CHAPTER FOUR

Database First

> *Stop mapping; start building. The plant is authored by hand, in SQL, before a single line of C# exists — keys, constraints, indexes, and a view — because the schema is the thing the rest of the system will be generated from.*

Chapters 1 through 3 drew the floor plan and laid the platform's shared foundations. This chapter picks up a trowel. It builds the **Reactor Fleet** sector — the running example carried, one table at a time, all the way through reverse engineering, Code First, and the Fluent API in the chapters that follow. The method is *database first*: the SQL Server schema is authored directly, as DDL, and is treated as the single source of truth. C# does not lead here; it follows in Chapter 5, scaffolded from what this chapter writes down.

What you already know

You have written `CREATE TABLE` on a whiteboard. You drew a box for the thing, listed its columns, underlined the identifier, and drew an arrow to the table it pointed at. Database first is that whiteboard sketch performed deliberately and in the right order: parents before children, primary keys before foreign keys, constraints stated rather than assumed, and an index put exactly where a query will look. Nothing in this chapter is a new idea. The discipline is only that the schema is written first, completely, and committed — not discovered later by an ORM's conventions.

The Reactor Fleet, as it really is

The fleet is not hypothetical; it is the one the console runs. Five units across three classes: a 900 MWe pressurised-water reactor (Unit 1), a 1300 MWe PWR (Unit 2), and three 170 MWe NUWARD small modular reactor modules (SMR A, B, and C). Each class has a fixed fuel-assembly count — 157, 193, and 76 respectively — and each PWR primary circuit runs at 155 bar with a thermal efficiency near 33.8 per cent. Those numbers are not chosen for the book; they are read straight from the simulator, and the seed data below reproduces them exactly.

Authoring order: classes, then units, then reactors

A foreign key cannot point at a table that does not yet exist, so the build order is the dependency order. The fleet's root is the class, because a unit is an instance of a class. We begin there. `UnitClass` follows the lookup shape from Chapter 2 — a code, a name, audit columns — and adds the three facts that define a reactor class:

```
CREATE TABLE ReactorFleet.UnitClass (
    UnitClassId       INT           IDENTITY PRIMARY KEY,
    Code              NVARCHAR(30)  NOT NULL UNIQUE,    -- 'PWR-900'
    Name              NVARCHAR(120) NOT NULL,
    ReactorFamily     NVARCHAR(10)  NOT NULL,         -- 'PWR' | 'SMR'
    RatedCapacityMWe  INT           NOT NULL,
    FuelAssemblyCount INT           NOT NULL,
    IsSmr             BIT           NOT NULL DEFAULT 0,
    CreatedDate       DATETIME2     NOT NULL DEFAULT SYSUTCDATETIME(),
    ModifiedDate      DATETIME2     NOT NULL DEFAULT SYSUTCDATETIME(),
    CONSTRAINT CK_UnitClass_Family
        CHECK (ReactorFamily IN ('PWR', 'SMR')),
    CONSTRAINT CK_UnitClass_Capacity CHECK (RatedCapacityMWe > 0),
    CONSTRAINT CK_UnitClass_Assemblies CHECK (FuelAssemblyCount > 0)
);
```

The three `CHECK` constraints are the point of authoring by hand. An ORM convention would not know that a capacity must be positive or that `ReactorFamily` takes only two values; here those rules are written into the table, enforced by the engine, and impossible to violate from any client. (In a stricter build `ReactorFamily` would be its own lookup table rather than a check — that refinement is Chapter 10.) The seed data is the real fleet's three classes:

```
INSERT ReactorFleet.UnitClass
    (Code, Name, ReactorFamily, RatedCapacityMWe, FuelAssemblyCount, IsSmr)
VALUES ('PWR-900',   '900 MWe Pressurised Water Reactor',  'PWR',  900,157,0),
       ('PWR-1300',  '1300 MWe Pressurised Water Reactor', 'PWR', 1300,193,0),
       ('NUWARD-170','NUWARD Small Modular Reactor',       'SMR',  170, 76,1);
```

`Unit` is the child: an instance of a class, with its own code and name. Its foreign key into `UnitClass` is the first passport authored in earnest, and it earns a dedicated index because the fleet is queried by class constantly. Note what the table does *not* hold — no power level, no online flag, no rod position. Those are runtime state, and they belong to telemetry, not to the unit's identity. `Unit` records what a unit *is*, never what it is currently *doing*:

```
CREATE TABLE ReactorFleet.Unit (
    UnitId        INT           IDENTITY PRIMARY KEY,
    Code          NVARCHAR(20)  NOT NULL UNIQUE,        -- 'NX1-U1'
    Name          NVARCHAR(100) NOT NULL,               -- 'Unit 1'
    UnitClassId   INT           NOT NULL
                  REFERENCES ReactorFleet.UnitClass(UnitClassId),
    CommissionedOn DATE         NULL,
    CreatedDate   DATETIME2     NOT NULL DEFAULT SYSUTCDATETIME(),
    ModifiedDate  DATETIME2     NOT NULL DEFAULT SYSUTCDATETIME()
);

CREATE INDEX IX_Unit_UnitClassId ON ReactorFleet.Unit(UnitClassId);
```

