1. Einführung

Vorstellung

Unser Entwicklungsteam besteht aus Paul und Erik, zwei Studenten des dualen Studiengangs, die beide bei ABB als Partner-Unternehmen tätig sind. Diese Konstellation bringt wertvolle Praxiserfahrung aus der Industrie in unser akademisches Projekt ein.

Erik ist im Bereich Home-Automation tätig und entwickelt intelligente Lösungen für moderne Wohnumgebungen. Seine Expertise umfasst IoT-Systeme, Benutzeroberflächen und die Integration verschiedener Technologien zu nahtlosen User Experiences. Diese Fähigkeiten sind besonders wertvoll für die Frontend-Entwicklung und die Gestaltung intuitiver Benutzerinteraktionen in unserem Projekt.

Paul arbeitet in einem spezialisierten Team, das MES-Lösungen (Manufacturing Execution Systems) entwickelt. In diesem Bereich beschäftigt er sich täglich mit komplexen Geschäftsprozessen, Datenintegration und der Entwicklung von Anwendungen, die kritische Produktionsabläufe steuern. Diese Erfahrung mit unternehmenskritischen Systemen und strikten Qualitätsanforderungen fließt direkt in die Architektur und Qualitätssicherung unseres Pokémon Arena-Projekts ein.

Verbindung zu Pokémon

Unsere gemeinsame Leidenschaft für Pokémon wurzelt in unserer Kindheit, als wir beide die ersten Generationen der Spiele erlebt haben. Diese frühe Begeisterung für das strategische Gameplay und die Mechaniken des Pokémon-Universums motiviert uns, eine Anwendung zu entwickeln, die sowohl technisch anspruchsvoll als auch für die Community wertvoll ist. Die Kombination aus professioneller Software-Entwicklungserfahrung bei ABB und persönlicher Begeisterung für das Pokémon-Franchise bildet die ideale Grundlage für ein Projekt, das sowohl die technischen Anforderungen unseres Software Engineering-Moduls erfüllt als auch echten Mehrwert für Pokémon-Enthusiasten schafft.

Motivation

Die Entscheidung für ein Pokémon Arena-Projekt basiert auf einer durchdachten Analyse verschiedener Faktoren, die sowohl die technischen Lernziele unseres Software Engineering-Moduls als auch die praktische Relevanz einer solchen Anwendung berücksichtigen.

Etabliertes Spielsystem mit bewährten Mechaniken

Pokémon ist eines der erfolgreichsten und langlebigsten Franchise der Videospielgeschichte mit über 25 Jahren kontinuierlicher Entwicklung. Die Spielmechaniken sind ausgereift, gut dokumentiert und bieten eine ideale Grundlage für ein Software-Projekt.

- **Komplexe, aber verständliche Regeln**: Das Kampfsystem kombiniert strategische Tiefe mit intuitiver Zugänglichkeit
- **Umfangreiche Datenbasis**: Über 1000 Pokémon mit jeweils individuellen Statistiken, Typen und Movesets
- **Klare Geschäftsregeln**: Eindeutige Validierungsregeln für Teams, Kämpfe und Pokémon-Konfigurationen

Lebendige Competitive Scene und Community

Die Pokémon Competitive Community ist eine der aktivsten Gaming-Communities weltweit und demonstriert den realen Bedarf für spezialisierte Tools:

Turnierszene und Esports:

- **Offizielle Weltmeisterschaften**: Jährliche Pokémon World Championships mit Preisgeld über $500.000
- **Regional Tournaments**: Hunderte lokale und regionale Turniere weltweit
- **Online Ladders**: Plattformen wie Pokémon Showdown mit über 10 Millionen monatlichen Battles
- **Content Creation**: Tausende Streamer und YouTuber produzieren täglich strategische Inhalte

Strategische Komplexität:

Das Pokémon-Kampfsystem bietet überraschende strategische Tiefe mit Elementen wie:

- **Team Synergy**: Pokémon müssen sich gegenseitig ergänzen und Schwächen kompensieren
- **Meta-Game Evolution**: Strategien entwickeln sich kontinuierlich weiter basierend auf neuen Entdeckungen
- **Prediction und Mindgames**: Psychologische Aspekte des Wettkampfs
- **Resource Management**: Optimale Nutzung von HP, PP und Statusveränderungen

