IT Project Guidance

Rationale and using Data Transfer Objects (DTOs)

Version: 0.1

Purpose

This document provides a rationale and best practice guidance for the use of Data Transfer Objects (DTOs) in API design. It aims to support enterprise system architects, developers, and integration leads in structuring system boundaries in a way that promotes security, maintainability, and long-term operability. The guidance is platform-agnostic and applies to any modern service-oriented or API-driven system.

Synopsis

Data Transfer Objects (DTOs) serve as the formal contract between services and their consumers. They represent the data structure intended for external consumption, separated from internal system models. This distinction is foundational for creating APIs that are secure, adaptable, and resilient to change. This document outlines why DTOs are necessary, how they differ from internal entities, and how to apply them in practice. It also addresses concerns around maintainability, versioning, and dynamic query support.

Contents

Introduction

As enterprise systems increasingly expose APIs to external consumers—ranging from internal teams to external partners and third-party developers—the need for clear, stable, and intentional data boundaries becomes critical. One of the most important tools for enforcing these boundaries is the use of Data Transfer Objects (DTOs). DTOs provide a mechanism for encapsulating and shaping data specifically for external consumption, decoupling external interfaces from internal representations. This approach enables a clear distinction between the service's internal model of the world and how it presents that model to others.

Data Transfer Objects (DTOs)

Data Transfer Object is a structured representation of data specifically designed for transfer across process boundaries. In the context of APIs, a DTO acts as an output-facing construct that is deliberately shaped for consumption by clients, whether they be other internal services, external partners, or user-facing applications. Unlike internal domain entities, DTOs are intentionally limited in scope. They usually contain only the fields required by the consumer, using simple types or flattened structures to facilitate compatibility, clarity, and ease of deserialization. DTOs help reduce cognitive and technical coupling between systems by limiting what is exposed and ensuring a stable, predictable contract.

Why Not Expose Internal Entities?

While modern development tools make it technically convenient to expose domain entities directly, this practice introduces several critical risks and should be avoided in all but the most limited and controlled cases. Internal entities are not designed with consumer requirements in mind—they often include system-specific metadata, internal identifiers, or navigation properties that are irrelevant or inappropriate for external users. Exposing entities directly can inadvertently disclose sensitive data or implementation details, increasing the attack surface and violating the principle of least privilege.

Beyond security, exposing internal models creates fragile coupling between internal system logic and external consumers. This makes routine internal changes—such as renaming a field or optimising a relationship—far more costly, as such changes could break external integrations. In contrast, DTOs allow internal models to evolve freely, as long as the contract represented by the DTO remains intact. This encapsulation fosters greater flexibility and long-term maintainability while improving consumer trust and autonomy.

The Role of DTOs in System Design

The Role of DTOs in System Design DTOs act as the formal boundary between the internal implementation of a service and the outside world. They are the basis of a service’s public contract, which is why their design must be intentional, governed, and version-controlled. By treating DTOs as a first-class artefact of system design, teams can better manage change, document functionality, and test interactions in isolation from business logic or storage models.

This separation is critical in enabling scalable, service-oriented architectures. With DTOs, each service defines its interface clearly and independently, which reduces interdependencies and encourages modularity. When implemented consistently, DTOs allow changes in one part of the system to occur without unintended ripple effects elsewhere. Their presence also improves observability and auditability by enabling predictable payload structures, which simplifies logging, validation, and diagnostics.

Versioning DTOs

DTOs must support the ability to evolve over time without disrupting existing consumers. Versioning is the mechanism that allows this. Several versioning strategies exist, including placing version numbers in URL paths (e.g., /v1/products), using custom media types to negotiate format versions via headers, or explicitly versioning the DTO class names within codebases (e.g., ProductV1Dto, ProductV2Dto).

The choice of strategy depends on the system’s design and governance model, but all approaches require consistent policy enforcement. Versioning enables services to introduce breaking changes in a controlled way, while still supporting clients that rely on older versions. It supports gradual migration and coexistence—essential in enterprises where consumers may have different release cadences or constrained ability to update.

