* 1. DRAFT - ICT Project

Guidance: Domain Driven Data Schema Design using ORMs

Version:

0.1

## Purpose

<TODO>

## Synopsis

<TODO>

## Contents

[Purpose 1](#_Toc195025947)

[Synopsis 1](#_Toc195025948)

[Contents 2](#_Toc195025949)

[Background 7](#_Toc195025950)

[Why This Guidance Focuses on Entity Framework Core 8](#_Toc195025951)

[Scope and Intent 8](#_Toc195025952)

[Domain Modelling 10](#_Toc195025953)

[But Why Not Keep the Modelling *Simple*? 11](#_Toc195025954)

[On the Perception of DDD’s Complexity 12](#_Toc195025955)

[Why Boundaries Matter — Even Before Writing Code 13](#_Toc195025956)

[The System Domain 14](#_Toc195025957)

[The User Entity — A System Actor, Not a Person 14](#_Toc195025958)

[A Tempting Mistake: Referencing Person from User 14](#_Toc195025959)

[A Common Correction — But Still Problematic 14](#_Toc195025960)

[A Correct Correction: A Join Entity 15](#_Toc195025961)

[Why This Pattern Matters 15](#_Toc195025962)

[On Navigability, Boundaries, and the Role of Services 16](#_Toc195025963)

[What about the DisplayName 17](#_Toc195025964)

[Permissions, Roles, and User-Level Overrides 18](#_Toc195025965)

[Permission — The Atomic Capability 18](#_Toc195025966)

[Role — A Reusable Bundle of Permissions 18](#_Toc195025967)

[RolePermission — Explicit Link with Optional Exclusion 19](#_Toc195025968)

[UserRole — Assigning Roles to Users 19](#_Toc195025969)

[UserPermissionOverride — Adding or Subtracting Specific Permissions 19](#_Toc195025970)

[On the question of Role Nesting 19](#_Toc195025971)

[The Social Domain 20](#_Toc195025972)

[Person — A Persistent Identity 20](#_Toc195025973)

[Persona — Contextual Identity 20](#_Toc195025974)

[Relationships — Connecting Personas 21](#_Toc195025975)

[Groups — Social and Organisational Units 21](#_Toc195025976)

[GroupMembership and GroupRoles 22](#_Toc195025977)

[Group Roles and System Permissions — Knowing the Boundary 22](#_Toc195025978)

[Example: Evaluating Access via a Service 23](#_Toc195025979)

[Simplicity, Deep Queries and Performance 24](#_Toc195025980)

[Group Types and Group Role Types 24](#_Toc195025981)

[Additional Domains – A Place for Everything 25](#_Toc195025982)

[A Note on Complexity and Querying 27](#_Toc195025983)

[The Education Domain 27](#_Toc195025984)

[Learner Profiles and the Separation of Purpose and Support 28](#_Toc195025985)

[The SchoolProfile 28](#_Toc195025986)

[Jobs and Enrolments 29](#_Toc195025987)

[Profiles Are Not Everything 29](#_Toc195025988)

[Appendices 30](#_Toc195025989)

[Appendix A - Document Information 30](#_Toc195025990)

[Versions 30](#_Toc195025991)

[Images 30](#_Toc195025992)

[Tables 30](#_Toc195025993)

[References 30](#_Toc195025994)

[Review Distribution 30](#_Toc195025995)

[Audience 30](#_Toc195025996)

[Structure 30](#_Toc195025997)

[Diagrams 31](#_Toc195025998)

[Acronyms 31](#_Toc195025999)

[API 31](#_Toc195026000)

[GUI 31](#_Toc195026001)

[ICT 31](#_Toc195026002)

[IT 31](#_Toc195026003)

[UI 31](#_Toc195026004)

[Terms 31](#_Toc195026005)

[Application Programming Interface 31](#_Toc195026006)

[User 31](#_Toc195026007)

[User Interface 31](#_Toc195026008)

[Appendix B –Relationship Patterns 31](#_Toc195026009)

[System Domain 31](#_Toc195026010)

[Social Domain 33](#_Toc195026011)

[Account Domain 34](#_Toc195026012)

[Artefact Domain 34](#_Toc195026013)

[Tasks Domain 37](#_Toc195026014)

[Scheduling Domain 39](#_Toc195026015)

[Appendix C – EF Relationship Schema Development 41](#_Toc195026016)

[Single Entities – Structure Without Relationships 41](#_Toc195026017)

[Models 41](#_Toc195026018)

[Fluent Configuration 41](#_Toc195026019)

[Key Points 42](#_Toc195026020)

[Optional One-to-One Single Value Object (1=>0–1), with no FK (flattened) 42](#_Toc195026021)

[Models 43](#_Toc195026022)

[Fluent Configuration 43](#_Toc195026023)

[Key Points 44](#_Toc195026024)

[Required One-to-One Single Value Object (1=>1-1), with no FK (flattened) 44](#_Toc195026025)

[Models 45](#_Toc195026026)

[Fluent Configuration 45](#_Toc195026027)

[Key Points 45](#_Toc195026028)

[Optional One-to-One Single Value Object (1=>0-1), with shadow FK (unflattened) 46](#_Toc195026029)

[When to Use 46](#_Toc195026030)

[Models 46](#_Toc195026031)

[Key Points 48](#_Toc195026032)

[Required One-to-One Single Value Object (1=>1-1), with shadow FK (unflattened) 48](#_Toc195026033)

[When to Use 48](#_Toc195026034)

[Models 49](#_Toc195026035)

[Fluent Configuration 49](#_Toc195026036)

[Key Points 50](#_Toc195026037)

[Optional One-to-Many Value Objects Collection (1=>0–\*), with shadow FK 50](#_Toc195026038)

[When to Use 51](#_Toc195026039)

[Models 51](#_Toc195026040)

[Fluent Configuration 51](#_Toc195026041)

[Key Points 52](#_Toc195026042)

[Required One-to-Many Value Objects Collection (1=>1–\*), with shadow FK 52](#_Toc195026043)

[When to Use 52](#_Toc195026044)

[Models 53](#_Toc195026045)

[Fluent Configuration 53](#_Toc195026046)

[Key Points 54](#_Toc195026047)

[Optional One-to-Many Value Objects Collection (1=>0–\*), with shadow FK, Bidirectional 54](#_Toc195026048)

[Required One-to-Many Value Objects Collection (1=>0–\*), with shadow FK, Bidirectional 55](#_Toc195026049)

[Optional One-to-Many Entity Collection (1=>0-\*), with shadow FK 55](#_Toc195026050)

[When to Use 55](#_Toc195026051)

[Models 56](#_Toc195026052)

[Fluent Configuration 56](#_Toc195026053)

[Key Points 57](#_Toc195026054)

[Required One-to-Many Entity Collection (1=>1–\*), with Navigation, with shadow FK 58](#_Toc195026055)

[When to Use 58](#_Toc195026056)

[Models 58](#_Toc195026057)

[Fluent Configuration 59](#_Toc195026058)

[Key Points 60](#_Toc195026059)

[Optional One-to-Many Entity Collection(1=>0–\*), with Navigation, with modelled FK, Bidirectional 60](#_Toc195026060)

[When to Use 60](#_Toc195026061)

[Models 61](#_Toc195026062)

[Fluent Configuration 61](#_Toc195026063)

[Required One-to-Many Entity Collection (1=>1-\*), with Navigation, with modelled FK, Bidirectional 62](#_Toc195026064)

[When to Use 63](#_Toc195026065)

[Models 63](#_Toc195026066)

[Fluent Configuration 64](#_Toc195026067)

[Key Points 65](#_Toc195026068)

[Optional One-to-Many Entity Collection (1=>0–\*), with Navigation, with shadow FK, Bidirectional 65](#_Toc195026069)

[When to Use 65](#_Toc195026070)

[Models 66](#_Toc195026071)

[Fluent Configuration 66](#_Toc195026072)

[Key Points 67](#_Toc195026073)

[Required One-to-Many Entity Collection (1=>1-\*), with Navigation, with shadow FK, Bidirectional 68](#_Toc195026074)

[When to Use 68](#_Toc195026075)

[Models 68](#_Toc195026076)

[Fluent Configuration 69](#_Toc195026077)

[Key Points 70](#_Toc195026078)

[Optional Many-to-Many Entity Collection (0–\* ⇄ 0–\*), without Payload, with Navigation, Implicit Join Entity 70](#_Toc195026079)

[When to Use 70](#_Toc195026080)

[Models 71](#_Toc195026081)

[Fluent Configuration 71](#_Toc195026082)

[Key Points 72](#_Toc195026083)

[Required Many-to-Many Entity Collection (1-\*=>1-\*), without Payload, with Navigation, Explicit Join (Join Must Exist, Link Must Be Validated) 72](#_Toc195026084)

[When to Use 73](#_Toc195026085)

[Models 73](#_Toc195026086)

[Fluent Configuration 74](#_Toc195026087)

[Key Points 75](#_Toc195026088)

[Optional Many-to-Many Entity Collection (0-\*=>0-\*), without Payload, with Navigation, Explicit Join Entity 75](#_Toc195026089)

[When to Use 75](#_Toc195026090)

[Models 76](#_Toc195026091)

[Fluent Configuration 76](#_Toc195026092)

[Key Points 77](#_Toc195026093)

[Required Many-to-Many Entity Collection (1-\*=>1-\*), without Payload, with Navigation, Explicit Join Entity (Join Must Exist, Link must be Validated) 78](#_Toc195026094)

[When to Use 78](#_Toc195026095)

[Models 79](#_Toc195026096)

[Fluent Configuration 79](#_Toc195026097)

[Optional Many-to-Many Entity Collection (0–\* ⇔ 0–\*), with Payload, with Navigation, with modelled FK 80](#_Toc195026098)

[When to Use 81](#_Toc195026099)

[Models 81](#_Toc195026100)

[Fluent Configuration 82](#_Toc195026101)

[Key Points 82](#_Toc195026102)

[Required|Required Many-to-Many Entity Collection (1-\*=>1-\*), with Payload, with Navigation, with modelled FK 83](#_Toc195026103)

[When to Use 83](#_Toc195026104)

[Models 84](#_Toc195026105)

[Fluent Configuration 85](#_Toc195026106)

[Key Points 85](#_Toc195026107)

[Optional|Optional Many-to-Many Entity Collection (0–\* ⇔ 0–\*), with Payload, with Navigation, with Shadow FKs 86](#_Toc195026108)

[When to Use 86](#_Toc195026109)

[Models 87](#_Toc195026110)

[Fluent Configuration 87](#_Toc195026111)

[Key Points 88](#_Toc195026112)

[Required|Required Many-to-Many Entity Collection (1–\* ⇒ 1–\*), with Payload, with Navigation, with shadow FKs 89](#_Toc195026113)

[When to Use 89](#_Toc195026114)

[Models 90](#_Toc195026115)

[Fluent Configuration 90](#_Toc195026116)

[Key Points 91](#_Toc195026117)

[Required|Optional Many-to-Many Entity Collection (1-\*=>0-\*), with Payload, with Navigation, with Modelled FK, One Side Must Participate 92](#_Toc195026118)

[When to Use 92](#_Toc195026119)

[Models 92](#_Toc195026120)

[Fluent Configuration 93](#_Toc195026121)

[Key Points 95](#_Toc195026122)

[Required|Required Many-to-Many Entity Collection (1=>1-\*), with Payload, without Navigation (Service-Only, Cross Domain, Join) 95](#_Toc195026123)

[When to Use 95](#_Toc195026124)

[Models 96](#_Toc195026125)

[Fluent Configuration 96](#_Toc195026126)

[Key Points 97](#_Toc195026127)

[Optional Self-Referencing Many-to-One(0-\*=>1), with modelled FK 98](#_Toc195026128)

[When to Use 98](#_Toc195026129)

[Models 98](#_Toc195026130)

[Fluent Configuration 98](#_Toc195026131)

[Key Points 99](#_Toc195026132)

[Optional Self-Referencing Many-to-One(0-\*=>1), with shadow FK 100](#_Toc195026133)

[Fluent Configuration 100](#_Toc195026134)

[Key Points 100](#_Toc195026135)

[Required Self-Referencing Many to One (1-\*=>1), with shadow FK 101](#_Toc195026136)

[When to Use 101](#_Toc195026137)

[Models 102](#_Toc195026138)

[Fluent Configuration 102](#_Toc195026139)

[Key Points 102](#_Toc195026140)

[Multiple Relationships to the Same Entity Type (Using Shadow Foreign Keys) 103](#_Toc195026141)

[When to Use 103](#_Toc195026142)

[Models 104](#_Toc195026143)

[Fluent Configuration 104](#_Toc195026144)

[Key Points 105](#_Toc195026145)

[Required|Required Many-to-Many Entity Collection (1-\*<=>1-\*), with Payload, with Surrogate Key (Instead of Composite Keys) 105](#_Toc195026146)

[When to Use 105](#_Toc195026147)

[Models 106](#_Toc195026148)

[Key Points 108](#_Toc195026149)

[Cross-Schema Relationship (Between Domains) 109](#_Toc195026150)

[When to Use 109](#_Toc195026151)

[Models 109](#_Toc195026152)

[Fluent Configuration 110](#_Toc195026153)

[Key Points 110](#_Toc195026154)

## Background

Systems store data for two primary reasons: they must handle more information than memory alone can hold, and they must preserve that data through power loss or restarts. While physical storage has evolved, our approaches to structuring data have also changed. Relational databases introduced a durable and general-purpose way to represent known entities and relationships. Over time, alternative storage models—key-value, document, graph, blob, time-series, and vector—emerged to serve more specialised or dynamic purposes. Despite this variety, relational thinking remains foundational, especially where clarity, governance, and interoperability are required.

While relational databases remain the backbone of most enterprise and public systems, both how schemas are defined and how data is manipulated have changed significantly. Schema design began with direct use of SQL’s Data Definition Language (DDL), written by hand—an error-prone and manually intensive approach. This evolved into database-first workflows, where the database was built and then documented through diagrams. Although this improved visibility, it did not resolve issues around version control, testing, or automation. It served the needs of maintenance stakeholders, but not those of agile service teams or their end users.

Model-first tools were introduced in an attempt to improve structure by generating both schemas and code from visual diagrams. But these failed to integrate cleanly with modern development practices. They lacked meaningful support for versioning and migration, were difficult to align with source-controlled pipelines, and often enforced design constraints disconnected from business behaviour. They reflected a tooling-first mentality that quickly fell out of step with DevOps.

Code-first approaches replaced this. In code-first development, schemas are derived directly from domain models written in application code. This allowed schema definition, logic, and service behaviour to be versioned together, tested together, and deployed together. Crucially, it marked a shift in ownership—from centralised database design to distributed, domain-aligned teams. This change enabled development to be driven by service outcomes, not schema constraints. It also introduced the need for application teams to manage their own schema evolution responsibly, including indexing, constraints, and data lifecycle.

In parallel, the way data is queried and manipulated also evolved. Initially, access was written directly in SQL—flexible but repetitive and error-prone. Early frameworks tried to standardise or simplify database access, but most failed to strike the right balance. By the early 2000s, ORMs had matured enough to offer a practical solution to this need for abstraction, consistency, and maintainability. Over time, Object-Relational Mapping (ORM) emerged as the dominant pattern. ORMs bridge object-oriented code and relational storage, allowing developers to work with structured data through familiar object models. They reduce boilerplate, encapsulate query logic, and support architectural patterns like Domain-Driven Design by aligning code with real-world concepts.

Security was a major driver in ORM adoption. By enforcing parameterised queries by default, ORMs eliminate most SQL injection risks—one of the most persistent attack vectors in web applications. This made ORMs an obvious choice in security-conscious environments, including those handling personal, financial, and legally protected data. But convenience and safety were only part of the story. ORMs also aligned closely with how modern teams build software: through loosely-coupled services, continuous integration, testability, and automation.

As ORM tooling matured, it began to support schema generation and migration management. This convergence—query abstraction, schema definition, and version-controlled migration—turned ORMs into the integration point between application code, data structures, and deployment pipelines. The transition from database-first to service-first became not just feasible, but preferred. Systems could now be shaped around the services they provide, not the tables they store.

Critics of this shift—often database administrators or data architects accustomed to earlier models—frequently raise concerns about visibility, maintainability, and performance. These concerns are not without merit, but they are often misplaced. The idea that schemas should be designed for maintainers before they are designed for services is fundamentally at odds with the goals of maintainability and evolvability. Designing for clarity at the code and API layer allows for encapsulation, consistency, and testability—exactly the characteristics that enable long-term maintenance. Domain-Driven Design, for instance, depends on the principle that data structures reflect domain behaviour, not storage convention.

It is also a mistake to judge a system’s quality by reading the SQL it generates. In a mature architecture, systems are accessed through services and APIs, not ad hoc queries against internal tables. If the only usable interface is SQL, then the system is incomplete. Complaints about the complexity of ORM-generated SQL often signal a deeper architectural flaw: the absence of proper boundaries and service abstractions.

We recognise that many legacy systems are caught between paradigms—lacking both well-structured schemas and modern service layers. But that is a transitional problem, not a design principle. New systems should not reproduce that compromise. They should provide APIs for access, use ORMs to model domain logic, and retain data expertise where it belongs: in the review of indexing, query plans, and constraints—not in hand-authoring schemas or forbidding automation.

Likewise, the belief that direct SQL delivers better performance is typically unfounded. Performance improvements come from understanding execution plans, selecting appropriate indexes, reducing round-trips, and applying caching—not from bypassing abstraction. In almost all cases, ORMs—used well—enable far greater control over performance by exposing structure, encouraging discipline, and supporting systematic tuning.

The role of data specialists remains critical—but their focus must shift. In modern systems, schema ownership sits with developers. They are responsible for modelling domain behaviour, defining structure, and managing schema evolution—including the safe migration of data as the design changes. However, they must actively draw on the expertise of data specialists when reviewing indexes, constraints, and query performance. An experienced database practitioner can often see in seconds what application developers might miss—particularly in high-volume or transactional systems. This is not the loss of discipline; it is its redirection—away from gatekeeping and toward collaboration, where insight is applied precisely where it delivers value.

A shift also occurs at the level of system maintenance. In traditional architectures, maintainers often accessed the database directly—using SQL, inspection tools, and views that were disconnected from how the system behaved in production. In a service-first model, maintenance no longer means working on the database component in isolation. Instead, maintainers interact with the system the same way clients do—through versioned, queryable APIs that reflect the system’s intended use. The tools and syntax may differ from raw SQL, but the fundamentals of data access, validation, and inspection remain fully recognisable and usable. This improves alignment between system behaviour, operational insight, and support capability. It ensures that what is tested, documented, and supported in production is exactly what is maintained.

## Why This Guidance mentions Entity Framework Core

It is difficult to discuss Object-Relational Mapping (ORM) practices entirely in the abstract. While the principles of schema modelling, relationship design, and migration management are broadly applicable, the implementation details are always shaped by the capabilities and constraints of a specific tool. For this reason, where needed, this guidance refers to Entity Framework Core (EF Core) as the working example.

EF Core is used because it meets a specific set of needs relevant to public sector systems, including performance at scale, tooling maturity, developer familiarity, and alignment with secure, maintainable deployment practices. It enables developers to model domains in code, define schema structure, manage migrations, and access data in a single, consistent framework that integrates tightly with the .NET ecosystem already common across many government systems.

The .NET Core runtime is consistently among the fastest general-purpose platforms available. Benchmark comparisons show that .NET Core handles approximately 80,000 requests per second under load—roughly twice as many as Node.js (~40,000), nearly three times more than Python with FastAPI (~30,000), and over five times that of PHP frameworks like Laravel or Symfony (~15,000). While Go achieves higher raw throughput (~100,000), it demands significantly more manual effort to implement the same operational standards for security, testing, and maintainability. In practical terms, .NET Core offers an optimal balance between high performance and ease of delivery—enabling cost-effective scaling while supporting structured development.

EF Core builds on over a decade of ORM evolution. It supports the full set of patterns required in modern development: code-first schema generation, fluent configuration, dependency injection, asynchronous execution, and query composition using LINQ. Unlike simpler ORMs, EF Core includes mature migration tooling, allowing schema changes to be tracked and applied with the same rigour and automation used for application code. This ensures that domain logic, data structure, and infrastructure can all be versioned, tested, and deployed through unified DevOps pipelines.

Its integration with the broader .NET ecosystem is another reason for its selection. EF Core fits cleanly into common development workflows, test frameworks, hosting environments, and identity management models. It eliminates the need for third-party glue code or brittle custom components. It also provides strong defaults for common concerns such as tenancy separation, multitenant indexing, and logging. These benefits make EF Core a natural choice for enterprise-grade systems where traceability, resilience, and maintainability are required.