Teambuilding als zentrale Herausforderung

Teambuilding ist der komplexeste und zeitaufwändigste Aspekt des competitive Pokémon-Spiels und bietet enormes Potenzial für Software-Unterstützung:

Aktuelle Herausforderungen für Spieler:

- **Overwhelming Choice**: Aus über 1000 Pokémon und tausenden Movesets optimale Teams zusammenstellen
- **Complex Calculations**: IV/EV-Optimierung, Schadenscalculations und Speed-Tiers manuell berechnen
- **Meta-Analysis**: Aktuelle Trends und Threats der Competitive Scene verfolgen
- **Testing und Iteration**: Teams gegen verschiedene Gegner testen und verfeinern

Unser Lösungsansatz:

Unsere Anwendung adressiert diese Pain Points durch:

- **Intuitive Team-Builder**: Intuitives Interface für schnelle Team-Erstellung
- **Automated Validation**: Sofortige Überprüfung auf legale Movesets und Regel-Compliance
- **Strategic Analysis**: Schwächen-Analyse und Verbesserungsvorschläge für Teams
- **Cloud Sync**: Teams geräteübergreifend verfügbar

Dateien im Anhang

- **Diese Datei im originalen markdown Format**
- **User-Stories:** Als Jira Export (png), sowie als lesbare Markdown
- **Code:** Abzug des Codes (pdf), strukturiert nach CLEAN Schichten

- **ER-Diagramm:** Ein png des in draw.io erstellten Entity-Relation Diagramms
- **Clean-Diagramm:** Ein Diagramm zur Übersicht des Backends ähnlich zu einem Klassen-Diagramm

PokeAPI

Die **PokeAPI** ist eine kostenlose und öffentlich zugängliche RESTful API, die umfassende Daten über das Pokémon-Universum bereitstellt. Sie fungiert als zentrale Datenquelle für alle Pokémon-bezogenen Informationen in unserem Projekt.

Was stellt die PokeAPI zur Verfügung?

Die PokeAPI bietet uns Zugang zu einer Vielzahl von Pokémon-Daten, die für unser Kampfsystem essentiell sind:

- **Pokémon-Grunddaten**: Namen, IDs, Typen, Sprites (Bilder) in verschiedenen Varianten
- **Detaillierte Stats**: Basis-Stats wie HP, Angriff, Verteidigung, Spezialangriff, Spezialverteidigung und Initiative
- **Movesets**: Alle verfügbaren Attacken pro Pokémon mit Details wie Stärke, Genauigkeit, PP und Schadensklasse
- **Fähigkeiten**: Verschiedene Abilities, die Pokémon besitzen können, inklusive versteckter Fähigkeiten
- **Evolutionsketten**: Verwandlungsmöglichkeiten und Evolutionsbedingungen
- **Items**: Berries, TMs, Held Items und andere spielrelevante Gegenstände

Die API folgt REST-Prinzipien und liefert Daten im JSON-Format, was eine einfache Integration in moderne Webanwendungen ermöglicht. Besonders wertvoll ist die Konsistenz der Datenstruktur, die es uns erlaubt, robuste Parser und Mapping-Funktionen zu implementieren.

Implementierung in unserem Projekt

Für die Integration nutzen wir die **PokeApiNet** NuGet-Library, die eine typisierte C#-Schnittstelle zur PokeAPI bereitstellt. Diese Library abstrahiert die HTTP-Anfragen und bietet uns stark typisierte Objekte für alle API-Endpunkte. Da wir eine lokale Instanz der PokeAPI verwenden, haben wir einen `CustomPokeApiClient` implementiert:

```
public class CustomPokeApiClient : PokeApiClient
{
    public CustomPokeApiClient() : base(new HttpClient
    {
        BaseAddress = new Uri("http://localhost:8080/api/v2")
    })
    {
    }
}
```

Diese Implementierung ermöglicht es uns, während der Entwicklung unabhängig von der Internetverbindung zu arbeiten und gleichzeitig bessere Performance durch reduzierte Latenz zu erreichen. Der lokale Server wird über Docker bereitgestellt und enthält eine vollständige Kopie aller PokeAPI-Daten.

Unsere Query-Handler nutzen die PokeAPI intensiv, wie beispielsweise der `GetAllPokemonsQueryHandler`, der Pokémon-Listen mit Paginierung abruft, oder der `GetPokemonDetailsQueryHandler`, der detaillierte Informationen inklusive aller verfügbaren Moves parallel lädt, um die Performance zu optimieren.