Mapping Between Entities and DTOs

DTOs are often derived from internal entities or aggregates, but this mapping must be intentional and unidirectional. The process of mapping ensures that only the appropriate data is exposed and that it is shaped in a way that reflects consumer expectations. Mapping rules define how to project internal fields, transform values, and resolve nested structures or flattened outputs.

This mapping layer is also where developers can enforce domain boundaries, apply validation rules, and introduce derived fields that improve client usability. It should be automated wherever possible, using tools like AutoMapper or similar, but also be auditable and testable to prevent regressions. Mapping logic is not a trivial afterthought—it is the connective tissue between internal logic and external usability.

Querying DTOs Dynamically

In many enterprise scenarios, service consumers need to retrieve data in flexible and context-specific ways. Rather than building one-off endpoints for each new use case, systems can expose queryable DTOs using mechanisms like OData or GraphQL. These allow clients to filter, sort, paginate, and project only the data they need—without expanding the backend codebase.

The .NET ecosystem, through its support for Language Integrated Query (LINQ), uniquely enables this pattern at scale. LINQ allows for client-composed queries to be expressed as strongly typed expression trees and then securely translated into database-level operations using OData. This deep integration eliminates boilerplate, simplifies DTO projection, and aligns closely with maintainable, typed development practices.

In contrast, ecosystems without LINQ must often fall back on GraphQL to achieve equivalent flexibility. While GraphQL is powerful, it requires complex schema definitions, resolver infrastructure, and a disciplined governance model to enforce contract and performance controls. It also introduces runtime complexity that .NET avoids through native query translation and middleware constraints.

The absence of built-in, declarative, and safe querying features in most stacks explains why developers in non-.NET environments reach for GraphQL once REST becomes unsustainable. In .NET, this same problem space is addressed more naturally, without bespoke middleware or client-overburdening workarounds.

This distinction has strategic consequences: it allows .NET-based APIs to expose DTOs that are both structured and deeply flexible. Teams can build one endpoint—governed, documented, and tested—and allow integrators, analysts, and third-party consumers to shape their own outputs within guardrails. This dramatically reduces delivery cycles, backlog volume, and integration friction.

Contrary to persistent myths, neither OData and GraphQL inherently create security risks. Denial-of-service concerns raised about complex or deeply nested queries are readily addressable with straightforward configuration, such as maximum query complexity or server-side query validation. When implemented with proper controls, queryable DTOs represent one of the most effective tools for reducing API bloat and increasing consumer independence.

Common Anti-Patterns

There are several recurring mistakes in the use of DTOs that weaken system boundaries and should be actively avoided. One is the direct exposure of internal entities via automated serialisation. This often begins innocently during prototyping but rapidly leads to entangled, brittle APIs.

Another issue is the overloading of DTOs with internal logic or transformation code, which turns them into hybrids of model and service layer concerns. DTOs should remain passive data structures—not execution points for business logic.

Other anti-patterns include the use of DTOs as database projection models (bypassing proper mapping), ignoring validation on inbound DTOs, or creating DTOs that mirror internal structures without abstraction or user-focused shaping. These patterns erode the intended decoupling and undermine the reliability of public interfaces.

When DTOs Aren’t Necessary

There are valid cases where DTOs may not be required. Internal microservices operating within a tightly coupled domain boundary, or systems where no external consumers are expected, may use shared models safely. However, this must be a conscious architectural decision, not a default.

As soon as a service boundary crosses team lines, exposes sensitive data, or risks unplanned schema evolution, DTOs become essential. Their introduction should be proactive, not reactive. The cost of introducing DTOs later—after coupling has taken hold—is far greater than defining them up front.

Demonstration

A basic illustrative example is included in Appendix B. It uses .NET Core and Entity Framework Core to demonstrate the conceptual parts of a DTO-based API pattern: how internal models are kept isolated, how mappings are registered, and how versioned DTOs can be exposed via an OData-enabled API. This demonstration is skeletal, intended to illustrate key architectural concepts without full production readiness -- it provides the structural outline needed to understand the rationale behind DTO usage in real-world systems.

