SAVING DATA WITH ENTITY FRAMEWORK CORE

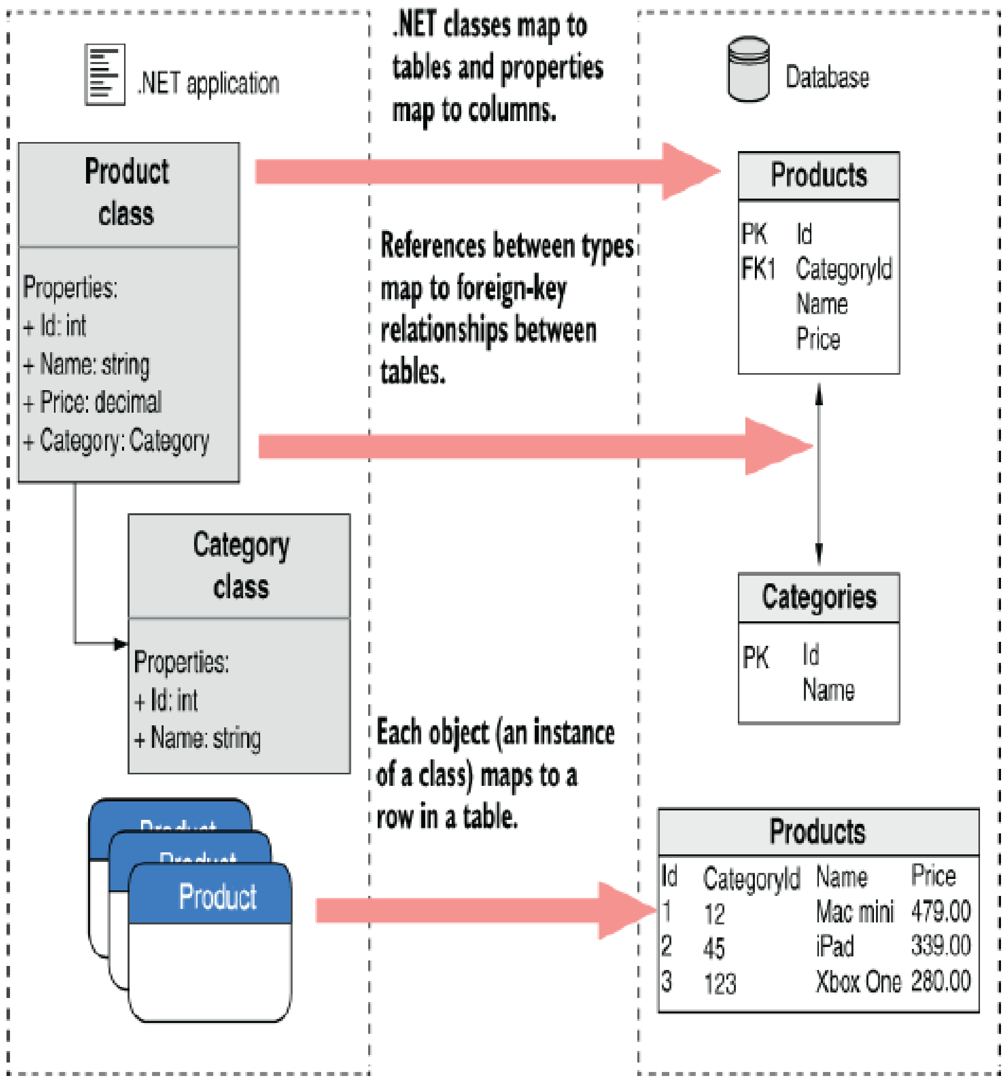
Kamal Beydoun
Lebanese University – Faculty of Sciences I
Kamal.beydoun@ul.edu.lb

INTRODUCING ENTITY FRAMEWORK CORE

- Unfortunately, **interacting with databases** from app code is often a messy affair, and you can take many approaches.
- A task as simple as reading data from a database, for example, requires **handling network connections**, **writing SQL statements**, and **handling variable result data**.

WHAT IS EF CORE?

- EF Core is a library that provides an **object-oriented way** to access databases.
- It acts as an object-relational mapper (ORM), **communicating** with the database for you and mapping database responses to .NET classes and objects.
- With an object-relational mapper (ORM), you can manipulate a database with object-oriented concepts such as classes and objects by mapping them to database concepts such as tables and columns.