Warum haben wir uns für die PokeAPI entschieden?

1. **Vollständigkeit**: Enthält alle offiziellen Pokémon-Daten bis zur aktuellen Generation mit regelmäßigen Updates
2. **Standardisierung**: Konsistente Datenstruktur und bewährte API-Patterns, die industriellen Standards entsprechen
3. **Kostenlos**: Keine Lizenzgebühren oder API-Limits, was besonders für Bildungsprojekte wichtig ist
4. **Community-Support**: Aktive Entwicklergemeinschaft mit über 10.000 GitHub-Stars und gute Dokumentation
5. **Lokale Verfügbarkeit**: Möglichkeit einer lokalen Instanz für bessere Performance und Offline-Entwicklung
6. **Rechtssicherheit**: Offizielle Unterstützung durch The Pokémon Company für nicht-kommerzielle Projekte

Alternative Datenquellen wie Scraping von offiziellen Websites oder proprietäre APIs hätten rechtliche Risiken oder hohe Kosten bedeutet. Die PokeAPI bietet uns die perfekte Balance zwischen Datenqualität, Zugänglichkeit und rechtlicher Sicherheit.

2. Planung

Github Versionskontrolle

Für die Versionskontrolle unseres Projekts setzen wir auf **GitHub**, das uns als zentrale Plattform für Code-Management und Kollaboration dient. GitHub hat sich als Industriestandard für Open-Source- und kommerzielle Projekte etabliert und bietet uns eine umfassende DevOps-Plattform.

Unsere GitHub-Strategie umfasst:

- **Zentrale Code-Verwaltung**: Alle Teammitglieder haben Zugriff auf den aktuellen Projektstand über ein einheitliches Repository
- **Branch-Management**: Feature-Branches für isolierte Entwicklung neuer Funktionalitäten, um Konflikte zu minimieren
- **Pull Request Workflow**: Systematische Code-Reviews vor dem Merge in den Hauptbranch zur Qualitätssicherung

Unser Branching-Modell orientiert sich an dem **Git Flow**-Pattern mit einem stabilen `main`-Branch, und Feature-Branches für einzelne User Stories. Dies ermöglicht uns parallele Entwicklung ohne Interferenzen und gewährleistet, dass der Hauptbranch immer deploymentfähig bleibt. Dies haben wir leider nicht immer zu 100% umgesetzt und sind deshalb auf mehr Probleme gestoßen als es bei ordentlicher Ausführung gegeben hätte.

Jira User Stories

Für die strukturierte Planung und Verwaltung unseres Entwicklungsprojekts setzen wir auf Atlassian Jira in Kombination mit agilen Methodiken. User Stories bilden dabei das Herzstück unserer anforderungsbasierten Entwicklung und ermöglichen es uns, Features aus der Perspektive der Endbenutzer zu betrachten.

Die detaillierten User Stories unseres Pokémon Arena Projekts sind im Anhang als Export aus Jira beigefügt und zeigen die praktische Anwendung dieser Methodik in unserem konkreten Entwicklungskontext.

User Stories als Grundlage agiler Entwicklung

User Stories folgen dem bewährten Format "Als [Rolle] möchte ich [Funktionalität], damit [Nutzen]" und dienen als leichtgewichtige Anforderungsdokumentation. Im Gegensatz zu traditionellen, umfangreichen Spezifikationsdokumenten fokussieren sich User Stories auf den Wert für den Benutzer und fördern die Kommunikation zwischen Entwicklungsteam und Stakeholdern. Jede User Story in unserem Projekt enthält:

- Titel und Beschreibung: Klare, verständliche Formulierung der gewünschten Funktionalität
- Akzeptanzkriterien: Messbare Bedingungen für die Definition of Done
- Story Points: Relative Aufwandsschätzung basierend auf Komplexität und Unsicherheit
- Priority: Geschäftswert und technische Abhängigkeiten bestimmen die Reihenfolge
- Labels und Components: Kategorisierung für bessere Filterung und Reporting

Jira als zentrale Planungsplattform

Jira Software bietet uns eine umfassende Toolchain für agiles Projektmanagement: **Hierarchische Strukturierung**:

- Epics: Große Themenbereiche wie "Battle System" oder "User Management"
- Stories: Einzelne, implementierbare Features innerhalb der Epics
- Subtasks: Technische Arbeitsschritte zur Umsetzung der Stories
- Bugs: Defects mit Verlinkung zu den ursprünglichen Stories