Other ORM tools remain viable and may offer specific advantages in certain scenarios. Lightweight options like Dapper can deliver faster raw query execution by removing abstraction overhead. Others may appeal through simplicity or fit better into non-.NET environments. However, these trade-offs often come at a cost: reduced integration, weaker support for migrations, more custom code to manage infrastructure concerns, and reintroduced risks—such as SQL injection—if query composition isn’t handled correctly.

In most cases, these marginal gains do not justify the operational risk or long-term complexity. Optimisation, when required, is better achieved through proper indexing, efficient data modelling, and understanding of query execution plans—not by abandoning abstraction. Introducing bespoke persistence layers or custom scripting strategies adds more surface area for bugs, regressions, and security flaws. EF Core, when used properly, already delivers strong performance while supporting the practices that actually lead to scalable and resilient systems.

All modelling principles in this guidance are tool-agnostic. Everything presented here can be implemented using any capable ORM. EF Core is used as the reference not because it is uniquely capable, but because it is complete, stable, and well-suited to the realities of government-grade systems. Other tools are possible, but the burden of proof—and operational cost—rests with those who choose to diverge from a tested and supported standard.

## Scope and Intent

This guidance outlines recommended practices for using Entity Framework Core (EF Core) in public sector application development. It focuses on modelling principles, schema generation, and the separation of concerns between code, persistence, and transport.

While EF Core is a powerful and flexible ORM, it is not opinionated by default — developers are responsible for establishing consistent patterns that reflect domain behaviour, data constraints, and long-term maintainability. This document provides those patterns, with justification for each.

Topics addressed include:

* **Entity design and configuration**  
  How to define entities using both attributes and Fluent API, with a strong preference for fluent configuration to retain full control over schema generation and naming.
* **Data types and field mapping**  
  Emphasis on selecting the most appropriate type for each field. For example, using decimal with fixed precision for currency, avoiding booleans when nullability is meaningful, and *avoiding enums* in persistent models due to versioning and compatibility concerns.
* **Key design**  
  Recommendations for primary keys, strongly favouring GUIDs over integers for system-generated identity. Guidance includes how to avoid collisions, how to generate them in code, and the rationale for not relying on database-generated values.
* **Indexing strategy**  
  Advice on single-field and composite indexes, with practical notes on tenancy-aware indexing. The guidance explains how tenancy, once seen as a natural partitioning strategy, has become an obstacle to collaboration and information portability across systems and organisations.
* **Relationship modelling**  
  Clear examples covering one-to-many (value objects and entities), many-to-one, and many-to-many with join entities carrying their own attributes. This includes a discussion on implicit vs explicit joins and the importance of modelling intent.
* **Persistence separation**  
  Demonstrating how to avoid polluting domain models with storage concerns such as foreign keys and table mappings. This includes the use of shadow properties and EF Core conventions to hide infrastructure from application logic.
* **Domain isolation and schema separation**  
  Justification for keeping database-specific concerns inside EF-only DTOs. Promotes modelling real-world behaviour through domain entities and aggregates, not transport or database shapes.
* **Working with multiple domains**  
  Guidance on structuring models across logical domains (e.g. System, Social) using schema separation. Includes techniques for crossing domain boundaries where appropriate, and explains how to maintain referential integrity without introducing coupling.
* **Versioning and historical records**  
  Strategies for modelling history tables for entities that require versioning. This includes rationale, patterns, and when to use shadow tables vs first-class entity versions.

## Domain Modelling

To demonstrate the concepts covered in this guidance, we will use simplified examples from the education sector — specifically the process of enrolling learners into education providers. While the surface narrative might appear straightforward, it offers rich opportunities to explore the kinds of modelling decisions that developers routinely get wrong, particularly in early-stage or single-purpose applications.

These examples will intentionally begin with common mistakes — often the kind that feel “obvious” or “natural” to developers — and progressively evolve toward cleaner, more sustainable designs. The aim is to show how small modelling changes can open up significant flexibility, avoid future rework, and support better long-term thinking.

At first glance, we might describe a system as one that allows a “Student” to enrol in a “School”. A naive model might represent this as a Student entity with a School property. It appears to work, until we remember that learners can be enrolled in more than one provider — dual enrolment is common across early childhood, secondary, and tertiary contexts. Another naive variation is to have a School entity with a Collection<Student>. This seems to offer navigability, but in practice it causes major performance issues. The only way to find out which schools a learner attends is to iterate through every school — which quickly becomes unworkable at scale.

Worse still, both models tightly bind enrolment to current state. If a learner leaves the school, they must be removed from the collection — which makes it impossible to track alumni, prior affiliations, or historical analysis.

A more robust design is to introduce an explicit Enrolment entity. Rather than treating enrolment as a property of the learner or the school, we recognise that enrolment is a *relationship with state*. It has a start date, an optional end date, a status (active, suspended, withdrawn), and possibly additional attributes like funding entitlement or learning mode. This is a classic example of a join object: an entity that connects two others while carrying meaningful information about the relationship between them.

Join objects are frequently overlooked by developers unfamiliar with relational modelling. They feel unnatural at first — “why do I need another table for this?” — but they become essential in any real-world domain. Being enrolled is not something that belongs to the student, or to the school. It is an independent thing — and it needs to be modelled that way.

There is a second, deeper mistake often made in beginner applications: the assumption that “Student” is a permanent state. In practice, it is not. A person becomes a student for a particular period of time, in a particular context. Outside of that context — at home, in other services, or after graduation — they are no longer a student. The core entity is not “Student”; it is “Person”.

This distinction is critical. Role and status are *temporal*. They exist in context and in time. A learner is a Person who has an active Enrolment in a Provider. The same person may simultaneously be a child in a household group, a user in a system, and eventually an alumni in a past association. These roles must be modelled separately from the person themselves — and from each other.

What follows are practical walkthroughs using EF Core to demonstrate how to model these scenarios correctly — and how to avoid design traps that will eventually limit the system’s capability, scalability, and reusability.

### But Why Not Keep the Modelling Simple?

A common question — and a fair one — is why not just keep the model simple? If the application is about students, why model Person at all? Why not just create a Student table and move on?

The answer is: *because that works — until it doesn’t*. Early systems are often built to serve one role, one context, or one business objective. But once a system proves useful, it evolves. Its value increases precisely because it begins to support adjacent needs — first one, then many.

It is only a matter of time before a system designed solely for students is asked to track teachers. Or parents. Or caregivers. A common response is to then introduce additional tables: one for Teacher, one for Parent, one for Student. But this approach leads to immediate duplication. A person who teaches in one school and parents in another will have multiple, disconnected records. So will an older sibling who picks up their younger brother and is also recorded as a caregiver. The model fragments, reporting becomes unreliable, and the system loses coherence.

Correct modelling practice avoids this from the outset. A Person is a unique, persistent identity. They may hold different roles in different places and at different times — but they are still the same person. The correct approach is to model Person as the base entity, and capture their involvement through temporal Relationships and contextual Roles.

Being a student, a teacher, or a parent is not something a person *is* — it is something a person *does*, in a particular context, for a particular time. It is the relationship that matters, not the label. By modelling it correctly at the beginning, the system remains extensible, reliable, and aligned with how people actually participate in education systems.

This design pattern is not unique to education. It holds across health, justice, social services, and nearly all public-sector systems. Roles are contextual. Relationships change. People do not.

### On the Perception of DDD’s Complexity

It is sometimes claimed that Domain-Driven Design is “too complex” or “too heavy” for practical use — particularly in early-stage projects. This perception is both understandable and misplaced.

Like any approach with structure, DDD went through an early adoption phase where enthusiasm sometimes overtook clarity. Teams new to the method often got caught up in its terminology — “ubiquitous” language, aggregates, value objects — without fully grasping its intent. The result was occasionally over-engineered solutions, with diagrams and jargon taking precedence over purpose.

But DDD itself is not complicated. At its core, it asks only one thing: *think clearly*. It encourages deliberate modelling. It requires that we name things honestly, structure things coherently, and separate what belongs together from what does not. That’s all. It is no more complex than the domain it reflects — and in fact, it exists to *make complex domains manageable*.

In practice, DDD is only “too much” when applied where it’s not needed — or by teams not yet ready to benefit from what it offers. It is not meant for disposable scripts, tightly scoped utilities, or pure data pumps. But enterprise systems — and certainly those developed by or for government agencies — are not short-lived. They carry obligations, entitlements, history, and meaning. These systems outlive projects, platforms, and even the people who created them.

In that context, minimal modelling is not efficiency — it is negligence. What are often described as Minimum Viable Products (MVPs) frequently become, in practice, **Missing Valuable Planning**, or **Most Vulnerable Projects** — where missing planning, unclear ownership, and rushed delivery accumulate long-term technical and organisational debt.

Domain-Driven Design is not the expensive option. It is the *preventative* option. It ensures that systems reflect the real-world structures they are responsible for serving, and that those structures remain robust even as features are added, teams change, or environments evolve.

Government agencies should not be developing proof-of-concept architectures for production systems. They should be modelling with care from the outset. DDD supports that — and when used as intended, it reduces the cost of complexity rather than adding to it.

## Why Boundaries Matter — Even Before Writing Code

Understanding the difference between a User and a Person — or between infrastructure and business domains — may seem abstract at first. But these distinctions have practical consequences that go far beyond technical modelling. When boundaries are ignored or misunderstood, the impact is not just felt in code, but in **procurement, platform selection, and operational viability**.

Public sector agencies frequently procure or adopt systems that *appear* to meet functional needs — they “have users”, “track students”, or “manage schools”. But these platforms are often built on simplistic assumptions: that users *are* people, that roles are fixed, or that relationships are permanent. When these assumptions are embedded into a system’s foundation, they become difficult or impossible to unwind later.

As real-world needs evolve — adding caregivers, enabling delegation, introducing multiple roles per person, integrating with third parties — the platform cannot adapt. The data model is too rigid, and the access model too naive. The result is a cycle of manual workarounds, bolt-on modules, duplicated data, and escalating integration costs.

In time, this leads not just to strained platforms but to **fragmented architectures**. Agencies are forced to procure *additional* systems to work around the limitations of earlier ones. The discussion shifts from designing a single system that models the domain correctly to orchestrating a set of loosely coupled tools that don’t. What began as a solution becomes a web of compromises — and the focus turns to *integration* as if it were a strategy. It isn’t. It’s often a desperate patch for poor modelling decisions made too early and under too little scrutiny.

The irony is that the “integration problem” is rarely technical. It is structural. If systems shared the same foundational understanding of people, roles, relationships, and time, there would be far less to reconcile. Instead, when each system defines these concepts differently — or embeds them inappropriately — integration becomes a permanent overhead, not a bridge.

Procurement decisions made without clear modelling boundaries often select platforms that cannot evolve to reflect actual use cases. These systems may deliver short-term results but become long-term liabilities — difficult to extend, difficult to trust, and impossible to unify with modern services.

This is why boundaries must be understood before writing code — and long before procuring technology. The first step in any system should be modelling the concepts it needs to reflect, and doing so in a way that keeps infrastructure, social context, and business rules apart. This requires careful separation of domains — not as a technical exercise, but as a design discipline that preserves flexibility and trust.

This guidance begins in the System domain — not because it is the most interesting, but because it is the most foundational. It deals with identity, permissions, and access — concerns that are present in *every* system, but that must be kept separate from domain logic. The goal is to establish, from the outset, the boundaries that let systems evolve safely — where infrastructure can support many domains, and where domain models reflect reality, not the limitations of early design choices.

## The System Domain

The **System domain** contains entities concerned with the operation of the application itself. These entities manage authentication, authorisation, and system-level configuration. They have no knowledge of the business logic or subject matter of the domains they support — and this separation is critical.

System entities operate the platform. They should not rely on, or be coupled to, business logic such as education or social roles. These concerns are layered *above* the system — and layering must be preserved to avoid structural entanglement.

### The User Entity — A System Actor, Not a Person

The most fundamental entity in this domain is the User.

It’s common — especially early in a system’s life — to assume a User is a person. And in simple cases, that might even appear to work. A user logs in, represents someone, performs actions. But this thinking breaks down quickly. Not all users are people (e.g. automated jobs). Not all people are users (e.g. children, caregivers). And the same person may hold multiple system identities (e.g. test, admin, or cross-organisation roles).

A User is a **system actor**. It exists to access and use the system. It may represent a person — but it is not a person.

#### A Tempting Mistake: Referencing Person from User

A common first instinct is to link User directly to Person, like this:

public Guid? PersonFK { get; set; }

This feels intuitive. In many systems, a user *is* a person — so linking them seems harmless. But this decision introduces a deep structural flaw: **it inverts the domain boundary**. The System domain — whose purpose is to manage infrastructure concerns — now depends on the Social domain, which models people, relationships, and identity. This upward dependency violates domain layering and invites ongoing coupling between concerns that should remain isolated.

#### A Common Correction — But Still Problematic

To correct this, one might flip the relationship and place a foreign key from Person to User:

public Guid? UserFK { get; set; }

This restores the layering: Social references System, not the other way around. But the structure is still wrong — just slightly less so. This relationship is now implied to be permanent and singular. That doesn’t reflect reality.

A person might:

* Use different user accounts over time (e.g. changed email, username, provider).
* Have accounts in different systems or environments.
* Temporarily be granted delegated access via another user.
* Lose and regain access under different credentials.
* Be referenced without ever logging in.

This means the relationship is not static. It is **temporal and contextual**. As such, it deserves to be treated as a **first-class join object**, not a static foreign key.

#### A Correct Correction: A Join Entity

The proper way to link User and Person is with an explicit join object — a separate entity that captures *when* the link was valid, and in what context.

public class PersonUserLink {

public Guid Id { get; set; }

public Guid PersonFK { get; set; } // Social domain

public Guid UserFK { get; set; } // System domain

public DateTime LinkedFrom { get; set; }

public DateTime? LinkedTo { get; set; }

public string? Context { get; set; } // e.g. sys environment, channel, purpose

}

This model provides full traceability and avoids inappropriate assumptions. You can now ask:

* Has this person ever held a system login?
* Which user identities have represented this person over time?
* Was this a long-term association, or temporary delegation?
* Which systems or subsystems did this link apply to?

It also means that User and Person remain clean, self-contained entities — unburdened by external references or hidden rules.

#### Why This Pattern Matters

Moving temporal data from entities into join objects is not just a more flexible design — it’s a correct one. Any time a relationship between two entities has state (dates, status, conditions), that relationship should be treated as its own object. It deserves its own table, its own constraints, and its own meaning.

This approach reinforces the broader principle at work in this guidance: that relationships are not always simple, and that modelling them honestly from the start avoids the slow accumulation of technical debt.

Systems built for government or public service must be able to support delegation, proxies, reauthentication, identity recovery, and external-facing users. None of that is possible if the relationship between person and user is flattened into a single ID field.

By introducing PersonUserLink, we make this relationship visible, trackable, and trustworthy — while keeping our domain boundaries intact.

This structure is not just good design — it’s a safeguard. It protects the system from growing into an unmanageable tangle and ensures that infrastructure, identity, and business logic can each be improved, replaced, or reused without unintended side effects.

#### On Navigability, Boundaries, and the Role of Services

Once the PersonUserLink entity has been introduced — a temporal join between a User in the System domain and a Person in the Social domain — it’s natural to ask whether it should be added as a navigable property to either entity.

The question seems technical, but the answer depends on remembering that we are designing a system, not just a database.

At first glance, it seems harmless to expose the link from both ends. After all, a person may often be connected to a user, and many users will be associated with a person. However, this overlooks the importance of both bound contexts, and domain layering. The Social domain sits above the System domain. Identity, context, and relationships are modelled above the machinery of access and authentication. If we expose PersonUserLink from the System domain — for example, by adding a property to the User entity — we reverse that layering. The System domain would now depend on Social, making the platform itself dependent on business or social rules. That coupling is subtle at first, but it spreads. It becomes harder to isolate infrastructure logic, harder to reason about boundaries, and ultimately harder to evolve.

So we consider placing the reference in the other direction — from Person to PersonUserLink. This appears to respect the boundary, since the Social domain is allowed to depend on System. But this introduces another modelling problem. The link between a person and a user is not universal, and it is not permanent. Many people will never hold a login. Some may hold more than one over time. Others may be temporarily linked to a delegated account. The relationship is not a property of the person. It is a contextual event. As such, it deserves to be modelled as its own object — and treated as a distinct entry point.

That brings us to the correct approach. PersonUserLink is a join object with context and time. It represents a real-world linkage that can be formed, ended, audited, or delegated. But it is not a navigable link from either User or Person. It is its own construct.

**However, as it is a constructs that lives *between* domains, it should be accessed not through object graphs but through explicit logic — a service.**

An example of how the parts are used would be when the request starts, the system resolves the user from a token or claim. From there, if the application needs to resolve the person associated with that user, it can do so through an identity resolution service — not by traversing a navigation property. Likewise, if a process needs to find which system user is linked to a given person, it can query for an active PersonUserLink record. This pattern keeps boundaries clean, and relationships explicit.

#### What about the DisplayName

A related common practical question follows: where should display name go? After all, the user needs to be greeted by name in the interface, and administrators need to know who the account belongs to.

It may be tempting to place display name directly on the User entity. This feels convenient but is problematic. It hardcodes naming into the system account — even though names may vary across contexts, or change over time.

An alternative would be to rely on the Persona — the contextual identity belonging to the person — but we have already agreed that User cannot reach up to it without breaking domain boundaries. That’s where a UserProfile comes in.

A UserProfile is part of the System domain and stores presentation preferences and settings associated with a given user. It can store the display name, not as a permanent representation of the person, but as a system-specific presentation layer. That display name can be populated from the linked Persona at the time of association — or overridden manually if needed. This ensures that names are accurate, but also that they are not embedded into the access model itself.

That said, in most systems, DisplayName is placed directly on the User entity for convenience — and this is an acceptable and common practice, as long as the reason is understood. It is not a canonical source of a person’s name. It is a cached label used for interface display and identification within the system. It may be initially sourced from a Persona, but it is not expected to remain in sync. If the system later supports richer user-managed settings, a UserProfile can be introduced at that point and take responsibility for it.

This small example — where to put a display name — demonstrates the value of deliberate boundaries between both domains and entities. What seems like a trivial design choice becomes significant when the system needs to evolve, scale, or integrate. And that’s the real value of respecting domains. Not that it solves every problem immediately, but that it allows the system to grow without collapsing under the weight of its own shortcuts.

### Permissions, Roles, and User-Level Overrides

In the **System domain**, user access is governed not by identity, but by **permission**. What a user can do is determined by which capabilities are granted to them. Permissions are the atomic unit of authorisation. Roles are simply bundles — reusable groupings of permissions applied to users for convenience.

Although often referred to as *role-based access control*, this is a misnomer. What matters is not the role a user holds, but the permissions they receive as a result. Roles are not an identity; they are a mechanism for assigning capability.

#### Permission — The Atomic Capability

A Permission represents a single, testable, system-defined capability. It should be:

* Specific enough to be meaningful (CanEditUserAccounts)
* Broad enough to be stable across minor system changes
* Not business-domain-dependent (these are infrastructure-level)

public class Permission {

public string PermissionCode { get; set; } // e.g. "CanManageUsers"

public string? Description { get; set; } // Optional explanation

}

Permissions are immutable and - in this simple system - system-scoped. They should be defined intentionally, centrally, and referenced throughout system logic via a service or policy layer — *not scattered as magic strings*.

#### Role — A Reusable Bundle of Permissions

A Role is a named grouping of permissions intended for reuse. Roles are created and managed within the system and serve as a convenient way to apply a set of access rights to multiple users.

public class Role {

public Guid RoleKey { get; set; }

public string Name { get; set; } // e.g. "SystemAdministrator"

public string? Description { get; set; }

public ICollection<RolePermission> IncludedPermissions { get; set; }

public ICollection<RolePermission> ExcludedPermissions { get; set; }

}

#### RolePermission — Explicit Link with Optional Exclusion

Permissions are linked to roles using a RolePermission join entity, with a flag to indicate whether the permission is being included or explicitly excluded.

public class RolePermission {

public Guid RoleFK { get; set; }

public string PermissionCode { get; set; }

public bool IsExcluded { get; set; } // false = included, true = explicitly removed

}

This allows a role to be customised with subtractive logic — for instance, “same as admin, but without deletion rights”.

#### UserRole — Assigning Roles to Users

At its most basic it’s as below, but you can always come back to add From/To (you may be setting up during the holidays rights for learners coming back at the beginning of next term).

public class UserRole {

public Guid UserFK { get; set; }

public Guid RoleFK { get; set; }

}

#### UserPermissionOverride — Adding or Subtracting Specific Permissions

In exceptional cases, a user may need a permission added or removed that differs from their assigned role(s). This is captured using a UserPermissionOverride.

These overrides allow administrators to grant or restrict access precisely, without redefining or duplicating entire roles. Use sparingly, and always with audit visibility.

