

Health & Fitness Tracker

CAPSTONE PROJECT



Name: Arjun Sethiya

Submission Date: 20 March

Batch Name:C3

Instructor Name: Mrs. Jyoti Patil

Table of Contents

S. No.	Section	Page No.
1	Abstract	1
2	Problem Definition and Objectives	2-3
3	How TO RUN	4-6
4	Configurations	7
5	Technology Stack	8
6	Frontend Architecture	
6.1	Directory Structure	9-10
6.2	Component Architecture	11-13
7	Backend Architecture	
7.1	Directory Structure	14
7.2	Controller Architecture	15-16
8	Database Design & Storage Optimization	17-21
9	API Documentation	22
10	Authentication & Authorization	21-27
11	Data Flow	28
12	Outputs	29-34
13	Conclusion	35

GITHUB LINK:-

<https://github.com/arjun79321/HealthFitnessTracker/>

(Health & Fitness Tracker)

ABSTRACT

The Health & Fitness Tracker is a web-based application designed to help users monitor their fitness journey effectively. The objective of this project is to provide a user-friendly interface for tracking workouts, diet, and health metrics. Built using ASP.NET Core for the backend, React for the frontend, and SQL Server for data storage, the system ensures a robust and scalable architecture.

The methodology involves designing RESTful APIs for seamless data exchange, implementing authentication mechanisms using JWT, and optimizing database queries for performance. Results indicate an intuitive dashboard that enhances user experience. This project demonstrates the integration of modern web technologies to support personalized health tracking.

PROBLEM DEFINITION AND OBJECTIVES

Problem Statement:

In today's fast-paced world, maintaining a healthy lifestyle has become increasingly challenging. Many individuals struggle to track their fitness progress, diet plans, and exercise routines efficiently. Existing solutions often lack personalized tracking, seamless integration across devices, and an intuitive user experience. Without a proper tracking system, users may find it difficult to measure their progress, stay motivated, and make data-driven decisions regarding their health.

Project Goals & Objectives:

- **Develop a User-Friendly Health Tracking Platform**
 - Create an intuitive and easy-to-use web application for users to log their fitness activities, dietary habits, and health metrics.
 - Ensure responsive design for seamless access across multiple devices.
- **Implement a Secure Authentication System:**
 - Utilize JSON Web Tokens (JWT) for secure authentication and authorization.
 - Ensure user data protection through proper encryption techniques.
- **Provide Real-Time Tracking and Analytics:**
 - Implement interactive dashboards that visualize progress and trends.
 - Allow users to set fitness goals and receive recommendations based on their health data.
- **Ensure Scalability and Performance Optimization:**
 - Use SQL Server for efficient data storage and retrieval.
 - Optimize database queries to handle a large volume of user records without performance degradation.
- **Enable Seamless Integration with Wearable Devices (Future Scope):**
 - a. Explore API integrations with fitness trackers and smartwatches to provide real-time insights.

(Health & Fitness Tracker)

The ultimate goal of the Health & Fitness Tracker is to empower users with actionable insights into their health, motivating them to stay consistent with their fitness journey.

Data Security and Privacy Concerns: Users often hesitate to share personal health information due to fears of data breaches and unauthorized access. Ensuring robust data protection mechanisms is crucial to build trust and encourage widespread adoption.

- **User Engagement and Retention:** Maintaining user interest over time is challenging. Many users discontinue app usage due to lack of motivation, monotonous content, or inadequate personalization.
- **Accuracy and Reliability of Data:** Wearable devices and fitness trackers may sometimes provide inaccurate data due to calibration issues or user error, leading to mistrust and reduced efficacy.
- **Integration with Existing Systems:** Seamless integration with other health platforms, wearables, and medical records is often lacking, resulting in fragmented data and a disjointed user experience.
- **Health Equity and Accessibility:** Ensuring that fitness applications cater to diverse populations, including those with disabilities or limited access to technology, remains a significant challenge

(Health & Fitness Tracker)

HOW TO RUN

Before you begin, ensure you have the following installed:

- [.NET 8 SDK](#)
- [Node.js](#) (v14 or later)
- [npm](#)
- [MySQL](#)
- [Visual Studio Code](#) or any preferred IDE

RUN THESE COMMANDS TO INSTALL NPM

```
npm install @reduxjs/toolkit@1.9.5 \
  @testing-library/dom@10.4.0 \
  @testing-library/jest-dom@6.6.3 \
  @testing-library/react@16.2.0 \
  @testing-library/user-event@13.5.0 \
  axios@1.4.0 \
  http-proxy-middleware@2.0.6 \
  lucide-react@0.482.0 \
  react@18.3.1 \
  react-dom@18.3.1 \
  react-icons@5.5.0 \
  react-redux@8.1.1 \
  react-router-dom@6.14.2 \
  recharts@2.15.1 \
  redux@4.2.1 \
  web-vitals@2.1.4
```

STEP 1. Install the required NuGet packages:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer --version 8.0.0
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore --version 8.0.3
dotnet add package Microsoft.AspNetCore.OpenApi --version 8.0.13
dotnet add package Microsoft.EntityFrameworkCore --version 8.0.13
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 8.0.13
dotnet add package Microsoft.EntityFrameworkCore.Tools --version 8.0.13
dotnet add package Microsoft.IdentityModel.Tokens --version 8.0.0
dotnet add package Pomelo.EntityFrameworkCore.MySql --version 8.0.3
dotnet add package Swashbuckle.AspNetCore --version 6.6.2
dotnet add package System.IdentityModel.Tokens.Jwt --version 8.0.0
```

- **STEP 2 Navigate to the client directory:**

(Health & Fitness Tracker)

- **cd ../client**
 - **Install npm packages:**
 - **npm install**
-
- **STEP3 Create a MySQL database named HealthFitnessTracker**
 - **Run the database creation script (HealthFitnessTracker_DB_Setup.sql): You can execute this script using MySQL Workbench, MySQL command line, or any other MySQL client.**
 - **CREATE DATABASE HealthFitnessTracker;**
 - **GO**
 - **USE HealthFitnessTracker;**
 - **GO**

IN SWAGGER AUTHORIZE THE TOKEN LIKE THIS

STEP4.

Bearer

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1laWRIbnRpZmlmlci6IjIiLCJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1lIjoiemFoaWRtdXN0YWZlIiwiaXhwIjojoxNzQyNDAA0Tc0LCJpc3MiOiJIZWZsdGhGaXRuZXNzVHJhY2tlckFOSSIsImF1ZCI6Ikh1YWx0aEZhZpdG5lc3NUcmFja2VyQVBJIIn0.LC6gv6cQSEePIuVy9xz0D8yx2GnbcFduDH994K2bGeg

Bearer should be written

Configuration

Backend Configuration

Update appsettings.json with your database connection and JWT settings:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost,1433;Database=HealthFitnessTracker;User
Id=sa;Password=arjun12344@@@;TrustServerCertificate=True;"
  },
  "AppSettings": {
    "Token":
"eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiYWRtaW4iOnRydWUsImhhdCI6MTUxNjIOTAYMn0uANCf_8p1AE4ZQs7
QuqGAyyfTEgYrKSjKWkhBk5cIn1_2QVr2jEjmM-1tu7EgnyOf_fAsvdFXva8Sv05iTGzETg",
    "Issuer": "HealthFitnessTrackerAPI",
    "Audience": "HealthFitnessTrackerAPI"
  }
}
```

