MediaRating - Projektdokumentation

Design, Lessons Learned, Unit Tests, SOLID u

Datum: 06.01.2026

Technologien: C# (.NET), HttpListener, ADO.NET (Npgsql), PostgreSQL, Docker Compose

Ziel der Anwendung
MediaRating ist eine einfache REST-ähnliche API, mit der Nutzer Medien (Movie/Series/Game) anlegen und Bewertungen (Ratings) erfassen können. Zusätzlich gibt es Authentifizierung per Token und Auswertungen wie Durchschnittsbewertung pro Medium.

Dokumentumfang

- App-Design: Architekturentscheidungen, Struktur und Diagramme
- Lessons Learned: Erkenntnisse aus Entwicklung und Debugging
- Unit-Testing-Strategie und Coverage (was getestet wird und warum)
- Mindestens zwei SOLID-Prinzipien mit Beispielen aus dem Projekt
- Zeiterfassung für zentrale Aufgaben

1. App-Design

1.1 Architekturentscheidungen

- Schichtenaufteilung: HTTP-Service (Transport), Controller (Use-Cases), Context (Datenzugriff).
- Persistenz: ADO.NET mit Npgsql, um SQL und Datenzugriff bewusst zu kontrollieren.
- IDs: GUIDs als öffentliche IDs (API), DB-IDs als interne Primary Keys, GUIDs werden in der DB per DEFAULT generiert.
- Containerisierung: PostgreSQL per Docker Compose, App kann lokal laufen

2. Lessons Learned

- Routing-Reihenfolge ist entscheidend: Spezifische Routen (z.B. /api/media/avg/{guid}) müssen vor generischen Routen (z.B. /api/media/{guid}) geprüft werden, sonst matcht der falsche Case.
- DB-Defaults reduzieren Fehler: Spalten wie guid, timestamp und confirmed sollten DEFAULT-Werte haben. Sonst schlagen Inserts mit NOT NULL-Spalten häufig fehl.
- FK-Constraints bewusst designen: Beim Löschen von MediaEntries sind Ratings betroffen. Mit ON DELETE CASCADE werden abhängige Ratings sauber mitgelöscht, alternativ kann man ein RESTRICT + sinnvolle Fehlermeldung wählen.
- DTOs schlank halten: Felder wie UserGuid sollten nicht vom Client kontrolliert werden, wenn der User ohnehin über Token bekannt ist. Das reduziert Sicherheitsrisiken und vereinfacht Tests.
- Fehlerdiagnose: Catch-all Exceptions verstecken Ursachen. Für Debugging hilft es, SQLState/MessageText temporär zu loggen.

3. Unit-Testing-Strategie und Coverage

3.1 Ziel

Unit Tests prüfen die Business-Logik in den Controllern und Modellen ohne echte Datenbank. Dazu wird der Datenzugriff über ein Interface (z.B. IMediaRatingContext) gemockt.

3.2 Testaufteilung

- UserControllerTests: Register/Login, Validierung, Statuscodes (400/404/409/200) und Token-Response.
- MediaControllerTests: Create/Update/Delete, Owner-Checks, NotFound etc.
- RatingControllerTests: Create (Stars-Validierung, Duplicate-Check), Update/Delete Owner-Checks und Statuscodes.
- Model-Tests: Reine Berechnungslogik wie MediaEntry.GetAverageScore() mit in-memory Ratings.

3.3 Abdeckung (Coverage)

Die Coverage konzentriert sich auf Validierung, Statuscodes, Fehlerpfade und Berechtigungen. SQL-Statements werden in Unit Tests nicht ausgeführt, deren Korrektheit würde man separat mit Integrationstests prüfen.

3.4 Mocking-Ansatz

Die Controller erhalten den Context als Interface. Im Test wird ein Mock erstellt, der definierte Rückgaben liefert (Arrange), dann wird die Controller-Methode aufgerufen (Act), und anschließend werden Status/Fehler/Response geprüft (Assert).

4. SOLID-Prinzipien mit Beispielen

4.1 Single Responsibility Principle (SRP)

SRP bedeutet: Eine Klasse hat genau eine Ursache, sich zu ändern. Im Projekt wurde SRP durch die Schichtenaufteilung umgesetzt:

- HTTP-Service ändert sich nur, wenn sich Routing/HTTP-Details ändern (Header, Pfade, Body parsing).
- Controller aendert sich nur, wenn sich Use-Cases/Business-Regeln ändern (Owner-Check, Validierung, Statuscodes).
- Context ändert sich nur, wenn sich das Datenmodell oder SQL ändert (Queries, Mapping)