#### On the question of Role Nesting

A common question is whether roles should be **nestable** — that is, should one role be allowed to include another?

While this might seem convenient in larger systems, the guidance here is **no**. Roles should not be nestable by default.

Nesting introduces **transitive permission inheritance**, which obscures the origin of a permission. It becomes difficult to understand why a user has access, or what will change if a nested role is altered. This reduces transparency, undermines auditability, and increases the likelihood of unintended side effects.

If nested behaviour is absolutely required, it should be modelled explicitly, as a separate structure with clear semantics. But for the majority of cases — and certainly for public-facing, high-trust systems — roles should remain flat, with permissions assigned directly and deliberately.

## The Social Domain

The **Social domain** models people — not as *system* users, but as real-world participants in social, educational, and organisational contexts. It exists independently of system access, and supports a much broader set of scenarios: people who have never logged in, people acting in multiple roles, and people participating across many contexts.

Where the **System** domain models *what someone can do* in a system, the **Social** domain models *who someone is*, *who they are to others*, and *how they participate in groups and relationships*. It is the human-facing layer — the bridge between domain logic and real-world experience.

### Person — A Persistent Identity

At the centre of the Social domain is the Person entity — a stable, non-temporal identity that persists across time, relationships, roles, and systems.

public class Person {

public Guid PersonKey { get; set; }

public DateTime DateOfBirth { get; set; }

public string? NationalIdentifier { get; set; }

public ICollection<Persona> Personas { get; set; }  
}

Note that the Person entity does not contain name, gender, or contact details. These attributes are not stable over time — they are **contextual**. They depend on where the person is, who they are interacting with, and what role they are playing. That context is captured in the Persona.

The only attributes appropriate to include on a Person are biological and factual: date of birth, place of birth, and (optionally) date of death. Even these are not as fixed as they may appear. Place of birth, for instance, may change in meaning due to political or cultural shifts — as seen in cases of diaspora, state dissolution, or refugee status. Everything else — name, gender identity, religion, citizenship, and social role — varies by time and place, and should be modelled accordingly.

### Persona — Contextual Identity

A Persona represents a **view of a Person in a specific context** — such as at a school, in a household, or in a government service. This is where name, gender, contact details, and role-specific data are held.

public class Persona {

public Guid PersonaKey { get; set; }

public Guid PersonFK { get; set; }

public string GivenName { get; set; }

public string Surname { get; set; }

public string? Gender { get; set; } // Optional, context-sensitive

public string? PreferredName { get; set; }

public ICollection<ContactChannel> ContactChannels { get; set; }

}

This separation allows for:

* Multiple representations of the same person in different contexts.
* Variation in name usage, pronouns, and contact information.
* Temporal and role-based associations without mutating core identity.

It also enables the system to support sensitive scenarios — such as where a person presents differently in different social groups — without imposing a single, authoritative view of their identity.

### Relationships — Connecting Personas

A Relationship defines a directional link between two Personas. It is **not** a property of either Persona, but a **first-class entity** with its own type and attributes.

public class PersonaRelationship {

public Guid PersonaRelationshipKey { get; set; }

public Guid FromPersonaFK { get; set; }

public Guid ToPersonaFK { get; set; }

public string RelationshipType { get; set; } // e.g. "Parent", "Sibling", "Caregiver"

public bool IsConfirmed { get; set; } // Optional two-way confirmation

}

This model allows for:

* Bi-directional or asymmetrical relationships
* Verification and approval flows (e.g. whānau guardianship)
* Flexible modelling of social and institutional ties

### Groups — Social and Organisational Units

People do not exist in isolation. They act within **Groups** — households, classes, organisations, clubs, providers. A Group is any collection of personas that has meaning.

public class Group{

public Guid GroupKey { get; set; }

public string Name { get; set; }

public string GroupType { get; set; } // e.g. "Whānau", "SchoolClass", "Iwi"

public ICollection<GroupMembership> Memberships { get; set; }

}

Groups exist solely in the Social domain — they are not system structures. They enable the modelling of collective relationships, shared responsibilities, and contextual groupings.

#### GroupMembership and GroupRoles

A Persona joins a Group via a GroupMembership, which may include a **Group Role** — a description of their role *within that group*. This is separate from system-level roles (defined in the System domain).

public class GroupMembership {

public Guid GroupMembershipKey { get; set; }

public Guid GroupFK { get; set; }

public Guid PersonaFK { get; set; }

public string GroupRole { get; set; } // e.g. "Parent", "Teacher", "Leader"

public DateTime? From { get; set; }

public DateTime? To { get; set; }

}

#### Group Roles and System Permissions — Knowing the Boundary

Group roles reflect real-world position. They describe how a person participates in a social context — such as a teacher in a class, a caregiver in a whānau group, or a coordinator in a working group. These roles live entirely in the **Social** domain. They are descriptive, contextual, and meaningful to the organisation or environment in which they appear.

System permissions, on the other hand, are entirely within the **System** domain. They control what a user is allowed to do within the software — view reports, edit records, approve requests. These permissions are technical, enforceable, and operational. They are infrastructure, not identity.

This distinction is essential. One describes a person’s *place* in a group; the other defines their *capability* in the system. Conflating the two leads to brittle logic, poor traceability, and security problems that are difficult to audit or fix.

It’s tempting to merge them directly. A developer might consider adding a Permissions collection to GroupMembership, or allowing a GroupRole to include system-level permissions. This seems to reduce duplication and create a convenient link. But in doing so, the model crosses domains. It leaks system-specific concerns into business data structures — and that compromises both.

Instead, the correct approach is to **separate modelling from enforcement**. The role a person holds in a group is stored in the GroupMembership. But whether that role should imply system access is **not** the job of that entity. That logic lives in a separate layer — in a **service** — which knows how to translate context into capability.

Just like the PersonUserLink earlier, any time a rule crosses from one domain into another, the logic should be expressed as a **service**, not a navigable property. This ensures each domain remains coherent and focused — and that policy can be updated without breaking the data model.

#### Example: Evaluating Access via a Service

Suppose we want to check whether a person has permission to enter assessment data in a school system. We do not ask the Person or the Group. Instead, we ask a service — one that understands how to resolve relationships, roles, and permissions.

var hasPermission = \_accessService.HasPermissionForPerson(

personKey,

"CanEnterAssessments",

context: new {

GroupKey = someClassGroupKey

}

);

This method encapsulates all the necessary steps:

* Resolve the current User for the Person.
* Determine their GroupMembership in the target group.
* Check whether their GroupRole maps to a system role.
* Aggregate roles and overrides (if any) into a permission set.
* Determine whether the requested permission is included.

This logic is explicit, testable, and isolated from the domain model. It allows for future complexity — such as delegated access, conditional permissions, or policy-based access — without having to refactor entity relationships or cascade schema changes.

#### Simplicity, Deep Queries and Performance

Concerns around performance often arise at this point — typically framed as a trade-off between a clean, layered design and the efficiency of a single, deep database query that joins everything from User to Person, to GroupMembership, to Permissions. However, this is a false choice. Systems that respect domain boundaries are not inherently less performant — they simply require performance to be addressed at the right layer. Access evaluation, relationship resolution, and permission loading are all highly cacheable. In fact, these are some of the best candidates for in-memory or distributed caching strategies, precisely because the data changes infrequently and is heavily reused across sessions. Optimisation belongs in service orchestration and caching — not in collapsing the model to shortcut design principles. Correct structure enables maintainable performance. Improper structure simply postpones inevitable rework.

#### Group Types and Group Role Types

Groups in the Social domain are deliberately abstract. They represent any meaningful collection of people — whether temporary, enduring, formal, or informal. Some are spontaneous (Adhoc), others institutional (System), and many sit somewhere in between. What distinguishes them is their **Group Type** — a declaration of what kind of context the group represents.

A group might describe a café meeting, a classroom, a whānau, or even an education provider. Each of these is valid, but the system must know *what kind* of group it is in order to reason about its behaviour. Group Types give structure to the otherwise shapeless variety of human arrangements.

Within a group, a person participates through a GroupMembership, and each membership may include a **Group Role** — a title or label that expresses the person’s role in that group. These titles are local. They may be Chairperson, Mentor, Observer, or simply Member. What they mean is specific to the group’s purpose. To help bring consistency, these labels may be aligned to a GroupRoleType — a classification that maps local language to structured meaning. One useful frame is RASCI: Responsible, Accountable, Consulted, Informed, and Ignored.

It is here — in these roles — that many systems quietly go astray.

Terms like Teacher, Student, or Caregiver feel intuitive, and they are common in modelling education systems. But they carry **sector meaning**, not social meaning. They are roles **within** an education context — and they depend on that context being clearly modelled. Too often, that context is *assumed* rather than *explicit*. Systems jump straight to classes and learners without ever modelling the **Education sector itself** — its boundaries, its structures, its authorities.

This is like forgetting to model the air. It is so pervasive, so foundational, that it disappears from view. Everything happens inside it, so we forget to name it, describe it, or give it form. But without it, we have no anchor — no object to associate people to, no structure to attach roles to. That is when developers begin to misuse the System domain. They assign system-level roles like Admin, Teacher, or Principal as if those are intrinsic access levels, when they are in fact **sector roles** — belonging to an education provider, not to the platform itself.

The correct approach is to **model the sector explicitly**. If we are building an education system, we must model the **Education context** as a domain. That context then contains its own objects — schools, classes, programmes, learning events. Once that structure exists, people can be associated to it through roles — using GroupMemberships or Enrolments — and permissions can be derived based on domain context, not hardcoded assumptions.

By keeping social structure distinct from sector logic, and sector logic distinct from infrastructure, we create systems that can model reality without conflating responsibilities. Group roles describe human context. System permissions enforce technical access. Sector domains — like Education — sit in between, giving those roles shape and meaning.

## Additional Domains – A Place for Everything

While this guidance focuses primarily on the System, Social, and the Education domains, it is important to acknowledge that most systems interact with other kinds of activity — and those activities deserve domains of their own. These domains are not described in detail here, but are introduced to illustrate the breadth of capability that a well-structured architecture can accommodate.

**Artefacts**  
Artefacts represent documents, media, and collaboratively developed outputs. They may originate in one domain — such as documents in a work context, a lesson plan in Education or a report in Health — but the Artefact domain handles them generically: as versioned, tagged, optionally shared files. This includes authored documents, uploaded content, recordings, or structured data assets. The Artefact domain supports systems where collaboration occurs around *objects* as creator, contributor, reviewer, approver and so forth. As systems mature the system gains the ability to manage workflows between these roles.

**Milestones and Objectives**

Define what Success is using Outcomes and Milestones.Outcomes represent *intentions* — what a person, group, or organisation *hopes* to achieve. They are often aspirational, qualitative, or high-level. In personal life, an outcome might be “build confidence in public speaking.” At work, “deliver projects more collaboratively.” In education, “understand quadratic equations.” Outcomes define *why* we pursue a path — they are purpose statements.

Milestones are *waypoints* — they break an outcome into recognisable steps that show movement. They’re usually structural rather than evaluative. For instance, “completed three presentations”, “participated in team training”, or “finished Unit 4.” Milestones show *where* someone is along a path but don’t necessarily indicate how well they’ve done — just that they’ve passed a point.

**Work and Tasks Domain**

The work domain captures the effort to achieve objectives.  
Tasks represent units of activity — both individual and collective — that are intended to be tracked, assigned, and progressed. This domain includes Tasks, Projects, and WorkItems. It may support delegation, escalation, parallelism, and linking to artefacts or schedules. Importantly, it is distinct from learning objectives or curriculum progress — this domain supports *work*, not *achievement*.

**Evaluation and Progress Domain**

Map alignment to objectives. When work needs to be measured against expectations — whether in education, case management, or performance — the Evaluation and Progress domain offers a space to model expectations, evaluations, and trajectory. This is distinct from tasks themselves. Tasks are performed; progress is judged.

By separating the three, the system remains flexible across sectors that evaluate progress in very different ways. A Task may exist without an Objective. An Objective may exist before any Work starts. An Evaluation might occur long after the original Task.

**Scheduling Domain**  
The Scheduling domain captures structured time — events, timeframes, appointments, and participation. This includes published calendars, recurring patterns, individual events, and ad hoc scheduling. Events may be associated with groups, persons, artefacts, or tasks. A scheduled event may carry participation expectations, exceptions (such as absences or excuses), and related metadata — but none of this should be collapsed into business logic elsewhere. Time deserves its own structure.

These domains are introduced not for completeness, but to reinforce the value of separation. Education systems, in particular, often appear to use “a little bit of everything” — objectives tasks, assessments, schedules, documents, communications — and as a result, there is a risk of modelling everything *inside* the education domain. This is a mistake. Education is a client of these capabilities, not their origin. Progress was not invented in schools. Homework is not the only form of work. Schedules are not uniquely academic. Files and handouts are not owned by learning systems. These are general capabilities, shared across domains — and they should be modelled that way.

### A Note on Complexity and Querying

It’s worth acknowledging that these boundaries and abstractions can frustrate some data specialists — particularly those accustomed to direct SQL access across a flat schema. They may argue that such separation makes the data harder to query. And in a narrow sense, they’re right — it does require more joins, more understanding, and more intentional access.

However, the point of a well-structured system is not to make life easier for the data analyst. It is to make life safer, clearer, and more purposeful for the people and services the system is built to support. These abstractions exist *not to hide the data*, but to represent its meaning — to ensure that what is stored reflects the real-world structures, responsibilities, and timeframes it claims to record. If a model is harder to query casually, that’s often a sign it is protecting complexity that matters. Querying becomes a thoughtful act, not a convenience. And thoughtful systems are the ones that last.

Boundary analysis is hard. It takes deliberate thought to separate what something *is* from how it is *used*. But it is also freeing. Once a domain is well defined, it becomes stable. It stops being reworked. It provides a reliable surface for other domains to use, without taking on their complexity. That is what makes rigorous domain modelling worth the effort. It saves teams from perpetual revision. It allows delivery stakeholders to move forward — to solve new problems instead of patching old ones, to build on solid ground instead of retrofitting unstable structures. And that, ultimately, is how complex systems remain usable.

## The Education Domain

The Education domain draws heavily on the Social, Artefact, Scheduling, and System domains — but it is distinct from them. It exists not to redefine people, documents, tasks, or access control, but to bring educational meaning to those elements. It is the layer that gives purpose and structure to learning activity — while respecting the architectural boundaries beneath it.

A common modelling mistake is to treat education as if it owns the concepts it relies on. It does not. Students are people. Teachers are people. Classes are groups. Documents are artefacts. Schedules are shared. Access is managed in the System domain. Education does not invent these things — it participates in them. The Education domain adds educational significance, structure, and interpretation — and it does so using clear, well-bounded objects.

### Learner Profiles and the Separation of Purpose and Support

A LearnerProfile captures a learner’s purpose within the education system. It includes their programme of study, learning level, enrolment intent, and curriculum pathway. It is linked not to a Person, but to a Persona, because the learner identity is contextual — a person is not a learner in all settings, and may be known differently in different environments.

Separately, a LearnerSupportProfile models the relationship between a learner and the systems that support their participation. This includes funding arrangements, special needs identification, learning support services, language assistance, or anything else that enables access. This is not part of the learner’s identity — it is part of how the system responds to it.

By splitting learning and support, the model remains honest and actionable. A learner’s intent is not determined by their needs, and support can evolve independently of academic progress. It also avoids the common trap of placing everything under one “profile” — which quickly becomes an unmanageable and conflated structure.

**Education Identifiers Are Contextual**

National identifiers, such as the NZ National Student Number (NSN), are not properties of a Person[[1]](#footnote-2), even if used that way in practice. They are contextual, issued by a specific system, and valid only within that system’s rules. They should be modelled as part of the **relationship** between a Persona and the issuing **System Group** — in this case, the national education system.

This applies at every scale. A learner receives a school ID when enrolled at a provider. That ID exists only in the context of that provider and does not follow the learner across institutions. It is not tattooed on the learner — it is part of their enrolment. This same pattern supports cross-border or long-term educational pathways. A learner may have started in New Zealand, continued in China, and moved on to Harvard — each relationship can be modelled, tracked, and contextualised, without collapsing identity or duplicating data.

The same is true for teachers. A TeacherProfile reflects their role in an education system. They may have credentials from New Zealand, but also from other countries. They may work in multiple systems, concurrently or over time. Their profile is not a fixed attribute of the person — it is a contextual role with specific system associations.

### The SchoolProfile

Schools themselves are not mere labels. They are structured entities with policies, roles, histories, and identifiers. A SchoolProfile captures those attributes — not just for public providers, but for any formal education group. This includes school codes, affiliations, system IDs, and governance relationships. Like all profiles, it is designed to be portable and cross-referenced without hardwiring structural assumptions into enrolments or user accounts.

### Jobs and Enrolments

A learner participates in a school through an Enrolment. A teacher participates through a Job. These are not symmetrical. An enrolment is about a learning journey; a job is about employment and role-based contribution. Both are temporal. Both may overlap with others. And both can reference the same group (SchoolProfile) without collapsing into each other.

### Profiles Are Not Everything

Each profile is scoped and purposeful. They do not contain everything about a person, nor even everything about their participation. For example, health information — while relevant — belongs in a HealthProfile, which is a relationship between a learner and the health system. That profile may be accessible to school nurses, doctors, or family members — but never to unrelated staff, and certainly not to teachers who only need to see academic progress. Likewise, a teacher may view curriculum alignment, but not funding eligibility. These separations allow the system to support privacy, accountability, and clarity — all at once.

Appendices

Appendix A - Document Information

Authors & Collaborators

* Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft

### Images

[Figure 1: TODO Image 2](#_Toc144995112)

### Tables

[Table 1: TODO Table 3](#_Toc145048484)

[Table 2: TODO Table 2 3](#_Toc145048485)

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### 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](#Term_ApplicationProgrammingInterface).

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. [IT](#Acronym_IT) is a subset of ICT.

IT

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

UI

: User Interface. Contrast with [API](#Acronym_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

: a human user of a system via its UIs.

User Interface

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

Appendix B –Relationship Patterns

### System Domain

**Session–Session**: use *Optional Self-Referencing One-to-One (0–1 ⇒ 1)*, with shadow FK. This models session transitions — for example, when an anonymous user logs in and a new session is created for their identity. The new session optionally references the original as its source. This preserves continuity while isolating authentication boundaries.

**Session–User**: use *Optional Many-to-One (0–* ⇒ 1)\*, with shadow FK. Sessions may be created anonymously — no User is linked until a person authenticates. Once authenticated, the Session is connected to a User. This avoids needing special AnonymousUser records. Periodic cleanup of anonymous sessions is done by a separate retention job.

**Session-SessionOperations**: use *Optional One-to-Many (0–*), with shadow FK\*. Each Session may generate many operation, therefore logs to audit them forever, but those logs are not structurally required. This allows sessions to exist even if no activity occurred.

**Session-ErrorLog**: use *Optional One-to-Many (0–*), with shadow FK\*. A session may produce error logs, but this relationship should be optional to reflect normal execution paths. Keep the ErrorLog entity clean of session identifiers using shadow keys.

**Session–ErrorLog**: use *Optional One-to-Many (0–*), with shadow FK\*. Not every session encounters an error. Keep the relationship optional and handled by service logic.

**Account–Session**: use *Optional Many-to-One (0–* ⇒ 1)\*, with shadow FK. If your system has Accounts or Tenancies, this pattern supports scoping sessions to an Account context. Some sessions (like global admin) may have no account. Sessions are always linked to an Account via a join — not embedded in the session.

**User–Identifier** (linking to an external IdP): use *Required Many-to-Many with Payload*, with shadow FKs. The payload stores Issuer, Subject, and LoginMethod. Each User can link to multiple identities (e.g. Google, AzureAD), and each login identity must belong to a User. Navigation is usually suppressed — this relationship is accessed via login services.

**User–UserProfile**: use *Required One-to-One (1–1)*, stored in a separate table with shadow FK. The UserProfile stores UI and preference data (e.g. timezone, display name). This ensures clean separation of infrastructure and experience.

**User–UserSecurityProfile**: use *Required One-to-One (1 ⇒ 1)* with a shadow FK. Every authenticated user must have exactly one security profile. The profile stores all access-control information — separate from identity. This allows access rights to be versioned, queried, or revoked without touching the User record. In multi-tenanted systems, multiple profiles can be allowed per user (e.g. one per Account), but in a single-tenant design, use a simple one-to-one.

**UserSecurityProfile–Permissions**: use *Optional Many-to-Many without Payload*, with an explicit join entity and shadow FKs. Permissions are atomic capabilities such as CanApproveInvoice or CanEditArtefact. These are rarely granted directly — but doing so via the profile instead of the user makes it possible to override role bundles selectively without contaminating the user entity.