(Health & Fitness Tracker)

```
    },
    "Logging": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft": "Warning",
        "Microsoft.Hosting.Lifetime": "Information"
      }
    },
    "AllowedHosts": "*"
  }
}
UPDATE THE CORS POLICY IN PROGRAM.CS IF NEEDED:
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowSpecificOrigin",
        builder => builder
            .WithOrigins("http://localhost:3000") // Update with your React app URL
            .AllowAnyMethod()
            .AllowAnyHeader()
            .AllowCredentials());
});
```


(Health & Fitness Tracker)

Frontend Configuration

Update setupProxy.js if needed:

```
const { createProxyMiddleware } = require('http-proxy-middleware');

module.exports = function(app) {
  app.use(
    '/api',
    createProxyMiddleware({
      target: 'http://localhost:5009',
      changeOrigin: true,
    })
  );
};
```

Running the Application

Backend

```
cd server
dotnet run
```

The API will be available at <http://localhost:5009> with Swagger documentation at <http://localhost:5009/swagger>

Frontend

```
cd client
npm start
```

The React app will be available at <http://localhost:3000>.

API Documentation

API documentation is available via Swagger. When the backend is running, navigate to:

Copy

<http://localhost:5009/swagger/index.html>

(Health & Fitness Tracker)

Technology Stack

Frontend

- **Framework:** React.js
- **State Management:** Redux
- **Routing:** React Router
- **UI Components:** Custom components with CSS/SCSS
- **HTTP Client:** Axios for API requests
- **Data Visualization:** Chart.js for fitness trends

Backend

- **Framework:** ASP.NET Core Web API
- **Database:** Microsoft SQL Server
- **ORM:** Entity Framework Core
- **Authentication:** JWT-based authentication

Frontend Architecture

Directory Structure

/client (React Frontend)

```
├── /public
├── /src
│   ├── /components
│   │   ├── /analytics    # Analytics components
│   │   ├── /auth        # Authentication components
│   │   ├── /calories     # Calorie tracking components
│   │   ├── /common       # Reusable UI components
│   │   ├── /dashboard    # Dashboard components
│   │   ├── /layout       # Layout components (header, footer)
│   │   ├── /nutrition    # Nutrition tracking components
│   │   └── /workout       # Workout tracking components
│   ├── /pages            # Page components for each route
│   │   ├── /auth         # Authentication pages (login, register)
│   │   ├── /Dashboard.js  # Main dashboard page
│   │   ├── /Home.js      # Home page
│   │   ├── /WorkoutLog.js # Workout logging page
│   │   ├── /CalorieTracker.js # Calorie tracking page
│   │   └── /FitnessAnalytics.js # Fitness analytics page
```

(Health & Fitness Tracker)

- | | | — /Profile.js # User profile page
- | | | — /NotFound.js # 404 page
- | | — /store # Redux store configuration
- | | | — /index.js # Store configuration
- | | | — /slices # Redux toolkit slices
- | | | — /alertSlice.js # Alert state management
- | | | — /authSlice.js # Authentication state management
- | | | — /calorieSlice.js # Calorie state management
- | | | — /workoutSlice.js # Workout state management
- | | — /services # API service calls
- | | | — authService.js # Authentication API calls
- | | | — workoutService.js # Workout API calls
- | | | — calorieService.js # Calorie API calls
- | | — /utils # Utility functions
- | | — App.js # Main application component
- | | — App.css # Main application styles
- | | — index.js # Entry point
- | | — index.css # Global styles
- | | — setupProxy.js # Proxy configuration for development
- | — package.json

(Health & Fitness Tracker)

Component Architecture

The frontend follows a component-based architecture with React:

1. **Common Components:** Reusable UI elements like buttons, inputs, and cards
2. **Layout Components:** Header, footer, and navigation elements
3. **Feature Components:** Specialized components for workouts, nutrition tracking, etc.
4. **Page Components:** Container components that represent different routes

State Management

The application uses Redux Toolkit for state management with a structured store that combines multiple slice reducers:

```
import { configureStore } from '@reduxjs/toolkit';

import authReducer from './slices/authSlice';

import workoutReducer from './slices/workoutSlice';

import calorieReducer from './slices/calorieSlice';

import alertReducer from './slices/alertSlice';

export const store = configureStore({
  reducer: {
    auth: authReducer,
    workouts: workoutReducer,
    calories: calorieReducer,
    alerts: alertReducer
  }
});
```

(Health & Fitness Tracker)

Each slice handles a specific domain of the application:

1. **authSlice**: Manages user authentication state
 - a. User login/registration
 - b. Current user data
 - c. Authentication status
2. **workoutSlice**: Handles workout-related data
 - a. Workout logs list
 - b. Current workout being edited
 - c. Loading and error states
3. **calorieSlice**: Manages calorie and nutrition tracking
 - a. Calorie entries
 - b. Daily calorie summaries
 - c. Loading and error states
4. **alertSlice**: Controls application notifications
 - a. Success/error messages
 - b. Toast notifications
 - c. Form validation alerts

The Redux Toolkit approach simplifies state management by combining reducers, actions, and action creators in a single file, reducing boilerplate code and making the application more maintainable.

