

Controlling Database Creation and Schema Changes with Migrations



Julie Lerman

EF Core Expert and Software Coach

@julielerman | thedatafarm.com

Module Overview



- Overview of EF Core Migrations API**
- Setting up your project and Visual Studio for migrations**
- Create and inspect a migration file**
- Using EF Core Migrations to create a database or database scripts**
- Reverse engineer an existing database into classes and DbContext**



Understanding EF Core Migrations



EF Core Needs to Comprehend the DB Schema

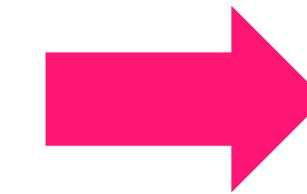
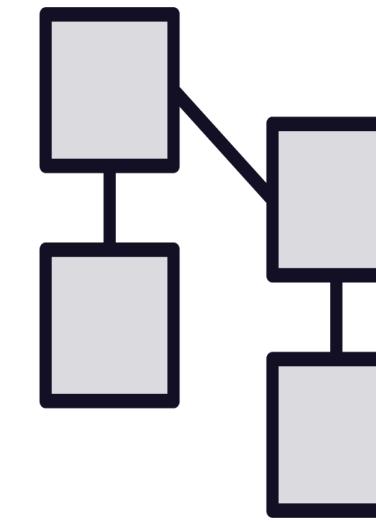
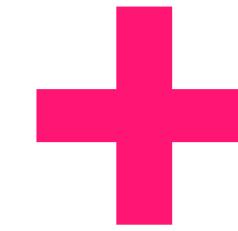
**Build SQL from
your LINQ queries**

**Materialize query
results into objects**

**Build SQL to save
data into database**



Mapping Your Data Model to the Database



DbContext

**Conventional and
custom mappings**

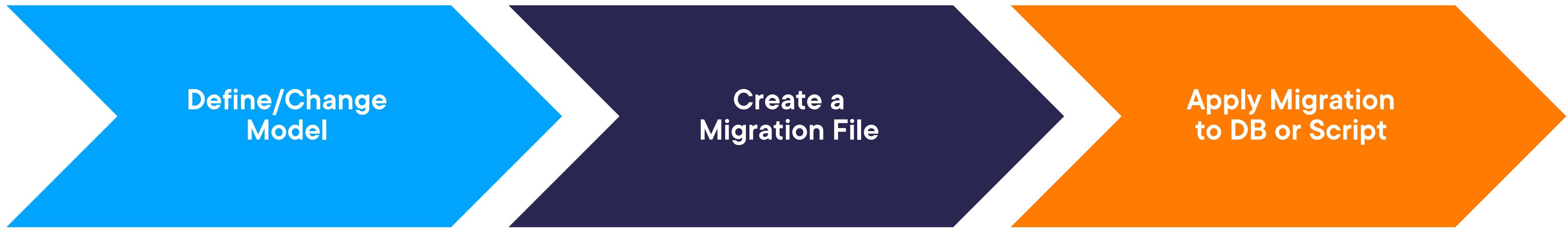
Database schema



**Mapping knowledge can
also be used to evolve the
database schema.**



EF Core Basic Migrations Workflow



Source-control friendly



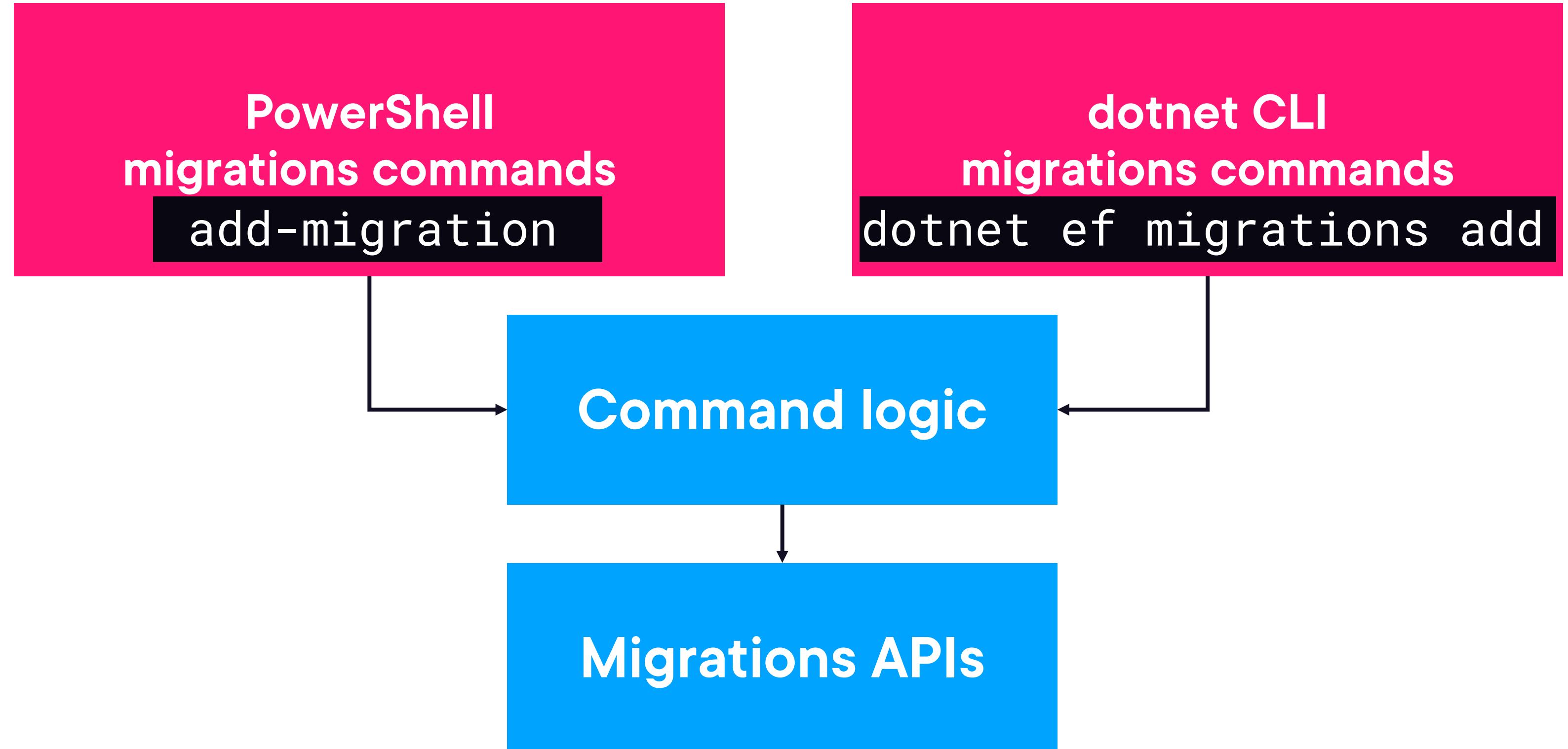
Getting and Understanding the Design-Time Migrations Tools