This example illustrates:

- Use of internal entity models
- Mapping logic (manual or using AutoMapper)
- Versioned controllers exposing DTOs
- Configuring OData filters and query limits

While implementation-specific, the principles shown are broadly applicable across platforms.

Conclusion

DTOs are not an implementation detail—they are a strategic tool in building APIs that are secure, sustainable, and fit for long-term use. They formalise the boundary between internal models and external consumers, enabling clarity of contract, governance of change, and separation of concerns. Properly applied, they reduce coupling, improve security, and support system adaptability.

The value of DTOs increases with system scale and lifespan. In environments where APIs must endure, evolve, and support multiple stakeholders over time, DTOs represent one of the most effective mechanisms for ensuring system health. They should be treated as first-class artefacts—reviewed, versioned, and governed alongside the code that depends on them.

Appendices

Appendix A - Document Information

Authors & Collaborators

- Sky Sigal, Solution Architect

Versions

0.1 Initial Draft

Images

No table of figures entries found.

Tables

No table of figures entries found.

References

There are no sources in the current document.

Review Distribution

The document was distributed for review as below:

Identity	Notes
Russell Campbell, Project Manager	
Carl Klitscher, Solution Architect	
Gareth Philpott, Solution Architect	
Vincent Wierdsma, Lead Developer	
Chanaka Jayarathne, Senior Developer	

Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

Structure

Where possible, the document structure is guided by either ISO-* standards or best practice.

Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

Acronyms

API : Application Programming Interface.

GUI: Graphical User Interface. A form of UI.

IT : acronym for Information, using Technology to automate and facilitate its management.

UI : User Interface. Contrast with API.

Terms

Refer to the project’s Glossary.

Application Programming Interface : an Interface provided for other systems to invoke (as opposed to User Interfaces).

User Interface : a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces (GUI) or Text/Console User Interfaces (TUI).

Appendix B - .NET Core Demonstration

This demonstration walks through the common evolution of API design, beginning with the exposure of internal entities and culminating in a proper DTO-based design with mapping and transformation logic. The example is implemented in EF Core and .NET Core but is relevant to any enterprise-grade service platform.

Step 1: Develop Internal Entities

Assume an internal domain model for Person, which includes:

```
public class Person {

    public int Id { get; set; }

    // Internal naming convention; not aligned with external expectations

    public string FName { get; set; }

    public string LName { get; set; }

    // Personal information that may be sensitive if exposed directly

    public DateTime DOB { get; set; }

    // Navigation property tied to an internal classification enum

    public int CitizenshipTypeId { get; set; }
    public CitizenshipType CitizenshipType { get; set; }
    // Many-to-many relation via membership table; implementation detail
    public ICollection<PersonGroupMembership> Memberships { get; set; }
}
public class CitizenshipType {
    public int Id { get; set; }
    public string Code { get; set; } // e.g., "NZ", "Other"
    public string Description { get; set; }
}
public class PersonGroupMembership {
    public int PersonId { get; set; }
    public Person Person { get; set; }
    public int GroupId { get; set; }
    public Group Group { get; set; }
}
public class Group {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Type { get; set; } // e.g., "School"
}
```

Step 2: Develop APIs to expose internal entities

An anti-pattern is to expose this model directly through the API. Doing so ties external consumers to internal naming (e.g., FName), structure (navigation properties, entity shape), and even logic (leaking internal enums or types).

Not only is it a security concern, but it's a contractual obligation risk in that if you correct the poor naming internally, all API consumers will have the link broken. Hence the code will probably never get updated or improved...