Id	CategoryId	Name	Price
1	12	Mac mini	479.00
2	45	iPad	339.00
3	123	Xbox One	280.00

WHAT IS EF CORE?

- Most commonly used family is **relational databases**, accessed via Structured Query Language (SQL).
  - This is the bread and butter of EF Core; it can map **Microsoft SQL Server**, **SQLite**, **MySQL**, **Postgres**, and many other relational databases.
  - It even has a cool **in-memory** feature you can use when testing to create a **temporary** database.
- EF Core uses a **provider model**, so support for other relational databases can be **plugged in** later as they become available.
- As of .NET Core 3.0, EF Core also works with **nonrelational, NoSQL, or document databases like Cosmos DB**, too.

WHY USE AN OBJECT-RELATIONAL MAPPER?

- One of the biggest **advantages** of an ORM is **the** speed with which it allows you to develop an application.
- You can stay in the **familiar** territory of **object-oriented .NET**, often **without** needing to manipulate a database directly or write **custom SQL**.
- Suppose that you have an e-commerce site, and you want to **load the details of a product from the database**. Using low-level database access code, you'd have to :
  - open a connection to the database;
  - write the necessary SQL with the correct table and column names;
  - read the data over the connection;
  - create a plain old CLR object (POCO) to hold the data;
  - set the properties on the object, converting the data to the correct format manually as you go.

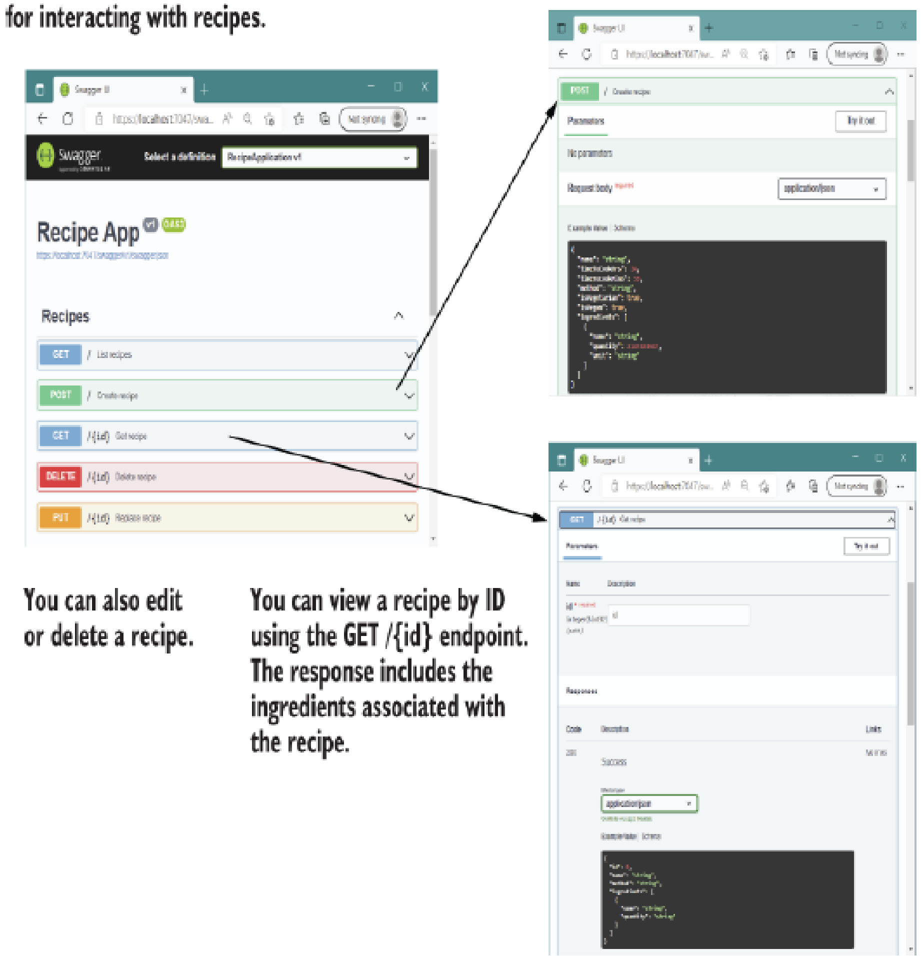
WHY USE AN OBJECT-RELATIONAL MAPPER?