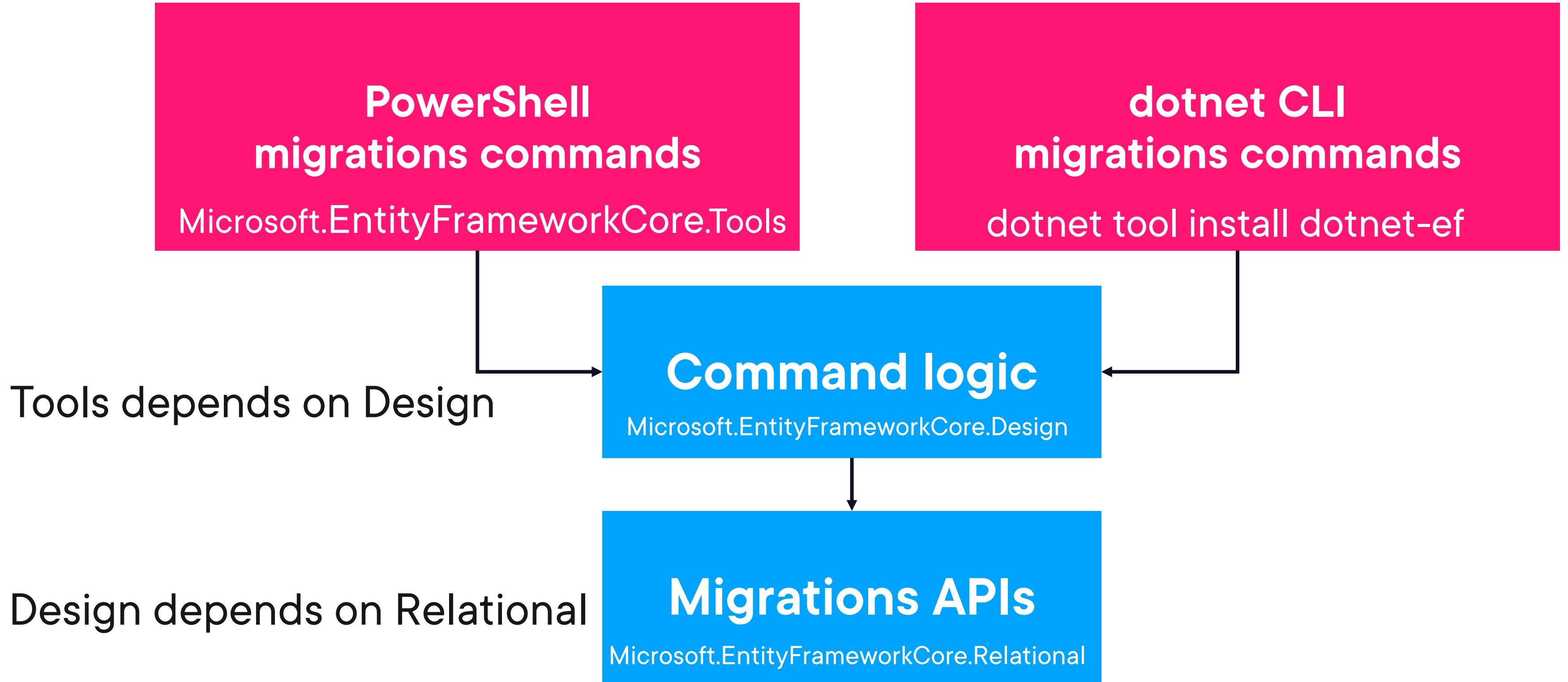
**Creating and executing
migrations happens at
design time**



Migration Commands Lead to Migrations APIs



Migration Commands Lead to Migrations APIs



Bottom Line

- Add Tools package to project
- Design comes for “free”

Visual Studio (Windows)

