1 · The Seam, Signed

Volume I closed by doing something implementation books rarely do: it itemized its debts. The epilogue of From Blueprint to Core listed, in writing, everything the core declared and did not build — ports, clauses, scars, and statutes — and named this volume as the debtor. This chapter therefore has three jobs and invents nothing in the course of doing them. It reads the inheritance aloud, so the whole book can be audited against one table. It keeps the first and most symbolic promise — the repository contract subclass, drafted in a Volume I comment, made to pass against a real database before this volume is twenty pages old. And it states the one split in the inheritance — the outbox — exactly once, so that no later chapter can soften it.

The inheritance, read aloud

Every row of Table 1.1 is a compile-time or test-time fact of the Volume I solution, quoted from its epilogue and its Part III audit — not a plan, an obligation. The right column is this volume's answer: the chapter where each debt is discharged. The table is printed first so that every later chapter is reachable from here, and so that the epilogue of this volume can audit the book against it, row by row.

INHERITED, IN WRITING (VOL. I)	DISCHARGED (THIS VOLUME)
Three ports — IRepository<TRoot,TId>, IUnitOfWork, IDateTimeProvider — and three read models, each with a named fake and its contract in XML documentation	Chs. 2–5 and 9: every fake replaced by a real adapter that passes the same contract tests
The repository contract class awaiting its adapter subclass — drafted verbatim in Vol. I, Listing 19.3	Ch. 1 — this chapter, kept on p. 11
The after-commit dispatch clause, contract language in the port's XML documentation	Ch. 5 — turned into code at the Unit of Work
The failed-commit scar, as law: a loaded aggregate never survives a failed commit	Ch. 5 — enforced and watched to fail against a real transaction
Optimistic concurrency, owed to Vol. I Chapter 8	Ch. 6 — rowversion columns; the concurrent-investigator race, lost on purpose; a named refusal
The outbox, owed to Vol. I Chapter 9	Ch. 7 — the write half; the dispatcher is Volume III's (the split, below)
Typed identifiers that must survive value conversion without leaking their primitive	Ch. 3 — every ID; the atlas's key shapes met in the open
A statute's worth of published refusal codes the API must carry to clients unedited	Ch. 12 — RFC 7807 problem details, codes verbatim
Authorization and idempotency, filed with this volume by the Part III audit	Chs. 13 and 15–17 — idempotency at the boundary; OperatorId proven, no longer on faith
Open reader exercises 16.1, 17.1, 17.2 — and the dispatcher of Vol. I awaiting its production caller	Carried on the ledger; Ch. 9 supplies the caller (the API is the caller)

Table 1.1 — The inheritance ledger: what Volume I handed over in writing, and where this volume answers.

Two projects, and the arrows that matter

The solution grows by exactly two projects: the Infrastructure layer, and the test project that watches it. Six commands, and the reference arrows they create are the chapter's first design decision:

```
$ dotnet new classlib -o src/Nexus.Infrastructure
$ dotnet new xunit    -o tests/Nexus.Infrastructure.Tests
$ dotnet sln add src/Nexus.Infrastructure tests/Nexus.Infrastructure.Tests
$ dotnet add src/Nexus.Infrastructure reference src/Nexus.Application src/Nexus.Domain
$ dotnet add tests/Nexus.Infrastructure.Tests reference src/Nexus.Infrastructure \
      tests/Nexus.Application.Tests
$ dotnet add src/Nexus.Infrastructure package Microsoft.EntityFrameworkCore.SqlServer \
      --version 8.0.8
```

Listing 1.1 — The solution extended. Appendix C pins every package version this volume adds.

Two of those arrows deserve their sentence. Nexus.Infrastructure references Application and Domain — never the reverse. The Dependency Rule that Chapter 2 of Volume I made a compiler error (CS0246, the missing-reference error) keeps working here without any new machinery: nothing above the seam can name a DbContext, because the reference that would let it does not exist to be used. And the test project references tests/Nexus.Application.Tests — a test project referencing a test project, which is unusual enough to justify aloud: RepositoryContract<TRoot,TId> lives there, and inheriting the law verbatim matters more than folder aesthetics. Extracting a shared test kit is a real alternative with a real price, and it is deferred, named, to Volume III's harness work.