(Health & Fitness Tracker)

Routing

React Router v6 is used for client-side routing with protected routes implemented using a custom PrivateRoute component:

// Actual implementation from App.js

```
<Route path="/" element={} />

<Route path="/login" element={} />

<Route path="/register" element={} />

{/* Protected Routes */}

<Route path="/dashboard" element={ } />

<Route path="/workouts" element={ } />

<Route path="/calories" element={ } />

<Route path="/analytics" element={ } />

<Route path="/profile" element={ } />

{/* 404 Page */} <Route path="/404" element={} />

<Route path="*" element={} />
```

The PrivateRoute component ensures that only authenticated users can access protected routes. If a user is not authenticated, they are redirected to the login page.

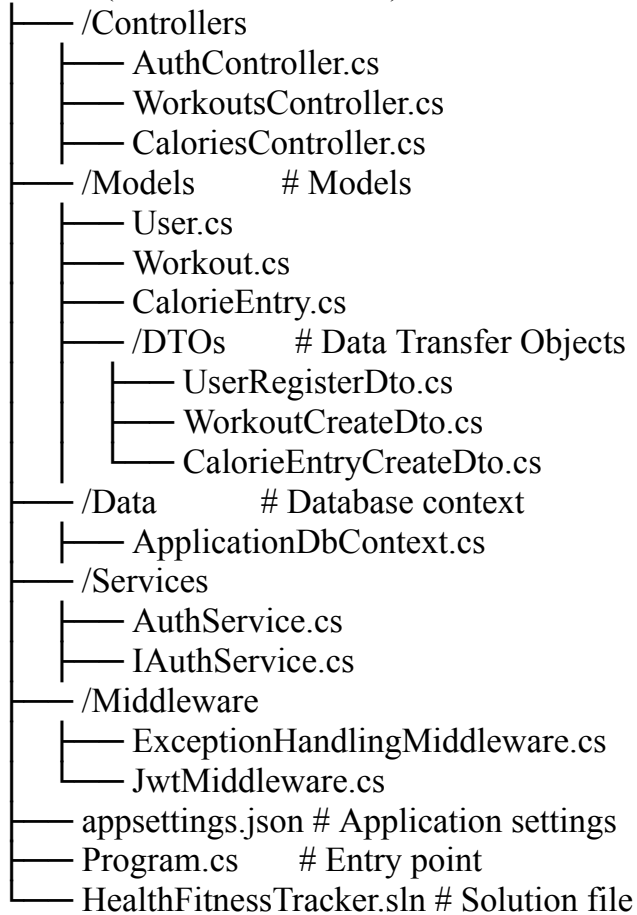
- A responsive layout with Navbar and Footer components
- Main content wrapped in a container for consistent styling
- Redux store provided at the application root level

(Health & Fitness Tracker)

Backend Architecture

Directory Structure

/server (ASP.NET Core API)



(Health & Fitness Tracker)

Controller Architecture

The backend follows the RESTful API design pattern with controllers handling specific resources:

1. **AuthController**: Handles user authentication and registration
 - a. POST /api/auth/register: Register a new user
 - b. POST /api/auth/login: Authenticate and generate JWT
 - c. GET /api/auth/current: Get current user details
2. **WorkoutsController**: Manages workout data
 - a. GET /api/workouts: Get all workouts for current user
 - b. GET /api/workouts/{id}: Get specific workout
 - c. POST /api/workouts: Create new workout
 - d. PUT /api/workouts/{id}: Update workout
 - e. DELETE /api/workouts/{id}: Delete workout
3. **CaloriesController**: Manages calorie/nutrition data
 - a. GET /api/calories: Get all calorie entries
 - b. GET /api/calories/{id}: Get specific calorie entry
 - c. POST /api/calories: Create new calorie entry
 - d. PUT /api/calories/{id}: Update calorie entry
 - e. DELETE /api/calories/{id}: Delete calorie entry

Each controller follows the standard RESTful conventions and includes appropriate authentication and authorization checks.

Service Layer

The service layer contains business logic and serves as an intermediary between controllers and data access:

1. **IAuthService/AuthService**: Handles user authentication and token generation
 - a. User registration and validation
 - b. Password hashing and verification
 - c. JWT token generation and validation

Additional services can be implemented as the application grows to handle more complex business logic for workouts and calories.

Data Access Layer

Entity Framework Core is used as the ORM to interact with the SQL Server database:

1. **ApplicationDbContext**: Defines the database context and sets up entity relationships
 - a. DbSet<User> Users
 - b. DbSet<Workout> Workouts
 - c. DbSet<CalorieEntry> CalorieEntries

(Health & Fitness Tracker)

Middleware

Custom middleware components handle cross-cutting concerns:

1. **ExceptionHandlerMiddleware**: Centralizes error handling and provides consistent error responses
2. **JwtMiddleware**: Validates JWT tokens and attaches user context to requests

(Health & Fitness Tracker)

Database Design

Database Design

Entity Relationship Diagram



(Health & Fitness Tracker)

Key Entities

1. **Users:** Stores user account information and profile details
 - a. Primary key: Id (auto-increment integer)
 - b. Properties: Username, Email, PasswordHash, PasswordSalt, FirstName, LastName, Weight, Height, DateOfBirth, Gender
 - c. Relationships: One-to-many with Workouts and CalorieEntries
2. **Workouts:** Records workout sessions
 - a. Primary key: Id (auto-increment integer)
 - b. Foreign key: UserId (references Users)
 - c. Properties: Type, Name, Duration, Calories, Date, Notes
 - d. Relationships: Many-to-one with Users
3. **CalorieEntries:** Tracks food intake and calorie consumption
 - a. Primary key: Id (auto-increment integer)
 - b. Foreign key: UserId (references Users)
 - c. Properties: FoodName, Calories, MealType, Date, Notes
 - d. Relationships: Many-to-one with Users

(Health & Fitness Tracker)