- Install the tools on your system
- Design package in your project

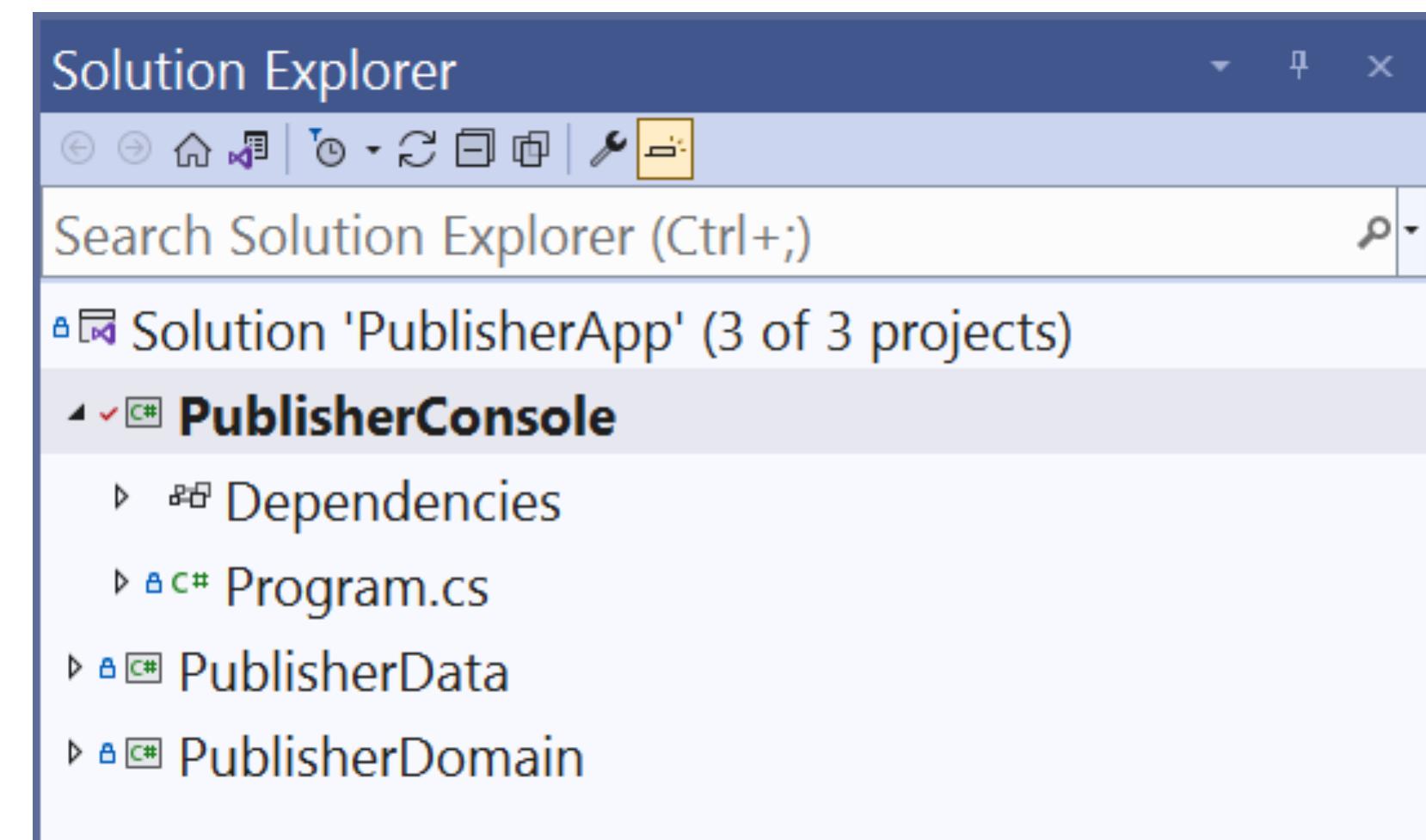
Command Line



Check IDE changes

Which Project Gets the Tools?

The executable project



Getting the Package Manager Console Ready to Use Migrations



Using Migrations in Visual Studio When EF Core Is in a Class Library Project



Install `Microsoft.EntityFrameworkCore.Tools` package into executable project (e.g., console)



Ensure the executable project is the startup project



Set Package Manager Console (PMC) “default project” to class library with EF Core (e.g., data)



Run EF Core Migration PowerShell commands in PMC



Using Migrations in dotnet CLI

Add EF Core tools
to system*

Add EF
Core Design**
package to
executable project

Use
“dotnet ef”
commands at
command line

*Command to install: `dotnet tool install -global dotnet-ef`

**`Microsoft.EntityFrameworkCore.Design`



Adding Your First Migration



Add-Migration Tasks for Initial Schema

Read DbContext and determine data model

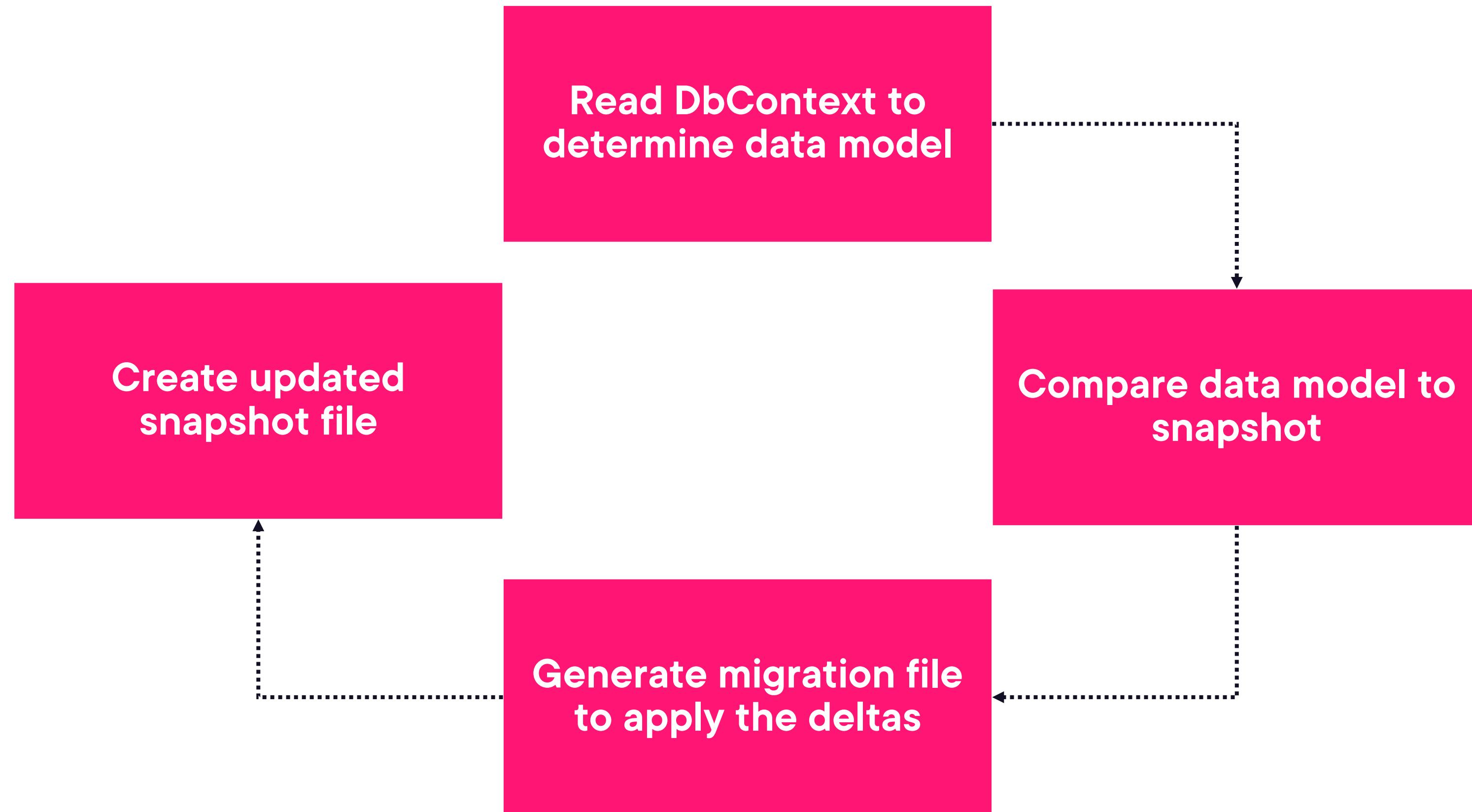
Create a migration file describing how to construct the database schema



Inspecting Your First Migration



Add-Migration Tasks for Model Changes



**For EF 6 users:
EF Core migrations is orders
of magnitude easier to use
in source control**



Some EF Core Mapping Conventions (Defaults)



DbSet name is the table name



Class property name is the column name



Strings are determined by provider
e.g., SQL Server: nvarchar(max)



Decimals are determined by provider
e.g., SQL Server: decimal(18,2)



“Id” or “[type]Id” are primary keys



Maps Strings to Non-Nullable Columns by Default

```
FirstName = table.Column<string>
(type: "nvarchar(max)", nullable: false),
LastName = table.Column<string>
(type: "nvarchar(max)", nullable: false)
```

SQL Server

```
FirstName = table.Column<string>
(type: "TEXT", nullable: false),
LastName = table.Column<string>
(type: "TEXT", nullable: false)
```