The smallest honest context

The contract needs a real database on the other side of the port, which needs a DbContext. Chapter 2 owns the NexusDbContext in full — seventeen schemas, assembly-scanned configurations, the atlas made executable. What this chapter builds is the same class at birth: born knowing exactly one citizen, so that the proof can run today without pretending the atlas work is done.

```
namespace Nexus.Infrastructure.Persistence;

/// <summary>The single DbContext of the NEXUS-1 backend. Born in Chapter 1
/// knowing one citizen; taught the seventeen-schema atlas in Chapters 2-3;
/// never taught anything above the seam.</summary>
public sealed class NexusDbContext(DbContextOptions<NexusDbContext> options)
    : DbContext(options)
{
    public DbSet<Policy> Policies => Set<Policy>();

    protected override void OnModelCreating(ModelBuilder modelBuilder) =>
        modelBuilder.ApplyConfigurationsFromAssembly(
            typeof(NexusDbContext).Assembly);
}
```

Listing 1.2 — src/Nexus.Infrastructure/Persistence/NexusDbContext.cs, complete as of this chapter. ApplyConfigurationsFromAssembly is the atlas's own discovery convention — one configuration class per table, no giant switch — adopted on day one so Chapter 2 extends this file without editing it.

One mapping class makes Policy persistable — Volume I's ReinforcementLearning aggregate, chosen here for the same reason Volume I chose it for the fake's contract: it is small, it has typed identifiers, and it is nobody's special case.

```
namespace Nexus.Infrastructure.Persistence.Configurations.ReinforcementLearning;

/// <summary>Chapter-1 scaffolding: exactly enough mapping to sign the
/// contract. The atlas's key shapes and lookup discipline are Chapter 3's,
/// and this file is revised there -- in the open.</summary>
public sealed class PolicyConfiguration : IEntityTypeConfiguration<Policy>
{
    public void Configure(EntityTypeBuilder<Policy> b)
    {
        b.ToTable("Policy", "ReinforcementLearning");
        b.HasKey(p => p.Id);
        b.Property(p => p.Id)
            .HasConversion(id => id.Value, v => new PolicyId(v));
        b.Property(p => p.UnitId)
            .HasConversion(id => id.Value, v => new UnitId(v));
        b.Property(p => p.SourceTable)
            .HasConversion(id => id.Value, v => new QTableId(v));
        b.Property(p => p.Status).HasConversion<string>().HasMaxLength(20);
        b.Property(p => p.EvaluationScore);
        b.Ignore(p => p.DomainEvents); // the mailbox is not a column (Ch. 7)
    }
}
```

Listing 1.3 — PolicyConfiguration.cs, first edition. Three value converters keep Chapter 4's typed identifiers intact across the seam: the wrapper evaporates at the column and reappears at rehydration, exactly as Volume I promised, and no naked Guid escapes into a signature on the way.

A reader with the schema atlas open has already noticed what this listing avoids saying: ReinforcementLearning.Policy in From Schema to System keys on INT IDENTITY, carries a status as a lookup foreign key, and audits itself with columns this mapping does not mention — while PolicyId wraps a Guid. That collision is real, it is the most interesting problem in Part I, and it belongs to Chapter 3, where key shapes are met honestly rather than in a footnote. What Chapter 1 proves is the seam; what it defers — printed in amber before this chapter closes — is the shape.

The three-line adapter

```
namespace Nexus.Infrastructure.Persistence;

/// <summary>The adapter Volume I promised: one member per port member,
/// no cleverness, no queries. Thickness here would betray the port.</summary>
public sealed class EfRepository<TRoot, TId>(NexusDbContext db)
    : IRepository<TRoot, TId>
    where TRoot : Entity<TId>, IAggregateRoot
    where TId : notnull
{
    public async Task<TRoot?> GetAsync(TId id, CancellationToken ct = default) =>
        await db.Set<TRoot>().FindAsync([id], ct);

    public Task AddAsync(TRoot root, CancellationToken ct = default) =>
        db.Set<TRoot>().AddAsync(root, ct).AsTask();

    public void Remove(TRoot root) => db.Set<TRoot>().Remove(root);
}
```

Listing 1.4 — EfRepository.cs, complete. Generic over the typed ID, as the port demands: the adapter cannot regress to naked Guids because its own signature will not let it.