Database Schema SQL

-- Create Users table

```
CREATE TABLE Users (  
    Id INT AUTO_INCREMENT PRIMARY KEY,  
    Username VARCHAR(50) NOT NULL,  
    Email VARCHAR(100) NOT NULL,  
    PasswordHash VARBINARY(255) NOT NULL,  
    PasswordSalt VARBINARY(255) NOT NULL,  
    FirstName VARCHAR(50) NULL,  
    LastName VARCHAR(50) NULL,  
    Weight INT NULL,  
    Height INT NULL,  
    DateOfBirth DATETIME NULL,  
    Gender VARCHAR(20) NULL,  
    CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    UpdatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE  
    CURRENT_TIMESTAMP  
);
```

-- Create indexes for Users table

```
CREATE UNIQUE INDEX IX_Users_Username ON Users(Username);  
CREATE UNIQUE INDEX IX_Users_Email ON Users(Email);
```

-- Create Workouts table

```
CREATE TABLE Workouts (  
    Id INT AUTO_INCREMENT PRIMARY KEY,  
    UserId INT NOT NULL,  
    Type VARCHAR(50) NOT NULL,  
    Name VARCHAR(100) NOT NULL,  
    Duration INT NOT NULL,  
    Calories INT NULL,  
    Date DATETIME NOT NULL,
```

(Health & Fitness Tracker)

```
Notes VARCHAR(500) NULL,  
CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
UpdatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE  
CURRENT_TIMESTAMP,  
FOREIGN KEY (UserId) REFERENCES Users(Id) ON DELETE CASCADE  
);
```

-- Create index for Workouts table

```
CREATE INDEX IX_Workouts_UserId ON Workouts(UserId);  
CREATE INDEX IX_Workouts_Date ON Workouts(Date);
```

-- Create CalorieEntries table

```
CREATE TABLE CalorieEntries (  
    Id INT AUTO_INCREMENT PRIMARY KEY,  
    UserId INT NOT NULL,  
    FoodName VARCHAR(100) NOT NULL,  
    Calories INT NOT NULL,  
    MealType VARCHAR(50) NOT NULL,  
    Date DATETIME NOT NULL,  
    Notes VARCHAR(200) NULL,  
    CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (UserId) REFERENCES Users(Id) ON DELETE CASCADE  
);
```

-- Create index for CalorieEntries table

```
CREATE INDEX IX_CalorieEntries_UserId ON CalorieEntries(UserId);  
CREATE INDEX IX_CalorieEntries_Date ON CalorieEntries(Date);
```

(Health & Fitness Tracker)

Database Indexes

To optimize query performance, the following indexes are implemented:

1. **Users Table:**
 - a. Unique index on Username for fast username lookups during authentication
 - b. Unique index on Email for fast email lookups during authentication
2. **Workouts Table:**
 - a. Index on UserId for fast filtering of workouts by user
 - b. Index on Date for efficient date range queries and reports
3. **CalorieEntries Table:**
 - a. Index on UserId for fast filtering of calorie entries by user
 - b. Index on Date for efficient date range queries and daily summaries

Data Relationships

1. **User to Workouts:** One-to-many relationship
 - a. A user can have multiple workout records
 - b. Each workout belongs to exactly one user
 - c. Implemented with a foreign key constraint with cascade delete
2. **User to CalorieEntries:** One-to-many relationship
 - a. A user can have multiple calorie/nutrition entries
 - b. Each calorie entry belongs to exactly one user
 - c. Implemented with a foreign key constraint with cascade delete

API DOCUMENTATION

Authentication Endpoints

Endpoint	Method	Description	Request Body	Response
<code>/api/auth/register</code>	POST	Register a new user	<code>{ username, email, password, firstName, lastName }</code>	<code>{ token, user }</code>
<code>/api/auth/login</code>	POST	Authenticate user	<code>{ email, password }</code>	<code>{ token, user }</code>

Workout Endpoints

Endpoint	Method	Description	Request Body	Response
<code>/api/workouts</code>	GET	Get all workouts for current user	None	Array of workouts
<code>/api/workouts/{id}</code>	GET	Get workout by ID	None	Workout object
<code>/api/workouts</code>	POST	Create new workout	Workout object	Created workout
<code>/api/workouts/{id}</code>	PUT	Update workout	Workout object	Updated workout
<code>/api/workouts/{id}</code>	DELETE	Delete workout	None	Success message

Nutrition Endpoints

Endpoint	Method	Description	Request Body	Response
<code>/api/nutrition</code>	GET	Get all nutrition entries	None	Array of nutrition entries
<code>/api/nutrition/{id}</code>	GET	Get nutrition entry by ID	None	Nutrition object
<code>/api/nutrition</code>	POST	Create new nutrition entry	Nutrition object	Created entry
<code>/api/nutrition/{id}</code>	PUT	Update nutrition entry	Nutrition object	Updated entry
<code>/api/nutrition/{id}</code>	DELETE	Delete nutrition entry	None	Success message

Authentication & Authorization

JWT Authentication Implementation

The application uses JSON Web Tokens (JWT) for secure authentication. JWT provides a stateless authentication mechanism that allows the backend to verify the identity of users without maintaining session state.

```
// JWT Configuration from Program.cs
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options =>
{
    options.SaveToken = true;
    options.RequireHttpsMetadata = false;
    options.TokenValidationParameters = new TokenValidationParameters()
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidAudience = builder.Configuration["AppSettings:Audience"],
        ValidIssuer = builder.Configuration["AppSettings:Issuer"],
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["AppSettings:
Token"!]))
    };
    options.Events = new JwtBearerEvents
    {
        OnAuthenticationFailed = context =>
        {
            var logger =
context.HttpContext.RequestServices.GetRequiredService<ILogger<Program>>();
            logger.LogError("Authentication failed: {0}", context.Exception.Message);
            return Task.CompletedTask;
        }
    };
});
```

(Health & Fitness Tracker)

Auth Controller Implementation

The authentication logic is implemented in the AuthController which provides endpoints for user registration, login, profile management, and password changes:

```
[Route("api/[controller]")]
[ApiController]
public class AuthController : ControllerBase
{
    private readonly ApplicationDbContext _context;
    private readonly IAuthService _authService;

    public AuthController(ApplicationDbContext context, IAuthService
authService)
    {
        _context = context;
        _authService = authService;
    }

    // Registration endpoint
    [HttpPost("register")]
    public async Task<ActionResult<UserDto>> Register(UserRegisterDto
request)
    {
        // Implementation details...
    }

    // Login endpoint
    [HttpPost("login")]
    public async Task<ActionResult<string>> Login(UserLoginDto request)
    {
        // Implementation details...
    }

    // Get current user profile - requires authentication
    [HttpGet("user"), Authorize]
    public async Task<ActionResult<UserDto>> GetCurrentUser()
    {
        // Implementation details...
    }

    // Update user profile - requires authentication
    [HttpPut("user"), Authorize]
```

