#25 TECHARTIKEL

Response Contract Namenskonventionen: Cross-Layer Lesbarkeit für Menschen und AI

10.12.2025

Ahmed Saleh

Inhalt

Response Contract Namenskonventionen: Cross-Layer Lesbarkeit für Menschen und AI

Data Contracts sind gute Kandidaten dafür, chaotisch zu werden, da sie verschiedene Formen haben. Hier etablieren wir Konventionen, die dabei helfen, die meisten verschiedenen Formen abzudecken und dabei Klarheit und Konsistenz zu bewahren.

Solide Konventionen dafür bieten besseres Verständnis und helfen AI Agents dabei, das Pattern zu erkennen und ähnliche Use-Cases zu implementieren, ohne sie durcheinander zu bringen.

1. Was man NICHT machen sollte: Schlechte Namens-Anti-Patterns

1.1. Mehrdeutiger Zweck

Namen wie *FacilityModel, FacilityData, FacilityOutput* geben keinen Hinweis darauf, ob es sich um eine Zusammenfassung, Detailansicht oder spezialisiertes Ergebnis handelt.

```
// SCHLECHT - Unklar, was das darstellt
public class FacilityModel { }

// GUT - Klarer Zweck
public class SingleFacilityResponse { }
```

1.2. Überladen über Endpoints

Den gleichen DTO-Namen für verschiedene API-Responses verwenden (z.B. FacilityResponse für sowohl List- als auch Detail-Endpoints).

```
// SCHLECHT - Gleicher Name für verschiedene Zwecke
public class FacilityResponse { } // Wird für sowohl Single als auch List verwendet

// GUT - Unterschiedliche Namen für unterschiedliche Zwecke
public class SingleFacilityResponse { } // Einzelne Entity
public class FacilityListResponse { }   // List Wrapper
```

1.3. An Request-Verben gekoppelt

Namen wie GetFacilityResponse, FetchFacilityOutput — macht es so, als wäre es an einen Endpoint gebunden, anstatt ein wiederverwendbarer Contract zu sein.

```
// SCHLECHT - An spezifisches HTTP Verb/Aktion gebunden
public class GetFacilityResponse { }

// GUT - Wiederverwendbarer Contract
```

```
public class SingleFacilityResponse { }
```

1.4. Zu Implementation-orientiert

Namen, die technische oder Persistence-Details offenlegen (z.B. FacilityDbRow, FacilityElasticResult).

```
// SCHLECHT - Legt Implementation-Details offen
public class FacilityDbRow { }
public class FacilityElasticResult { }

// GUT - Fokussiert auf Business-Zweck
public class SingleFacilityResponse { }
```

1.5. Versioniert im Namen

FacilityResponseV2 oder *NewFacilityOutput* — altert schlecht und erzeugt Namens-Chaos.

```
// SCHLECHT - Versionierung im Namen erzeugt Chaos
public class FacilityResponseV2 { }
public class NewFacilityOutput { }

// GUT - Verwende ordentliche Versionierungs-Strategien
public class SingleFacilityResponse { } // Aktuelle Version
// Handle Versions durch API-Versionierung, nicht durch Namensgebung
```

1.6. Abgekürzt oder unklar

Akronyme oder Kurzschreibweisen wie *FacResp*, *FcltyDto*, die nicht universell offensichtlich sind.

```
// SCHLECHT - Unklare Abkürzungen
public class FacResp { }
public class FcltyDto { }

// GUT - Klare, lesbare Namen
public class SingleFacilityResponse { }
```

```
public class FacilityDto { }
```

2. Namens-Komponenten

Unsere Konvention verwendet diese Komponenten in verschiedenen Kombinationen:

- **Context**: *Single, Detailed, B2C, B2B*, etc.
- **ModelName**: Der Domain-Entity-Name (z.B. *Facility, Product*)
- **Extension**: *Summary, List, PagedList, With[RelatedEntity]*, etc.
- **Suffix**: *Response, Dto, ReadModel* (optional)

Hinweis: Komponenten werden basierend auf dem spezifischen Response-Typ und Zweck kombiniert, nicht immer in der gleichen Reihenfolge.