**UserSecurityProfile–Roles**: use *Optional Many-to-Many without Payload*, with an explicit join entity and shadow FKs. Roles are named collections of permissions (e.g. “SchoolAdmin”, “FinanceManager”). They simplify assignment and revocation. A profile may reference multiple roles — and vice versa — forming a many-to-many.

**Role–Permissions**: use *Required Many-to-Many*, without Payload, with shadow FKs. This is a declarative mapping. Roles define which permissions they include. These associations are managed centrally and change infrequently.

**PermissionOverrides**: use *Optional One-to-Many* from UserSecurityProfile, with shadow FK. These represent granular grant or deny entries for specific permissions, overriding those assigned via roles. Model these as explicit override entries, with fields like PermissionKey, OverrideType (Allow/Deny), and Source (manual, automated, imported). This allows for precise exception handling without redefining the role model.

### Social Domain

**Person-Persona**: Use a *One-to-Many (1⇒0–*) relationship, with shadow FK.\*  
A Person can have multiple Personas across different contexts (e.g., schools, platforms, institutions). This allows contextual information like names, gender, and roles to change while keeping biological identity separate. Shadow FK is used to avoid polluting the domain model and maintain clear separation between Person and Persona.

**Persona-Relationship-Persona**: Use a *Self-Referencing Many-to-Many (0–*⇔0–*) with Payload, with shadow FK.*  
Relationships between Personas (e.g., sibling, parent, friend) are stateful and potentially temporal. This requires a rich join to store relationship type, status, and any lifecycle metadata. Shadow FKs help preserve the neutrality and reusability of Persona.

**Group-GroupType**: Use a *Many-to-One (1–*⇒1), with shadow FK.\*  
Groups of all kinds (schools, clubs, forums) are typed for structure and behaviour (e.g., School, Household, Discussion). GroupType defines expectations, constraints, and default roles. The link is required but shouldn't clutter the Group model.

**GroupType-GroupRoleType**: Use a *Many-to-Many (1–*⇔1–*), with Payload, explicit Join Entity, shadow FK.*  
Each GroupType defines what kinds of roles it supports (e.g., a School allows roles like Principal, Teacher, Learner). This model allows role definitions to vary by group context and carry metadata like visibility, constraints, or descriptions.

**Persona-GroupRoleType-Group**: Use a *Many-to-Many (1–*⇔1–*) with Payload, using a Join Entity with shadow FK.*  
This is how roles are actually assigned. The join (e.g., GroupMembership or GroupParticipation) captures when a Persona holds a particular role within a Group. Roles are contextual, temporal, and may carry audit or lifecycle data. Navigation is often required in both directions.

### Account Domain

Note: “tenancies” was an anti-pattern for collaboration enabled by SaaS.

**Account-AccountProfile**  
Use: Required One-to-One (1 ⇒ 1), with Shadow FK  
Each Account must have a profile, which stores mutable attributes (name, subtitle, logo, branding, etc). Shadow the FK so application logic doesn’t need to be concerned with the database linkage.

**Account-User/Persona Relationship**  
Use: Many-to-Many with Payload, Required Join (1-\* ⇒ 1-\*), with modelled FK  
Model explicitly via AccountMembership. This allows assigning a Persona or User to an Account with a role (e.g. Owner, Administrator, Member) and join time. Allows for governance, auditing, and delegating account control.  
Tip: use Persona rather than User for better domain separation.

**Account-Service**  
Use: Many-to-Many Without Payload, Explicit Join, Required (1-\* ⇒ 1-\*), with modelled FK  
Pattern allows an Account to subscribe to multiple Services. Use a join entity like AccountServiceSubscription.  
Use a surrogate key if you anticipate needing to track billing, activation status, etc.  
Otherwise, composite keys are fine.

**Service**  
Use: Standalone Entity  
Represents a system-level capability (e.g. Learning Journal, Messaging, Reporting Suite). Each Service is defined once and referenced by subscriptions.

### Artefact Domain

The Artefacts domain manages stored resources such as documents, images, videos, and audio files, supporting both individual recordkeeping (e.g. scanned documents) and collaborative workflows (e.g. teachers issuing work or learners submitting responses). Artefacts themselves are immutable and uniquely stored. Contexts of use — such as names, purpose, accessibility metadata, and approval status — are captured via **ArtefactContext**. These contexts are shareable and role-aware, allowing flexible, accessible, and controlled document handling across different domains.

**Artefact ⇔ ArtefactContext**:  
Use **Required Many-to-One (1–\*⇒1)**.  
Each context wraps one artefact. The same artefact can be referenced by multiple contexts. ArtefactContext holds naming, media type, access length, accessibility flags, and language-specific labels. It also includes status fields (e.g. Draft, Submitted, Reviewed).

**ArtefactContext ⇔ Folder**:  
Use **Optional Many-to-One (0–\*⇒1)**.  
Folders are optional containers for grouping multiple ArtefactContexts. Contexts may exist without folders (e.g. standalone documents), or appear in multiple views.

**ArtefactContext ⇔ User/Persona** (Owner):  
Use **Required Many-to-One (1–\*⇒1)**, likely via **shadow FK**.  
Every ArtefactContext has a creator or owner — typically the person who uploaded or issued the artefact. For collaboration, sharing relationships are defined separately.

**ArtefactContext ⇔ Tags**:  
Use **Optional Many-to-Many without Payload, Explicit Join**.  
Used for keyword filtering, classification, or theming. Tags are domain-defined and loosely structured. Link table may include source (manual, AI-suggested, inherited).

**ArtefactContext ⇔ Approvals/Review**:  
Use **Optional One-to-Many Entity Collection**.  
Each ArtefactContext may be reviewed, approved, or commented on by authorised Personas. This is logged using a separate entity like *ReviewRecord* that records actor, role, action, and time.

**Accessibility metadata**:  
Model as a required value object inside ArtefactContext (e.g. AccessibilityInfo), containing content length, media type, captioning, or readability status.

**Account–Session**: use *Optional Many-to-One (0–* ⇒ 1)\*, potentially with a shadow FK. A Session may belong to an Account (if scoped by tenancy), but not all systems enforce this strictly. Avoid navigational cycles — model only one direction.

**User–Identifier** (linking internal User to an external IdP identity): use *Required Many-to-Many with Payload*, with shadow FK. The payload (e.g. Issuer, Subject, LoginStrategy) carries context. This relationship must exist to enable login but should not entangle the User object directly — service access is recommended.

**User–UserProfile**: use *Required One-to-One (1–1)*, with separate table (not flattened), and shadow FK. This ensures system preferences are available for every User, but separate from core identity or access logic.

**User–Permissions**: use *Optional Many-to-Many*, without Payload, with explicit Join Entity. Permissions are system primitives, not tied to business role. Keep this association loose and replaceable — driven via services not direct navigation.

**Permission–Role**: use *Optional Many-to-Many*, without Payload, with explicit Join Entity. Roles are logical bundles — this join enables reuse and abstraction.

**User–Role**: use *Optional Many-to-Many with Payload* if role assignment needs dates, status, or sources. Otherwise, *pure many-to-many without payload* is sufficient — though it should be replaced if you ever need role history.

**ArtefactContext ⇔ ReviewRecord** : use Optional One-to-Many Entity Collection (0–\*⇒1) with Shadow FK. Each ArtefactContext can be reviewed by zero or more people. Each review is a distinct entity with actor, decision, and timestamp.

**ArtefactContext ⇔ Tag**: use Optional Many-to-Many Without Payload, Explicit Join Entity (0–\* ⇔ 0–\*). ArtefactContexts may be associated with many Tags, and Tags reused across contexts.

**Objectives, Milestones & Progress Domain**

This domain models structured learning outcomes, achievement markers, and the evidence or assessment of advancement. It is designed to be reused across educational, professional, and personal development contexts. Rather than being tightly coupled to “schooling,” this domain supports the expression of intent (Objectives), breakpoints or evaluation moments (Milestones), and progress recording mechanisms (ProgressEvidence or similar).

This separation enables clear reuse: teachers may define outcomes, students may track progress toward them, and systems can evaluate completion in different ways — without reworking the underlying model. Importantly, none of these entities assume ownership by a particular sector. They can be adopted by educators, employers, parents, or self-guided learners.

**Entities and Value Objects**

**Objective**  
Represents a formalised intent or goal. Can be grouped (e.g. part of a standard, curriculum, or job description) and often associated with a subject, domain, or role. Objectives are reusable across learners.

**Milestone**  
Describes a measurable waypoint toward an objective. This might be a level (e.g. NCEA Level 2), a specific standard (e.g. “Interpret Statistical Reports”), or an informal checkpoint (e.g. “Complete 10 hours of reading”). Milestones may link to one or more Objectives and can form ordered sequences.

**ProgressEvidence**  
Represents an observed, self-reported, or assessed event contributing to a learner’s advancement toward a Milestone. It may point to work completed, behaviours demonstrated, or criteria met.

**Rubric / AssessmentCriteria (VO)**  
Value object used to describe how milestones are assessed — could include rating scales, success descriptors, evidence types, or thresholds.

**ProgressState (VO)**  
Captures the status of a learner or actor in relation to a Milestone (e.g. NotStarted, InProgress, Demonstrated, Mastered, etc.)

**Links and Relationships**

**Objective → Milestone**  
Pattern: Required One-to-Many (1–\*), with shadow FK  
Every Milestone must relate to an Objective. A single Objective may have many Milestones.

**Milestone → Rubric / AssessmentCriteria**  
Pattern: Optional One-to-One (0–1), flattened VO  
Not all milestones require rubrics, but when present they define how performance is interpreted.

**Milestone → ProgressEvidence**  
Pattern: Optional One-to-Many (0–\*), with shadow FK  
Milestones may accumulate evidence of progress. Each evidence record is scoped to a milestone but may have its own creation method (manual entry, derived, imported, etc.).

**Persona → ProgressEvidence**  
Pattern: Required Many-to-Many (1–\* ⇔ 1–\*), with Payload  
Each piece of evidence is linked to at least one Persona (the learner, actor, or contributor). In some contexts, it may also record who observed, submitted, or verified it.

**ProgressEvidence → Artefact**  
Pattern: Optional One-to-One (0–1), with Payload  
Evidence can optionally point to a supporting Artefact — a document, video, submission, or other media.

### Tasks Domain

The Tasks domain exists to coordinate human effort — individual or collective — toward the achievement of goals. It operates as the execution layer that links abstract intent (from the Objectives, Milestones & Progress domain) to concrete action. It is sector-agnostic and models how people plan, distribute, and complete work, whether in an educational setting, professional environment, voluntary organisation, or personal project.

Tasks are structured, temporal commitments — things that someone should or could do. The domain supports both formal task tracking and informal coordination. It accommodates solo to-do items, collaborative workstreams, recurring obligations, and delegated responsibilities.

Scheduling is handled in its own domain. Tasks may later be associated with scheduled events, but this link is not defined within the Tasks domain itself. Time, participation, location, and availability are outside this scope.

**Task**  
A defined unit of effort. May include a name, description, priority, status, deadlines, and associations to objectives, artefacts, or other tasks. Can be created by or assigned to a person, persona, or group.

**TaskAssignment**  
A join object linking a Task to its assignees. Includes optional role, responsibility level, and participation status (e.g. accepted, delegated, rejected). Enables both individual and group responsibility.

**TaskStatus (VO)**  
Captures a task's lifecycle state — e.g. NotStarted, InProgress, Blocked, Completed, Cancelled. Useful for tracking or filtering.

**TaskPriority (VO)**  
Describes urgency or importance — e.g. High/Medium/Low, or sector-specific prioritisation models.

**TaskRecurrence (VO)**  
Optional value object supporting recurring or scheduled tasks. Used where a task pattern repeats (daily, weekly, per term, etc.).

**TaskDependency**  
Optional link showing that a Task depends on another. May include type (e.g. blocking, related, grouped). Enables sequencing, progress roll-up, and critical path analysis.

**Relationships and Patterns**

**Task → TaskAssignment**  
Pattern: Required One-to-Many (1–\*), with shadow FK  
Each task may be assigned to one or more actors (Personas or Groups). The assignment holds the role and status of the relationship.

**Persona or Group → TaskAssignment**  
Pattern: Required Many-to-Many with Payload (1–\* ⇔ 1–\*), with shadow FKs  
Supports collaboration — many people or roles working together. Assignments may carry metadata (who is responsible, who is informed, etc.).

**Task → TaskDependency**  
Pattern: Optional One-to-Many (0–\*), with shadow FK  
Tasks can depend on others. Dependencies are useful for sequencing and managing blockers or grouped work.

**Task → Objective / Milestone**  
Pattern: Optional One-to-Many (0–\*), cross-domain  
Tasks may contribute to or be derived from a formal goal. This is not enforced — the domain does not assume all work has formal objectives.

**Task → Artefact**  
Pattern: Optional One-to-Many (0–\*), with shadow FK  
A task may result in (or be supported by) one or more Artefacts — submitted work, shared documents, plans, or outcomes.

### Scheduling Domain

The Scheduling domain coordinates *when* things happen, *where* they happen, and *who* is involved. It manages the orchestration of time-bound events — whether system-triggered (like maintenance windows), human-participated (like classes or meetings), or hybrid (like automated deadlines). It does **not** concern itself with task intent (Work), assessment (Progress), or permissions (System); it solely owns the structured representation of time, space, and presence.

This domain is especially important when modelling across educational, organisational, or service delivery contexts where multiple people and systems must align for a moment in time. It allows for temporal coordination without leaking logic into domains that should remain agnostic of time or participation.

**Schedule → ScheduleEvent**  
Pattern: Required One-to-Many (1–\*), with shadow FK  
A Schedule groups recurring or related events under a common label — e.g. Term 1, Room 5 timetable, or PLD calendar. Each event belongs to one schedule but can be independently manipulated.

**ScheduleEvent → Location**  
Pattern: Optional Many-to-One (0–\* ⇒ 1), with shadow FK  
Events may be assigned to a physical or virtual location. Locations are generic (not education-specific) and can include room names, URLs, or virtual placeholders.

**ScheduleEvent → Participation**  
Pattern: Optional Many-to-Many (0–\* ⇔ 0–\*), with payload, with shadow FKs  
People (via Personas or System Users) can be associated to ScheduleEvents as Participants. The join entity (Participation) may include role (e.g. Presenter, Learner), status (Confirmed, Tentative), and visibility preferences. These relationships are not directly attached to the Person — they represent contextual engagement.

**ScheduleEvent → Excuse**  
Pattern: Optional One-to-Many (0–\*), with shadow FK  
Participants may supply an Excuse object for why they missed or declined an event. This can be linked back to the ScheduleEvent and/or the Participation if tracked at that granularity.

**ScheduleEvent → Reminder**  
Pattern: Optional One-to-Many (0–\*), with shadow FK  
Reminders (system or user-defined) can be attached to a scheduled event for pre-event notifications. Not required by default.

**ScheduleEvent → Task (Work Domain)**  
Pattern: Optional One-to-One (0–1), cross-domain  
Some tasks have an associated event (e.g. a parent-teacher meeting). The link is optional and declared in the *Scheduling* domain to avoid polluting the Work domain with temporal logic.

**ScheduleEvent → Objective (Progress Domain)**  
Pattern: Optional Cross-Domain Reference, via join entity (Service-only access)

This relationship **crosses from Scheduling into Progress**, and should not be a direct navigation. Objectives are **not** time-bound by nature — they exist as intended outcomes independent of schedule. However, scheduled events (like exams, lessons, or review sessions) may **support** the delivery or assessment of those objectives.

To avoid polluting either domain, this relationship is best represented by a **join entity** like ScheduledObjectiveLink, defined in the **Scheduling domain**, containing:

* ScheduleEventId (shadow FK)
* ObjectiveId (shadow FK)
* Optional metadata (e.g. purpose, weight, context)

This allows services to associate events to goals without implying that objectives “belong” to a calendar or that events are “owned” by the curriculum.

**Participation → Persona**  
Pattern: Required Many-to-One (1–\* ⇒ 1), with shadow FK  
Every participant represents a contextual identity. Participation is not attached directly to a Person to maintain domain clarity.

**Participation → ScheduleEvent**  
Pattern: Required Many-to-One (1–\* ⇒ 1), with shadow FK  
Each Participation links to a specific ScheduleEvent, defining who is involved in what, and under what terms.

Appendix C – EF Relationship Schema Development

### Single Entities – Structure Without Relationships

This first example demonstrates how to define a simple standalone entity using EF Core's Fluent API. Even without relationships, it’s important to configure entities deliberately — particularly how keys are generated, named, and stored.

#### Models

The following are the use case’s relevant models:

public class Vehicle {

public Guid Id { get; set; } // Deliberately named "Id", not "VehicleId"

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public DateOnly FirstRegistered { get; set; }

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class VehicleConfiguration : IEntityTypeConfiguration<Vehicle> {

public void Configure(EntityTypeBuilder<Vehicle> builder) {

builder.ToTable("Vehicle", "EFExamples");

builder.HasKey(v => v.Id);

builder.Property(v => v.Id)

.HasColumnName("VehicleKey") // Rename the column in DB

.ValueGeneratedNever(); // Must be set in code

builder.Property(v => v.Make)

.IsRequired()

.HasMaxLength(100);

builder.Property(v => v.Model)

.IsRequired()

.HasMaxLength(100);

builder.Property(v => v.FirstRegistered)

.HasColumnType("date")

.IsRequired();

}

}

#### Key Points

* The table is deliberately named ( “Vehicle”) and not just accepting defaults
* The table is within a specific Domain based Schema (“EFExamples”, not “dbo”)
* The key is not database-generated. It must be assigned in code using Guid.NewGuid() or equivalent.
* The property is called Id in code for clarity, but renamed to VehicleKey in the database schema to avoid ambiguity and enforce an arbitrary naming conventions.
* The properties are given space constraints (HasMaxLength) that provides dbms tier validation expectations of size.   
  **Tip: It should be *also* done at the API level, but better late than never.**
* FirstRegistered uses DateOnly, which requires a deliberate mapping to date in SQL Server (EF Core 6+ supports this natively).   
  ***Tip: Always use Universal types (ie, UTC)***

### Optional One-to-One Single Value Object (1=>0–1), with no FK (flattened)

This pattern models an **optional value object** that is **flattened into the parent entity’s table**. The value object:

* Has no identity or key of its own
* Cannot exist without the parent
* Is not independently queryable or referencable
* Is deleted automatically when the parent is deleted
* Is accessed only through the parent (e.g. person.Profile)
* Is **optional** — the parent may or may not include this structure

EF Core flattens the value object’s properties directly into the parent entity’s table columns. If the value object is null, those columns are left NULL.

This is the most compact and common form of optional value object modelling.

**Note:** The flattening means no separate table is created. All value object fields are persisted inline as nullable columns on the parent’s table.

#### Models

The following are the use case’s relevant models:

##### Entity

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public Registration? Registration { get; set; } // Optional

}

##### Value Object

public class Registration {

public string PlateNumber { get; set; } = null!;

public DateOnly Expiry { get; set; }

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class VehicleConfiguration : IEntityTypeConfiguration<Vehicle> {

public void Configure(EntityTypeBuilder<Vehicle> builder) {

builder.ToTable("Vehicle", "EFExamples");

builder.HasKey(v => v.Id);

builder.Property(v => v.Id)

.HasColumnName("VehicleKey")

.ValueGeneratedNever();

builder.Property(v => v.Make).IsRequired().HasMaxLength(100);

builder.Property(v => v.Model).IsRequired().HasMaxLength(100);

builder.OwnsOne(v => v.Registration, reg => {

reg.Property(r => r.PlateNumber)

.HasColumnName("PlateNumber")

.HasMaxLength(20);

reg.Property(r => r.Expiry)

.HasColumnName("RegistrationExpiry")

.HasColumnType("date");

reg.WithOwner(); // Required for ownership, optional for fluent clarity

});

}

}

#### Key Points

* A value object like Registration is **owned** by the entity — it cannot exist on its own.
* Since it's optional (Registration?), EF allows the related columns to be NULL.
* The value object's properties are **flattened into the same table** as the owner (EFExamples.Vehicle).
  + EF will inline PlateNumber and RegistrationExpiry into the EFExamples.Vehicle table. Since Registration is nullable, the columns allow NULL.
* Value objects have **no primary key** of their own and should not contain foreign keys.
* Always configure column names explicitly to avoid long defaults like Registration\_PlateNumber.

### Required One-to-One Single Value Object (1=>1-1), with no FK (flattened)

This pattern models a **required value object** that is **flattened directly into the parent entity’s table**. The value object:

* Has no identity or key of its own
* Cannot exist without the parent
* Is not queryable on its own
* Is deleted automatically when the parent is deleted
* Is always present — the parent **must have** the value object

EF Core embeds the value object’s properties as **non-nullable columns** in the parent table. The object is not stored in a separate table, and no joins are used.

This is the default and most efficient representation for required value objects that have a fixed structure and are integral to the parent’s state.