3. Backend

Datenbank

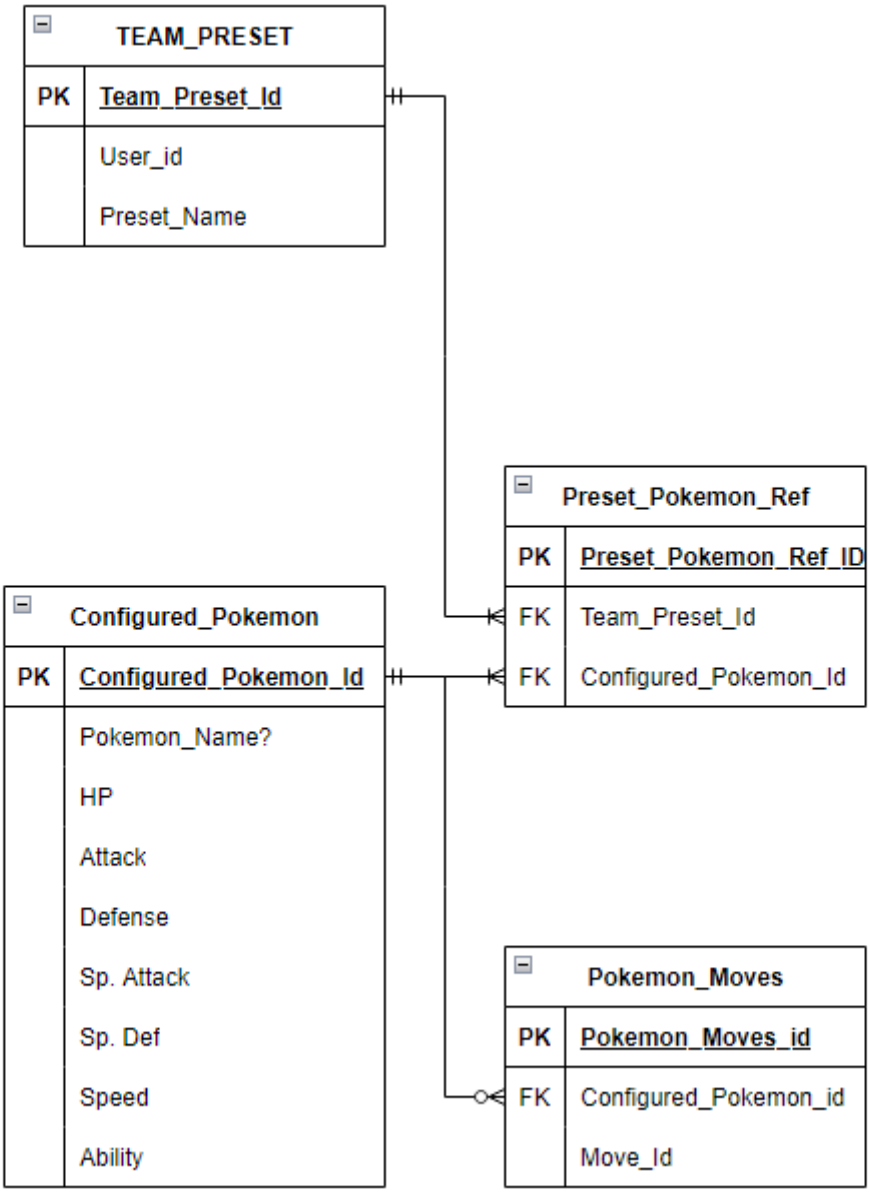
ER-Diagramm und Datenbankdesign

Unsere Datenbank basiert auf einem relationalen Modell, das die komplexen Beziehungen zwischen Pokémon, Teams und Benutzern abbildet. Das Design folgt den Prinzipien der Normalisierung bis zur dritten Normalform, um Datenredundanz zu minimieren und Integrität zu gewährleisten.