3. Konventionen & Beispiele

3.1. Einzelne Entity Responses

Namens-Pattern: *[Single][ModelName]Response*

SingleFacilityResponse

```
// Verwendung: GET /facilities/{id} gibt SingleFacilityResponse zurück
public class SingleFacilityResponse
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public int TotalRooms { get; set; }
    public DateTime CreatedAt { get; set; }
    // andere Properties...
}
```

3.2. Zusammengefasste Entity Responses

Namens-Pattern: *[ModelName][Summary]*

FacilityResponse (Summary)

```
// Verwendung: GET /facilities gibt IEnumerable<FacilityResponse> zurück
public class FacilityResponse
{

    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    // andere Properties...
}
```

3.3. List Responses

Namens-Pattern: *[ModelName][List]*

FacilityList

```
// Verwendung: GET /facilities gibt FacilityList zurück
public class FacilityList
{
    public IEnumerable<FacilityResponse> Data { get; set; }
}
```

3.4. Paged List Responses

Namens-Pattern: *[ModelName][Paged][List]*

FacilityPagedList

```
// Verwendung: GET /facilities?page=1&size=20 gibt FacilityPagedList zurück
public class FacilityPagedList : PaginatedList<FacilityResponse>
{
}

// Generische Basis-Klasse für Wiederverwendbarkeit
```

```
public class PaginatedList<T>
{
    public IEnumerable<T> Data { get; set; }
    public int PageNumber { get; set; }
    public int PageSize { get; set; }
    public int TotalCount { get; set; }
    public int TotalPages { get; set; }
}
```

3.5. Context-Aware Extended Responses

Namens-Pattern: *[Single][ModelName][Context][Response]*

SingleProductB2CResponse

```
// Verwendung: GET
/products/{id}?context=b2c gibt SingleProductB2CResponse zurück
public class SingleProductB2CResponse
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public decimal RetailPrice { get; set; }
    public string CustomerFriendlyDescription { get; set; }
    public List<string> CustomerBenefits { get; set; }
    // andere Properties...
}
```

SingleProductB2BResponse

```
// Verwendung: GET /products/{id}?context=b2b gibt SingleProductB2BResponse
zurück
public class SingleProductB2BResponse
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public decimal WholesalePrice { get; set; }
    public string TechnicalSpecifications { get; set; }
    public List<string> BusinessFeatures { get; set; }
    // andere Properties...
}
```

3.6. Managed Aggregates & Reports

Namens-Pattern: *[ModelName][WithRelatedEntity][List]* oder *[ModelName][WithRelatedEntity][Report]*

FacilityWithRoomsRenovationList

```
// Verwendung: GET /facilities/reports/rooms-renovation gibt FacilityWithRoomsRenovationList zurück
public class FacilityWithRoomsRenovationList
{
    public IEnumerable<FacilityWithRoomsRenovationData> Data { get; set; }
}


public class FacilityWithRoomsRenovationData
{
    public Guid FacilityId { get; set; }
    public string FacilityName { get; set; }
    public int RoomsRequiringRenovation { get; set; }
    public decimal TotalSquareMeters { get; set; }
    // andere Properties...
}
```

FacilityWithRoomsRenovationReport

```
// Verwendung: GET /facilities/reports/rooms-renovation gibt FacilityWithRoomsRenovationReport zurück
public class FacilityWithRoomsRenovationReport
{
    public IEnumerable<FacilityWithRoomsRenovationData> Facilities { get; set; }
    public int TotalFacilities { get; set; }
    public int TotalRoomsRequiringRenovation { get; set; }
    public decimal TotalSquareMeters { get; set; }
    public DateTime ReportGeneratedAt { get; set; }
    // andere Properties...
}
```

3.7. Alternative Plural-Formen

Namens-Pattern: *[ModelName|PL|][WithRelatedEntity]* (wenn kein Suffix Pluralität impliziert)

FacilitiesWithRoomsRenovation

```
// Verwendung: GET /facilities/reports/rooms-renovation gibt FacilitiesWithRoomsRenovation zurück
public class FacilitiesWithRoomsRenovation
{
    public IEnumerable<FacilityWithRoomsRenovationData> Data { get; set; }
}
```

4. Clean Architecture Prinzipien: DTO Contract Trennung über Layers

4.1. Die Dependency Rule für Data Contracts

Clean Architecture Prinzipien folgend sollten DTOs und Data Contracts nie über architektonische Grenzen hinweg durchsickern. Jeder Layer sollte seine eigenen Data Transfer Objects definieren, die seinem spezifischen Zweck dienen, das Dependency Inversion Principle beibehalten und verhindern, dass Infrastructure-Concerns Business Logic kontaminieren.