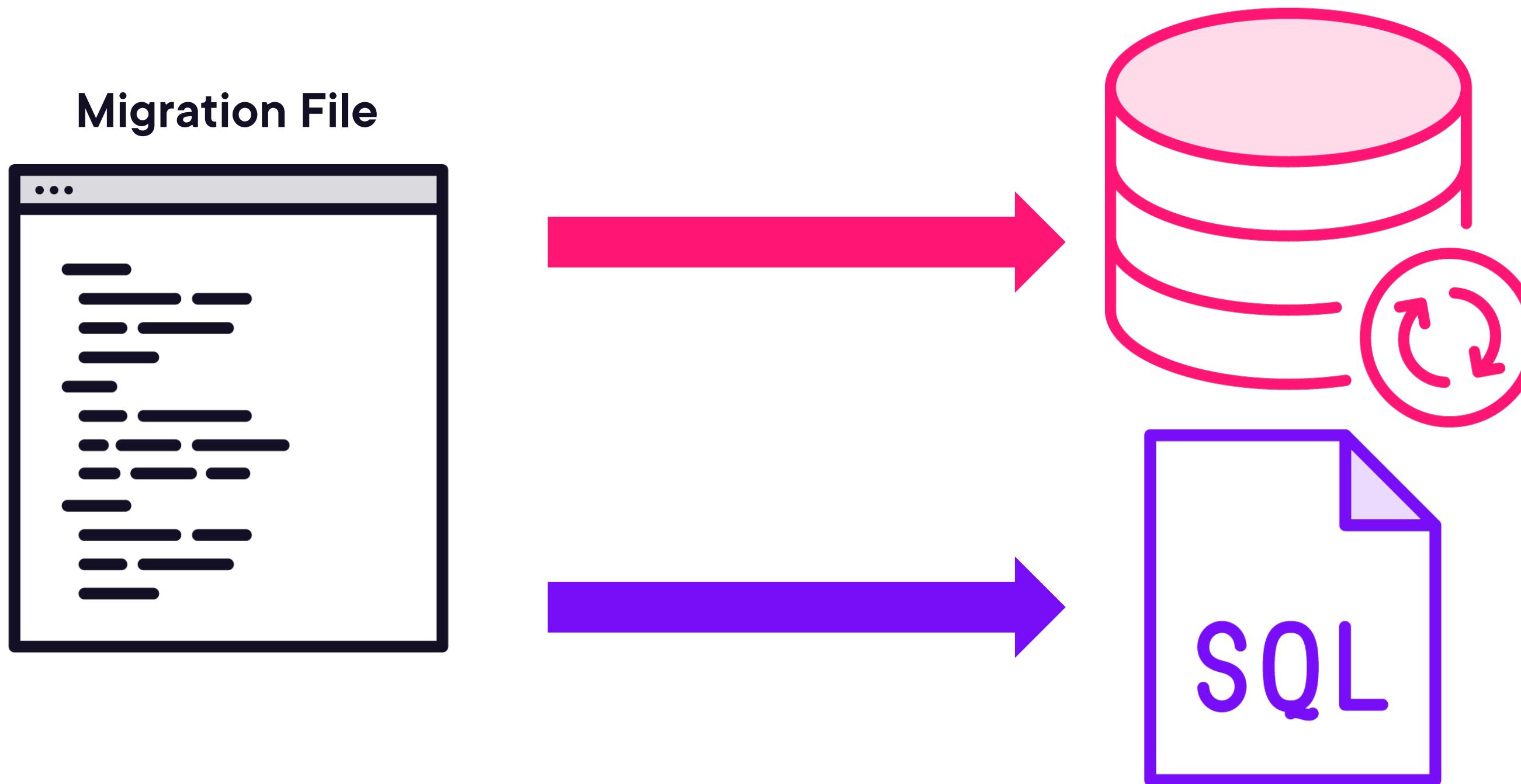
SQLite



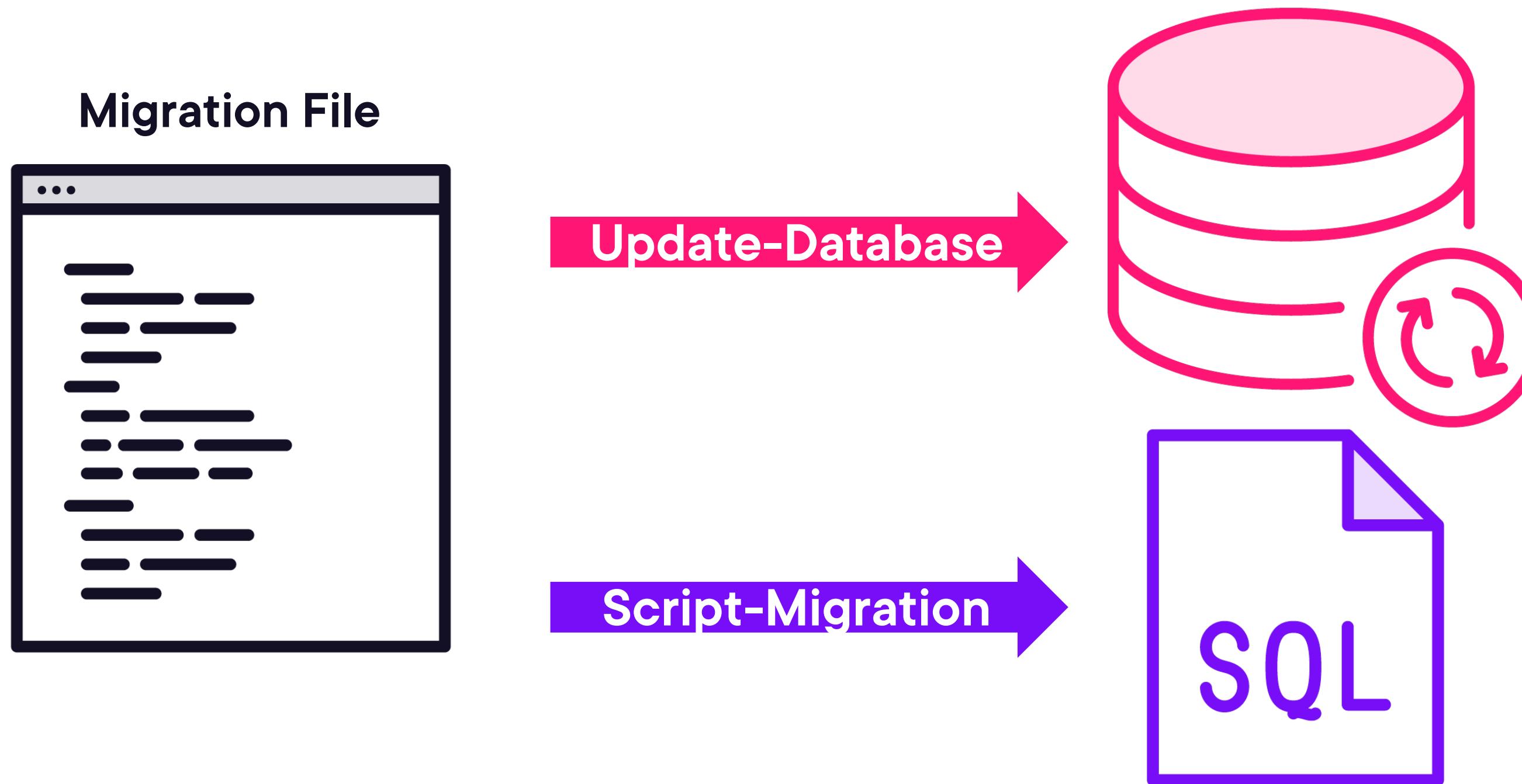
Using Migrations to Script or Directly Create the Database