(Health & Fitness Tracker)

```
public async Task<ActionResult> UpdateUser(UserUpdateDto request)
{
    // Implementation details...
}

// Change password - requires authentication
[HttpPost("change-password"), Authorize]
public async Task<ActionResult> ChangePassword(ChangePasswordDto
request)
{
    // Implementation details...
}
}
```

Authentication Flow

1. **User Registration:**
 - a. Client sends user registration details to /api/Auth/register
 - b. Server validates username and email uniqueness
 - c. Server hashes the password with a secure algorithm and salt
 - d. User is stored in the database
 - e. JWT token is generated and returned with user details
2. **User Login:**
 - a. Client sends username and password to /api/Auth/login
 - b. Server validates credentials against stored hash and salt
 - c. If valid, JWT token is generated and returned with user details
3. **Protected Routes:**
 - a. Client includes JWT token in Authorization header
 - b. ASP.NET Core's [Authorize] attribute enforces authentication
 - c. Token is validated for integrity, expiration, issuer, and audience
 - d. Claims like ClaimTypes.NameIdentifier are used to identify the user
4. **User Profile Management:**
 - a. Authenticated users can retrieve their profile via /api/Auth/user
 - b. Users can update profile information via /api/Auth/user (PUT)
 - c. Password changes are handled securely via /api/Auth/change-password

Security Implementation

1. **Password Security:**
 - a. Passwords are hashed using a cryptographic algorithm via `IAuthService.CreatePasswordHash`
 - b. Each user has a unique salt to prevent rainbow table attacks
 - c. Password verification uses constant-time comparison to prevent timing attacks
2. **Token Security:**
 - a. Tokens include user ID and expiration time
 - b. Tokens are signed with a secret key to prevent tampering
 - c. Full validation includes issuer, audience, and signature checks
 - d. Error logging for authentication failures

3 CORS Security:

// CORS Policy from Program.cs

`builder.Services.AddCors(options =>`

(Health & Fitness Tracker)

```
{  
  options.AddPolicy("AllowSpecificOrigin",  
    builder => builder  
      .WithOrigins("http://localhost:3000") // React frontend URL  
      .AllowAnyMethod()  
      .AllowAnyHeader()  
      .AllowCredentials());  
})
```

4 API Security:

- Input validation on all requests
- Proper HTTP status codes (401, 403) for authentication/authorization failures
- User-friendly error messages that don't leak sensitive information
- JWT integrated with Swagger UI for API testing# Frontend & Backend Architecture

Data Flow

Request Lifecycle

1. User interacts with React UI component
2. Component dispatches Redux action
3. Action creator makes API call to ASP.NET Core backend
4. Backend controller receives request and validates authentication
5. Controller delegates to service layer for business logic
6. Service layer interacts with repository for data access
7. Repository performs CRUD operations on the database
8. Response flows back through the same layers
9. Redux updates state based on API response
10. React components re-render with updated data