Zentrale Entitäten und ihre Beziehungen:

- **TEAM_PRESET**: Speichert Team-Konfigurationen mit `Team_Preset_Id` (PK), `User_Id` (FK) und `Preset_Name`
- **Configured_Pokemon**: Enthält individuell konfigurierte Pokémon mit vollständigen Statistiken, IVs, EVs, Fähigkeiten und Referenz zum Team
- **Pokemon_Moves**: N:M-Beziehungstabelle zwischen konfigurierten Pokémon und ihren Attacken mit `Pokemon_Moves_id` (PK)

- **Preset_Pokemon_Ref**: Verknüpfungstabelle zwischen Teams und konfigurierten Pokémon für flexible Team-Zusammenstellungen



Das Datenbankschema berücksichtigt die Anforderungen von Pokémon-Daten, einschließlich der Tatsache, dass jedes Pokémon individuell konfigurierbar ist (IVs, EVs, Movesets). Foreign Key Constraints stellen sicher, dass referentielle Integrität gewährleistet ist

Die Konfiguration erfolgt über Entity Framework Databaseode-First Migrations, was uns ermöglicht, Datenbankänderungen versioniert und reproduzierbar zu verwalten. Besondere Aufmerksamkeit haben wir der Behandlung von NULL-Werten gewidmet, da Pokémon-Daten viele optionale Felder enthalten können.

SQL Server als Datenbankmanagementsystem

Als Datenbankmanagementsystem nutzen wir **Microsoft SQL Server 2022**, der sich als robuste und skalierbare Lösung für unsere Anforderungen erwiesen hat:

Technische Vorteile:

- **ACID-Compliance**: Gewährleistung der Datenintegrität auch bei gleichzeitigen Transaktionen während Kämpfen
- **Advanced Security**: Row-Level Security für Benutzerdaten und Always Encrypted für sensitive Informationen
- **Integration**: Nahtlose Anbindung an das .NET-Ecosystem mit optimierten Datentypen und Funktionen
- **Skalierbarkeit**: Skalierung für wachsende Benutzerzahlen

C# .NET API

Backend Architecture (Clean Architecture)

Unser Backend implementiert die **Clean Architecture** nach Robert C. Martin, die eine klare Trennung der Verantwortlichkeiten und hohe Testbarkeit gewährleistet. Diese Architektur ermöglicht es uns, Geschäftslogik von technischen Details zu entkoppeln und macht unser System wartbar und erweiterbar.

Domain Layer (Innerste Schicht): Die Domain Layer enthält die reine Geschäftslogik ohne externe Abhängigkeiten:

- **Team Aggregate**: `Team.cs` mit Factory Methods für sichere Objekterstellung, `ConfiguredPokemon.cs` mit komplexer Validierungslogik, `Move.cs` für Attacken-Management
- **User Aggregate**: `User.cs` mit Team-Management-Funktionalitäten
- **Interfaces**: Definiert Contracts wie `TeamRequest`, `PokemonRequest` für typsichere Datenübertragung

Application Layer (Anwendungslogik): Orchestriert Geschäftsprozesse und implementiert Use Cases:

- **Commands**: `CreateTeamCommand` mit zugehörigem `CreateTeamCommandHandler` für Write-Operationen
- **Queries**: `GetAllPokemonsQuery`, `GetPokemonDetailsQuery` für Read-Operationen nach CQRS-Pattern
- **Extensions**: `MappingExtensions` für effiziente DTO-Konvertierung