Applying Migrations



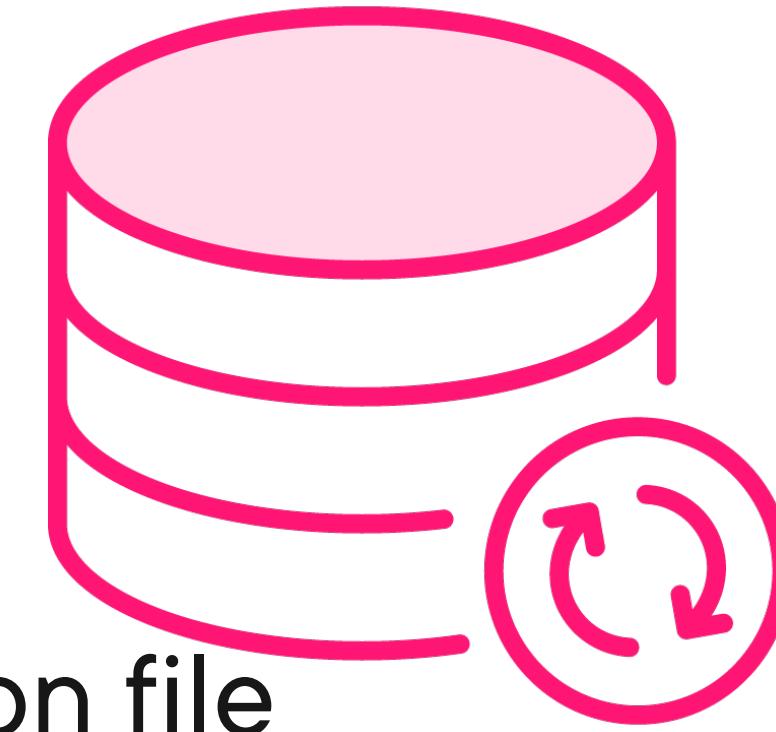
Applying Migrations



Applying Migrations Directly to the Database



Update-Database



- Reads migration file
- Generates SQL in memory
- Creates the database if needed
- Runs SQL on the database

CLI: `dotnet ef database update`



Applying Migrations into a SQL Script

Migration File



- Reads migration file
- Generates SQL
- Default: Displays SQL in editor
- Use parameters to target file name etc.

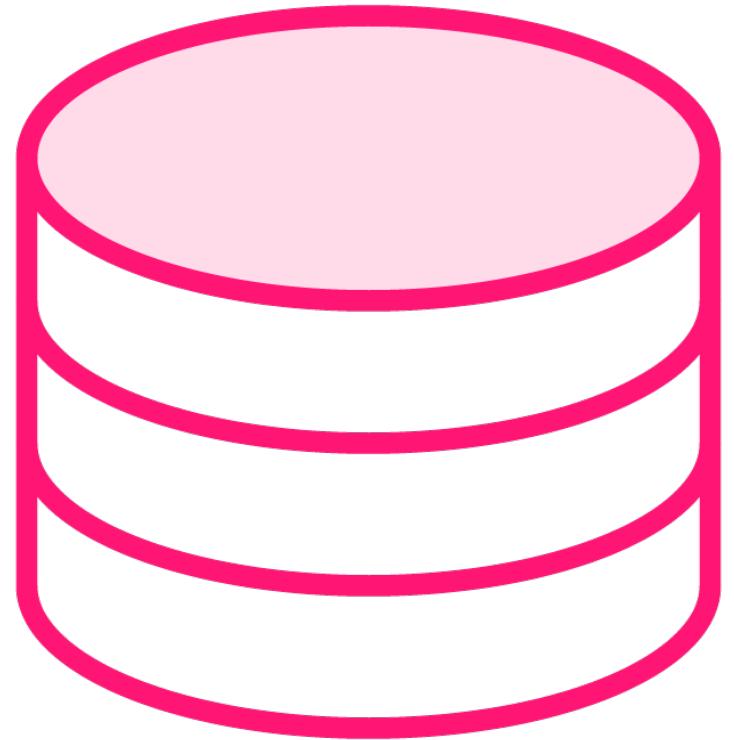
Script-Migration →



CLI: `dotnet ef migrations script`



Migrations Recommendation



Development database
update-database



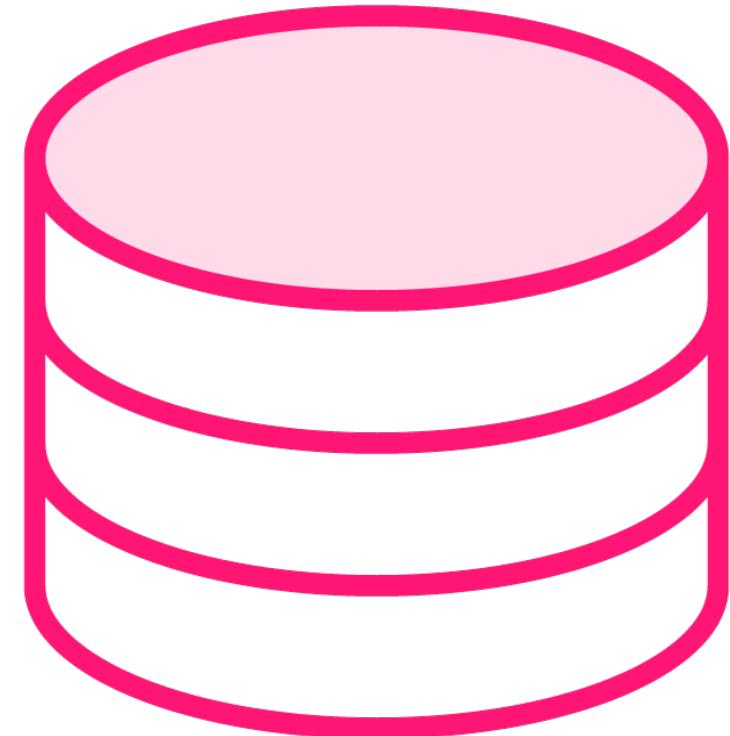
Production database
script-migration



**We will create a fresh
database in this module.**



What If Database Does Not Exist?