Chapter 4 of the outline calls these the three-line adapters, and the name is a design assertion, not a boast. Everything a repository might be tempted to add — queries, includes, save calls, caching — is either the query side's business (Chapter 15 of Volume I, and Chapter 9 here), the Unit of Work's business (Chapter 5), or nobody's. FindAsync is the one considered choice: it consults the change tracker before the database, and that behavior is about to matter more than it appears.

The promise, kept verbatim

Volume I, Listing 19.3, closed with a comment: the adapter's contract subclass, drafted in advance — “two overrides, one real adapter, the same three laws.” Here is that comment, made code, with one helper it was always going to need:

```
namespace Nexus.Infrastructure.Tests;

/// <summary>Volume I, Listing 19.3, its closing comment made code --
/// verbatim by design. The laws are inherited, not restated.</summary>
public sealed class EfRepositoryContract
    : RepositoryContract<Policy, PolicyId>, IDisposable
{
    private readonly NexusDbContext _db = ThrowawayDatabase.Open();

    protected override IRepository<Policy, PolicyId> NewRepository() =>
        new EfRepository<Policy, PolicyId>(_db);

    protected override Policy NewRoot() =>
        Policy.Draft(UnitId.New(), QTableId.New());

    public void Dispose() => _db.Dispose();
}

internal static class ThrowawayDatabase
{
    // CONFESSED CONSTANT: configuration environments are Volume III's.
    // Until then the learning machine's connection string lives here,
    // once, named. Appendix A installs LocalDB.
    private const string Cs =
        @"Server=(localdb)\MSSQLLocalDB;Database=Nexus_Ch01_Proof;" +
        @"Trusted_Connection=True";

    public static NexusDbContext Open(bool wipe = true)
    {
        var options = new DbContextOptionsBuilder<NexusDbContext>()
            .UseSqlServer(Cs).Options;
        var db = new NexusDbContext(options);
        if (wipe)
        {
            db.Database.EnsureDeleted();  // every run starts from nothing
            db.Database.EnsureCreated();  // Chapter-8 debt: migrations, not this
        }
        return db;
    }
}
```

Listing 1.5 — EfRepositoryContract.cs and its throwaway database. Two overrides, one real adapter, the same three laws — the diff Volume I wrote in advance, plus the fixture it predicted it would need.

Run it. The standing rule says watched to fail, and the seam obliges on the first try:

```
$ dotnet test tests/Nexus.Infrastructure.Tests
Failed!  System.InvalidOperationException : No suitable constructor was
found for entity type 'Policy'. ... cannot bind 'source' in
'Policy(PolicyId id, UnitId unitId, QTableId source)'.
```

The first run: red, and usefully so.

EF Core rehydrates through the private constructor by matching parameter names to properties — id to Id, unitId to UnitId — and stalls on the third: the parameter is named source, the property SourceTable. The fix is a one-word rename inside a private constructor. No behavior changes; no caller exists to notice. And it does not matter that the change is trivial, because Volume I's promise was not trivial: entities shaped to satisfy the atlas

without modification. The Domain just changed in order to be persisted, and the standing rule does not grade on intent. The ledger this opens stays open to the epilogue, which audits it.

```
// src/Nexus.Domain/ReinforcementLearning/Policy.cs -- the one-word audit:
private Policy(PolicyId id, UnitId unitId, QTableId sourceTable) : base(id)
{ UnitId = unitId; SourceTable = sourceTable; Status = PolicyStatus.Draft; }
```

Listing 1.6 — The first entry in the Broken Promise Ledger, printed rather than patched.

> **TABLE 1.2 · THE BROKEN PROMISE LEDGER (OPENED)**
>
> Entry 1 — Policy.cs: private constructor parameter renamed source → sourceTable, so that EF Core constructor binding can find SourceTable. Behavior changed: none. Callers affected: none (the constructor is private). Counted anyway: the Domain changed to be persisted. Every future entry takes this row's shape — what changed, why persistence required it, what it cost — and the volume's epilogue audits the ledger in full.

```
$ dotnet test tests/Nexus.Infrastructure.Tests
Passed!  - Failed: 0, Passed: 3, Skipped: 0, Duration: 4.6 s
```

The promise, kept: the fake's three laws, signed by a real adapter against a real database.

