DRAFT - ICT Project

<TODO>

Version:

0.1

## Purpose

<TODO>

## Synopsis

<TODO>

## Contents

[Description 1](#_Toc154927865)

[Synopsis 1](#_Toc154927866)

[Contents 2](#_Toc154927867)

[Purpose 3](#_Toc154927868)

[Background 3](#_Toc154927869)

[Outcomes [Objectives] 3](#_Toc154927870)

[Options [Considered & Selected] 3](#_Toc154927871)

[Constraints 3](#_Toc154927872)

[Assumptions 3](#_Toc154927873)

[Dependencies 3](#_Toc154927874)

[Decisions 3](#_Toc154927875)

[Deliverables/Outputs 3](#_Toc154927876)

[Heading Level 3 3](#_Toc154927877)

[Heading Level 4 4](#_Toc154927878)

[Appendices 5](#_Toc154927879)

[Appendix A - Document Information 5](#_Toc154927880)

[Versions 5](#_Toc154927881)

[Images 5](#_Toc154927882)

[Tables 5](#_Toc154927883)

[References 5](#_Toc154927884)

[Review Distribution 5](#_Toc154927885)

[Audience 5](#_Toc154927886)

[Structure 5](#_Toc154927887)

[Diagrams 6](#_Toc154927888)

[Terms 6](#_Toc154927889)

## Background

Object-Relational Mapping (ORM) is a mature and widely adopted pattern that emerged in mainstream software development in the early 2000s. It provides a structured method for bridging object-oriented programming and relational database systems. By abstracting database operations behind a consistent object model, ORMs reduce direct dependency on SQL and help developers focus on modelling real-world behaviour and relationships in code.

The primary benefit of ORMs is the decoupling of business logic from storage concerns. They remove the need for repetitive, boilerplate SQL and allow developers to express intent in terms of domain entities and their relationships. This has led to more maintainable and flexible codebases, and has enabled architectural patterns such as Domain-Driven Design (DDD) to flourish.

From a security perspective, ORMs offer a major advantage: the near-total removal of SQL injection as an attack vector. By default, all query parameters are automatically escaped and parameterised, preventing user-supplied input from manipulating the underlying query structure. This has made ORMs a standard recommendation in security-conscious environments. Their use is now considered best practice for most government and enterprise applications, particularly when handling personal, financial, or legally protected data.

Over time, ORM tooling has matured. What began as a mechanism for managing queries eventually extended into database schema generation. This shifted responsibility for schema design from database administrators (DBAs) and data architects to application developers. While this enabled greater agility in development, it also exposed systems to new risks — primarily from developers lacking a deep understanding of indexing strategies, constraint modelling, or query planning.

The evolution moved through distinct stages: Database First (where schemas were manually created and reverse-engineered into code), Model First (where visual models generated both schema and code), and finally Code First — where the structure of the codebase itself becomes the single source of truth, with the schema generated from it automatically. Code First is now the dominant approach in modern software development due to its flexibility and alignment with iterative, test-driven, and domain-led methods.

However, the concerns raised by data architects remain valid. The introduction of ORMs into schema design did not remove the need for sound relational modelling or indexing discipline. It simply moved the responsibility. Data specialists still have a critical role in the development lifecycle — particularly in reviewing constraint definitions, query paths, and expected volumes. The tools may have changed; the fundamentals of effective database design have not.

## Why This Guidance Focuses on Entity Framework Core

There are many ORMs available to developers. This guidance focuses on EF Core because it satisfies several key criteria relevant to government systems:

1. It has a long development pedigree, with decades of enterprise usage and continual refinement. EF Core builds on earlier versions of Entity Framework and is actively supported and maintained by Microsoft.
2. It is one of a small number of ORM libraries that fully support both query generation *and* schema creation. This dual capability makes it well-suited to modern development workflows where schema evolves with application logic.
3. This schema generation capability also aligns well with current deployment methodologies, where pipelines create the infrastructure by code, then the schemas by code, then the integrations by code, the provisioning by code, and so on.
4. It is among the very few ORMs that natively support the Code First approach, enabling schemas to be expressed directly from application logic without requiring separate schema design tools or database-first processes.
5. It integrates deeply with the .NET.Core ecosystem, including support for dependency injection, async execution, LINQ-based querying, and migration management. This makes it a natural fit for the technology stacks already widely adopted across public sector systems.

Some alternatives to EF Core offer individual features that may appeal in specific cases. However, those features often come with trade-offs — including reduced long-term support, less mature tooling, or weaker integration with security and dependency management standards. While it is technically true that direct SQL or another ORM might deliver marginal performance improvements in highly specific cases, these are almost always outweighed by other factors. Meaningful performance gains are the result of appropriate indexing, caching, and load balancing — not query syntax.

The marginal advantage of handcrafted SQL is rarely worth the cost of increased complexity, tighter coupling, reduced testability, and the heightened risk of introducing easily avoidable security vulnerabilities. EF Core, used properly, provides a strong baseline of safety, consistency, and operational flexibility that meets the demands of modern application development in government.

## 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, **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

This 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.

### 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; }

}

For now, these group roles are **descriptive**, not access-based. They carry no system permissions. Their purpose is to model **social structure** — who someone is *to a group*, not what they are allowed to do in the system.

This distinction is important: Group roles reflect real-world position; system roles reflect technical authority. Conflating the two leads to confusion, brittle logic, and security risks.

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).