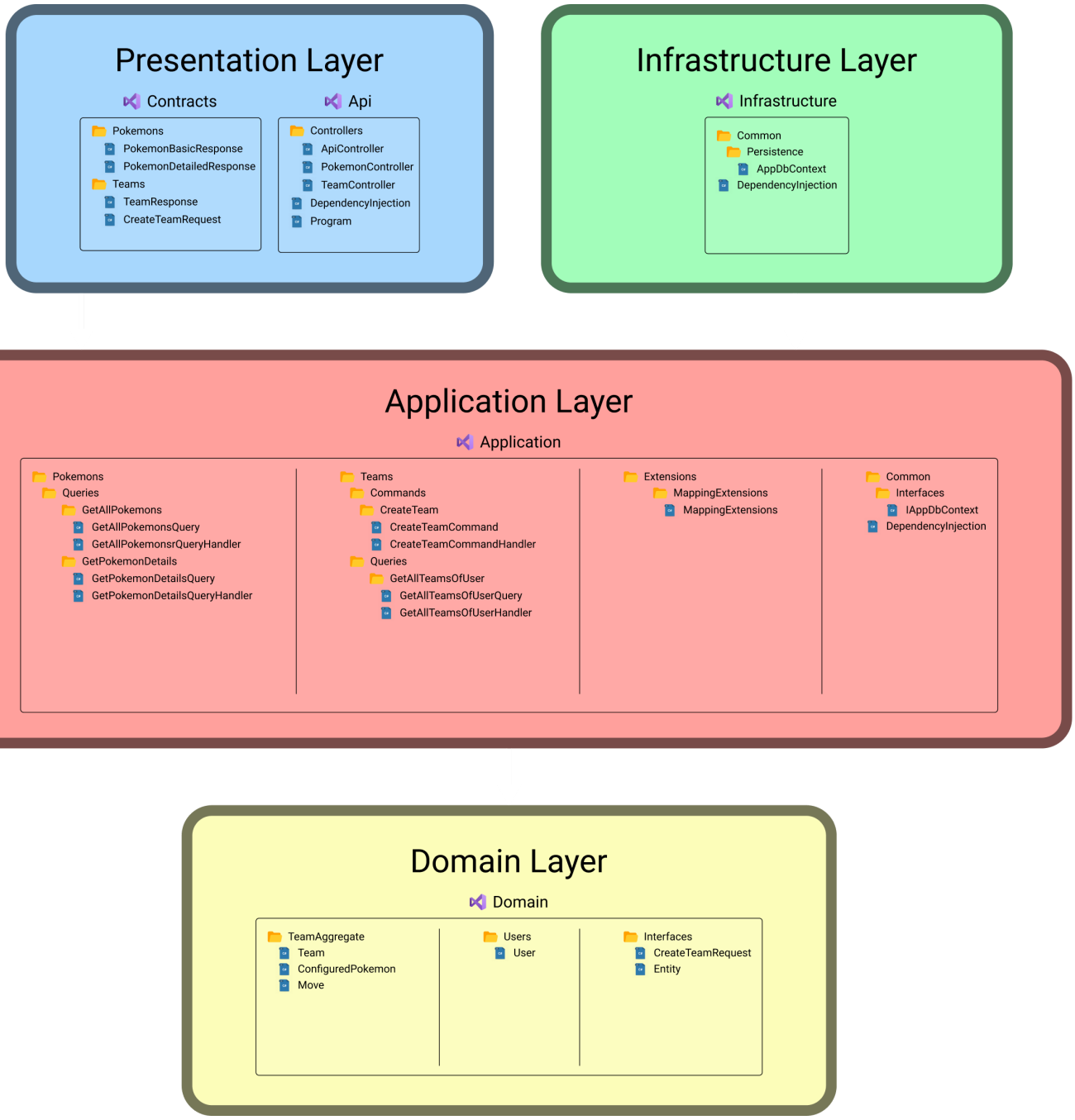
Infrastructure Layer (Äußerste Schicht): Implementiert technische Details und externe Abhängigkeiten:

- **Persistence**: `AppDbContext` für Entity Framework mit optimierten Queries und Change Tracking
- **External APIs**: `CustomPokeApiClient` für PokeAPI-Integration mit Retry-Policies und Caching

Presentation Layer (API-Schicht): Stellt HTTP-Endpunkte bereit und handhabt Request/Response-Zyklen:

- **Controllers**: `PokemonController`, `TeamController` mit RESTful Endpoints und Swagger-Dokumentation
- **Contracts**: DTOs wie `PokemonBasicResponse`, `TeamResponse` für optimierte Datenübertragung
- **Middleware**: Globale Error Handling, Logging und Authentifizierung

Die Dependency Inversion ermöglicht es uns, Abhängigkeiten durch Dependency Injection zu verwalten und macht unser System einfach testbar. Jede Schicht kann isoliert getestet werden, was zu einer hohen Code-Qualität führt.



Entity Framework Core ORM

Wir verwenden **Entity Framework Core 7.0** als Object-Relational Mapper, der uns eine moderne und performante Datenzugriffsschicht bietet:

Implementierungsdetails:

- **DbContext**: `AppDbContext` verwaltet alle Entitäten
- **Migrations**: Durch den Databse-First approach wurden Migrations als Basis für die Entititäten verwendet
- **LINQ-Unterstützung**: Typsichere Datenbankabfragen mit Expression Trees für optimale SQL-Generierung

EF Core bietet uns erweiterte Features wie Lazy Loading für verwandte Entitäten, Global Query Filters für Soft Deletes und Shadow Properties für Audit-Felder, was uns maximale Kontrolle über das Mapping gibt:

```
modelBuilder.Entity<ConfiguredPokemon>(entity =>
{
    entity.HasMany(p => p.Moves)
          .WithOne()
          .HasForeignKey(m => m.ConfiguredPokemonId);

    entity.Property(e => e.Name)
          .HasMaxLength(50)
          .IsRequired();
});
```