`update-database`

**API's internal code will create the database
before executing migration code**



`script-migration`

**You must create the database before
running the script**



Seeding a Database via Migrations



```
modelBuilder.Entity<EntityType>().HasData(parameters)
```

```
modelBuilder.Entity<Author>().HasData(new Author {Id=1, FirstName="Julie", ..});
```

Specify Seed Data with ModelBuilder HasData Method

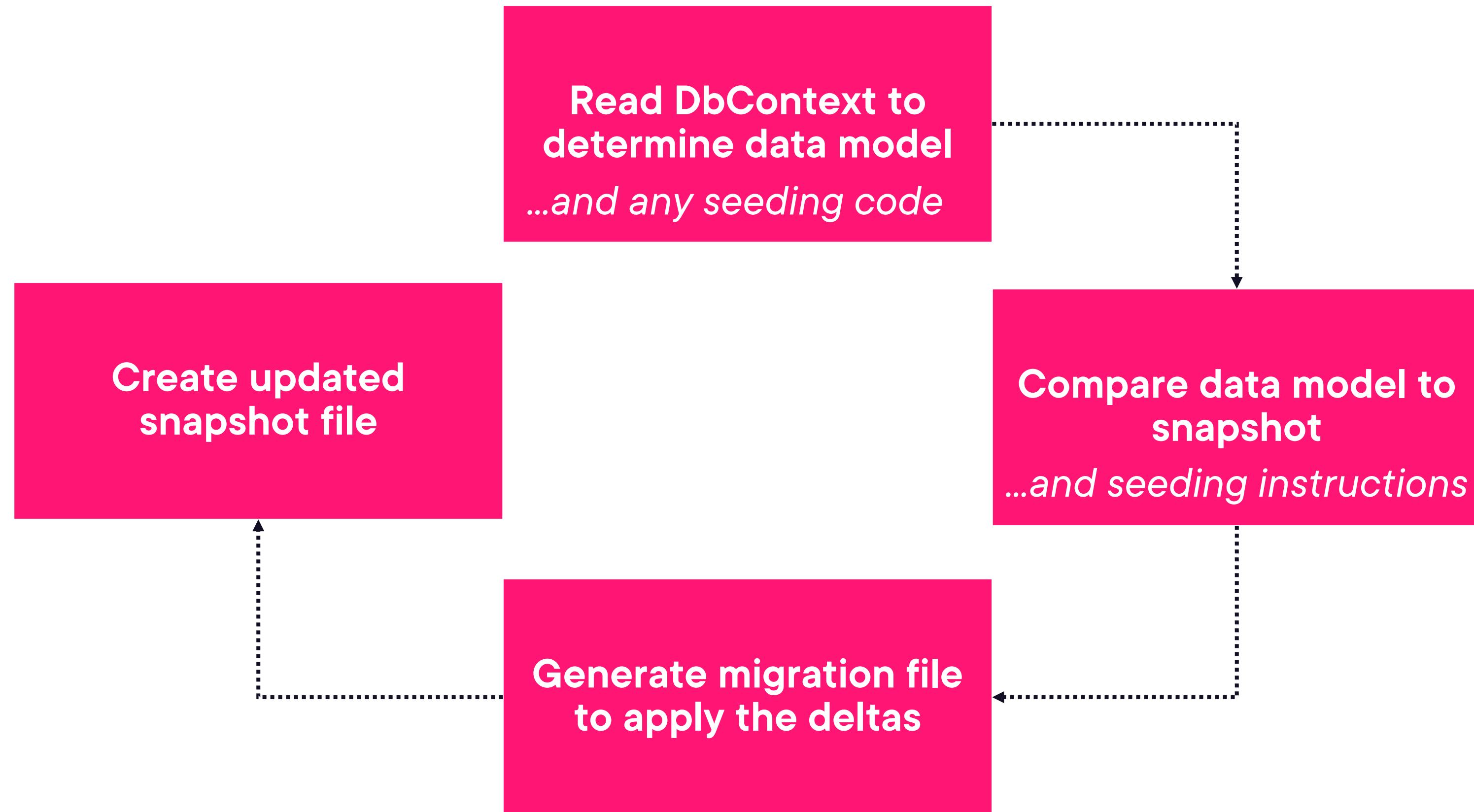
Provide all non-nullable parameters including keys and foreign keys

HasData will get translated into migrations

Inserts will get interpreted into SQL

Data will get inserted when migrations are executed

Add-Migration Tasks for Model Changes



Seeding with HasData will not cover all use cases for seeding



Use Cases for Seeding with HasData

Mostly static seed data

Sole means of seeding

**No dependency on anything
else in the database**

**Provide test data with a
consistent starting point**



**HasData will also be
recognized and applied by
EnsureCreated().**



Scripting Multiple Migrations



**Scripting migrations
requires more control, so it
works differently than
update-database.**



Some Scripting Options

`script-migration`

Default: Scripts every migration

`script-migration
-idempotent`

Scripts all migrations but checks for each object first e.g., table already exists