Three laws, one identity map — the confession

Before the census is celebrated, the run deserves an honest reading. Two of the three laws barely consulted SQL Server. What-is-added-is-what-is-returned passed because FindAsync found the instance in the change tracker — EF Core's identity map — and handed back the very object AddAsync had tracked; no INSERT had run, no row existed, and Assert.Equal was satisfied by reference identity it did not ask for. The contract is not wrong — a law must be one every implementor can sign, including one with a tracker — but a chapter that stopped here would have proven the seam and only gestured at the database. The proof the volume actually owes is rehydration: the same identity, in a stranger's body, after the first context is gone. Volume I named this exact gap at Listing 12.5 — code may rely on Equals, never on ReferenceEquals — so the test asserts both sides of it:

```
namespace Nexus.Infrastructure.Tests;

public sealed class RehydrationTests
{
    [Fact]
    public async Task What_is_committed_survives_a_new_context_as_an_equal_stranger()
    {
        var root = Policy.Draft(UnitId.New(), QTableId.New());

        using (var writing = ThrowawayDatabase.Open())
        {
            await new EfRepository<Policy, PolicyId>(writing).AddAsync(root);
            await writing.SaveChangesAsync(); // Chapter 5 owns this line's fate
        }

        using var reading = ThrowawayDatabase.Open(wipe: false);
        var loaded = await new EfRepository<Policy, PolicyId>(reading)
            .GetAsync(root.Id);

        Assert.Equal(root, loaded);    // Chapter 4's equality: type and Id
        Assert.NotSame(root, loaded);  // the gap Listing 12.5 named, closed
        Assert.Equal(PolicyStatus.Draft, loaded!.Status);
    }
}
```

Listing 1.7 — RehydrationTests.cs: an INSERT, a disposed context, a SELECT, and a stranger that passes for the original everywhere except ReferenceEquals. The direct SaveChangesAsync call is infrastructure testing infrastructure — lawful here, and never how a handler will commit.

```
$ dotnet test
Passed!  - Failed: 0, Passed: 54, Skipped: 0, Duration: 6.8 s
  Nexus.Domain.Tests          36   ~0.1 s   SDK only, as always
  Nexus.Application.Tests     14   ~0.5 s   named fakes, as always
  Nexus.Infrastructure.Tests   4   ~6 s     one real database, wiped per run
```

Table 1.3 — The census after Chapter 1: fifty inherited, four new — and the first suite that is slow on purpose. Volume I priced this in advance: Volume II's tests buy different guarantees, at a different speed, by design.

> **HONEST BOUNDARY**
>
> Three deferrals shaped this chapter and are printed where they apply. EnsureCreated is scaffolding: it creates whatever tables the current model implies, which is not the same as the 654 tables of the schema atlas — the Guid-keyed Policy table this chapter created will not survive Chapter 3, where the atlas's INT IDENTITY key shape meets PolicyId's Guid in the open, and Chapter 8 replaces EnsureCreated with migrations that hit the atlas exactly. The connection string is a confessed constant: configuration environments are Volume III's, and until then it lives in one named place. And nothing in this chapter is an integration test suite: these four tests are a promise being kept in public, not systematic coverage — the harness, TestContainers and all, is Volume III's opening act.

The outbox split, stated once

One inherited row of Table 1.1 arrives split, and the split is stated here — once, authoritatively — so no later chapter can soften it by restatement. Volume I's Chapter 9 owes the trilogy an outbox: domain events persisted atomically with aggregate state, so that a commit-then-crash loses nothing. This volume builds the write half, in Chapter 7: the table, the envelope, the idempotency key, the same-transaction guarantee. The dispatcher that drains that table — scheduling, retries, failure isolation between subscribers — is a background service, background services are runtime, and runtime is Volume III. The consequence is inherited too: wiring flood verdicts to alarm reactions and policy activations to the DigitalTwin stays deferred, because reliable delivery is

exactly the part that does not exist yet. Volume I Chapter 9's waiting rule remains in force, and Chapter 7 will point back to this paragraph rather than negotiate with it.

> **CHECKPOINT · CHAPTER 1**
>
> 1. Recite five rows of the inheritance ledger from memory — the obligation, and the chapter that discharges it.
> 2. Two of the three contract laws passed before any INSERT ran. Name the mechanism responsible, and explain why the contract is still right to be signable by it.
> 3. Defend the rehydration test's three asserts: what does each prove, and which Volume I listing predicted the NotSame?
> 4. State the Broken Promise Ledger's first entry: what changed, why EF Core required it, and why the standing rule counts a one-word rename in a private constructor as a broken promise at all.
> 5. State the outbox split in one sentence, and name what stays deferred — and to whom — because of it.