The five units are seeded by resolving each class by its natural key, so the insert never hard-codes a generated `UnitClassId`:

```
INSERT ReactorFleet.Unit (Code, Name, UnitClassId)
SELECT v.Code, v.Name, c.UnitClassId
FROM (VALUES
    ('NX1-U1','Unit 1',      'PWR-900'),
    ('NX1-U2','Unit 2',      'PWR-1300'),
    ('NX1-SA','SMR Module A','NUWARD-170'),
    ('NX1-SB','SMR Module B','NUWARD-170'),
    ('NX1-SC','SMR Module C','NUWARD-170')
) v(Code, Name, ClassCode)
JOIN ReactorFleet.UnitClass c ON c.Code = v.ClassCode;
```

`Reactor` completes the spine: exactly one reactor per unit, holding the design constants the simulator uses — primary pressure, thermal rating, and the number of coolant loops. The one-reactor-per-unit rule is not a comment; it is a `UNIQUE` constraint on the foreign key, so the database itself refuses a second reactor for a unit:

```
CREATE TABLE ReactorFleet.Reactor (
    ReactorId        INT     IDENTITY PRIMARY KEY,
    UnitId           INT     NOT NULL UNIQUE          -- one reactor per unit
                     REFERENCES ReactorFleet.Unit(UnitId),
    DesignPressureBar DECIMAL(5,1) NOT NULL,    -- 155.0 for the PWRs
    ThermalRatingMWt INT     NOT NULL,             -- ≈ MWe / 0.338
    CoolantLoopCount TINYINT NOT NULL,
    CreatedDate      DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME(),
    CONSTRAINT CK_Reactor_Pressure CHECK (DesignPressureBar > 0),
    CONSTRAINT CK_Reactor_Loops CHECK (CoolantLoopCount BETWEEN 1 AND 4)
);
```

The reactor’s contents

A reactor contains a core of fuel assemblies and a set of control-rod banks; both are substantive tables, not lookups, because each row is a real object with its own state. `ControlRodBank` holds the five banks the Control Rods screen drives — two shutdown banks and three control banks — each travelling 228 steps:

```
CREATE TABLE ReactorFleet.ControlRodBank (
    ControlRodBankId INT          IDENTITY PRIMARY KEY,
    ReactorId        INT          NOT NULL
                     REFERENCES ReactorFleet.Reactor(ReactorId),
    Code             NVARCHAR(20) NOT NULL,        -- 'Control D'
    BankType         NVARCHAR(12) NOT NULL,        -- 'Shutdown' | 'Control'
    StepsTotal       SMALLINT     NOT NULL DEFAULT 228,
    CONSTRAINT CK_RodBank_Type
        CHECK (BankType IN ('Shutdown', 'Control')),
    CONSTRAINT UQ_RodBank_Reactor_Code UNIQUE (ReactorId, Code)
);
```

The composite `UNIQUE (ReactorId, Code)` says a bank code is unique *within* a reactor, not across the whole fleet — every reactor has its own “Control D.” That is a deliberate hand-authored decision an ORM convention would not make for you. `FuelAssembly` follows the same pattern, keyed uniquely by core position within its reactor; its rows are generated — 157 for a PWR-900, 193 for a PWR-1300, 76 for an SMR module — by the seed routine of Appendix H rather than hand-listed here. The remaining sector tables are summarised below; each is grounded in a quantity the simulator already computes.