The first approach is to expose the internal object.

```
GET /api/people → returns raw Person entity
```

Step 3: Develop DTOs

We define clean, stable DTOs aligned to consumer understanding and long-term usage needs:

```
public class StudentDto {
    public string GivenName { get; set; } // Exposed from internal FName
    public string Surname { get; set; }   // Exposed from internal LName
    // Derived flags replace DOB for privacy-conscious eligibility logic
    public bool EligibleForCompulsoryEducation { get; set; }
    public bool EligibleForFreeCompulsoryEducation { get; set; }
    public List<EnrolmentDto> Enrolments { get; set; }
}
public class SchoolDto {
    public string Name { get; set; }
    public string Location { get; set; }
}
public class EnrolmentDto {
    public string SchoolName { get; set; }
    public DateTime? DateEnrolled { get; set; } // Optional metadata
    public bool IsActive { get; set; }
}
```

These DTOs isolate external contracts from internal structures. Field naming is aligned with consumer expectations and international conventions (e.g. GivenName, Surname). Computed values such as eligibility flags help meet privacy-by-design principles, removing the need to expose raw personal data (DOB).

Step 4: Registering Mappings with AutoMapper

To use DTOs via APIs first Mapping is required.

Mapping is defined unidirectionally using AutoMapper profiles:

```
CreateMap<Person, StudentDto>()
    .ForMember(dest => dest.GivenName, opt => opt.MapFrom(src => src.FName))
    .ForMember(dest => dest.Surname, opt => opt.MapFrom(src => src.LName))
    .ForMember(dest => dest.EligibleForCompulsoryEducation, opt => opt.MapFrom(src =>
AgeBetween(src.DOB, 5, 19)))
    .ForMember(dest => dest.EligibleForFreeCompulsoryEducation, opt => opt.MapFrom(src
=> src.CitizenshipType.Code == "NZ" && AgeBetween(src.DOB, 5, 18)))
    .ForMember(dest => dest.Enrolments, opt => opt.MapFrom(src =>
src.Memberships.Where(m => m.Group.Type == "School")));
```

Step 5: Developing DTO API Endpoints

We refactor the APIs to expose DTOs instead of the entities. The corrected solution now exposes stable and secure DTOs rather than leaking internal structures.

```
[ApiController]
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/students")]
public class StudentsController : ControllerBase {
    private readonly IMapper _mapper;
    private readonly IPersonRepository _personRepository;
    public StudentsController(IMapper mapper, IPersonRepository personRepository) {
        _mapper = mapper;
        _personRepository = personRepository;
    }
    [HttpGet]
    public ActionResult<IEnumerable<StudentDto>> Get() {
        var people = _personRepository.GetAll();
        var students = _mapper.Map<List<StudentDto>>(people);
        return Ok(students);
    }
}
```

This separation allows internal structures (e.g. Person) to evolve independently of external contracts (e.g. StudentDto). The controller only concerns itself with delivering the externally expected shape, ensuring that internal models can change without breaking public interfaces.

The corrected solution now expose stable and secure DTOs instead of exposing internal entities.

```
GET /api/students → returns list of StudentDto
```

Step 4: Versioning DTOs

If a field such as GivenName changes meaning, we create a StudentV2Dto and expose it through a versioned endpoint one now would have two endpoints:

```
GET /api/v1/students
```

And

```
GET /api/v2/students
```

By using DTOs and versioned APIs, internal refactors (e.g., renaming FName to FirstName) do not break consuming systems.

This approach enforces contract stability, privacy, and long-term service adaptability—goals which are difficult or impossible to sustain when using internal entities as API contracts.

```
The following shows an example of second API, with second API, using a different map,
but querying the same person table:

[ApiController]
[ApiVersion("2.0")]
[Route("api/v{version:apiVersion}/students")]
public class StudentsV2Controller : ControllerBase {
    private readonly IMapper _mapper;
    private readonly IPersonRepository _personRepository;
    public StudentsV2Controller(IMapper mapper, IPersonRepository personRepository){
        _mapper = mapper;
        _personRepository = personRepository;
    }
    [HttpGet]
    public ActionResult<IEnumerable<StudentV2Dto>> Get() {
        var people = _personRepository.GetAll();
        var students = _mapper.Map<List<StudentV2Dto>>(people);
        return Ok(students);
    }
}
```

Step 5: Upgrading to an OData Controller

However, while versioning adds value, there is a tremendous amount of functionality that is still not delivered by conventional REST endpoints alone. For example, if you want the students to come back with or without enrolments and associated schools, you would typically need to define two different endpoints or response modes. You're also missing capabilities for clients to define how the data should be filtered, sorted, projected, or paged. Each additional use case introduces more endpoint complexity, business analysis, and test coverage. Most projects either never complete this work or draw an arbitrary line—resulting in a backlog of unresolved requests and frustrated consumers.