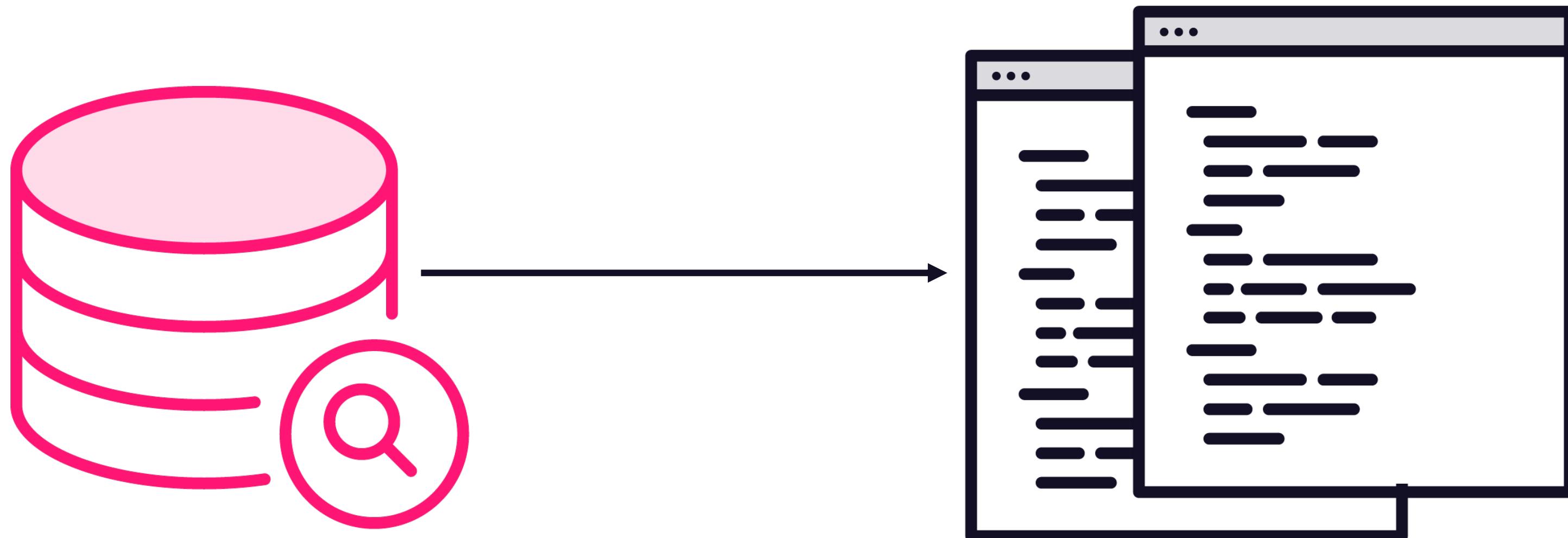
`script-migration
FROM TO`

FROM: Specifies last migration run, so start at the next one
TO: final one to apply

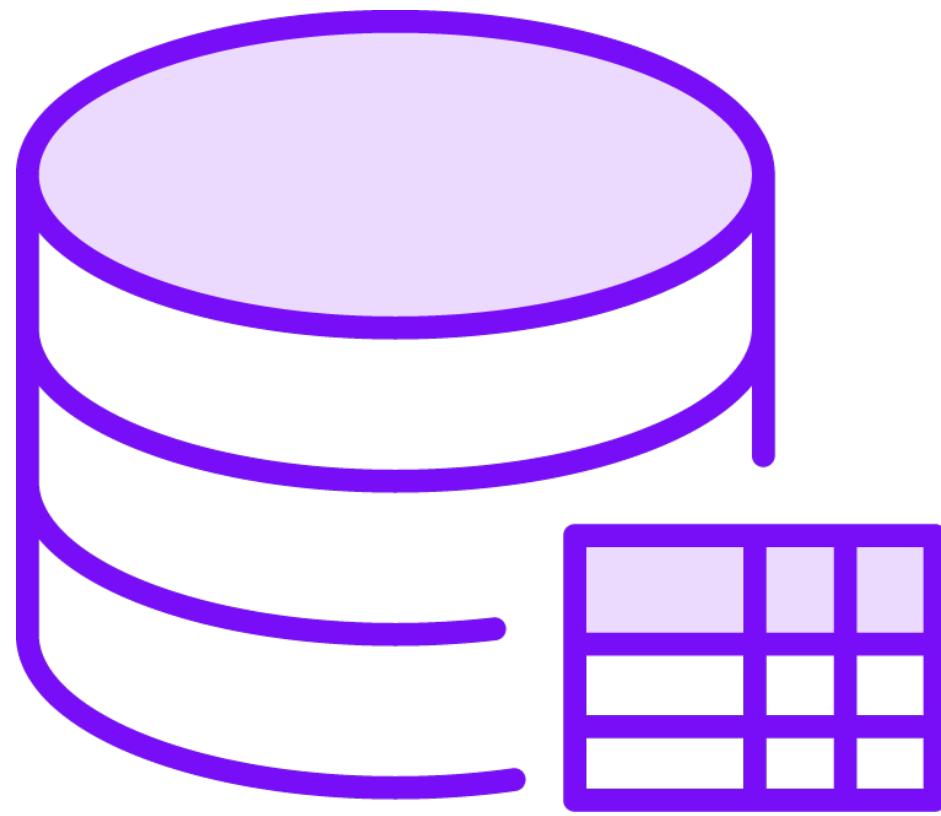
Reverse Engineering an Existing Database



Scaffolding Builds DbContext and Entity Classes



Creates Entities from Discovered Database Objects



Tables



Views



Reverse Engineer with the Scaffold Command

PowerShell
Scaffold-DbContext

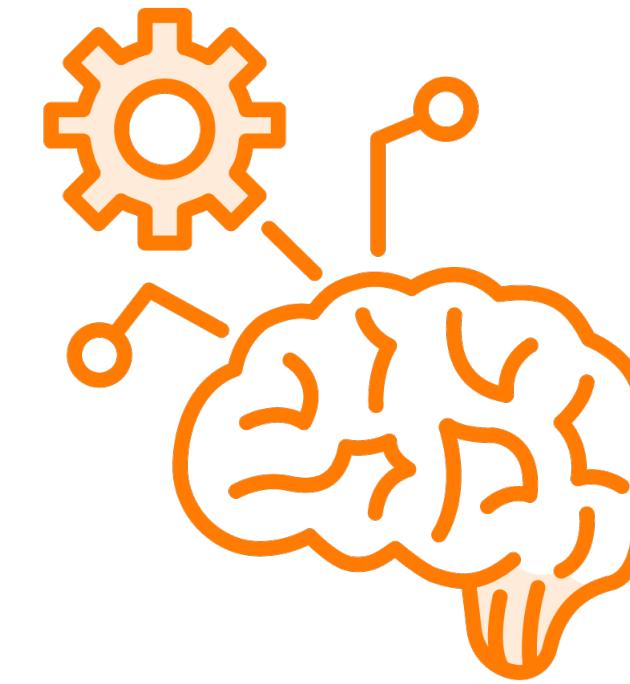
EF Core CLI
dotnet ef dbcontext
scaffold



Scaffolding Limitations



**Updating model when
database changes is not
currently supported**



**Transition to migrations is not
pretty. Look for helpful link in
resources**



The Many Parameters of scaffold-dbcontext

-Connection <String>

The connection string to the database.

-Provider <String>

The provider to use. (E.g. Microsoft.EntityFrameworkCore.SqlServer)

-OutputDir <String>

The directory to put files in. Paths are relative to the project directory.

-ContextDir <String>

The directory to put the DbContext file in. Paths are relative to project directory.

-Context <String>

The name of the DbContext. Defaults to the database name.

-Schemas <String[]>

The schemas of tables to generate entity types for.

-Tables <String[]>

The tables to generate entity types for.

-DataAnnotations [<SwitchParameter>]

Use attributes to configure the model (where possible). If omitted only the fluent API is

-UseDatabaseNames [<SwitchParameter>]

Use table and column names directly from the database.

-Force [<SwitchParameter>]

Overwrite existing files.

-NoOnConfiguring [<SwitchParameter>]

Don't generate DbContext.OnConfiguring.

-Project <String>

The project to use.

-StartupProject <String>

The startup project to use. Defaults to the solution's start project.

-Namespace <String>

The namespace to use. Matches the directory by default.