REACTOR FLEET — REMAINING TABLES, AND WHAT GROUNDS THEM

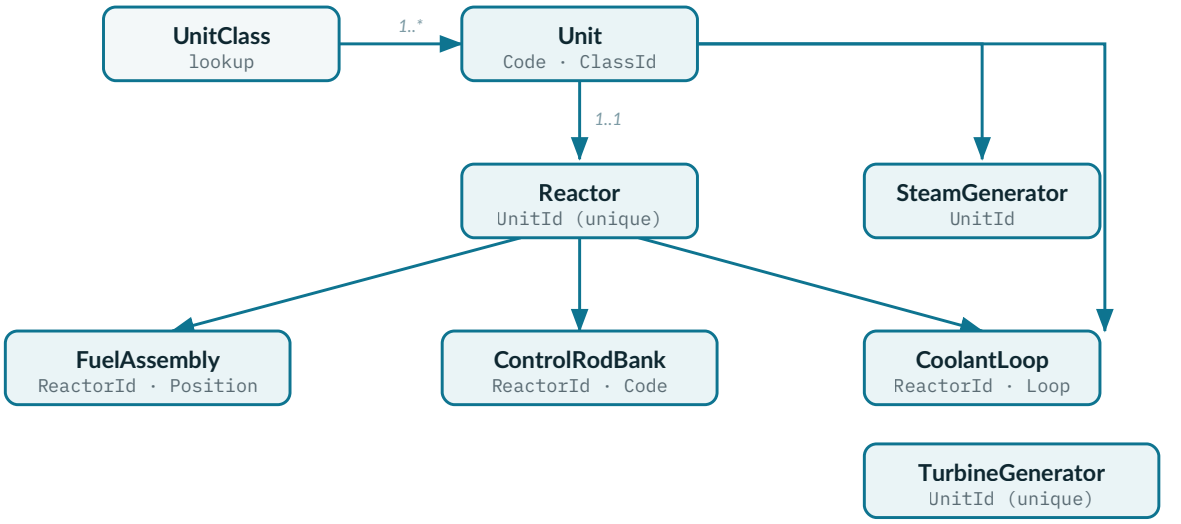
FuelAssembly	`ReactorId, PositionCode, EnrichmentPercent, OutletTemp-pC` core map: 157 / 193 / 76 cells, coloured by outlet temp
CoolantLoop	`ReactorId, LoopCode, HotLegTempC, ColdLegTempC` simulator: hot leg 323 °C, cold leg 289 °C
SteamGenerator	`UnitId, Code, NominalLevelPercent` simulator: SG level 62 %
TurbineGenerator	`UnitId, RatedSpeedRpm, TerminalVoltageKv, PowerFactor` simulator: 3000 rpm, 21.0 kV, pf 0.98, 50 Hz

A view the console already draws

The Plant Fleet screen shows each unit with its class, capacity, and assembly count. That is a join, and authoring it once as a view means every client reads the fleet the same way:

```
CREATE VIEW ReactorFleet.vw_FleetSummary AS
SELECT u.Code, u.Name,
       c.Code              AS Class,
       c.RatedCapacityMWe  AS CapacityMWe,
       c.FuelAssemblyCount AS Assemblies,
       c.IsSmr
FROM   ReactorFleet.Unit      u
JOIN   ReactorFleet.UnitClass c ON c.UnitClassId = u.UnitClassId;
```

FIGURE 4.1 · REACTOR FLEET SECTOR — KEYS AND RELATIONSHIPS



UnitClass classifies many Units; each Unit has exactly one Reactor (a unique foreign key) and exactly one TurbineGenerator, and many SteamGenerators; each Reactor contains many FuelAssemblies, ControlRodBanks, and CoolantLoops. Every arrow is a foreign key authored by hand, parent before child.

SEE IT IN NEXUS-1

Three screens show exactly what this sector stores. Open **Plant Fleet**: the five units, two online by default, each labelled with its class and capacity — the rows of `Unit` joined to `UnitClass`, which is precisely what `vw_FleetSummary` returns. Switch the selected unit and open **Reactor Core**: the header reads *PWR · 157 Assemblies* for Unit 1, and the core map draws one cell per fuel assembly, coloured by outlet temperature — one cell per `FuelAssembly` row. Open **Control Rods** and you see the five banks, Shutdown A and B fully withdrawn and the control banks regulating, each measured in steps out of 228 — the rows of `ControlRodBank`. The console shows the values; the schema is where they would persist.

TRY IT

In the console, open **Plant Fleet** and toggle SMR Module B online; watch installed capacity rise by 170 MWe. That capacity is the sum of a column you are about to own. Then, in the database you build, run the seed above and confirm the fleet matches the screen:

```
-- The fleet, the way the Plant Fleet screen reads it.
SELECT * FROM ReactorFleet.vw_FleetSummary ORDER BY CapacityMWe DESC;

-- Total installed capacity across the fleet.
SELECT SUM(CapacityMWe) AS InstalledMWe FROM ReactorFleet.vw_FleetSummary;
```

The sum is 2510 MWe with the whole fleet present — 900 plus 1300 plus three times 170 — the same arithmetic the Plant Fleet screen performs when every module is online.

HONEST BOUNDARY

The `Unit` table holds identity, not condition. Whether a unit is online, what power it is at, where its rods sit — none of that lives here, because it changes by the second and belongs to telemetry and the digital twin (Chapters 6, 8, and 16). Confusing the two is the most common data-modelling error in plant software: a current-power column on the `Unit` row turns a record of what a unit *is* into a stale guess about what it is *doing*. Identity is slow and authoritative; state is fast and lives elsewhere.

Two grounding caveats. The capacities, assembly counts, pressures, and rod-bank structure are read from the simulator and are real in that sense; the commissioning dates and per-assembly enrichments are illustrative, because the console does not model them. And database first means this SQL is the source of truth *for now* — Chapter 7 rebuilds the identical model from C#, and Chapter 13 reconciles the two builds, naming every physical difference out loud.

Chapter 5 hands this hand-authored schema to `Scaffold-DbContext` and reads the result honestly — including the places where the scaffolder infers something the DDL did not say, and gets it subtly wrong.