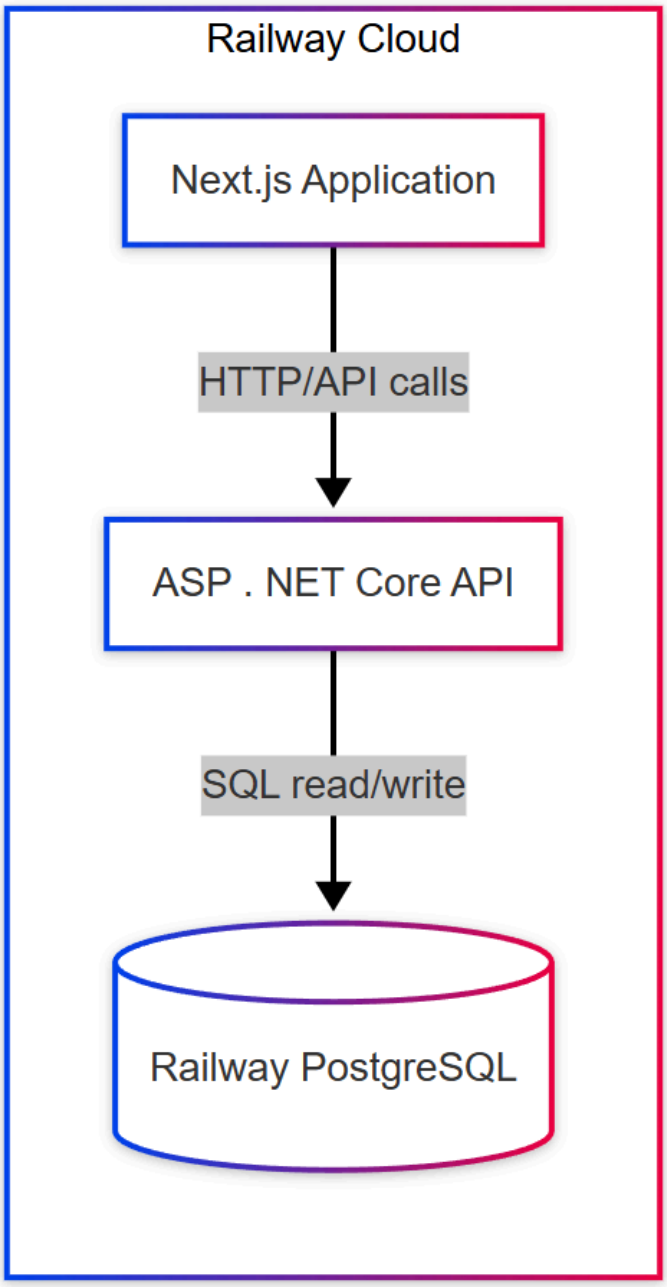


Figure 5. Deployment diagram

This deployment runs entirely on Railway: a Next.js application handles the user interface and sends HTTP requests to an ASP .NET Core API, which contains your business logic and talks to a Railway-hosted PostgreSQL database for data storage and retrieval.

3.7 System Security

Our platform enforces a multi-layered security posture, covering authentication, authorization, input protection, and data encryption.

Authentication & Authorization

- JWT and Cookie Support
    - We issue JSON Web Tokens for our single-page apps and third-party integrations. Tokens are signed with a rotating asymmetric key pair, scoped with audience and issuer claims, and have short-lived expirations plus refresh tokens stored in HTTP-only, Secure cookies.

Role Management (ASP.NET Identity)

- We leverage ASP.NET Identity for user and role stores:
    - Role Hierarchies & Claims: Administrators, Managers, and Users are mapped to roles, each carrying a set of claims.
    - Extensible Stores: Backed by Entity Framework, the user store can be swapped out or extended (e.g. to import from LDAP or external OAuth providers) without touching controllers.

Protections Against Common Web Attacks

- XSS
    - Any user-generated content is sanitized with a whitelist approach, and we deploy a strict Content Security Policy (CSP) to block inline scripts and unauthorized domains.
- SQL Injection
    - All database operations go through parameterized ORM calls (Entity Framework LINQ) or stored procedures—no string-concatenated SQL. We also employ query logging and periodic scans for any non-parameterized access.

Together, these controls provide a resilient defense-in-depth strategy, guarding against unauthorized access, injection attacks, and data disclosure both in flight and at rest.

4. Methodology and Implementation

4.1 Technologies and Tools

- **Backend**: .NET Core 8.0, Entity Framework Core
- **Frontend**: Node JS

- **Database**: PostgreSQL
- **CI/CD**: Github Actions pipelines automate builds for deploying in Vercel and Railway.

4.2 Development Team Structure

Our development team consists of four members, two focused on building the .NET Core APIs and database layer, and two crafting the frontend. We all pitched in wherever help was needed, reviewing each other's work and smoothing the integration between backend and UI.

5. Evaluation and Testing

Testing Strategy

We adopt a layered approach to ensure quality across the entire stack:

- **Unit Tests** verify individual components in isolation, business services, utility classes, and repository methods.
- **Integration Tests** exercise the interaction between multiple layers against an in-memory or test database to catch wiring and configuration issues.
- **End-to-End (E2E) Tests** simulate real user flows through the UI, covering scenarios such as login, data entry, and quiz completion to validate the system end-to-end.

Tools

- **Backend**
    - **xUnit** (primary) and **NUnit** (where preferred) for writing and running unit and integration tests in .NET Core.
    - **Entity Framework Core In-Memory Provider** for fast, reliable integration tests without a real database.
- **Frontend**
    - **Cypress** for automated browser-based E2E tests against the Next.js UI.

6. Results

During the hackathon we built a fully interactive prototype with a polished, user-friendly interface that lets learners move through lessons, take quizzes, tackle challenges, set personal goals, and maintain streaks—all while chatting with an AI tutor and competing on a live leaderboard. In our demo sessions, judges and peers praised the seamless navigation and gamified hooks. Although it isn't deployed to production, our MVP proves the core mechanics work end-to-end and lays a solid foundation for future hosting, scalability, and additional content.

7. Conclusions and Future Work

The current setup leverages Railway's managed infrastructure to deliver a simple yet robust deployment: a Next.js frontend for fast, server-rendered UI; an ASP .NET Core API for business logic; and a PostgreSQL database for reliable persistence. Hosting all components within Railway ensures seamless networking, unified environment configuration, and minimal operational overhead.

Future Work

- CI/CD Enhancements: Integrate GitHub (or GitLab) workflows to automate testing and deployment, with rollbacks on failure.
- Caching & CDN: Introduce edge caching (e.g., Cloudflare or Vercel) for static assets and API responses to improve performance and reduce load.
- Monitoring & Alerts: Hook in application performance monitoring (e.g., Railway Metrics, Datadog, or Prometheus) to track latency, errors, and resource usage, with proactive alerting.
- Scalability: Configure horizontal autoscaling rules for the API and database replicas to handle traffic spikes.
- Security: Implement secrets management best practices, network policies, and periodic vulnerability scans to safeguard data.
- Feature Roadmap: Explore adding background job processing (e.g., with Hangfire or BullMQ) and real-time functionality (e.g., using WebSockets or server-sent events) to support richer user experiences.