**Note:** If any property of the value object can be null, you must explicitly allow it in the schema. Otherwise, EF treats the entire value object as required and enforces non-nullability on all flattened columns.

Enforced by EF and constructor logic.

#### Models

The following are the use case’s relevant models:

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public Registration Registration { get; set; } = null!;

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

builder.OwnsOne(v => v.Registration, reg => {

reg.Property(r => r.PlateNumber)

.HasColumnName("PlateNumber")

.HasMaxLength(20)

.IsRequired();

reg.Property(r => r.Expiry)

.HasColumnName("RegistrationExpiry")

.HasColumnType("date")

**.IsRequired();**

});

#### Key Points

* Making the navigation non-nullable (Registration) signals to EF that this is required.
* All properties inside the VO must also be marked .IsRequired() to prevent nulls.
* This enforces that every Vehicle must have a Registration at creation — in code and in schema.
* Still flattened — no separate table, no joins, no foreign keys.

### Optional One-to-One Single Value Object (1=>0-1), with shadow FK (unflattened)

This pattern models a **one-to-one value object** that is **not flattened** into its owning entity's table, but stored in its **own separate table**. It remains a **true value object** — with no identity, no navigation from outside, and ownership by the parent — but its structure and storage are isolated. This approach improves clarity, avoids table bloat, and supports structured querying or reporting without promoting the object to a full entity.

The value object:

* Has no identity
* Cannot exist on its own
* Is deleted when the parent is deleted
* Is still accessed through the parent (e.g. person.Profile)
* But is **optional** — it may or may not be present

EF Core will generate a **separate table** and allow the owned row to be absent (i.e., no profile until created).

#### When to Use

Use this when:

* You want to model a value object that is **not required at creation**.
* The VO contains non-essential or deferrable data (e.g. bios, preferences, history).
* You want to separate storage, but retain **ownership and lifecycle coupling**.
* The parent should be valid without the VO — and the VO should not live independently.

Avoid it when:

* The VO is required — then use a required version instead.
* The VO might be reused, queried directly, or updated independently — then promote to an entity.

#### Models

The following are the use case’s relevant models:

public class Person {

public Guid Id { get; set; }

public string LegalName { get; set; } = null!;

public PersonProfile? Profile { get; set; } // Optional

}

public class PersonProfile {

public string Bio { get; set; } = null!;

public DateOnly RegisteredSince { get; set; }

}

**Fluent Configuration**

The above models are configured as part of the domain as follows:

public class PersonConfiguration : IEntityTypeConfiguration<Person> {

public void Configure(EntityTypeBuilder<Person> builder) {

builder.ToTable("Person", "EFExamples");

builder.HasKey(p => p.Id);

builder.Property(p => p.Id).HasColumnName("PersonKey").ValueGeneratedNever();

builder.Property(p => p.LegalName).HasMaxLength(200).IsRequired();

builder.OwnsOne(p => p.Profile, profile => {

profile.ToTable("PersonProfile", "EFExamples");

profile.WithOwner().HasForeignKey("PersonFK");

profile.Property<Guid>("PersonProfileKey"); // Shadow PK

profile.HasKey("PersonProfileKey");

profile.Property(pp => pp.Bio).HasMaxLength(1000).IsRequired();

profile.Property(pp => pp.RegisteredSince).HasColumnType("date").IsRequired();

});

}

}

#### Key Points

* PersonProfile is declared as **nullable (?)**, making it optional in both the domain model and schema.
* The table EFExamples.PersonProfile may have **no row** for some people.
* Ownership is still strict — if the parent Person is deleted, the profile (if present) is deleted as well.
* No navigation or ID exists for the value object. It is **not reusable or queryable on its own**.
* The **shadow FK** (PersonFK) and shadow PK (PersonProfileKey) keep persistence logic out of the domain models.
* Use this structure when deferrable, soft-configurable, or non-core data is required — but keeping it in the parent table would cause bloat or confusion.

**Note:** You can check for presence with if (person.Profile is not null) and create/update accordingly. The profile remains attached solely through the parent.

### Required One-to-One Single Value Object (1=>1-1), with shadow FK (unflattened)

This pattern models a **one-to-one value object** that is **not flattened** into its owning entity's table, but stored in its **own separate table**. It remains a **true value object** — with no identity, no navigation from outside, and ownership by the parent — but its structure and storage are isolated. This approach improves clarity, avoids table bloat, and supports structured querying or reporting without promoting the object to a full entity.

The value object:

* Has no identity
* Cannot exist on its own
* Is deleted when the parent is deleted
* Is still accessed through the parent (e.g. person.Profile)
* But is **required** — it must be present

#### When to Use

Use this when:

* The value object is conceptually part of the parent but too complex or large to flatten sensibly.
* You want to preserve the **DDD principle** of value objects — no identity, owned lifecycle — but separate their persistence.
* You need better **query performance, indexing, or isolation** for large or semi-structured embedded objects.

Avoid it if:

* The value object is trivially small — then flattening is simpler and faster.
* The object might evolve into a shared or referencable model — then promote to an entity.
* You need polymorphic behaviour or alternate parent associations — those break the VO contract.

#### Models

The following are the use case’s relevant models:

public class Person {

public Guid Id { get; set; }

public string LegalName { get; set; } = null!;

public PersonProfile Profile { get; set; } = null!;

}

public class PersonProfile {

public string Bio { get; set; } = null!;

public DateOnly RegisteredSince { get; set; }

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class PersonConfiguration : IEntityTypeConfiguration<Person> {

public void Configure(EntityTypeBuilder<Person> builder) {

builder.ToTable("Person", "EFExamples");

builder.HasKey(p => p.Id);

builder.Property(p => p.Id).HasColumnName("PersonKey").ValueGeneratedNever();

builder.Property(p => p.LegalName).HasMaxLength(200).IsRequired();

builder.OwnsOne(p => p.Profile, profile => {

profile.ToTable("PersonProfile", "EFExamples"); // Stored separately

profile.WithOwner().HasForeignKey("PersonFK");

profile.Property<Guid>("PersonProfileKey"); // Shadow PK

profile.HasKey("PersonProfileKey");

profile.Property(pp => pp.Bio).HasMaxLength(1000).IsRequired();

profile.Property(pp => pp.RegisteredSince).HasColumnType("date").IsRequired();

});

}

}

#### Key Points

* The PersonProfile is still a **value object** — it has no standalone identity, is not navigated to directly, and is removed when the parent Person is deleted.
* Unlike flattened value objects, this one is mapped to its **own table** (EFExamples.PersonProfile) using .ToTable(...).
* EF Core adds a **shadow foreign key** (PersonFK) linking back to the parent and a **shadow primary key** (PersonProfileKey) to track row identity internally.
* You get **structured separation** for indexing, maintenance, or schema clarity — without breaking value object principles.
* The value object is still loaded through the parent, never by ID.
* This is especially useful when VOs:
  + Contain audit data or text blocks
  + Are queried/reportable as a whole
  + Must avoid cluttering the root entity’s table

**Note:** You still access the profile via person.Profile in code. The storage is separate, but the usage remains inline.

### Optional One-to-Many Value Objects Collection (1=>0–\*), with shadow FK

This pattern models a **collection of optional value objects** owned by a parent entity. Each value object instance is flattened into its own table, but lacks identity or persistence outside its parent. This variant allows **zero or more instances** — the collection may be empty, reflecting an optional aspect of the parent’s history, context, or descriptive attributes.

The following changes simulate a registration history.

#### When to Use

Use this pattern when: The parent may accumulate repeated lightweight facts, states, or observations over time, such as change history, previous names, contact methods, audit trails, or episodic activity. These objects are structurally uniform but **not worth modelling as full entities**. They are unshared, unlabeled, and disposable outside the context of the parent.

Avoid this pattern if: The items require separate querying across parents, cross-linking to other entities, or lifecycle independence. In those cases, promote to a full entity with its own schema, identifier, and lifecycle. Avoid it too if the parent is not the natural lifecycle anchor.

#### Models

The following are the use case’s relevant models:

public class Vehicle {

public Guid Id { get; set; }

public List<Registration> PreviousRegistrations { get; set; } = new();

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

builder.OwnsMany(v => v.PreviousRegistrations, reg => {

reg.ToTable("PreviousRegistrations", "EFExamples");

reg.WithOwner().HasForeignKey("VehicleFK");

reg.Property<Guid>("RegistrationKey"); // Shadow primary key

reg.HasKey("RegistrationKey");

reg.Property(r => r.PlateNumber)

.HasColumnName("PlateNumber")

.HasMaxLength(20)

.IsRequired();

reg.Property(r => r.Expiry)

.HasColumnName("Expiry")

.HasColumnType("date")

.IsRequired();

});

EF will generate a table called EFExamples.PreviousRegistrations with a shadow FK to Vehicle, and each row represents one registration in the history.

#### Key Points

* Collections of value objects **do create their own table** — but still have no entity identity.
* Use ToTable() explicitly to control the name and schema of the value object collection table.
* All rows are still tightly owned by the parent — if the parent is deleted, so are its value objects.
* A **shadow foreign key** (VehicleFK) is used to link the owned items back to the parent.
* **Note:** RegistrationKey and VehicleFK are *shadow properties* — they exist in the database and EF model but are not declared in the Registration class.  
  The reason it’s a shadow key is because EF knows about it – we tod EF to treat RegistrationKey as the PrimaryKey, therefore the DB table has it – but the model used in the rest of the app doesn’t (it wasn’t defined on the Entity itself).  
    
  While **invisible in code** shadow keys can be queryed explicitly using EF.Property<T>().
  + reg.WithOwner().HasForeignKey("VehicleFK");

### Required One-to-Many Value Objects Collection (1=>1–\*), with shadow FK

This pattern models a collection of value objects where the parent entity **must** contain **at least one**. Like all value objects, the items have no identity, are not shared, and are tightly owned. The difference here is that the presence of at least one instance is **structurally or logically required**.

The following changes simulate a registration history.

#### When to Use

Use this when: The parent cannot exist without one or more of these structured items — such as one or more contact methods, funding sources, tags, or definitions. The value objects **do not require identity**, but their **presence is essential** to the parent’s meaning or operation.

Avoid it if: You cannot enforce the presence of at least one in application logic or through input validation. EF Core itself **does not support enforcing non-empty collections at the database level**, so this requirement must be handled in service logic or validation layers. If a countable requirement is critical and externally enforced (e.g., “must have exactly three reviewers”), consider promoting to a formal entity and enforcing constraints in the schema.

#### Models

The following are the use case’s relevant models:

public class Document {

public Guid Id { get; set; }

public string Title { get; set; } = null!;

public List<Keyword> Keywords { get; set; } = new(); // Must contain at least one

}

public class Keyword {

public string Term { get; set; } = null!;

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class DocumentConfiguration : IEntityTypeConfiguration<Document> {

public void Configure(EntityTypeBuilder<Document> builder) {

builder.ToTable("Document", "EFExamples");

builder.HasKey(d => d.Id);

builder.Property(d => d.Id).HasColumnName("DocumentKey").ValueGeneratedNever();

builder.Property(d => d.Title).IsRequired().HasMaxLength(200);

builder.OwnsMany(d => d.Keywords, kw => {

kw.ToTable("DocumentKeywords", "EFExamples");

kw.WithOwner().HasForeignKey("DocumentFK");

kw.Property<Guid>("KeywordKey"); // Shadow PK

kw.HasKey("KeywordKey");

kw.Property(k => k.Term)

.HasMaxLength(50)

.IsRequired();

});

}

}

#### Key Points

* Keyword is an owned value object. It exists only inside its parent Document.
* The value object is persisted in its own table (EFExamples.DocumentKeywords) with:
  + A shadow primary key (KeywordKey)
  + A shadow foreign key (DocumentFK)
* The presence of one or more keywords is a **logical requirement**, not a database constraint — it must be enforced in application logic or via service validation.
* You may throw in the domain constructor, use model validation, or enforce in the UI/API — but EF Core cannot require “at least one” via Fluent alone.
* This pattern is ideal when the value object is lightweight, tightly bound, and never accessed independently.

**Note:** EF Core deletes the keyword records automatically when the Document is deleted. They are fully owned, and always queried or updated through the parent entity.

### Optional One-to-Many Value Objects Collection (1=>0–\*), with shadow FK, Bidirectional

This pattern is **not valid**. A Value Object cannot exist independently — it is always part of its owning Entity. While EF Core allows collections of value objects to be stored in a separate table, the relationship is **not optional** from the value object’s perspective: it cannot exist without a parent. Therefore, an optional one-to-many with value objects does not make conceptual sense.

**Clarification:** The collection itself on the owning entity may be empty, but each value object in that collection must always belong to the parent. The link is mandatory; the quantity is optional.

### Required One-to-Many Value Objects Collection (1=>0–\*), with shadow FK, Bidirectional

This pattern is **not valid**. A Value Object cannot exist independently — it is always part of its owning Entity. While EF Core allows collections of value objects to be stored in a separate table, the relationship is **not optional** from the value object’s perspective: it cannot exist without a parent. Therefore, an optional one-to-many with value objects does not make conceptual sense.

**Clarification:** The collection itself on the owning entity may be empty, but each value object in that collection must always belong to the parent. The link is mandatory; the quantity is optional.

### Optional One-to-Many Entity Collection (1=>0-\*), with shadow FK

This pattern models a **zero-or-many relationship** where child entities may or may not be assigned to a parent. The relationship is optional and unidirectional — the parent may know its children, but the child does **not reference the parent in code**. Instead, EF Core uses a **shadow foreign key** to persist the optional link.

Let’s model DriverNote — comments or notes that *may* be associated with a Driver, but don’t require one immediately.

#### When to Use

Use this when:

* The child entity may **exist independently** but can optionally belong to a parent.
* You want the parent to access and manage a collection of child items, but **not expose** the parent from the child side.
* You want to **enforce loose coupling** and preserve domain separation between aggregates.
* You prefer keeping **foreign keys out of the entity** and relying on EF to manage storage.

Avoid this when:

* The child needs to reference the parent directly for logic or traversal — then use a navigation property or surface the FK.
* The relationship is required — then use .IsRequired() and skip the nullable FK.

#### Models

The following are the use case’s relevant models:

public class Driver {

public Guid Id { get; set; }

public string FullName { get; set; } = null!;

public List<DriverNote> Notes { get; set; } = new();

}

public class DriverNote

{

public Guid Id { get; set; }

public string NoteText { get; set; } = null!;

public DateTime CreatedAt { get; set; }

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class DriverConfiguration : IEntityTypeConfiguration<Driver> {

public void Configure(EntityTypeBuilder<Driver> builder) {

builder.ToTable("Driver", "EFExamples");

builder.HasKey(d => d.Id);

builder.Property(d => d.Id)

.HasColumnName("DriverKey")

.ValueGeneratedNever();

builder.Property(d => d.FullName)

.IsRequired()

.HasMaxLength(200);

builder.HasMany(d => d.Notes)

.WithOne()

.HasForeignKey("DriverFK") // Shadow FK

.IsRequired(false); // Optional link

}

}

public class DriverNoteConfiguration : IEntityTypeConfiguration<DriverNote> {

public void Configure(EntityTypeBuilder<DriverNote> builder) {

builder.ToTable("DriverNote", "EFExamples");

builder.HasKey(n => n.Id);

builder.Property(n => n.Id)

.HasColumnName("NoteKey")

.ValueGeneratedNever();

builder.Property(n => n.NoteText)

.IsRequired()

.HasMaxLength(500);

builder.Property(n => n.CreatedAt)

.IsRequired();

}

}

#### Key Points

* This is a **unidirectional** optional one-to-many.
* The parent Driver has a collection of child DriverNote items, but each note may or may not belong to a driver.
* There is **no navigation or foreign key** declared on the DriverNote class.
* The foreign key (DriverFK) is configured as a **shadow property**, and marked .IsRequired(false) to allow nulls.
* This means:
  + Orphaned notes are permitted
  + Notes can be added or removed from the parent without enforcing referential constraints in code
* This pattern avoids coupling storage detail (like foreign keys) into the child domain model.
* The database schema still includes a DriverFK column, but the application doesn't need to see it.

**Note:** Use this when you want a read-through relationship, not a bidirectional model. EF tracks the link internally — you access it only through navigation from the parent.

### Required One-to-Many Entity Collection (1=>1–\*), with Navigation, with shadow FK

This pattern models a **required one-to-many relationship** between a parent entity and a collection of child entities. Each child entity must belong to exactly one parent. The foreign key is configured as a **shadow property**, which keeps the domain models clean and focused on relationships, not persistence concerns.

#### When to Use

Use this when:

* Each child entity is **meaningless on its own** — it must always be associated with a parent.
* The parent entity can contain **multiple children**, but none can exist without a parent.
* You want to preserve referential integrity while keeping foreign key properties **out of your domain models**.

Avoid this when:

* The child must be able to exist independently or change parents later — in that case, consider making the relationship optional.
* You require child entities to reference the parent explicitly in logic or via public API — in which case, surface the FK in the model.

#### Models

The following are the use case’s relevant models:

public class Driver {

public Guid Id { get; set; }

public string FullName { get; set; } = null!;

public List<DrivingLicence> Licences { get; set; } = new();

}

public class DrivingLicence {

public Guid Id { get; set; }

public string LicenceNumber { get; set; } = null!;

public string Category { get; set; } = null!;

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class DriverConfiguration : IEntityTypeConfiguration<Driver> {

public void Configure(EntityTypeBuilder<Driver> builder) {

builder.ToTable("Driver", "EFExamples");

builder.HasKey(d => d.Id);

builder.Property(d => d.Id)

.HasColumnName("DriverKey")

.ValueGeneratedNever();

builder.Property(d => d.FullName)

.IsRequired()

.HasMaxLength(200);

builder.HasMany(d => d.Licences)

.WithOne() // No navigation back to Driver

.HasForeignKey("DriverFK") // Shadow FK

.IsRequired(); // Required parent

}

}

public class DrivingLicenceConfiguration : IEntityTypeConfiguration<DrivingLicence> {

public void Configure(EntityTypeBuilder<DrivingLicence> builder) {

builder.ToTable("DrivingLicence", "EFExamples");

builder.HasKey(dl => dl.Id);

builder.Property(dl => dl.Id)

.HasColumnName("LicenceKey")

.ValueGeneratedNever();

builder.Property(dl => dl.LicenceNumber)

.IsRequired()

.HasMaxLength(20);

builder.Property(dl => dl.Category)

.IsRequired()

.HasMaxLength(5);

}

}

#### Key Points

* DrivingLicence is a full entity with its own primary key and table.
* Each DrivingLicence **must** be associated with exactly one Driver.
* There is **no navigation** back to Driver from DrivingLicence. The relationship is unidirectional.
* The foreign key (DriverFK) is a **shadow property**, defined only in the Fluent configuration.
* This prevents the domain model from being polluted with storage concerns.
* EF Core will automatically generate the DriverFK column in the database and enforce referential integrity.
* This pattern works well when:
  + You only need to traverse one way.
  + The child doesn’t need to know or operate on the parent explicitly.
  + You want to keep infrastructure (like FKs) out of the public domain surface.

**Note:** If you ever need to query based on the parent, use EF.Property<Guid>(licence, "DriverFK") in LINQ. Or wrap access in a service method.

### Optional One-to-Many Entity Collection(1=>0–\*), with Navigation, with modelled FK, Bidirectional

**Important:**

This configuration is only included for education purposes – we suggest you use instead

Optional One-to-Many Entity Collection(1=>0–\*) –with shadow FK, Bidirectional.

This pattern defines a **zero-or-many** relationship where the parent may own multiple children, and the child may or may not reference the parent. Both entities support **navigation**, and the relationship is **optional** — a child can exist independently or with a parent. The foreign key is typically nullable and visible in the child’s model to support two-way traversal.

#### When to Use

Use this when:

* A parent entity may optionally relate to multiple children, and the relationship is meaningful in both directions.
* The child may exist without a parent and might be associated later.
* Application logic requires traversing from both parent to child and child to parent, or filtering based on FK.
* The FK needs to be exposed for business logic, DTOs, or external systems.

Avoid this when:

* You want to keep the child model clean and decoupled from the parent — then hide the FK and use a shadow property.
* Only one-way navigation is required.
* Parent-child attachment must be enforced structurally — in which case, make the FK required.