There is a more powerful, standardised approach. To enable consumer-driven filtering, sorting, projection, and paging, the DTO endpoint can be upgraded to an OData controller. This capability is rarely matched in other frameworks and is uniquely enabled in .NET by its integration with Language Integrated Query (LINQ). LINQ allows expression trees defined by clients to be transformed directly into backend queries with security, type safety, and runtime efficiency. Other ecosystems lacking LINQ must rely on more complex, boilerplate-heavy solutions such as GraphQL, which—while powerful—require explicit schema authoring, resolver logic, and additional validation mechanisms.

The absence of native, safe, and declarative querying in most stacks is the reason developers gravitate toward GraphQL when basic API endpoints become unscalable. .NET’s OData support addresses the same problem space with less ceremony and better alignment with existing code. Rather than building new endpoints for every reporting or filtering requirement, queryable DTOs shift power to consumers while reducing backend maintenance. This drastically changes the development model: teams can deliver more value with fewer iterations, and consumers gain autonomy without introducing risk.

For example: To enable consumer-driven filtering, sorting, projection, and paging, the DTO endpoint can be upgraded to an OData controller.

For example:

```
[EnableQuery(PageSize = 50)]

[ApiVersion("1.0")]

[Route("api/v{version:apiVersion}/students")]

public class StudentsController : ODataController {

    private readonly IMapper _mapper;

    private readonly DbContext _db;

    public StudentsController(IMapper mapper, DbContext db) {

        _mapper = mapper;

        _db = db;

    }

    [HttpGet]

    public IQueryable<StudentDto> Get() {

        return _mapper.ProjectTo<StudentDto>(_db.Set<Person>());

    }

}
```

This enables clients to use query strings to tailor the response:

```
GET /api/v1/students?$filter=EligibleForFreeCompulsoryEducation eq
true&$orderby=Surname&$select=GivenName,Surname&$top=10
```

With the appropriate AddOData() and model configuration in Startup.cs, the API becomes queryable while preserving DTO abstraction. This dramatically shifts the service's capability profile: consumers can retrieve tailored data slices through a single endpoint, without requiring additional development, deployment, or interface negotiation. It eliminates the overhead of defining, testing, and maintaining countless bespoke endpoints for each query permutation.

The result is transformative. Teams can deliver broader functionality with less code, while external consumers gain control over how they interact with data. Filtering, sorting, projection, and pagination become dynamic features—not static releases. Combined with proper DTO design, queryable APIs give integrators, analysts, and partners a safe, governed, and scalable way to access structured data at depth and at speed.

This model not only reduces delivery overhead, it fundamentally enhances the adaptability and service value of the API, enabling long-lived public contracts to serve a wide and evolving range of consumers—without constant redevelopment.

What makes this approach even more significant is that, to our knowledge, no other mainstream framework or language ecosystem delivers this level of seamless, type-safe, end-to-end queryability. This capability is enabled by the unique integration of .NET's Language Integrated Query (LINQ) with its OData infrastructure and expression tree capabilities. These allow client-defined queries to be safely translated into database-level operations with minimal code and maximal safety.

Other ecosystems attempt to achieve a similar outcome—most commonly through GraphQL—but require significantly more effort, boilerplate, and specialised schema and resolver management. While GraphQL is a powerful option, it often introduces greater operational and cognitive complexity, especially when paired with DTO versioning, validation, and access controls.

By contrast, .NET's OData pipeline allows APIs to remain lean and versioned, while still offering highly flexible access to consumers. This makes the platform exceptionally well-suited to systems that must support external reporting, integration, and analytics without constant developer intervention. The team can move on to new capabilities while the existing APIs remain adaptive, governed, and high-performing.