The seam is signed; the signature is four green tests and one honest ledger. What the signature covers is still one aggregate and a scaffolded table — the next two chapters widen it to the atlas: first the context that can hold seventeen schemas without learning anything above the seam, then the citizens mapped as the atlas actually drew them, key shapes and all.

The Adapter, the Identity Map, and the First Broken Promise

WHERE WE ARE IN THE SOLUTION

```
 Nexus.sln
 |- src/Nexus.SharedKernel
 |- src/Nexus.Domain
 |- src/Nexus.Application
>|- src/Nexus.Infrastructure          NEW
 |- tests/Nexus.Domain.Tests
 |- tests/Nexus.Application.Tests
>|- tests/Nexus.Infrastructure.Tests  NEW
```

Tests green: 54 (50 inherited · 4 new)
Migrations: 0 · Databases: 1 (throwaway, wiped per run)

The chapter's story in one panel: two new projects, four new tests, zero migrations — and the first suite in the trilogy that needs anything installed beyond the SDK. The count matters less than the asymmetry: fifty tests still run in half a second with no database at all, because the core still depends on nothing. Only the new four pay the price of the real world, which is the Onion working as drawn.

What an adapter actually is

Volume I's Application layer never says “database.” It says IRepository — a port: a demand, written from the consumer's side, for someone to store and return aggregate roots. An adapter is any class that answers the demand. Volume I answered it with a dictionary (the fake); this chapter answers it with EF Core and SQL Server (the real one). The crucial property is that the consumer cannot tell — CloseCaseHandler compiled in Volume I and was not touched, recompiled, or even re-tested to make this chapter work. That is what “changing nothing above the seam” means in practice: the adapter is new, the demand is not.

The identity map, in plain language

A DbContext keeps a private registry of every entity it has loaded or been handed — the change tracker. When FindAsync is asked for a key, it checks that registry before writing SQL, and if the entity is already there it returns that exact object; this is the identity map pattern, and it exists so that one transaction never holds two disagreeing copies of the same row. It is also why two of the three contract laws passed without an INSERT: Add put the policy in the registry, Get found it there, and SQL Server was a bystander. The map is a feature, not a cheat — but a proof of persistence must evict it, which is what disposing the writing context does in Listing 1.7. Figure D1.1 traces both paths.

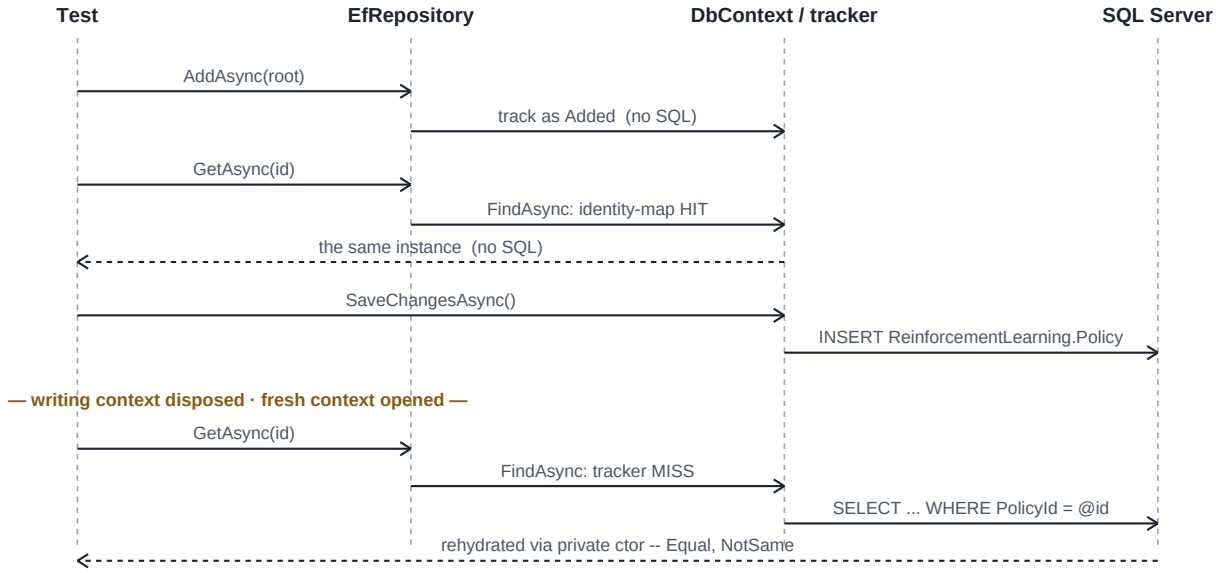

Figure D1.1 — Two lives of GetAsync: answered by the identity map while the writing context lives, answered by SQL and the private constructor once it is gone.