#### Models

The following are the use case’s relevant models:

public class Fleet {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public List<Vehicle> Vehicles { get; set; } = new();

}

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public Guid? FleetId { get; set; } // Nullable FK

public Fleet? Fleet { get; set; } // Optional navigation

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class FleetConfiguration : IEntityTypeConfiguration<Fleet> {

public void Configure(EntityTypeBuilder<Fleet> builder) {

builder.ToTable("Fleet", "EFExamples");

builder.HasKey(f => f.Id);

builder.Property(f => f.Id).HasColumnName("FleetKey").ValueGeneratedNever();

builder.Property(f => f.Name).IsRequired().HasMaxLength(100);

builder.HasMany(f => f.Vehicles)

.WithOne(v => v.Fleet)

.HasForeignKey(v => v.FleetId) // Exposed nullable FK

.IsRequired(false); // Optional relationship

}

}

public class VehicleConfiguration : IEntityTypeConfiguration<Vehicle> {

public void Configure(EntityTypeBuilder<Vehicle> builder) {

builder.ToTable("Vehicle", "EFExamples");

builder.HasKey(v => v.Id);

builder.Property(v => v.Id).HasColumnName("VehicleKey").ValueGeneratedNever();

builder.Property(v => v.Make).IsRequired().HasMaxLength(100);

builder.Property(v => v.Model).IsRequired().HasMaxLength(100);

}

}

### Required One-to-Many Entity Collection (1=>1-\*), with Navigation, with modelled FK, Bidirectional

**Important:**

This configuration is only included for education purposes – we suggest you use instead

Required One-to-Many Entity Collection (1=>0–\*) –with shadow FK, Bidirectional.

This pattern models a **required**, **bidirectional** one-to-many relationship — where each child **must** belong to a parent, and both sides can **navigate** to each other. The foreign key is declared explicitly in the child entity, making it directly accessible to application code. This is common in models where both entities are tightly bound and frequently traversed.

We’ll model a Fleet of Vehicles.

Each Fleet can contain multiple Vehicles, and each Vehicle must belong to exactly one Fleet.

Navigation exists **on both sides** of the relationship.

#### When to Use

Use this when:

* You need to navigate both ways — from parent to children and child to parent — in application logic or queries.
* The child entity **must belong** to a parent (no orphans).
* The foreign key value itself may be used in filters, DTOs, or auditing.
* The relationship is part of the **business surface** of both entities — not just storage.

Avoid this when:

* You want to keep application models free from persistence concerns — then use a **shadow FK** instead.
* The relationship is optional or unidirectional.
* You’re trying to keep the child fully decoupled from its parent aggregate.

#### Models

The following are the use case’s relevant models:

public class Fleet {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public List<Vehicle> Vehicles { get; set; } = new();

}

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public Guid FleetId { get; set; }

public Fleet Fleet { get; set; } = null!;

}

#### Fluent Configuration

public class FleetConfiguration : IEntityTypeConfiguration<Fleet> {

public void Configure(EntityTypeBuilder<Fleet> builder) {

builder.ToTable("Fleet", "EFExamples");

builder.HasKey(f => f.Id);

builder.Property(f => f.Id)

.HasColumnName("FleetKey")

.ValueGeneratedNever();

builder.Property(f => f.Name)

.IsRequired()

.HasMaxLength(100);

builder.HasMany(f => f.Vehicles)

.WithOne(v => v.Fleet)

.HasForeignKey(v => v.FleetId)

.IsRequired(); // Every vehicle must be in a fleet

}

}

public class VehicleConfiguration : IEntityTypeConfiguration<Vehicle> {

public void Configure(EntityTypeBuilder<Vehicle> builder) {

builder.ToTable("Vehicle", "EFExamples");

builder.HasKey(v => v.Id);

builder.Property(v => v.Id)

.HasColumnName("VehicleKey")

.ValueGeneratedNever();

builder.Property(v => v.Make)

.IsRequired()

.HasMaxLength(100);

builder.Property(v => v.Model)

.IsRequired()

.HasMaxLength(100);

}

}

#### Key Points

* This is a **1–**\* relationship: a Fleet has many Vehicles, and each Vehicle must belong to exactly one Fleet.
* The relationship is **bidirectional**: both Fleet.Vehicles and Vehicle.Fleet are present and used.
* The foreign key (FleetId) is explicitly declared in the Vehicle model — it’s part of the object’s public surface.
* EF uses the declared FK (FleetId) to enforce referential integrity and link the two entities in the schema.
* The application can now query directly on FK values without needing EF.Property, and project to DTOs or APIs without additional joins.
* This is appropriate when the FK value itself is needed in logic, or when identity traversal is part of your model's surface.

**Note:** This is the most direct and explicit form of required one-to-many — but it **exposes persistence detail** in your model. Consider using a shadow FK in more decoupled or DDD-centric designs.

Hence the next example.

### Optional One-to-Many Entity Collection (1=>0–\*), with Navigation, with shadow FK, Bidirectional

This pattern defines a **zero-or-many** relationship where the parent may own multiple children, and the child may or may not reference the parent. Both entities support **navigation**, and the relationship is **optional** — a child can exist independently or with a parent. The foreign key is typically nullable and visible in the child’s model to support two-way traversal.

In this example, Vehicle records can exist on their own but *may* be assigned to a Fleet. This is useful when vehicles are registered in the system before they are assigned to a formal fleet.

#### When to Use

Use this when:

* A parent entity may optionally relate to multiple children, and the relationship is meaningful in both directions.
* The child may exist without a parent and might be associated later.
* Application logic requires traversing from both parent to child and child to parent, or filtering based on FK.
* The FK needs to be exposed for business logic, DTOs, or external systems.

Avoid this when:

* You want to keep the child model clean and decoupled from the parent — then hide the FK and use a shadow property.
* Only one-way navigation is required.
* Parent-child attachment must be enforced structurally — in which case, make the FK required.

#### Models

public class Fleet {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public List<Vehicle> Vehicles { get; set; } = new();

}

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

~~public Guid? FleetId { get; set; } // Optional FK~~

~~public Fleet? Fleet { get; set; } // Optional navigation~~

}

#### Fluent Configuration

public class FleetConfiguration : IEntityTypeConfiguration<Fleet> {

public void Configure(EntityTypeBuilder<Fleet> builder) {

builder.ToTable("Fleet", "EFExamples");

builder.HasKey(f => f.Id);

builder.Property(f => f.Id)

.HasColumnName("FleetKey")

.ValueGeneratedNever();

builder.Property(f => f.Name)

.IsRequired()

.HasMaxLength(100);

builder.HasMany(f => f.Vehicles)

.WithOne() // No navigation back from Vehicle

.HasForeignKey("FleetFK") // Shadow FK

.IsRequired(false); // Optional relationship

}

}

public class VehicleConfiguration : IEntityTypeConfiguration<Vehicle>

{

public void Configure(EntityTypeBuilder<Vehicle> builder)

{

builder.ToTable("Vehicle", "EFExamples");

builder.HasKey(v => v.Id);

builder.Property(v => v.Id)

.HasColumnName("VehicleKey")

.ValueGeneratedNever();

builder.Property(v => v.Make)

.IsRequired()

.HasMaxLength(100);

builder.Property(v => v.Model)

.IsRequired()

.HasMaxLength(100);

}

}

#### Key Points

* This is a **unidirectional** optional one-to-many — only Fleet knows about the vehicles it owns.
* The Vehicle class has **no knowledge** of the fleet — no navigation, no foreign key.
* EF uses a **shadow property** (FleetFK) to store the optional relationship in the database.
* The shadow foreign key is configured in Fluent, and exists only in the EF model and the database — **not** in the C# code.
* The database table EFExamples.Vehicle will include a nullable FleetFK column, but you’ll never see or use that property in the Vehicle class unless you access it via EF.Property<Guid?>() manually.
* This structure avoids polluting the Vehicle model with persistence concerns and preserves domain layering.

**Note:** FleetFK is a *shadow property* — it exists in the database and EF model but is not defined in the Vehicle class. This is the recommended pattern when the application doesn’t need to navigate from the child back to the parent.

### Required One-to-Many Entity Collection (1=>1-\*), with Navigation, with shadow FK, Bidirectional

This pattern defines a **required one-to-many relationship** between two entities: a Fleet and the Vehicles it owns. Each Vehicle must belong to one Fleet, and both ends of the relationship are navigable. The foreign key is stored as a **shadow property**, meaning it does not appear in the Vehicle entity, keeping it free of storage concerns.

#### When to Use

Use this pattern when:

* The parent-child relationship is **structurally required** — each child must have a parent.
* Business logic or UI needs to traverse in **both directions** (e.g. get a vehicle's fleet and all vehicles in a fleet).
* You want to enforce referential integrity, but **keep the model focused on relationships**, not database keys.

Avoid it when:

* The child must exist without a parent (then make the FK optional).
* The relationship should only be navigated one way — then remove the reverse navigation.

#### Models

The following are the use case’s relevant models:

public class Fleet {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

// Note the missing FK property.

public List<Vehicle> Vehicles { get; set; } = new();

}

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public Fleet Fleet { get; set; } = null!;

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class FleetConfiguration : IEntityTypeConfiguration<Fleet> {

public void Configure(EntityTypeBuilder<Fleet> builder) {

builder.ToTable("Fleet", "EFExamples");

builder.HasKey(f => f.Id);

builder.Property(f => f.Id).HasColumnName("FleetKey").ValueGeneratedNever();

builder.Property(f => f.Name).HasMaxLength(100).IsRequired();

builder.HasMany(f => f.Vehicles)

.WithOne(v => v.Fleet)

.HasForeignKey("FleetFK")

.IsRequired();

}

}  
public class VehicleConfiguration : IEntityTypeConfiguration<Vehicle> {

public void Configure(EntityTypeBuilder<Vehicle> builder) {

builder.ToTable("Vehicle", "EFExamples");

builder.HasKey(v => v.Id);

builder.Property(v => v.Id).HasColumnName("VehicleKey").ValueGeneratedNever();

builder.Property(v => v.Make).HasMaxLength(100).IsRequired();

builder.Property(v => v.Model).HasMaxLength(100).IsRequired();

}

}

#### Key Points

* This is a **required one-to-many** relationship — a Vehicle **must** belong to a Fleet.
* Both ends of the relationship are navigable — Fleet.Vehicles and Vehicle.Fleet.
* The foreign key (FleetFK) is a **shadow property**, not visible in the Vehicle class.
* This keeps the model clean and focused on **navigations**, not storage mechanics.
* The relationship is enforced by the database, but the domain logic remains untouched by the persistence detail.
* Queries can filter by FK using EF.Property<Guid>(v, "FleetFK") if needed.

**Note:** Use shadow FKs unless you explicitly need to manipulate or expose the foreign key outside of EF.

### Optional Many-to-Many Entity Collection (0–\* ⇄ 0–\*), without Payload, with Navigation, Implicit Join Entity

This pattern defines a **many-to-many relationship** using EF Core’s shortcut .HasMany().WithMany() syntax. EF silently generates a linking table in the background.

While this is the **simplest** way to represent a many-to-many relationship in code, it should be used **with caution**. The link is implicit, uncustomisable, and does not support storing any additional data (e.g. role, status, dates, metadata) about the relationship.

This example is included for **contrast and completeness**. It demonstrates the shortcut syntax but is not recommended for real-world systems unless the relationship is truly static and trivial.

**Important:**While convenient, it should generally be avoided in real-world systems because it cannot capture context (such as role, status, dates, or metadata) on the relationship itself. This example is included for contrast and completeness.

#### When to Use

Use this pattern only when:

* The relationship is **purely structural**, with no additional attributes, logic, or state.
* The link is **permanent, uniform, and always reciprocal** — e.g. tag assignments, reference links, static groupings.
* Neither side ever needs to record metadata about the relationship itself.
* Simplicity is more important than flexibility — e.g. in internal tooling, admin panels, or prototypes.

Avoid this pattern when:

* You may need to add **role**, **status**, **timestamps**, or **auditing** to the relationship.
* You want **clear control** over the linking table schema or indexes.
* You need to **query the relationship** directly or independently.
* You are working in government, education, or enterprise contexts where data usually carries meaning beyond the link.

#### Models

The following are the use case’s relevant models:

public class Student {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public List<Course> Courses { get; set; } = new();

}

public class Course {

public Guid Id { get; set; }

public string Title { get; set; } = null!;

public List<Student> Students { get; set; } = new();

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class StudentConfiguration : IEntityTypeConfiguration<Student> {

public void Configure(EntityTypeBuilder<Student> builder) {

builder.ToTable("Student", "EFExamples");

builder.HasKey(s => s.Id);

builder.Property(s => s.Id).HasColumnName("StudentKey").ValueGeneratedNever();

builder.Property(s => s.Name).IsRequired().HasMaxLength(100);

builder.HasMany(s => s.Courses).WithMany(c => c.Students)

.UsingEntity(j => j.ToTable("StudentCourse", "EFExamples"));

}

}

public class CourseConfiguration : IEntityTypeConfiguration<Course> {

public void Configure(EntityTypeBuilder<Course> builder) {

builder.ToTable("Course", "EFExamples");

builder.HasKey(c => c.Id);

builder.Property(c => c.Id).HasColumnName("CourseKey").ValueGeneratedNever();

builder.Property(c => c.Title).IsRequired().HasMaxLength(200);

}

}

#### Key Points

* EF Core silently generates a **linking table** (EFExamples.StudentCourse) with just two columns: StudentKey and CourseKey.
* There is **no join entity**, so:
  + You **cannot add properties** like EnrolmentDate, Status, or Role.
  + You **cannot control FK names**, column types, or constraints on the join.
* The relationship is **fully navigable** from both directions (bidirectional) and symmetrical.
* The link is purely structural — the existence of a Student ↔ Course link means nothing more than “is enrolled”.
* This is useful only for **trivial, state-free joins**. Once your business model evolves, this pattern becomes a **liability**.
* Refactoring this into a **join entity with payload** later requires replacing all usage patterns, including projections, updates, and navigation logic.

**Note:** You cannot use shadow properties for the join keys or apply indexes or constraints directly on the link — EF manages it internally. If you need control, you must define a join entity.

### Required Many-to-Many Entity Collection (1-\*=>1-\*), without Payload, with Navigation, Explicit Join (Join Must Exist, Link Must Be Validated)

This pattern models a **many-to-many relationship** using an **explicit join entity**, even though the join carries no additional data (no payload). It is an extension of the optional version — except this time, we **require that the join exists** and **enforce referential integrity** on both sides.

However, EF Core and relational databases **do not support enforcing that each root entity participates in at least one join**. That constraint must be enforced in **application logic** (e.g., aggregate or service layer).

This pattern is ideal when you want schema control and consistency, but still keep the join entity lightweight. You may later extend the join to carry timestamps, roles, or status without breaking schema or navigation.

#### When to Use

Use this pattern when:

* Both sides must always participate via the join entity
* You want control over schema/table names and relationships
* You want to **require** that every join has valid references to both entities
* You're prepared to enforce **"must belong to at least one"** in domain logic

Avoid it if:

* The relationship is truly optional, or only used for convenience
* You don't want to manage validation for empty associations

#### Models

TODO

public class Student {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public List<StudentCourse> Enrolments { get; set; } = new();

}

public class Course {

public Guid Id { get; set; }

public string Title { get; set; } = null!;

public List<StudentCourse> Enrolments { get; set; } = new();

}

public class StudentCourse {

public Guid StudentId { get; set; }

public Guid CourseId { get; set; }

public Student Student { get; set; } = null!;

public Course Course { get; set; } = null!;

}

#### Fluent Configuration

TODO

public class StudentConfiguration : IEntityTypeConfiguration<Student> {

public void Configure(EntityTypeBuilder<Student> builder) {

builder.ToTable("Student", "EFExamples");

builder.HasKey(s => s.Id);

builder.Property(s => s.Id).HasColumnName("StudentKey").ValueGeneratedNever();

builder.Property(s => s.Name).IsRequired().HasMaxLength(100);

builder.HasMany(s => s.Enrolments)

.WithOne(e => e.Student)

.HasForeignKey(e => e.StudentId)

.IsRequired(); // Required participation

}

}

public class CourseConfiguration : IEntityTypeConfiguration<Course> {

public void Configure(EntityTypeBuilder<Course> builder) {

builder.ToTable("Course", "EFExamples");

builder.HasKey(c => c.Id);

builder.Property(c => c.Id).HasColumnName("CourseKey").ValueGeneratedNever();

builder.Property(c => c.Title).IsRequired().HasMaxLength(200);

builder.HasMany(c => c.Enrolments)

.WithOne(e => e.Course)

.HasForeignKey(e => e.CourseId)

.IsRequired(); // Required participation

}

}

public class StudentCourseConfiguration : IEntityTypeConfiguration<StudentCourse> {

public void Configure(EntityTypeBuilder<StudentCourse> builder) {

builder.ToTable("StudentCourse", "EFExamples");

builder.HasKey(e => new { e.StudentId, e.CourseId }); // Composite key

}

}

#### Key Points

* This is a **many-to-many** relationship using a **composite join entity** (StudentCourse) with no payload.
* The join entity is required to reference both Student and Course.
* EF enforces **referential integrity** at the database level — each join must be valid.
* However, **EF does not** enforce that **every student or course has at least one join row**.
  + That must be enforced in code (e.g., when saving a student, ensure .Enrolments.Count > 0).
* The schema is robust and future-proof: if enrolments later need a timestamp or status, the model can evolve without breaking navigation.
* This pattern is **recommended** over EF's shortcut .HasMany().WithMany() even when no payload is needed — due to its clarity and safety.

**Note:** This is not a “strict required many-to-many” — the **join is required**, but **participation is not enforced at schema level**. Application logic must guard against unlinked entities if that matters.

### Optional Many-to-Many Entity Collection (0-\*=>0-\*), without Payload, with Navigation, Explicit Join Entity

This pattern models a **many-to-many relationship** using a manually defined **join entity**, even though the relationship currently has **no additional fields**. The join entity acts purely as a bridge — but unlike EF Core’s shortcut .HasMany().WithMany() method, this approach gives you **full control** over the linking table.

While it may seem unnecessary for simple links, this pattern is often the right choice in enterprise or government systems where schema control, naming conventions, and future extensibility matter. It also avoids disruption if the join ever needs to carry **role**, **dates**, or **status** fields later.

#### When to Use

Use this pattern when:

* You want a many-to-many link with **explicit schema control**, even without payload.
* You anticipate adding metadata to the relationship later.
* You want to define **naming**, **indexing**, or **audit rules** on the join table.
* You need **shadow FK access**, which .WithMany() does not expose.

Avoid it when:

* You are certain the join will **never** hold state or grow in complexity.
* Simplicity and rapid iteration outweigh long-term schema flexibility.

#### Models

The following are the use case’s relevant models:

public class Student {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public List<StudentCourse> Enrolments { get; set; } = new();

}

public class Course {

public Guid Id { get; set; }

public string Title { get; set; } = null!;

public List<StudentCourse> Enrolments { get; set; } = new();

}

public class StudentCourse {

public Guid StudentId { get; set; }

public Guid CourseId { get; set; }

public Student Student { get; set; } = null!;

public Course Course { get; set; } = null!;

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class StudentConfiguration : IEntityTypeConfiguration<Student> {

public void Configure(EntityTypeBuilder<Student> builder) {

builder.ToTable("Student", "EFExamples");

builder.HasKey(s => s.Id);

builder.Property(s => s.Id).HasColumnName("StudentKey").ValueGeneratedNever();

builder.Property(s => s.Name).IsRequired().HasMaxLength(100);

builder.HasMany(s => s.Enrolments)

.WithOne(e => e.Student)

.HasForeignKey(e => e.StudentId)

.IsRequired();

}

}

public class CourseConfiguration : IEntityTypeConfiguration<Course> {

public void Configure(EntityTypeBuilder<Course> builder) {

builder.ToTable("Course", "EFExamples");

builder.HasKey(c => c.Id);

builder.Property(c => c.Id).HasColumnName("CourseKey").ValueGeneratedNever();

builder.Property(c => c.Title).IsRequired().HasMaxLength(200);

builder.HasMany(c => c.Enrolments)

.WithOne(e => e.Course)

.HasForeignKey(e => e.CourseId)

.IsRequired();

}

}

public class StudentCourseConfiguration : IEntityTypeConfiguration<StudentCourse> {

public void Configure(EntityTypeBuilder<StudentCourse> builder) {

builder.ToTable("StudentCourse", "EFExamples");

builder.HasKey(e => new { e.StudentId, e.CourseId }); // Composite key

// No additional payload or columns

}

}

#### Key Points

* This is a **manual join entity** — StudentCourse has no payload but represents the link.
* The **composite key** enforces uniqueness of enrolment per student-course pair.
* Navigation is **bidirectional** and exposed through the root entities.
* This gives you **control** over:
  + Table name and schema (EFExamples.StudentCourse)
  + Foreign key naming
  + Future extensibility (e.g. EnrolmentDate, Status)
  + Indexing and constraints
* The domain model is already designed for **future enrichment**.
* This is functionally equivalent to the shortcut .HasMany().WithMany() — but far safer in systems that evolve.

**Note:** This version is functionally identical to EF’s automatic many-to-many join — but unlike the automatic version, this pattern supports shadow properties, payloads, navigation flexibility, and structured growth without schema rewrite.