This is one of the clearest demonstrations of the value of a complete, mature application platform: not just in enabling functionality, but in enabling architectural patterns that reduce ongoing cost and risk while increasing delivered value over time.

Appendix C –DTOs in no/low Code Environments

Low-code and integration platforms often claim to support API design that aligns with enterprise-grade best practices, including structured data exchange and entity abstraction. While it is technically possible to implement DTOs within these platforms, they do not offer native, idiomatic, or first-class support for the DTO pattern as defined in robust software architecture. Their emphasis remains on reducing delivery time, automating integration steps, and abstracting away system complexity—at the expense of long-term control over internal versus external system boundaries.

Salesforce, for example, is centred around Apex classes and platform-managed data models like SObjects. While developers can manually create DTO-style classes and perform custom transformations, these steps are not supported by default tooling or encouraged by platform workflows. Instead, it is common for SObjects to be directly exposed through REST endpoints, binding internal representations to external contracts. Versioning support is limited and lacks formal enforcement. This turns DTO use into an optional and unsupported pattern, implemented only through custom workarounds. For any meaningful separation of contracts, the developer must sacrifice the platform's low-code advantages and revert to extensive custom Apex code within a constrained IDE.

In MuleSoft, a more middleware-oriented integration platform, DTO principles are not enforced by design. RAML-based schemas and DataWeave mappings can technically implement a form of DTO translation, but they are entirely manual and must be defined and governed by the team. Templates and tutorials frequently promote straight-through mapping from source to target, which undermines the boundary between internal structure and external interface. The platform does not support or promote lifecycle management of externally-facing models; versioning and structural abstraction are the responsibility of developers and architects. Any robust DTO usage nullifies the core premise of MuleSoft’s convenience and instead introduces a level of complexity usually handled more effectively in full-code platforms.

In Azure Logic Apps and AWS Step Functions, integration workflows favour pass-through data shaping between cloud connectors or function invocations. Payloads are typically treated as ungoverned JSON, derived from upstream services or storage accounts. Mapping can be inserted via Functions or Lambdas, but this is not part of the standard low-code flow. Instead, any attempt to enforce DTO shaping, contract isolation, or versioning consistency must be implemented manually in external code, detached from the platform’s orchestration model. The more structure and discipline one tries to impose, the more one abandons the platform's value proposition.

This leads to the central observation: implementing DTOs within these platforms is possible, but not natural. None of these platforms offer:

- built-in support for DTO definition,
- automatic mapping utilities aligned to DTO conventions,
- lifecycle or versioning enforcement for external contracts,
- nor platform tooling to identify and manage internal–external model drift.

To implement DTOs rigorously, developers must circumvent the low-code path entirely, building and maintaining manual transformation layers that are neither transparent nor easily governed. The result is a contradiction: the very effort to establish proper interface discipline strips away the advertised benefits of the platform. Instead of reducing development overhead, it increases it. Instead of accelerating delivery, it introduces brittleness and opaque complexity. Instead of improving maintainability, it creates new layers that are hard to audit, test, and evolve.

In practice, organisations are often left with a poor compromise—superficially fast delivery but structurally weak systems. The appearance of simplicity conceals the reality of ungoverned interfaces and unstable contracts. This is not a matter of capability, but of architectural fit. These platforms were designed for integration speed, not for software boundary definition. Where external interfaces must be versioned, secured, governed, and modelled cleanly—DTOs are essential. But where DTOs are essential, low-code platforms rarely offer the tooling or patterns to support them without significant manual work and compromise.

In conclusion, while low-code integration platforms may allow for DTOs to be implemented, they do so only through non-native patterns, manual configuration, and a level of customisation that erodes their central value proposition. In enterprise systems with long-term obligations to stability, auditability, and external integration, these trade-offs should be recognised clearly. Without architectural counterweights and committed discipline, the promise of low-code too often becomes a path to technical debt, not long-term value.