4.2. Layer Contract Verantwortlichkeiten

Domain Layer

- Zweck: Reine Business Logic und Entities
- Contracts: Domain Entities, Value Objects, Domain Services
- Regel: Keine externen Dependencies, keine Infrastructure-Concerns

Application Layer

- Zweck: Use Cases, Application Services, Orchestration
- Contracts: Commands, Queries, Application DTOs, Response Contracts
- Regel: Hängt nur von Domain ab, definiert Input/Output Contracts

Infrastructure Layer

- Zweck: Externe Concerns (Databases, APIs, Messaging)
- Contracts: Database Entities, External API Models, Persistence Models
- Regel: Implementiert Interfaces, die vom Application Layer definiert werden

Presentation Layer

- Zweck: User Interface, API Controllers, Views
- Contracts: View Models, API Request/Response DTOs
- Regel: Hängt von Application Layer Contracts ab, nicht von Infrastructure

4.3. Contract Transformations-Strategie

Wenn Data zwischen Layers fließt, verwende explizite Transformation anstatt geteilte Contracts:

```
// Domain Entity (reine Business Logic)
public class Facility
{
    public Guid Id { get; private set; }
    public string Name { get; private set; }
    public Address Address { get; private set; }

    public void UpdateName(string newName)
    {
        if (string.IsNullOrWhiteSpace(newName))
            throw new ArgumentException("Name cannot be empty");
        Name = newName;
    }
}

// Application Response Contract
public class SingleFacilityResponse
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string AddressLine1 { get; set; }
    public string City { get; set; }
```

```
    public string State { get; set; }
}

// Infrastructure Persistence Model
public class FacilityDbEntity
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string AddressLine1 { get; set; }
    public string AddressLine2 { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string PostalCode { get; set; }
    public DateTime CreatedAt { get; set; }
    public DateTime ModifiedAt { get; set; }
}
```

4.4. Mapping-Strategien zwischen Layer Contracts

Wenn du Data zwischen Layers transformierst, verwende explizites Mapping anstatt geteilte Contracts. Hier sind häufige Ansätze:

AutoMapper/Mappster (Library-basiertes Mapping)

```
// Mapping Profile konfigurieren
CreateMap<FacilityDbEntity, SingleFacilityResponse>();
// Verwendung: _mapper.Map<SingleFacilityResponse>(dbEntity)
```

Implicit Operators (Sprach-Feature)

```
public static implicit operator SingleFacilityResponse(FacilityDbEntity entity)
// Verwendung: SingleFacilityResponse response = dbEntity;
```

Manual Assignment (Manuelles Property-Kopieren)

```
return new SingleFacilityResponse { Id = entity.Id, Name = entity.Name };
```

4.5. Vorteile der Contract-Trennung

- **Wartbarkeit**: Änderungen in einem Layer beeinflussen andere nicht
- **Testbarkeit**: Jeder Layer kann isoliert getestet werden
- **Flexibilität**: Infrastructure kann sich ändern, ohne Business Logic zu beeinflussen
- **Klarheit**: Klare Grenzen und Verantwortlichkeiten
- **Evolution**: Jeder Layer kann sich unabhängig entwickeln

4.6. Wann man die Regel brechen sollte (Pragmatische Überlegungen)

Während strikte Trennung ideal ist, gibt es pragmatische Szenarien, wo etwas Contract-Wiederverwendung akzeptabel sein könnte:

- **Read-only DTOs**: Wenn der Contract dem gleichen Zweck über Layers hinweg dient
- **Geteilte Konstanten**: Enums, Status Codes oder Konfigurations-Werte
- **Cross-cutting Concerns**: Logging, Metrics oder Audit-Informationen

Aber bevorzuge immer explizite Transformation über geteilte Contracts, wenn du unsicher bist.

4.7. Praktisches Beispiel: Bewusstes Regel-Brechen

Manchmal ist der pragmatische Ansatz, ein geteiltes Responses-Projekt zu erstellen, das sowohl Application- als auch Presentation-Layer referenzieren können:

```
AwesomeSolution/
├── AwesomeSolution.Domain/           # Reine Business Logic
├── AwesomeSolution.Application/      # Use Cases, Services
├── AwesomeSolution.Infrastructure/   # Externe Concerns
├── AwesomeSolution.Presentation/     # Controllers, Views
└── AwesomeSolution.Responses/        # Geteilte Response Contracts
```