### Required Many-to-Many Entity Collection (1-\*=>1-\*), without Payload, with Navigation, Explicit Join Entity (Join Must Exist, Link must be Validated)

This pattern models a **many-to-many relationship** using a manually defined **join entity** that holds **no payload**, but is **required to exist** for linking two valid entities. Each join record must reference both sides — the **foreign keys are required**, and **referential integrity is enforced**.

This version is very similar to the optional version, but it takes deliberate steps to require the relationship from both sides — making the join entity part of the core model rather than an optional link.

**Important:** EF Core and relational databases do not allow constraints like “every student must be in at least one course.” This constraint must be enforced in application logic. The requirement here is that each *join record* is valid — not that each *entity* participates in a join.

#### When to Use

Use this pattern when:

* You want full schema control over a many-to-many link.
* You need the ability to evolve the join later (e.g. add timestamps or roles).
* You want both foreign keys to be **required**, enforcing **valid linkage only**.
* You intend to enforce **at least one participation** in your domain logic.

Avoid it when:

* The relationship is temporary or trivial.
* The join should allow payload — then use the "rich join with payload" instead.

#### Models

TODO

public class Author {

public Guid Id { get; set; }

public string FullName { get; set; } = null!;

public List<AuthorBook> Books { get; set; } = new();

}

public class Book {

public Guid Id { get; set; }

public string Title { get; set; } = null!;

public List<AuthorBook> Authors { get; set; } = new();

}

public class AuthorBook {

public Guid AuthorId { get; set; }

public Guid BookId { get; set; }

public Author Author { get; set; } = null!;

public Book Book { get; set; } = null!;

}

#### Fluent Configuration

TODO

public class AuthorConfiguration : IEntityTypeConfiguration<Author> {

public void Configure(EntityTypeBuilder<Author> builder) {

builder.ToTable("Author", "EFExamples");

builder.HasKey(a => a.Id);

builder.Property(a => a.Id).HasColumnName("AuthorKey").ValueGeneratedNever();

builder.Property(a => a.FullName).IsRequired().HasMaxLength(150);

builder.HasMany(a => a.Books)

.WithOne(ab => ab.Author)

.HasForeignKey(ab => ab.AuthorId)

.IsRequired(); // Require FK

}

}

public class BookConfiguration : IEntityTypeConfiguration<Book> {

public void Configure(EntityTypeBuilder<Book> builder) {

builder.ToTable("Book", "EFExamples");

builder.HasKey(b => b.Id);

builder.Property(b => b.Id).HasColumnName("BookKey").ValueGeneratedNever();

builder.Property(b => b.Title).IsRequired().HasMaxLength(200);

builder.HasMany(b => b.Authors)

.WithOne(ab => ab.Book)

.HasForeignKey(ab => ab.BookId)

.IsRequired(); // Require FK

}

}

public class AuthorBookConfiguration : IEntityTypeConfiguration<AuthorBook> {

public void Configure(EntityTypeBuilder<AuthorBook> builder) {

builder.ToTable("AuthorBook", "EFExamples");

builder.HasKey(ab => new { ab.AuthorId, ab.BookId }); // Composite key

// No payload here — just enforcing valid links

}

}

### Optional Many-to-Many Entity Collection (0–\* ⇔ 0–\*), with Payload, with Navigation, with modelled FK

This pattern models a **bidirectional many-to-many relationship** where both ends are optional and the relationship itself is represented as a **first-class join entity**. The join entity carries context (payload) and uses **explicitly declared foreign keys**.

Unlike the required version, this pattern allows:

* Drivers to exist with no associated vehicles
* Vehicles to exist with no assigned drivers

The join still has mandatory references per record — each assignment must link to a Driver and a Vehicle — but those entities may exist independently without any assignments.

#### When to Use

Use this pattern when:

* Drivers and Vehicles are **independent domain entities**.
* The relationship is **optional on both sides** (e.g. pending assignment, unallocated resource).
* The relationship has business meaning (e.g. Role, StartDate, Status).
* You want to expose or query **assignment data** directly.

Avoid this pattern if:

* The relationship must exist at all times for one or both sides — use the required variant instead.
* The relationship has no state or data — then consider .HasMany().WithMany() with no join object.

#### Models

TODO

public class Driver {

public Guid Id { get; set; }

public string FullName { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class DriverVehicleAssignment {

public Guid DriverId { get; set; }

public Guid VehicleId { get; set; }

public string Role { get; set; } = null!;

public DateOnly StartDate { get; set; }

public DateOnly? EndDate { get; set; }

public Driver? Driver { get; set; } // Optional nav

public Vehicle? Vehicle { get; set; } // Optional nav

}

#### Fluent Configuration

TODO

public class DriverVehicleAssignmentConfiguration : IEntityTypeConfiguration<DriverVehicleAssignment> {

public void Configure(EntityTypeBuilder<DriverVehicleAssignment> builder) {

builder.ToTable("DriverVehicleAssignment", "EFExamples");

builder.HasKey(a => new { a.DriverId, a.VehicleId });

builder.Property(a => a.Role).HasMaxLength(50).IsRequired();

builder.Property(a => a.StartDate).HasColumnType("date").IsRequired();

builder.Property(a => a.EndDate).HasColumnType("date");

builder.HasOne(a => a.Driver)

.WithMany(d => d.Assignments)

.HasForeignKey(a => a.DriverId)

.IsRequired(false) // Optional link

.OnDelete(DeleteBehavior.NoAction); // Optional pattern

builder.HasOne(a => a.Vehicle)

.WithMany(v => v.Assignments)

.HasForeignKey(a => a.VehicleId)

.IsRequired(false)

.OnDelete(DeleteBehavior.NoAction);

}

}

#### Key Points

* This is a **true many-to-many** pattern with an explicit, contextual join entity.
* The join carries **payload** (e.g. role, dates), making it meaningful beyond the link.
* Foreign keys are **declared in the model**, not shadowed.
* Participation is **optional on both sides** — Drivers and Vehicles may exist without assignments.
* The join still requires valid references when used — but entities can exist independently.
* Navigation properties are nullable (Driver?, Vehicle?) to reflect optional participation.
* Use DeleteBehavior.NoAction to avoid accidental cascading deletes.
* This structure supports real-world scenarios like:
  + Vehicle assignment history
  + Pending or expired relationships
  + Cross-functional participation (temporary/project-based)

**Warning:**  
An entry can be created with both FKs empty. Ensure that’s what you want (versus a one side being required, the other not).

### Required|Required Many-to-Many Entity Collection (1-\*=>1-\*), with Payload, with Navigation, with modelled FK

We’ll model a scenario where multiple Drivers can be assigned to multiple Vehicles — but each assignment carries context: a Role (e.g. Primary, Backup) and assignment dates.

This is an explicit many-to-many relationship where the *link itself is required* — each assignment must reference both entities — but *participation is optional* from either side. Drivers and Vehicles may exist independently of each other, and only become associated through a navigable, stateful join.

This is often called a **rich join** or **explicit many-to-many**. The join is mandatory for each relationship instance, but neither entity is required to participate.

#### When to Use

Use this pattern when:

* The **relationship is not just a link**, but carries **business-relevant information** (e.g. role, timeframe, status).
* You want to enable **queries, updates, or navigation** based on the relationship itself.
* You need **fine control over keys, constraints, and lifecycle**.
* You may need to reference, filter, or track the join independently of either parent.
* Your domain model includes concepts like **enrolments, assignments, collaborations, memberships, contracts**, or **authorisations**.

Avoid this pattern only when:

* You’re linking entities without any additional state or metadata — use HasMany().WithMany() instead.
* The join entity will never be queried directly or extended.

#### Models

public class Driver {

public Guid Id { get; set; }

public string FullName { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class DriverVehicleAssignment {

public Guid DriverId { get; set; }

public Guid VehicleId { get; set; }

public string Role { get; set; } = null!; // e.g. Primary, Secondary

public DateOnly StartDate { get; set; }

public DateOnly? EndDate { get; set; }

public Driver Driver { get; set; } = null!;

public Vehicle Vehicle { get; set; } = null!;

}

#### Fluent Configuration

public class DriverVehicleAssignmentConfiguration : IEntityTypeConfiguration<DriverVehicleAssignment> {

public void Configure(EntityTypeBuilder<DriverVehicleAssignment> builder) {

builder.ToTable("DriverVehicleAssignment", "EFExamples");

builder.HasKey(a => new { a.DriverId, a.VehicleId }); // Composite key

builder.Property(a => a.Role)

.HasMaxLength(50)

.IsRequired();

builder.Property(a => a.StartDate)

.HasColumnType("date")

.IsRequired();

builder.Property(a => a.EndDate)

.HasColumnType("date");

builder.HasOne(a => a.Driver)

.WithMany(d => d.Assignments)

.HasForeignKey(a => a.DriverId)

.OnDelete(DeleteBehavior.Cascade);

builder.HasOne(a => a.Vehicle)

.WithMany(v => v.Assignments)

.HasForeignKey(a => a.VehicleId)

.OnDelete(DeleteBehavior.Cascade);

}

}

#### Key Points

* This is a **true many-to-many relationship** with a **join object that has meaning**. It's not just a link — it's a thing: an assignment.
* The DriverVehicleAssignment join entity has:
  + Navigation properties to both sides
  + A **composite key** made from both foreign keys
  + Additional properties (Role, StartDate, EndDate) that give context to the relationship
* This pattern is the **recommended** way to model many-to-many relationships in real systems — particularly in government or enterprise — because relationships are rarely binary. They usually carry state, time, or permission implications.
* EF Core’s default HasMany().WithMany() pattern **cannot** support properties — avoid using it unless you're absolutely sure you only need a dumb link.
* We expose both Assignments collections in the root entities (Driver and Vehicle) so queries can include, traverse, or project them easily.

**Note:** Foreign keys are declared explicitly in the join entity and configured using .HasForeignKey(). In this case, we do not use shadow properties — because the relationship itself is a first-class object and deserves explicit keys.

This structure is more maintainable than composite keys over time — but still uses a composite key for uniqueness.  
Consider promoting the join to its own surrogate-keyed entity if it will be versioned, linked externally, or reused across contexts.

### Optional|Optional Many-to-Many Entity Collection (0–\* ⇔ 0–\*), with Payload, with Navigation, with Shadow FKs

This pattern models a **fully optional**, explicit many-to-many relationship where the **join entity carries business data**, and **foreign keys are hidden** using **shadow properties**. The core model remains free from persistence details.

Both participating entities may exist independently. The relationship is navigable and structured but does not require either side to be present initially. This is suitable for systems using **Domain-Driven Design** or **clean architecture** that separate logical models from storage logic.

#### When to Use

Use this pattern when:

* The **relationship has contextual data** (role, dates, metadata).
* You want to allow **Drivers and Vehicles to exist independently**, with optional assignments.
* You want to **keep models clean** by avoiding clutter from foreign keys.
* You want to **persist and query** a join entity with minimal coupling to entity storage logic.

Avoid this pattern if:

* You must **filter or update** the join based on FK values in application code — use modelled FKs instead.
* The relationship is **structurally required** from one or both sides — use a required variant instead.

#### Models

TODO

public class Driver {

public Guid Id { get; set; }

public string FullName { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class DriverVehicleAssignment {

public string Role { get; set; } = null!;

public DateOnly StartDate { get; set; }

public DateOnly? EndDate { get; set; }

public Driver? Driver { get; set; } // Optional

public Vehicle? Vehicle { get; set; } // Optional

}

#### Fluent Configuration

TODO

public class DriverVehicleAssignmentConfiguration : IEntityTypeConfiguration<DriverVehicleAssignment> {

public void Configure(EntityTypeBuilder<DriverVehicleAssignment> builder) {

builder.ToTable("DriverVehicleAssignment", "EFExamples");

builder.Property<Guid>("DriverFK");

builder.Property<Guid>("VehicleFK");

builder.HasKey("DriverFK", "VehicleFK");

builder.Property(a => a.Role)

.HasMaxLength(50)

.IsRequired();

builder.Property(a => a.StartDate)

.HasColumnType("date")

.IsRequired();

builder.Property(a => a.EndDate)

.HasColumnType("date");

builder.HasOne(a => a.Driver)

.WithMany(d => d.Assignments)

.HasForeignKey("DriverFK")

.IsRequired(false)

.OnDelete(DeleteBehavior.NoAction);

builder.HasOne(a => a.Vehicle)

.WithMany(v => v.Assignments)

.HasForeignKey("VehicleFK")

.IsRequired(false)

.OnDelete(DeleteBehavior.NoAction);

}

}

#### Key Points

* This pattern supports a **navigable join entity** without requiring either side to participate.
* Foreign keys are **shadow properties**:
  + They exist in the database and EF Core model
  + But are **not declared in the C# model**
* The entity’s logic is focused entirely on domain concerns (Role, StartDate, etc.).
* **Deletion does not cascade** — appropriate when either side may be independently maintained.
* This model is great for:
  + Temporary or partial assignments
  + Data import, staging, or external mapping scenarios
  + Loose participation in workflows

To filter by FK, use:

EF.Property<Guid?>(assignment, "DriverFK") == someId

**Warning:**  
An entry can be created with both FKs empty. Ensure that’s what you want (versus a one side being required, the other not).

### Required|Required Many-to-Many Entity Collection (1–\* ⇒ 1–\*), with Payload, with Navigation, with shadow FKs

This pattern models a **required, explicit many-to-many relationship** where the join carries contextual data (**payload**), but **foreign keys are hidden** via EF Core’s **shadow properties**. The join is still navigable and structured, but storage concerns do not pollute the domain model.

This is ideal for systems following **clean architecture** or **DDD**, where FKs are considered persistence details, and the join object is treated as a **first-class domain concept**.

#### When to Use

Use this pattern when:

* You need a join entity with business meaning (e.g. an assignment, enrolment, access record).
* The join **must reference both sides** — it is structurally required.
* You want to **keep your domain model clean**, avoiding FK clutter in code.
* Foreign keys are **never used in code** for logic or filtering — navigations are preferred.

Avoid this pattern if:

* You need to **query or filter by FK IDs** inside your application.
* You plan to expose the join in APIs where its identifiers are required.
* You want to manage the join table outside of EF (e.g. shared schema).

#### Models

TODO

public class Driver {

public Guid Id { get; set; }

public string FullName { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class DriverVehicleAssignment {

public string Role { get; set; } = null!;

public DateOnly StartDate { get; set; }

public DateOnly? EndDate { get; set; }

public Driver Driver { get; set; } = null!;

public Vehicle Vehicle { get; set; } = null!;

}

#### Fluent Configuration

TODO

public class DriverVehicleAssignmentConfiguration : IEntityTypeConfiguration<DriverVehicleAssignment> {

public void Configure(EntityTypeBuilder<DriverVehicleAssignment> builder) {

builder.ToTable("DriverVehicleAssignment", "EFExamples");

builder.Property<Guid>("DriverFK");

builder.Property<Guid>("VehicleFK");

builder.HasKey("DriverFK", "VehicleFK");

builder.Property(a => a.Role)

.HasMaxLength(50)

.IsRequired();

builder.Property(a => a.StartDate)

.HasColumnType("date")

.IsRequired();

builder.Property(a => a.EndDate)

.HasColumnType("date");

builder.HasOne(a => a.Driver)

.WithMany(d => d.Assignments)

.HasForeignKey("DriverFK")

.OnDelete(DeleteBehavior.Cascade);

builder.HasOne(a => a.Vehicle)

.WithMany(v => v.Assignments)

.HasForeignKey("VehicleFK")

.OnDelete(DeleteBehavior.Cascade);

}

}

#### Key Points

* The join is a **first-class domain object**, not just a link.
* Both **foreign keys are shadow properties**:
  + They exist in the model and database
  + But are **not declared** in the DriverVehicleAssignment class
* The composite key is defined using those hidden FKs.
* Navigation properties (Driver, Vehicle) allow full inclusion and traversal.
* The join entity holds business data: Role, StartDate, EndDate.
* **No FK clutter** in your C# class — cleaner models, safer refactoring.

To filter by FK in queries, use:

EF.Property<Guid>(assignment, "DriverFK") == someId

### Required|Optional Many-to-Many Entity Collection (1-\*=>0-\*), with Payload, with Navigation, with Modelled FK, One Side Must Participate

We’ll extend our earlier model to require that **each Driver must be assigned to at least one Vehicle** (but vehicles can exist with or without drivers).

We keep:

* Driver – required to have one or more Assignments
* Vehicle – may have none

We’ll **enforce the required assignment** at the model level — and **optionally reinforce it in application logic** or DB constraints.

#### When to Use

TODO

#### Models

public class Driver {

public Guid Id { get; set; }

public string FullName { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class Vehicle {

public Guid Id { get; set; }

public string Make { get; set; } = null!;

public string Model { get; set; } = null!;

public List<DriverVehicleAssignment> Assignments { get; set; } = new();

}

public class DriverVehicleAssignment {

public Guid DriverId { get; set; }

public Guid VehicleId { get; set; }

public string Role { get; set; } = null!;

public DateOnly StartDate { get; set; }

public DateOnly? EndDate { get; set; }

public Driver Driver { get; set; } = null!;

public Vehicle Vehicle { get; set; } = null!;

}

#### Fluent Configuration

The key difference in the configuration is:

builder.HasMany(d => d.Assignments)

.WithOne(a => a.Driver)

.HasForeignKey(a => a.DriverId)

.IsRequired(); // Must be present for every assignment

Which gives:

public class DriverConfiguration : IEntityTypeConfiguration<Driver> {

public void Configure(EntityTypeBuilder<Driver> builder) {

builder.ToTable("Driver", "EFExamples");

builder.HasKey(d => d.Id);

builder.Property(d => d.Id)

.HasColumnName("DriverKey")

.ValueGeneratedNever();

builder.Property(d => d.FullName)

.IsRequired()

.HasMaxLength(200);

builder.HasMany(d => d.Assignments)

.WithOne(a => a.Driver)

.HasForeignKey(a => a.DriverId)

.IsRequired(); // Every assignment must reference a Driver

}

}

public class VehicleConfiguration : IEntityTypeConfiguration<Vehicle> {

public void Configure(EntityTypeBuilder<Vehicle> builder){

builder.ToTable("Vehicle", "EFExamples");

builder.HasKey(v => v.Id);

builder.Property(v => v.Id)

.HasColumnName("VehicleKey")

.ValueGeneratedNever();

builder.Property(v => v.Make)

.IsRequired()

.HasMaxLength(100);

builder.Property(v => v.Model)

.IsRequired()

.HasMaxLength(100);

builder.HasMany(v => v.Assignments)

.WithOne(a => a.Vehicle)

.HasForeignKey(a => a.VehicleId)

.IsRequired(); // Every assignment must reference a Vehicle

}

}

public class DriverVehicleAssignmentConfiguration : IEntityTypeConfiguration<DriverVehicleAssignment> {

public void Configure(EntityTypeBuilder<DriverVehicleAssignment> builder){

builder.ToTable("DriverVehicleAssignment", "EFExamples");

builder.HasKey(a => new { a.DriverId, a.VehicleId });

builder.Property(a => a.Role)

.HasMaxLength(50)

.IsRequired();

builder.Property(a => a.StartDate)

.HasColumnType("date")

.IsRequired();

builder.Property(a => a.EndDate)

.HasColumnType("date");

builder.HasOne(a => a.Driver)

.WithMany(d => d.Assignments)

.HasForeignKey(a => a.DriverId)

.OnDelete(DeleteBehavior.Cascade);

builder.HasOne(a => a.Vehicle)

.WithMany(v => v.Assignments)

.HasForeignKey(a => a.VehicleId)

.OnDelete(DeleteBehavior.Cascade);

}

}

#### Key Points

* This is a **many-to-many join with required participation on one side**.
* The **join itself is mandatory** (you can’t have an assignment without a driver *or* a vehicle).
* This model adds an **application-level constraint**: each Driver must be assigned to at least one Vehicle.
* EF does **not** support collection-level constraints (e.g. “must have at least one”) — this is enforced in code, not Fluent API.
* Use this pattern when business logic expects or demands full participation from one entity type — e.g. “no unassigned teachers,” “no unenrolled students,” “every employee must belong to a team.”
* Alternatively, model this as a **bounded lifecycle**: you can't *create* a Driver until a Vehicle is assigned.

**Note:** This pattern is a design constraint, not just a schema detail — it’s about what the *system allows to exist*. EF can model it, but the enforcement must come from logic or policy.

### Required|Required Many-to-Many Entity Collection (1=>1-\*), with Payload, without Navigation (Service-Only, Cross Domain, Join)

This pattern models a join or log table that links two entities but **does not expose any navigation properties**. It is intentionally decoupled from either side of the relationship in the application layer. The entity exists for persistence and querying, not for traversing. It is commonly used in **cross-domain joins**, **audit logs**, **access traces**, or **integration records**, where domain entities should remain clean and unaware.

