# ClubberApp - My Sports Playlist

This project is a web application built to fulfill the “My Sports Playlist” challenge. It allows users to register, log in, view a list of sports matches (from a mock source), and manage their personal playlist of matches.

The application follows Clean Architecture principles for the backend and utilizes NgRx for state management in the Angular frontend.

## Project Structure

/clubber\_app  
├── frontend/ # Angular Frontend Application  
│ ├── src/  
│ ├── angular.json  
│ ├── package.json  
│ └── tsconfig.json  
├── src/ # .NET Backend Source Code  
│ ├── ClubberApp.Api/  
│ ├── ClubberApp.Application/  
│ ├── ClubberApp.Domain/  
│ └── ClubberApp.Infrastructure/  
├── tests/ # .NET Backend Tests  
│ ├── ClubberApp.Api.Tests/  
│ └── ClubberApp.Application.Tests/  
├── ClubberApp.sln # .NET Solution File  
└── README.md # This file

## Technology Stack

* **Backend:** .NET 8, C#, ASP.NET Core Web API, Entity Framework Core (for data persistence concept), AutoMapper, Clean Architecture, xUnit, Moq, BCrypt.Net (for password hashing), JWT (for authentication)
* **Frontend:** Angular 19, TypeScript, NgRx (Store, Effects, Selectors), Tailwind CSS v3, RxJS, Karma, Jasmine

## Backend (.NET)

### Architecture

The backend follows Clean Architecture principles, separating concerns into distinct layers:

* **Domain:** Contains core entities (User, Match, Playlist) and domain logic.
* **Application:** Contains application logic, interfaces (repositories, services, unit of work), DTOs, and service implementations.
* **Infrastructure:** Provides implementations for interfaces defined in the Application layer, such as repositories using EF Core (conceptual) and potentially external services.
* **Api:** The ASP.NET Core Web API layer, responsible for handling HTTP requests, routing, controllers, and interacting with the Application layer.

## Technical Documentation: API Design & Best Practices

### Overview

The ClubberApp backend API is designed with modern, production-grade best practices in mind. The architecture and implementation focus on maintainability, security, scalability, and developer experience. Below is a summary of the key design decisions, their implementation status, and considerations for future growth.

### API Design Summary

| Area | Status | Details/Notes ||------------------------------------------|-------------|-------------------------------------------------------------------------------|| **Model validation** | ✅ Done | DTOs use [Required], [StringLength]; [ApiController] enables validation || **Consistent error handling** | ✅ Done | All controllers use Problem() for RFC 7807-compliant error responses || **Pagination (Matches & Playlists)** | ✅ Done | Generic PaginatedResult<T> used in both controllers and services || **Integration/unit testing** | ✅ Done | Integration tests for AuthController, including model validation || **Rate limiting** | ✅ Done | Middleware enabled in Program.cs || **Response caching** | ✅ Done | Middleware enabled for suitable endpoints (not on health check) || **Secure secret management** | ✅ Done | JWT secret read from environment variable if available || **Monitoring in health checks** | ✅ Done | Health check endpoint checks DB connectivity || **Clean code for health checks** | ✅ Done | Health logic encapsulated in HealthCheckService and injected into controller|| **Removal of obsolete TODOs** | ✅ Done | All TODO comments removed from codebase || **Built-in ASP.NET Core health checks** | ⚠️ Optional | Custom service used; built-in system available for advanced scenarios |

### Design Decisions & Rationale

* **Validation:**

All input DTOs are decorated with validation attributes, and [ApiController] ensures automatic model validation and 400 responses for invalid input.

* **Error Handling:**

The API uses the Problem() method for all error responses, ensuring consistency and compliance with RFC 7807 (Problem Details for HTTP APIs).

* **Pagination:**

Both matches and playlists endpoints use a generic PaginatedResult<T> for consistent, DRY pagination logic. This pattern is easily reusable for future endpoints.

* **Testing:**

Integration and unit tests are in place, especially for authentication and validation logic, ensuring reliability and maintainability.

* **Rate Limiting:**

Implemented at the API level using .NET middleware. For production, it is recommended to enforce global rate limits at the API gateway (e.g., NGINX, Azure API Management, AWS API Gateway) for better scalability and protection.

* **Response Caching:**

Enabled for suitable endpoints to improve performance. Caching strategy (API, gateway, or CDN) should be tailored to business needs and discussed with stakeholders.

* **Secret Management:**

JWT secrets are read from environment variables when available, following security best practices.

* **Health Checks:**

The health check endpoint verifies database connectivity in real time. The logic is encapsulated in a dedicated service for clean code and testability.> **Note:** For advanced scenarios, consider using the built-in ASP.NET Core health check system for ecosystem integration and UI/dashboard support.