Substitutability, continued from Volume I

Chapter 19 of Volume I retired Assert.Same from the contract because only a fake could pass it — a portrait of one signatory dressed as a law. This chapter is the payoff of that repair: the laws that remained are ones the EF adapter can sign without special pleading, and Figure D1.2 shows the shape that makes it work. One abstract class holds the laws; each implementation supplies only itself; the tests are inherited, never copied. When Chapter 6 adds concurrency laws and Chapter 7 adds outbox laws, they will extend this same family — laws the fake cannot honestly sign will live in a Volume II extension, so the fake is never asked to lie.

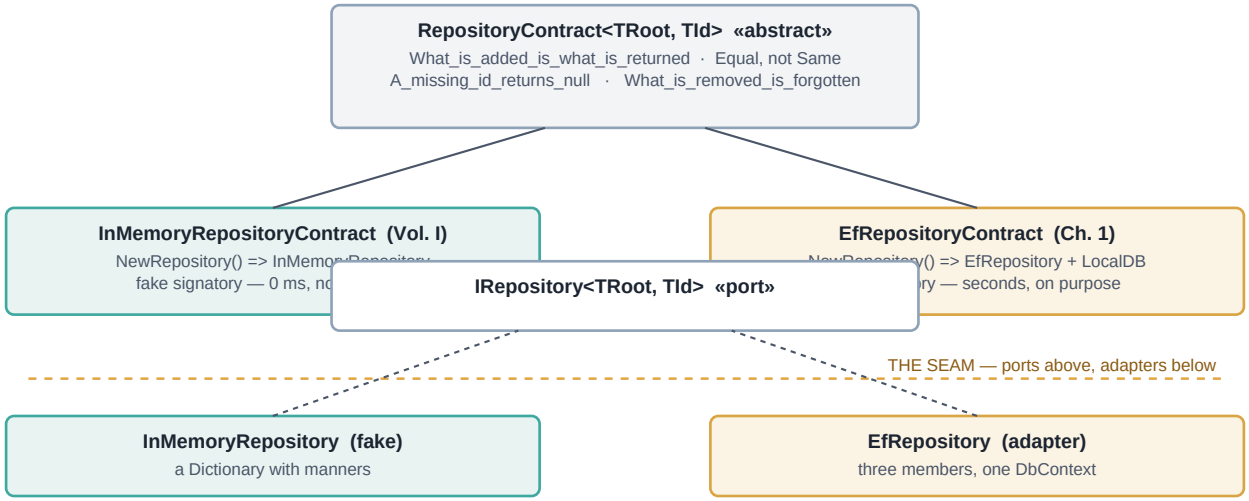

Figure D1.2 — One law, two signatories, twice: the contract family above the seam, the port and its implementations across it. Dashed arrows are «implements»; the amber dashed line is the seam of the cover, at code scale.

Rehydration: the same identity in a stranger's body

When the second context materializes the policy, it allocates a brand-new object and fills it through the private constructor and private setters. Nothing about the original instance is consulted — it may not even exist anymore. What makes the stranger “the same policy” is Chapter 4 of Volume I: equality by type and Id, on purpose, precisely so that rehydration would pass and reference comparison would fail. If you have ever wondered why that chapter refused to compare entities by their data, this test is the answer, two volumes later: data is what changed while you were away; identity is what survived it.

Why a one-word rename deserves a ledger

The temptation was real: rename the parameter, say nothing, and no reader would ever know. The ledger exists because the promise was public and auditable, so its violations must be too — otherwise “entities persist without modification” silently degrades from a checkable claim into marketing. A one-word entry also calibrates the instrument: if the ledger's first row is this small, a reader can trust that anything larger would have been printed larger. Chapter 3 will test that trust against a genuinely hard collision — the atlas's INT IDENTITY keys against the Domain's Guid-backed identifiers — with the ledger open on the table.