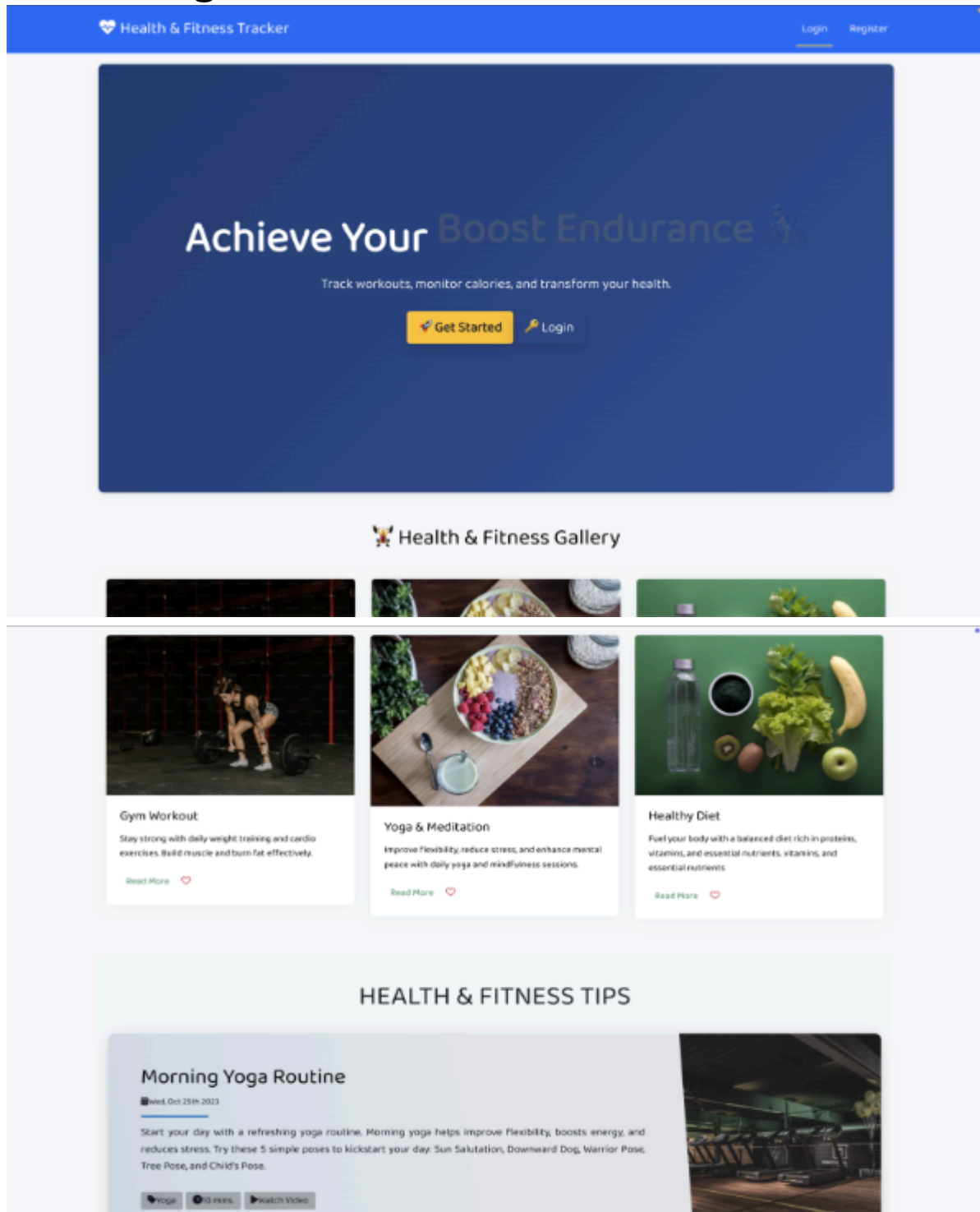
Example Flow - Logging a Workout

1. User fills out workout form and submits
2. Form submission triggers Redux action addWorkout
3. Action creator makes POST request to /api/workouts
4. WorkoutsController processes request and validates data
5. WorkoutService handles business logic
6. WorkoutRepository saves data to the database
7. Response returns to client with created workout data
8. Redux updates workout list in store
9. React components re-render to show updated workout list

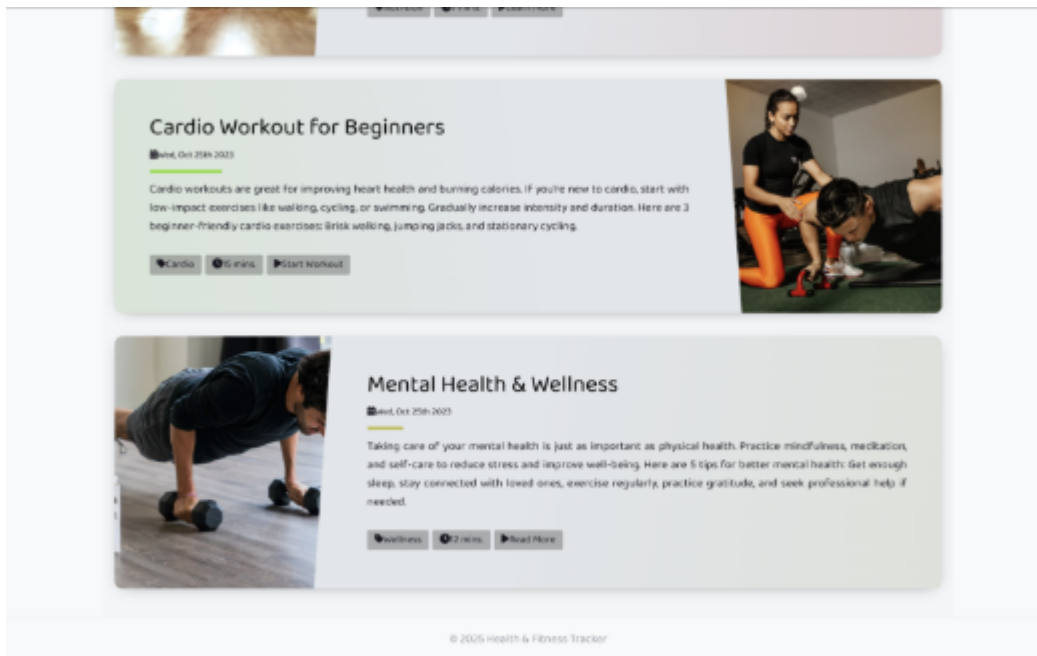
(Health & Fitness Tracker)