* **Code Cleanliness:**

All obsolete TODOs have beenremoved from the codebase, ensuring clarity and maintainability for future developers and reviewers.

### App-Level Limits & Caching Strategy

* **App-Level Limits:**

While rate limiting is implemented at the API level, it is best practice to enforce global limits at the API gateway layer (such as NGINX, Azure API Management, or AWS API Gateway) for comprehensive protection and scalability.

* **Caching:**

Response caching is enabled for suitable endpoints within the API. However, the overall caching strategy—including whether to cache at the API, gateway, or CDN level—should be discussed and tailored to the application's business and infrastructure requirements.

### Recommendations for Future Growth

* **Consider adopting the built-in ASP.NET Core health check system** for advanced scenarios, such as integration with cloud-native health probes, dashboards, or multi-service health aggregation.
* **Expand integration and unit test coverage** as new features are added.
* **Review and update rate limiting and caching policies** as usage patterns and business needs evolve.
* **Document any changes to security or infrastructure practices** to ensure ongoing compliance and maintainability.

For further details or architectural discussions, please refer to the project maintainers or the extended technical documentation.

### Setup & Running

1. **Navigate to the API directory:** bash cd /home/ubuntu/clubber\_app/src/ClubberApp.API
2. **Restore dependencies:** bash dotnet restore ../../ClubberApp.sln
3. **Database Setup (Conceptual):**
   * This implementation uses an in-memory database for simplicity during development and testing. For a production scenario, you would configure a connection string in appsettings.json for a persistent database (e.g., PostgreSQL, SQL Server).
   * EF Core migrations would be used to manage the database schema: bash # Add migration (run from Infrastructure project directory or specify with --project) dotnet ef migrations add InitialCreate --startup-project ../ClubberApp.Api # Apply migration (run from Api project directory) dotnet ef database update
4. **JWT Configuration:**
   * Update the JwtSettings section in /home/ubuntu/clubber\_app/src/ClubberApp.Api/appsettings.json with your own secret key, issuer, and audience.

* "JwtSettings": {  
   "Key": "YOUR\_VERY\_SECURE\_SECRET\_KEY\_HERE\_LONGER\_THAN\_32\_BYTES",  
   "Issuer": "YourAppIssuer",  
   "Audience": "YourAppAudience"  
  }

1. **Run the API:** bash dotnet run The API will typically be available at http://localhost:5000 or https://localhost:5001 (check console output).

### Running Backend Tests

1. **Navigate to the solution root or tests directory:** bash cd /home/ubuntu/clubber\_app
2. **Run tests:** bash dotnet test *Note: Some API integration tests might still have compilation errors related to CustomWebApplicationFactory accessibility that need resolving.*

### API Endpoints

* POST /api/Auth/register: Register a new user.
* POST /api/Auth/login: Log in an existing user, returns JWT.
* GET /api/Matches: Get all available matches.
* GET /api/Playlist: Get the authenticated user’s playlist.
* POST /api/Playlist/{matchId}: Add a match to the playlist.
* DELETE /api/Playlist/{matchId}: Remove a match from the playlist.

## Frontend (Angular)

### Setup & Running

1. **Navigate to the frontend directory:** bash cd /home/ubuntu/clubber\_app/frontend
2. **Install dependencies:** bash npm install
3. **Configure API URL:**
   * Update the apiUrl in /home/ubuntu/clubber\_app/frontend/src/environments/environment.ts and environment.development.ts to point to your running backend API URL (e.g., http://localhost:5000).
4. **Run the development server:** bash ng serve The application will typically be available at http://localhost:4200/.

### Running Frontend Tests

1. **Navigate to the frontend directory:** bash cd /home/ubuntu/clubber\_app/frontend
2. **Run tests:** bash ng test --watch=false --browsers=ChromeHeadless *Note: The tests should now pass after the recent fixes. If issues persist, further debugging might be needed.*

## Next Steps & Improvements

* **Implement Real Data Source:** Replace the mock match data source with a real one (e.g., fetch from a third-party API or a database).
* **Database Persistence:** Fully implement database persistence using EF Core and a chosen database provider.
* **EF Core Migrations:** Implement and manage database schema changes using EF Core migrations.
* **API Integration Tests:** Fix the remaining compilation errors in the API integration tests.
* **Frontend Test Coverage:** Add more comprehensive unit and integration tests for the frontend components, services, and NgRx store.
* **Error Handling:** Enhance error handling on both backend and frontend.
* **UI/UX:** Improve the user interface and user experience.
* **Security:** Implement more robust security measures (e.g., input validation, rate limiting, HTTPS enforcement).
* **Deployment:** Create build pipelines and deployment scripts.