-ContextNamespace <String>

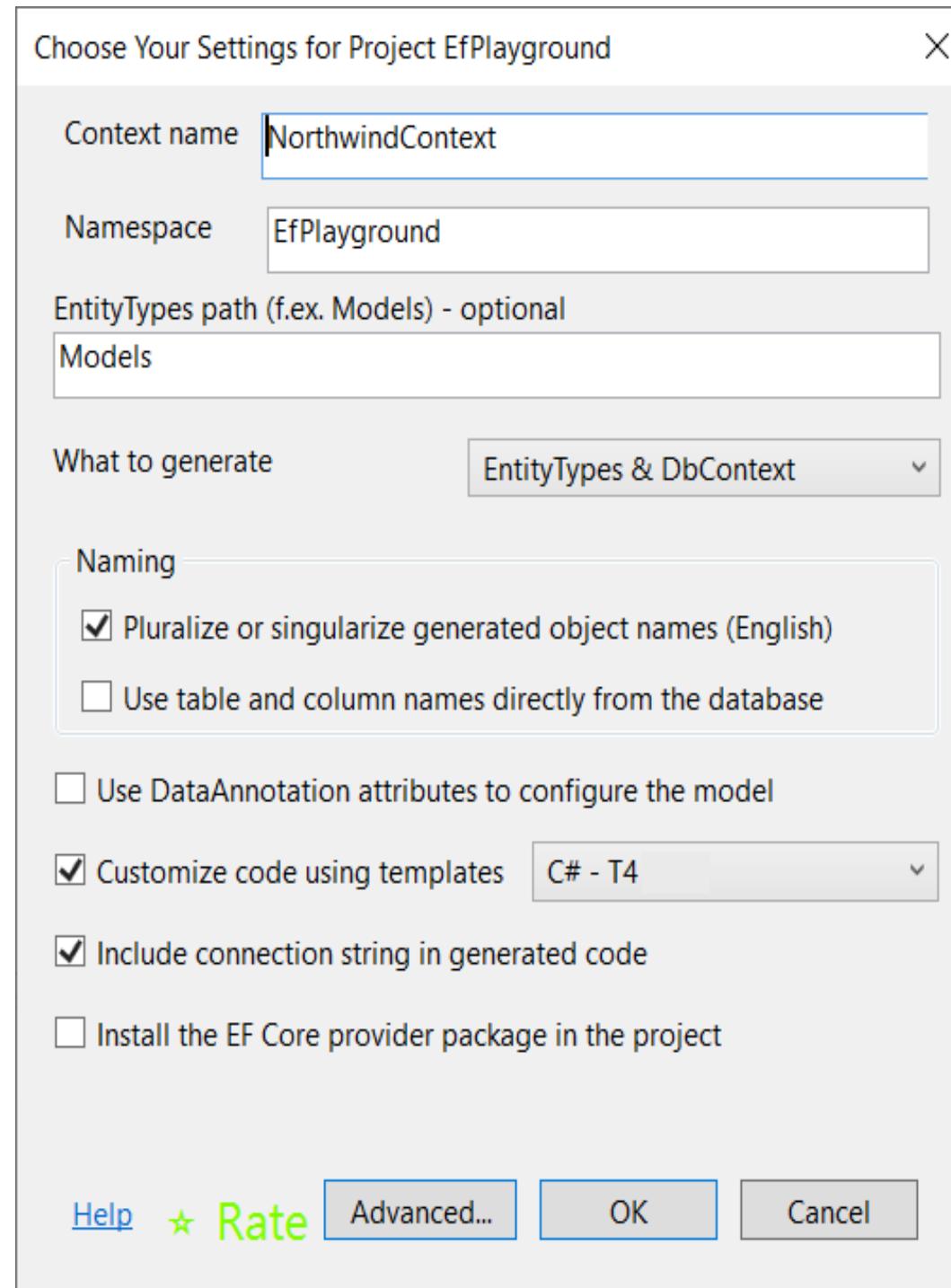
The namespace of the DbContext class. Matches the directory by default.

-NoPluralize [<SwitchParameter>]

Don't use the pluralizer.



EF Core Power Tools for Visual Reverse Engineering



VS extension: ErikEJ.EFCorePowerTools

Free

**Open-source
(github.com/ErikEJ/EFCorePowerTools)**

Built and maintained by Erik Ejlskov Jensen

Many more features besides reverse engineer



How EF Core Determines Mappings to DB

Conventions

Default assumptions

```
property name=column name
```

Override with Fluent Mappings

Apply in DbContext using Fluent API

```
modelBuilder.Entity<Book>()  
.Property(b => b.Title)  
.HasColumnName("MainTitle");
```

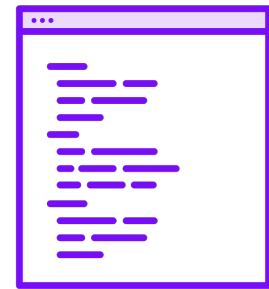
Override with Data Annotations

Apply in entity

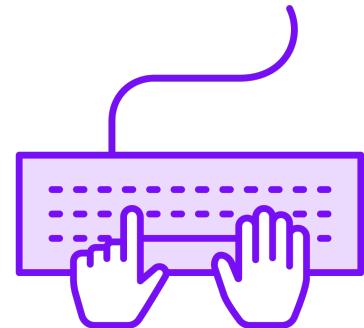
```
[Column("MainTitle")]  
public string Title{get;set;}
```



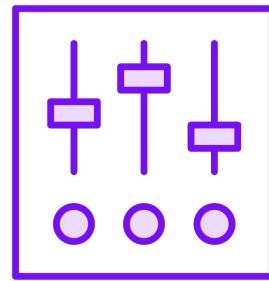
Scaffolding



Scaffolded classes are a “stake in the ground” to get you started



You can refactor to incorporate your business logic



There is a lot of flexibility and even other tools



Review



Workflow of how EF Core determines database schema

Where Migrations API and tools fit in

PowerShell or CLI commands for creating and executing migrations

Created and explored a migrations file

Used migrations commands to generate script or create a new database directly

Reverse engineer existing database into classes and DbContext



Up Next:

Defining One-to-Many Relationships



Resources

Entity Framework Core on GitHub: github.com/dotnet/efcore

EF Core Tools Documentation: docs.microsoft.com/ef/core/cli/

EF Core Power Tools Extension (model visualizer, scaffold and more):
<https://github.com/ErikEJ/EFCorePowerTools>



Resources Cont.

EF Core migrations with existing database schema: cmatskas.com/ef-core-migrations-with-existing-database-schema-and-data

Scott Hanselman “Magic Unicorn” blog post:
hanselman.com/blog/entity-framework-magic-unicorn-and-much-more-is-now-open-source-with-take-backs