OUTPUTS

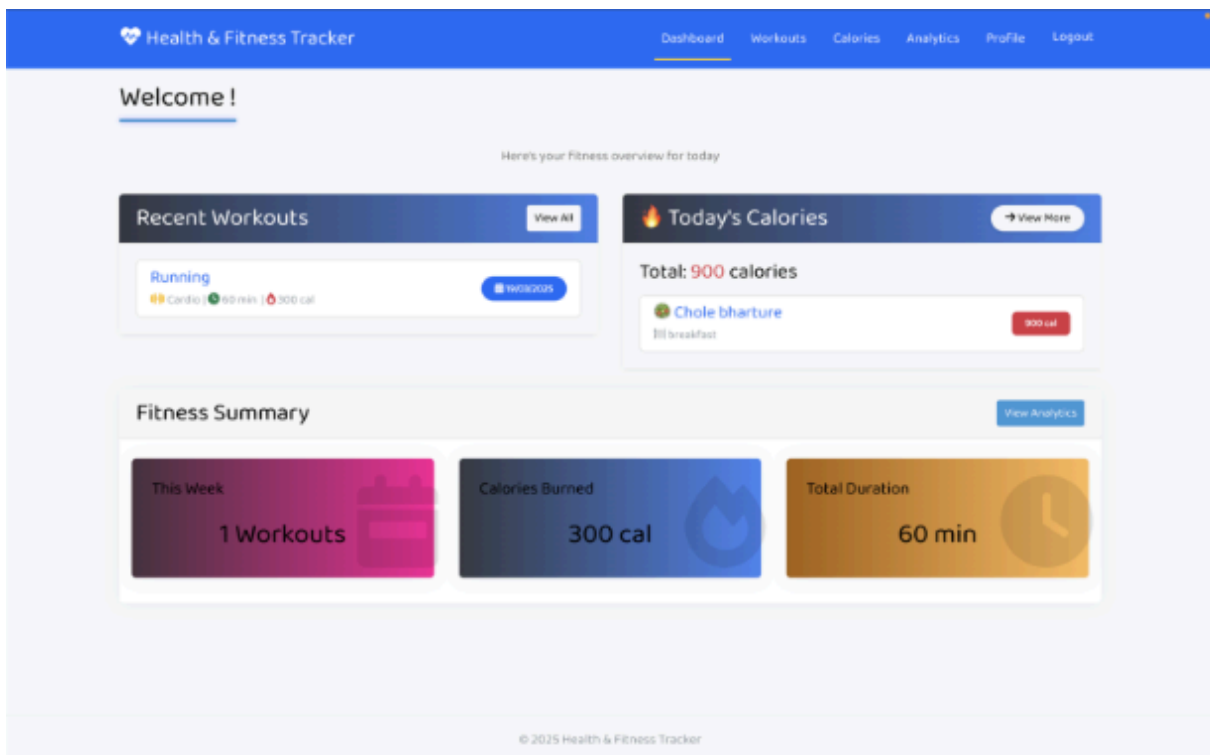
Home Page



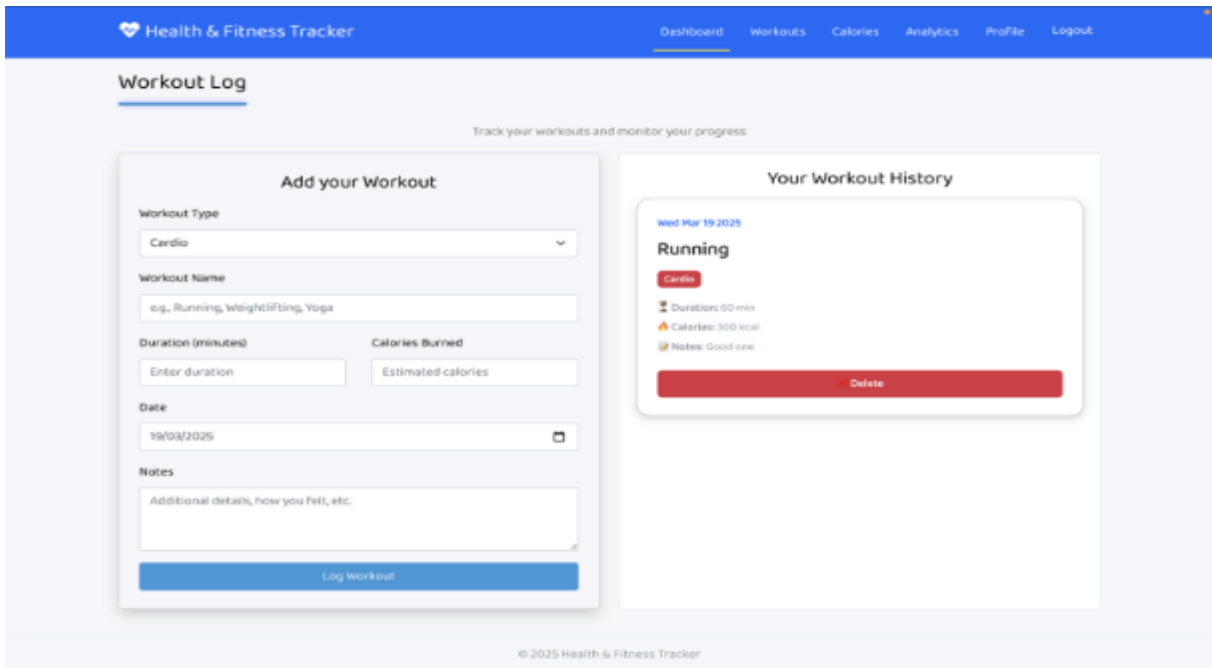
(Health & Fitness Tracker)