ErrorOr für funktionale Fehlerbehandlung

Das **ErrorOr** Pattern verbessert unsere Fehlerbehandlung durch funktionale Programmierkonzepte und macht Fehler zu "First-Class Citizens" in unserem System:

```
    public ErrorOr<Success> AddMove(ConfiguredMove move)
    {
        if (_moves.Count >= 4)
            return Error.Conflict(description: "A Pokémon cannot have more than 4
moves");

        if (_moves.Any(m => m.MoveId == move.MoveId))
            return Error.Conflict(description: "This move is already added to this
Pokémon");

        _moves.Add(move);
        return Result.Success;
    }
```

Kernvorteile des ErrorOr Patterns:

- **Railway-Oriented Programming**: Explizite Erfolgs- und Fehlerbehandlung ohne Exception-Overhead
- **Typsicherheit**: Compiler-unterstützte Fehlerbehandlung
- **Komposition**: Verkettung von Operationen mit automatischer Fehler-Propagation
- **Lesbarkeit**: Klare Trennung zwischen Success Path und Error Handling
- **Performance**: Keine Exception-Kosten für erwartete Fehlerfälle

ErrorOr integriert sich nahtlos mit unserem MediatR-basierten CQRS-Pattern und ermöglicht konsistente Fehlerbehandlung über alle Anwendungsschichten hinweg.

Unit Tests

Umfassende Äquivalenzklassen-Analyse

Für unsere Unit Tests haben wir eine systematische Äquivalenzklassen-Analyse durchgeführt, die alle kritischen Geschäftsregeln und Edge Cases abdeckt:

Testbereich	Gültige Klassen	Ungültige Klassen
Team Creation	Name: 1-50 Zeichen, User ID > 0	Name: leer/null/51+ Zeichen, User ID ≤ 0
Pokemon Configuration	IVs: 0-31, EVs: 0-252	HP: ≤ 0 oder ≥ 1000, Stats/IVs/EVs außerhalb
Move Validation	Move ID > 0, max. 4 Moves, legale Kombinationen	Move ID ≤ 0, > 4 Moves, illegale Kombinationen
Team Size	1-6 Pokémon, keine Duplikate	0 oder > 6 Pokémon, identische Pokémon

Diese Klassifizierung berücksichtigt die komplexen Regeln des Pokémon-Universums und stellt sicher, dass unser System robust gegen alle möglichen Eingabeszenarien ist.

XUnit Framework mit modernen Testing Patterns

Wir verwenden **XUnit 2.4** als primäres Test-Framework, ergänzt durch moderne Testing-Bibliotheken für maximale Effektivität:

Kern-Features unserer Test-Suite:

- **Fact/Theory Attributes**: Unterscheidung zwischen einfachen Tests und datengetriebenen Parametertests
- **Parallel Execution**: Bis zu 80% Zeitersparnis durch parallele Testausführung
- **Dependency Injection**: Nahtlose Integration mit .NET Core DI Container für realistische Tests
- **Custom Attributes**: Eigene Test-Kategorien für Integration-, Unit- und Performance-Tests

Test-Organisation:

```
    [Theory]
    [InlineData(null)]
    [InlineData("")]
    [InlineData(" ")]
    [InlineData("\t")]
    [InlineData("\n")]
    public void CreateFromRequest_WithInvalidName_ShouldReturnValidationError(string
invalidName)
    {
        // Arrange
        // Act
        // Assert
    }
```

Fluent Assertions für ausdrucksstarke Tests

Fluent Assertions 6.0 transformiert unsere Tests in selbstdokumentierenden Code mit natürlichsprachlichen Assertions:

Assertion-Beispiel:

```
// Assert
result.IsError.Should().BeTrue();
result.FirstError.Type.Should().Be(ErrorType.Validation);
result.FirstError.Description.Should().Be("Pokemon name cannot be empty");
```

Testabdeckung: Unser aktueller Test-Suite umfasst 50+ **Unit Tests** mit 95%+ Code Coverage der Domain Layer

Die Tests werden automatisch bei jedem Build ausgeführt.

4. Frontend

Integrationstest Beispiel Konzept (login)??

-Regresseionstest

-top-down, bottom-up

6. Fazit und Ausblick