Hinweis: Wenn du die gleiche Sprache für das Frontend verwendest (z.B. Blazor auf ASP.NET Core), kannst du den Responses-Ordner eine Ebene nach oben verschieben, um Wiederverwendung sowohl in Frontend- als auch API-Contracts zu ermöglichen:

```
AwesomeSolution/
├── AwesomeSolution.Responses/        # Geteilte Response Contracts (nach oben verschoben)
├── AwesomeSolution.Server/           # ASP.NET Core API
│   ├── AwesomeSolution.Domain/       # Reine Business Logic
│   ├── AwesomeSolution.Application/  # Use Cases, Services
│   ├── AwesomeSolution.Infrastructure/ # Externe Concerns
│   └── AwesomeSolution.Presentation/ # Controllers, Views
└── AwesomeSolution.Client/           # Blazor Frontend
```

4.8. Timeline-Vergleich

Timeline 1: Explizite Isolation mit Mapping

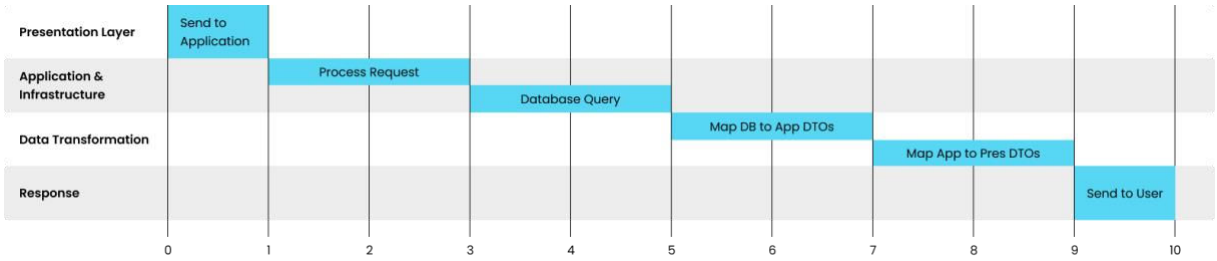

Abbildung 1: Explizite Isolation: Data Flow Timeline

Timeline 2: Geteilte Responses (Pragmatischer Ansatz)

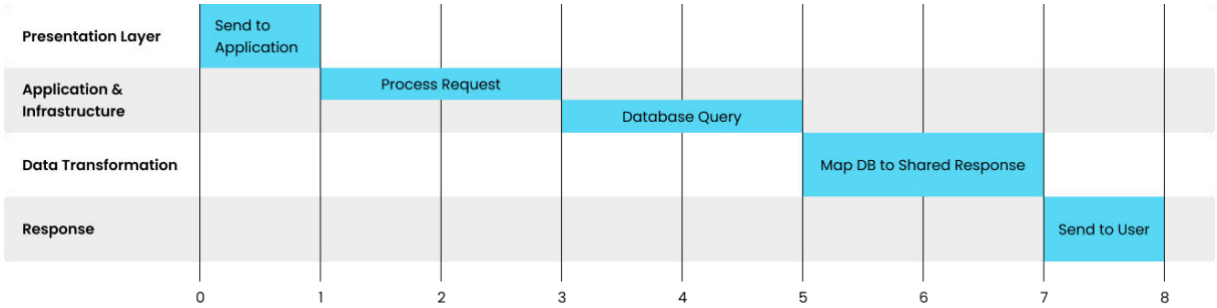

Abbildung 2: Geteilte Responses: Data Flow Timeline

Hinweis: Beachte den reduzierten Data Transformation Lane - nur ein Mapping-Schritt (DB → Shared Response) anstatt zwei separate Transformationen. Das eliminiert die Notwendigkeit, zwischen Application- und Presentation-DTOs zu mappen, reduziert Komplexität und potenzielle Mapping-Fehler.

5. Zusammenfassung

Es gibt keine Einheitslösung für alle. Jedes Team kann seine eigenen Namens-Patterns definieren, solange sie klar, konsistent, ausdrucksstark und erweiterbar bleiben. Das Ziel ist es, Contracts sowohl für Menschen als auch für AI lesbar zu halten und dabei an reale Bedürfnisse anzupassen.

Wichtige Prinzipien:

1. Konsistenz: Verwende das gleiche Pattern über ähnliche Response-Typen hinweg
2. Klarheit: Namen sollten selbsterklärend sein
3. Erweiterbarkeit: Patterns sollten zukünftige Variationen ermöglichen
4. Eindeutigkeit: Vermeide Namenskonflikte in Namespaces/Packages

Denk dran: Gute Namensgebung bedeutet klare Kommunikation. Deine Contracts sollten für sich selbst sprechen, ohne dass Entwickler in Implementation-Details oder Dokumentation graben müssen.