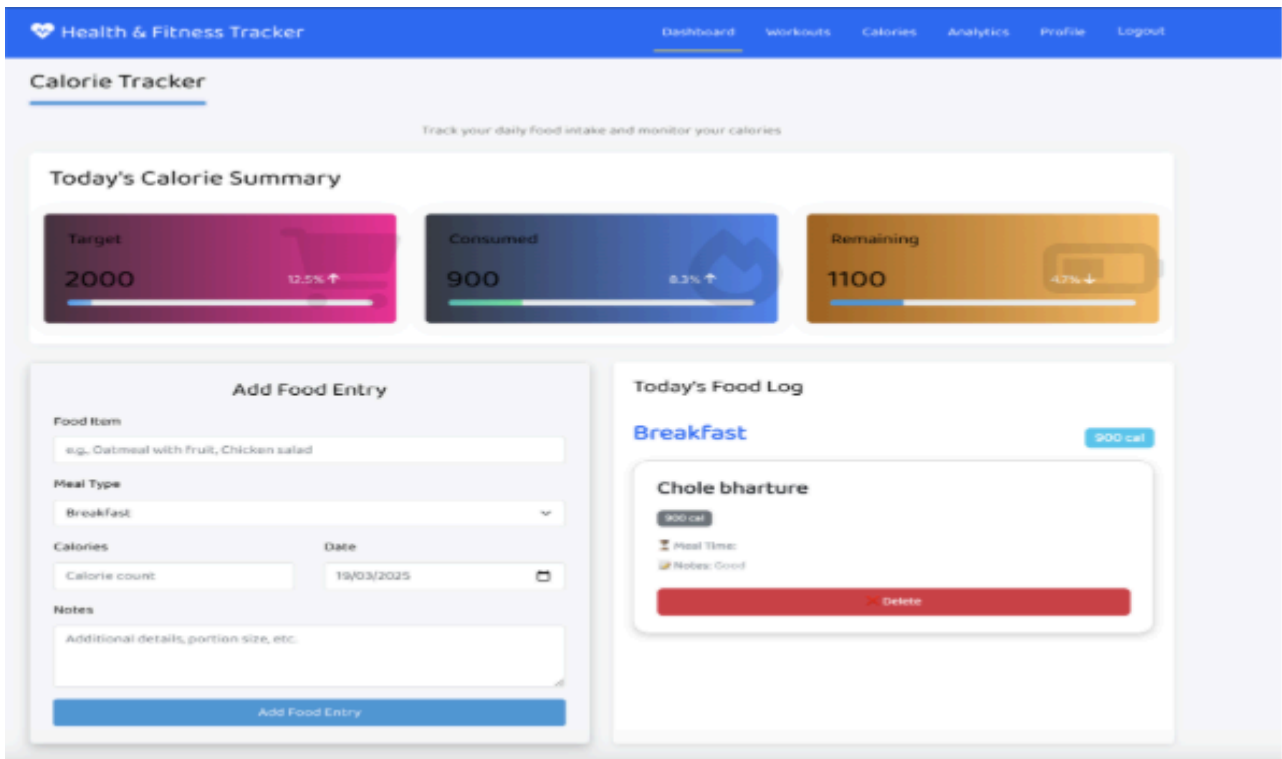
Dashboard



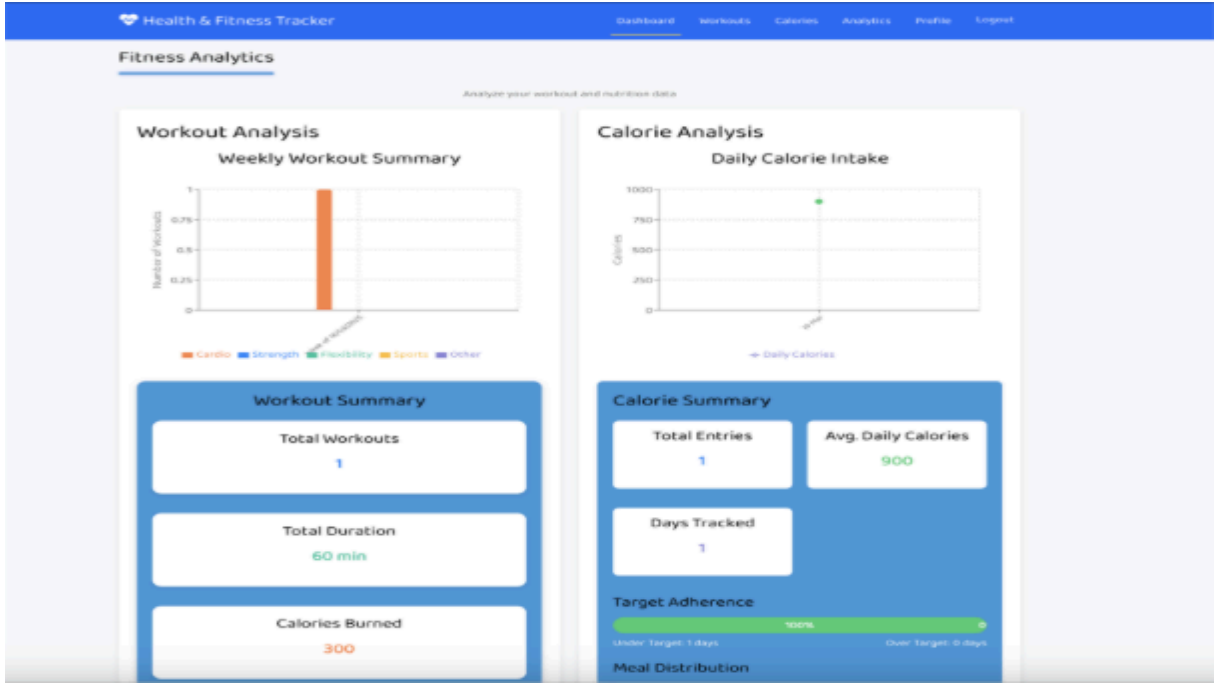
Workout



CalorieTracker



Analytics



Profile

Health & Fitness Tracker

Dashboard Workouts Calories Analytics Profile Logout

Your Profile

arjun793111

Fitness

Information

Username	arjun793111	Name	Arjun Sethiya
Email	arjun793111@gmail.com	Gender	Male
Weight	165 cm (65.0 inches)	Height	165 cm (65.0 inches)
Date of Birth	01/01/1999		
BMI	15.0 - Underweight		

© 2025 Health & Fitness Tracker

(Health & Fitness Tracker)

Sign In

Health & Fitness Tracker

[Login](#)[Register](#)

Sign In

Username

Password

Login

Don't have an account? [Sign Up](#)

© 2025 Health & Fitness Tracker

Register

Health & Fitness Tracker

[Login](#)[Register](#)

Create an Account

First Name

Last Name

Username

Email Address

Date of Birth

dd/mm/yyyy

Gender

Select Gender

Weight (lbs)

Enter weight in pounds

Height (cm)

Enter height in centimeters

Password


Confirm Password

Register

Password must be at least 6 characters

(Health & Fitness Tracker)

Swagger



Swagger

OpenAPI Specification

Select a definitionserver v1

Health & Fitness Tracker API v1 OAS 3.0
<http://localhost:5000/swagger/v1/swagger.json>

Authorize

Auth

POST

/api/Auth/register

POST

/api/Auth/login

GET

/api/Auth/user

PUT

/api/Auth/user

POST

/api/Auth/change-password

Calories

GET

/api/Calories

POST

/api/Calories

GET

/api/Calories/{id}

PUT

/api/Calories/{id}

DELETE

/api/Calories/{id}

GET

/api/Calories/daily-summary

GET

/api/Calories/stats

Workouts

GET

/api/Workouts

POST

/api/Workouts

GET

/api/Workouts/{id}

PUT

/api/Workouts/{id}

DELETE

/api/Workouts/{id}

GET

/api/Workouts/stats

Schemas

CalorieEntryCreateDto >

CalorieEntryDto >

CalorieEntryUpdateDto >

CalorieStatsDto >

ChangePasswordDto >

DailyCalorieSummaryDto >

MealTypeStats >

UserData >

UserLoginDto >

UserRegisterDto >

UserUpdateDto >

WorkoutCreateDto >

WorkoutDto >

WorkoutStatsDto >

WorkoutTypeStats >

Conclusion

The Health & Fitness Tracker application (Project ID: P8) demonstrates a comprehensive full-stack architecture using React for the frontend and ASP.NET Core for the backend. This document has outlined the architectural design decisions, component structures, and data flows that make up the application.

Key Architecture Highlights

1. **Clean Separation of Concerns**
 - a. Clear division between frontend (React) and backend (ASP.NET Core) components
 - b. Well-defined API contracts between client and server
 - c. Proper use of Redux for state management with Redux Toolkit slices
2. **Security-First Approach**
 - a. Robust JWT authentication implementation
 - b. Secure password storage with hashing and salting
 - c. Protected routes with proper authorization checks
 - d. CORS configuration to prevent unauthorized access
3. **Scalable Database Design**
 - a. Normalized MySQL schema with proper relationships
 - b. Strategic indexing for query performance
 - c. Transaction support for data integrity
 - d. Optimized entity models mapped via Entity Framework Core
4. **Efficient Data Flows**
 - a. Well-defined request lifecycles
 - b. Optimized API endpoints for different data needs
 - c. Proper error handling throughout the stack
 - d. Performance considerations with caching strategies