#### When to Use

Use this pattern when:

* The join is purely infrastructural (e.g. logging access between systems or services).
* The association crosses **domain boundaries** (e.g. Social → System).
* You want to avoid creating navigation loops or polluting domain models with technical joins.
* Relationships are transient, indirect, or sensitive (e.g. health records accessed by teachers).

It is *not* appropriate when the join needs to drive navigation or hold state meaningful to the core business logic — for that, use a rich join with navigation.

#### Models

The following are the use case’s relevant models:

public class User {

public Guid Id { get; set; }

public string Username { get; set; } = null!;

}

public class Group {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

}

public class UserGroupAccessLog {

public Guid Id { get; set; }

public Guid UserId { get; set; }

public Guid GroupId { get; set; }

public DateTime AccessedAt { get; set; }

public string AccessMethod { get; set; } = null!;

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class UserConfiguration : IEntityTypeConfiguration<User> {

public void Configure(EntityTypeBuilder<User> builder) {

builder.ToTable("User", "EFExamples");

builder.HasKey(u => u.Id);

builder.Property(u => u.Id).HasColumnName("UserKey").ValueGeneratedNever();

builder.Property(u => u.Username).IsRequired().HasMaxLength(100);

}

}

public class GroupConfiguration : IEntityTypeConfiguration<Group> {

public void Configure(EntityTypeBuilder<Group> builder) {

builder.ToTable("Group", "EFExamples");

builder.HasKey(g => g.Id);

builder.Property(g => g.Id).HasColumnName("GroupKey").ValueGeneratedNever();

builder.Property(g => g.Name).IsRequired().HasMaxLength(100);

}

}

public class UserGroupAccessLogConfiguration : IEntityTypeConfiguration<UserGroupAccessLog> {

public void Configure(EntityTypeBuilder<UserGroupAccessLog> builder) {

builder.ToTable("UserGroupAccessLog", "EFExamples");

builder.HasKey(log => log.Id);

builder.Property(log => log.Id).HasColumnName("AccessLogKey").ValueGeneratedNever();

builder.Property(log => log.UserId).IsRequired();

builder.Property(log => log.GroupId).IsRequired();

builder.Property(log => log.AccessedAt).IsRequired();

builder.Property(log => log.AccessMethod).IsRequired().HasMaxLength(50);

}

}

#### Key Points

* This join entity (UserGroupAccessLog) does **not reference User or Group via navigation**. There are no .WithOne() or .WithMany() calls.
* The IDs are treated as pure data — no back-references, no EF navigation, no circular loading.
* The entity exists independently and is typically read or written via a **service layer**, not through aggregate roots.
* This is a good pattern when logs, traces, authorisations, or external interactions need to be recorded without polluting domain models.
* It helps enforce **domain purity**, especially across bounded contexts or services.
* Queries using this object typically project it or filter by ID — e.g., context.AccessLogs.Where(x => x.UserId == ...).

**Note:** This is a great fit for **multi-tenant**, **cross-context**, or **security-focused** systems where relationships may exist without business logic binding them together.

### Optional Self-Referencing Many-to-One(0-\*=>1), with modelled FK

This pattern models an entity that relates to another instance of **itself** — forming either a hierarchy (e.g. organisational tree) or a peer relationship (e.g. referrals or mentorships). EF Core supports this natively using .HasOne().WithMany() and a self-referencing foreign key. These relationships can be required or optional.

#### When to Use

Use this when:

* Entities are part of a hierarchy (e.g. organisational units, file systems, taxonomies).
* There is a recursive or peer-based relationship (e.g. mentors, delegates, reports).
* You want to load or reason about parent-child relationships without creating separate types.

Avoid it if:

* The relationship crosses types or bounded contexts (then a join object may be better).
* You need to store metadata about the relationship — use a rich join entity instead.

#### Models

The following are the use case’s relevant models:

public class OrgUnit {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public Guid? ParentId { get; set; }

public OrgUnit? Parent { get; set; }

public List<OrgUnit> Children { get; set; } = new();

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class OrgUnitConfiguration : IEntityTypeConfiguration<OrgUnit> {

public void Configure(EntityTypeBuilder<OrgUnit> builder) {

builder.ToTable("OrgUnit", "EFExamples");

builder.HasKey(o => o.Id);

builder.Property(o => o.Id).HasColumnName("OrgUnitKey").ValueGeneratedNever();

builder.Property(o => o.Name).IsRequired().HasMaxLength(100);

builder.HasOne(o => o.Parent)

.WithMany(o => o.Children)

.HasForeignKey(o => o.ParentId)

.IsRequired(false); // Root nodes have no parent

}

}

#### Key Points

* This is a **self-referencing optional one-to-many**. Each OrgUnit can have zero or one parent, and zero or many children.
* ParentId is a nullable foreign key to the same table (EFExamples.OrgUnit), enabling root-level entities.
* EF Core supports this natively — no special syntax or reflection needed.
* This pattern is useful for **organisation trees**, **menus**, **nested groups**, or **concept taxonomies**.
* EF can load the entire tree with .Include(o => o.Children).ThenInclude(...) as needed.
* Be cautious when eager-loading large trees — use projection or depth-limited queries when possible.
* Recursive operations (e.g. calculating total depth or flattened descendants) are best handled outside EF (via recursion or raw SQL).

**Note:** Self-referencing foreign keys work exactly like normal foreign keys. Nullable keys represent root nodes. You can apply this same pattern to model delegates, replacements, or fallback relationships.

### Optional Self-Referencing Many-to-One(0-\*=>1), with shadow FK

This pattern defines an optional parent-child hierarchy using a shadow property for the foreign key. The OrgUnit model remains free of storage concerns — ParentId is declared and mapped only in Fluent configuration.

This variation is cleaner in code and maintains the same database structure.

**Models**

The following are the use case’s relevant models:

public class OrgUnit {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public OrgUnit? Parent { get; set; }

public List<OrgUnit> Children { get; set; } = new();

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class OrgUnitConfiguration : IEntityTypeConfiguration<OrgUnit> {

public void Configure(EntityTypeBuilder<OrgUnit> builder) {

builder.ToTable("OrgUnit", "EFExamples");

builder.HasKey(o => o.Id);

builder.Property(o => o.Id).HasColumnName("OrgUnitKey").ValueGeneratedNever();

builder.Property(o => o.Name).IsRequired().HasMaxLength(100);

builder.HasOne(o => o.Parent)

.WithMany(o => o.Children)

.HasForeignKey("ParentOrgUnitFK") // Shadow FK

.IsRequired(false);

}

}

#### Key Points

* This is a **self-referencing hierarchy**, where each OrgUnit may have a parent and any number of children.
* The foreign key ParentOrgUnitFK is a **shadow property** — it is not declared in the entity class, only in Fluent.
* This keeps the application model clean and focused on relationships, not database mechanics.
* EF Core will still create the correct column in the OrgUnit table and maintain referential integrity.
* This pattern is preferable when:
  + You do not need to query or update the FK manually.
  + Navigation is all that's needed for traversal or loading.
* You **should surface the FK** (i.e., make it a declared property) if:
  + You filter or join on the FK value regularly in application logic.
  + You update or manipulate it outside of object navigation.
* In service-layer-driven or DDD models, shadow FKs are usually preferable — they reduce coupling between the domain and persistence layers.

**Note:** With shadow FKs, any filtering must use EF.Property<Guid?>(o, "ParentOrgUnitFK") in LINQ queries.

### Required Self-Referencing Many to One (1-\*=>1), with shadow FK

This pattern models a **self-referencing structure** in which **every entity must have a parent** — that is, no root nodes are allowed. Each entity belongs to exactly one other instance of the same type. The relationship is **required** and configured using a **shadow foreign key**, so the model remains free of storage concerns.

This is an **unusual but valid pattern**, typically seen in systems where a strict hierarchy is imposed without a top-level root — for example, in artificial structures like **token ring networks**, **non-rooted taxonomies**, or **cyclic social graphs**.

However, the pattern can easily become **incoherent** or dangerous if the relationship is assumed to be hierarchical but the data forms **cycles** — e.g. A → B → C → A. EF Core does not detect or prevent these loops — you must guard against them in logic or via recursive constraints in the database (which are rare and complex).

#### When to Use

Use only when:

* You need a **parent** on every record — even the first one created must reference something.
* You are modelling a **looped** or **non-rooted** structure.
* You are explicitly designing a **graph**, not a hierarchy.
* You’re handling data carefully to prevent infinite recursion or cycles.

Avoid if:

* You are trying to model a traditional tree or org chart — those need **optional** parents.
* You assume a clear root or depth — that would violate this required-parent rule.

#### Models

TODO

public class Node {

public Guid Id { get; set; }

public string Label { get; set; } = null!;

public Node Parent { get; set; } = null!;

public List<Node> Children { get; set; } = new();

}

#### Fluent Configuration

TODO

public class NodeConfiguration : IEntityTypeConfiguration<Node> {

public void Configure(EntityTypeBuilder<Node> builder) {

builder.ToTable("Node", "EFExamples");

builder.HasKey(n => n.Id);

builder.Property(n => n.Id).HasColumnName("NodeKey").ValueGeneratedNever();

builder.Property(n => n.Label).IsRequired().HasMaxLength(100);

builder.HasOne(n => n.Parent)

.WithMany(n => n.Children)

.HasForeignKey("ParentNodeFK") // shadow FK

.IsRequired(); // every node must have a parent

}

}

#### Key Points

* This is a **self-referencing required many-to-one**.
* Every Node **must** have a parent (.IsRequired() on the foreign key).
* The foreign key is a **shadow property** (ParentNodeFK) — it’s stored but not exposed in the model.
* EF will enforce that no row exists without a valid parent.
* **Cycles are not prevented** — use application logic or validation to avoid infinite loops.
* There is **no root node** — not even one. This can be problematic in practice.

**Warning:**In most systems, **at least one node needs to be rootless** — e.g., an organisation must have a top-level unit. This pattern **disallows that**, which makes it risky unless you're explicitly designing a **circular, connected structure**.

If your intent is to model:

* A traditional hierarchy: use an **optional** parent relationship
* A directed acyclic graph: you’ll need **extra constraints** to enforce DAG structure
* A general graph: ensure your model supports **cycles** and avoid recursive traversal errors

### Multiple Relationships to the Same Entity Type (Using Shadow Foreign Keys)

This pattern models an entity that references the **same target entity type more than once**, but for **different roles** or purposes. For example, a Job may be created by one User and approved by another. This introduces a **key disambiguation requirement**: EF Core needs to know which FK goes with which navigation.

We’ll define both foreign keys as **shadow properties** to keep the Job model clean and focused.

#### When to Use

Use this when:

* The same target type (e.g. User) plays **multiple roles** in relation to the same entity (e.g. CreatedBy, ApprovedBy).
* You want to keep your model clean, avoid FK clutter, and represent roles explicitly.
* You need to support different filtering paths or permissions based on who did what.

#### Models

The following are the use case’s relevant models:

public class Job {

public Guid Id { get; set; }

public string Title { get; set; } = null!;

public User CreatedBy { get; set; } = null!;

public User ApprovedBy { get; set; } = null!;

}

public class User {

public Guid Id { get; set; }

public string DisplayName { get; set; } = null!;

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class UserConfiguration : IEntityTypeConfiguration<User> {

public void Configure(EntityTypeBuilder<User> builder) {

builder.ToTable("User", "EFExamples");

builder.HasKey(u => u.Id);

builder.Property(u => u.Id).HasColumnName("UserKey").ValueGeneratedNever();

builder.Property(u => u.DisplayName).HasMaxLength(100).IsRequired();

}

}

public class JobConfiguration : IEntityTypeConfiguration<Job> {

public void Configure(EntityTypeBuilder<Job> builder) {

builder.ToTable("Job", "EFExamples");

builder.HasKey(j => j.Id);

builder.Property(j => j.Id).HasColumnName("JobKey").ValueGeneratedNever();

builder.Property(j => j.Title).IsRequired().HasMaxLength(200);

builder.HasOne(j => j.CreatedBy)

.WithMany()

.HasForeignKey("CreatedByUserFK")

.IsRequired();

builder.HasOne(j => j.ApprovedBy)

.WithMany()

.HasForeignKey("ApprovedByUserFK")

.IsRequired(false);

}

}

#### Key Points

* This pattern uses **two navigations to the same type**: one required (CreatedBy), one optional (ApprovedBy).
* EF Core **must be told** which FK belongs to which navigation using .HasForeignKey("...").
* These FKs are implemented as **shadow properties**, so they don’t appear in the Job class.
* This keeps your model free of confusing UserId, CreatedByUserId, or similarly ambiguous fields.
* The generated table will contain CreatedByUserFK and ApprovedByUserFK, but these are only configured, not declared.
* You can still query or filter by FK using EF.Property<Guid>(j, "CreatedByUserFK").

**Note:** Always use .HasForeignKey() with disambiguating names when referencing the same entity type more than once — and prefer shadow FKs to avoid polluting your domain model with internal structure.

### Required|Required Many-to-Many Entity Collection (1-\*<=>1-\*), with Payload, with Surrogate Key (Instead of Composite Keys)

This pattern is similar to *Reqiuired|Required Many-To-Many with Entity Collection (1-\*,1-\*), with Payload, with Navigation* – however it has been elevated to being an Entity in its own right, with its own Id.

But instead of using a composite key ([LeftId, RightId]), it introduces a **surrogate primary key** (usually a GUID or int). This gives the relationship a standalone identity — useful when the join is edited independently, referenced externally, or extended with history.

#### When to Use

Use this pattern when:

* The join is meaningful — not just a link, but an entity in its own right (e.g. Enrolment, JobAssignment, Contract).
* The join may later be **queried, updated, or versioned** independently of the root entities.
* You want to simplify navigation, updates, or make the join addressable (e.g. by API or document reference).
* The relationship has **more than two keys** (e.g. also scoped to a School or Period).

Avoid it when:

* The join has no identity or state and is only ever traversed — then composite keys are fine.

#### Models

The following are the use case’s relevant models:

public class Teacher {

public Guid Id { get; set; }

public string FullName { get; set; } = null!;

public List<JobAssignment> Assignments { get; set; } = new();

}

public class School {

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public List<JobAssignment> Assignments { get; set; } = new();

}

public class JobAssignment {

public Guid Id { get; set; }

public string Position { get; set; } = null!;

public DateOnly StartDate { get; set; }

public DateOnly? EndDate { get; set; }

public Teacher Teacher { get; set; } = null!;

public School School { get; set; } = null!;

}

**Fluent Configuration**

The above models are configured as part of the domain as follows:

public class TeacherConfiguration : IEntityTypeConfiguration<Teacher> {

public void Configure(EntityTypeBuilder<Teacher> builder) {

builder.ToTable("Teacher", "EFExamples");

builder.HasKey(t => t.Id);

builder.Property(t => t.Id).HasColumnName("TeacherKey").ValueGeneratedNever();

builder.Property(t => t.FullName).HasMaxLength(200).IsRequired();

builder.HasMany(t => t.Assignments)

.WithOne(a => a.Teacher)

.HasForeignKey("TeacherFK")

.IsRequired();

}

}

public class SchoolConfiguration : IEntityTypeConfiguration<School> {

public void Configure(EntityTypeBuilder<School> builder) {

builder.ToTable("School", "EFExamples");

builder.HasKey(s => s.Id);

builder.Property(s => s.Id).HasColumnName("SchoolKey").ValueGeneratedNever();

builder.Property(s => s.Name).HasMaxLength(200).IsRequired();

builder.HasMany(s => s.Assignments)

.WithOne(a => a.School)

.HasForeignKey("SchoolFK")

.IsRequired();

}

}

public class JobAssignmentConfiguration : IEntityTypeConfiguration<JobAssignment> {

public void Configure(EntityTypeBuilder<JobAssignment> builder) {

builder.ToTable("JobAssignment", "EFExamples");

builder.HasKey(a => a.Id);

builder.Property(a => a.Id).HasColumnName("AssignmentKey").ValueGeneratedNever();

builder.Property(a => a.Position).IsRequired().HasMaxLength(100);

builder.Property(a => a.StartDate).HasColumnType("date").IsRequired();

builder.Property(a => a.EndDate).HasColumnType("date");

builder.HasOne(a => a.Teacher)

.WithMany(t => t.Assignments)

.HasForeignKey("TeacherFK")

.OnDelete(DeleteBehavior.Cascade);

builder.HasOne(a => a.School)

.WithMany(s => s.Assignments)

.HasForeignKey("SchoolFK")

.OnDelete(DeleteBehavior.Cascade);

}

}

#### Key Points

* The JobAssignment entity is a **first-class object** with its own identity (AssignmentKey).
* This lets it be referenced elsewhere — e.g. linked to timesheets, evaluations, or audit logs.
* It avoids the complexity of composite keys and makes updates easier (no need to reassign both keys).
* Navigation is enabled in both directions — Teacher → Assignments, School → Assignments.
* Foreign keys to Teacher and School are **shadow properties**, keeping the domain model clean.
* This pattern is common in education (Enrolment), HR (EmploymentContract), and project work (Assignment, Participation).

**Note:** Always prefer a surrogate key when the join may become a **referencable, editable, or standalone** entity. Composite keys are cheaper up front, but harder to evolve.

**Performance and Indexing Note:**  
While this pattern uses a surrogate primary key for simplicity and referential clarity, it's often valuable to **add a composite unique index** on the foreign keys (e.g. TeacherFK and SchoolFK). This ensures that no duplicate assignment pairs can exist and improves query performance when filtering by both keys. EF Fluent configuration:

builder.HasIndex("TeacherFK", "SchoolFK").IsUnique();

This does not replace the surrogate key but complements it — combining the benefits of **standalone identity** with **efficient lookup** and **integrity constraints**. Use this approach especially when the join carries meaningful payload and is expected to grow or be queried heavily.

### Cross-Schema Relationship (Between Domains)

This pattern models a relationship between entities that belong to **different logical domains**, each of which is mapped to a separate **database schema**. It is a practical application of **Domain-Driven Design** (DDD), where schemas are used to enforce boundaries and clarify context.

Cross-schema navigation should be **deliberate and rare**. Objects should not freely reach across domains. When they do, it should be a **clear and explicit relationship** — not an accident of convenience.

This example shows how a LearnerProfile in the Education domain relates to a User in the System domain. The reference is one-way: education knows about system access, but the system does not know about specific learners.

#### When to Use

Use this when:

* A domain needs to reference another domain’s entity for integration, access, or lifecycle alignment.
* You want to **enforce separation** between bounded contexts via schema, while still allowing controlled links.
* The referencing domain is **dependent**, but not authoritative, over the other domain.

Avoid it if:

* You are frequently navigating across domains — that may indicate a domain boundary has been misplaced.
* You are attempting to reuse foreign keys or navigation for convenience instead of business intent.

#### Models

The following are the use case’s relevant models:

public class LearnerProfile {

public Guid Id { get; set; }

public string PreferredName { get; set; } = null!;

public User User { get; set; } = null!;

}

public class User {

public Guid Id { get; set; }

public string Username { get; set; } = null!;

}

#### Fluent Configuration

The above models are configured as part of the domain as follows:

public class UserConfiguration : IEntityTypeConfiguration<User> {

public void Configure(EntityTypeBuilder<User> builder) {

builder.ToTable("User", "System");

builder.HasKey(u => u.Id);

builder.Property(u => u.Id).HasColumnName("UserKey").ValueGeneratedNever();

builder.Property(u => u.Username).HasMaxLength(100).IsRequired();

}

}

public class LearnerProfileConfiguration : IEntityTypeConfiguration<LearnerProfile> {

public void Configure(EntityTypeBuilder<LearnerProfile> builder) {

builder.ToTable("LearnerProfile", "Education");

builder.HasKey(lp => lp.Id);

builder.Property(lp => lp.Id).HasColumnName("LearnerProfileKey").ValueGeneratedNever();

builder.Property(lp => lp.PreferredName).HasMaxLength(100).IsRequired();

builder.HasOne(lp => lp.User)

.WithMany()

.HasForeignKey("UserFK") // Shadow FK

.IsRequired();

}

}

#### Key Points

* User is defined in the System schema; LearnerProfile is in the Education schema — this reinforces domain boundaries in both code and database.
* The navigation is one-way: LearnerProfile → User. The system does not care about education profiles.
* The foreign key (UserFK) is a **shadow property**, keeping the LearnerProfile model clean of persistence concerns.
* This is an intentional **cross-domain relationship** — not a default or accidental join.
* Such links should be **rare**, documented, and often mediated through **services** when access is optional or conditional.
* The database enforces referential integrity, but the model stays logically clear — Education depends on System, not the other way around.

**Note:** Use schemas to reflect bounded contexts, and shadow properties to avoid coupling storage with business logic. If navigation is needed across schemas, prefer one-way directionality and make the dependency explicit.

1. Not tatoo’ed on the learner either. [↑](#footnote-ref-2)