- An ORM such as EF Core **takes care of most of this work for you**.
  - It handles the connection to the database, generates the SQL, and maps data back to your POCO objects.
  - All you need to provide is a **LINQ query** describing the data you want to retrieve.
- ORMs serve as **high-level abstractions over databases**, so they can significantly reduce the amount of plumbing code you need to write to interact with a database.
- ORMs like EF Core **keep track** of which **properties** have changed on any objects they retrieve from the database, which lets you load an object from the database by **mapping it from a database table**, **modify** it in .NET code, and then ask the ORM to **update** the associated record in the database.

MAPPING A DATABASE TO YOUR APPLICATION CODE

The recipe app provides an API for interacting with recipes.

Use the POST / endpoint to create a recipe.



You can also edit or delete a recipe.

You can view a recipe by ID using the GET /{id} endpoint. The response includes the ingredients associated with the recipe.

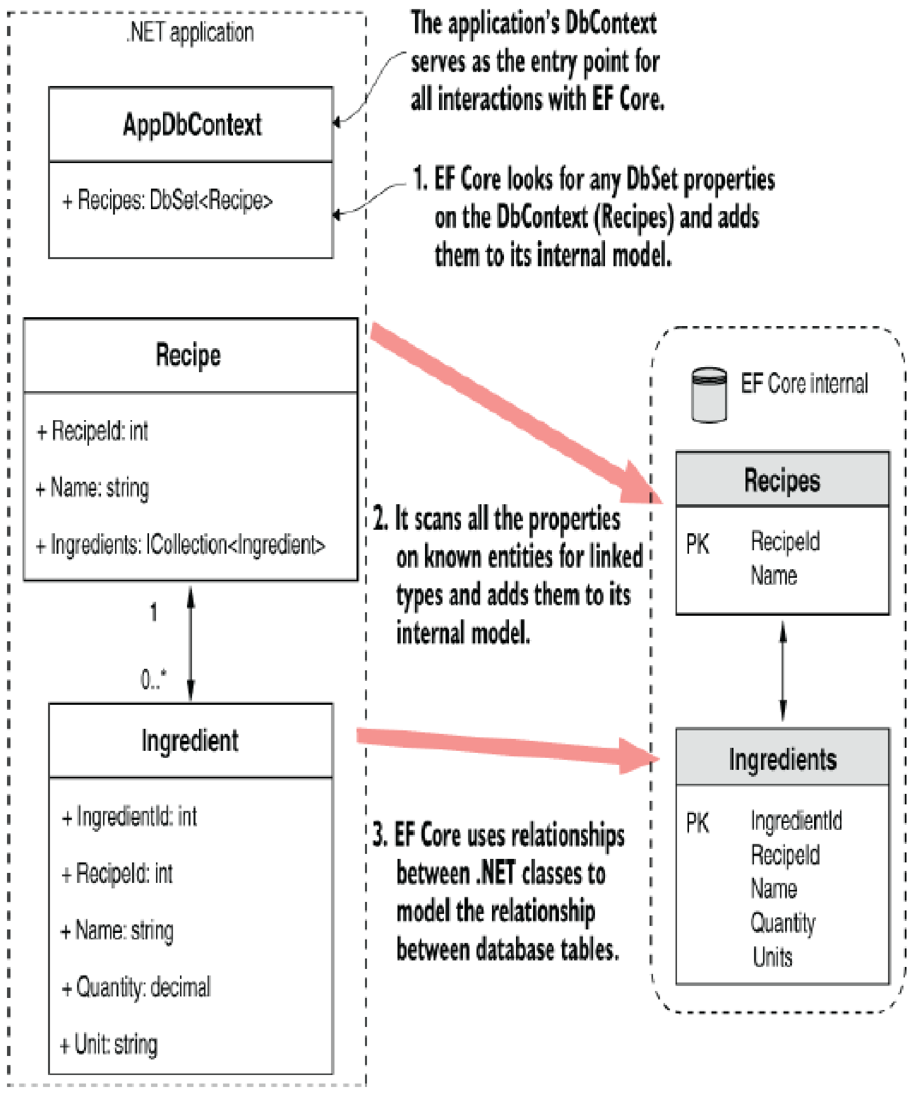
An **entity** is a **.NET class** that's mapped by EF Core to the database. These are classes you define, typically as POCO classes, that can be saved and loaded by mapping to database tables using EF Core.

MAPPING A DATABASE TO YOUR APPLICATION CODE

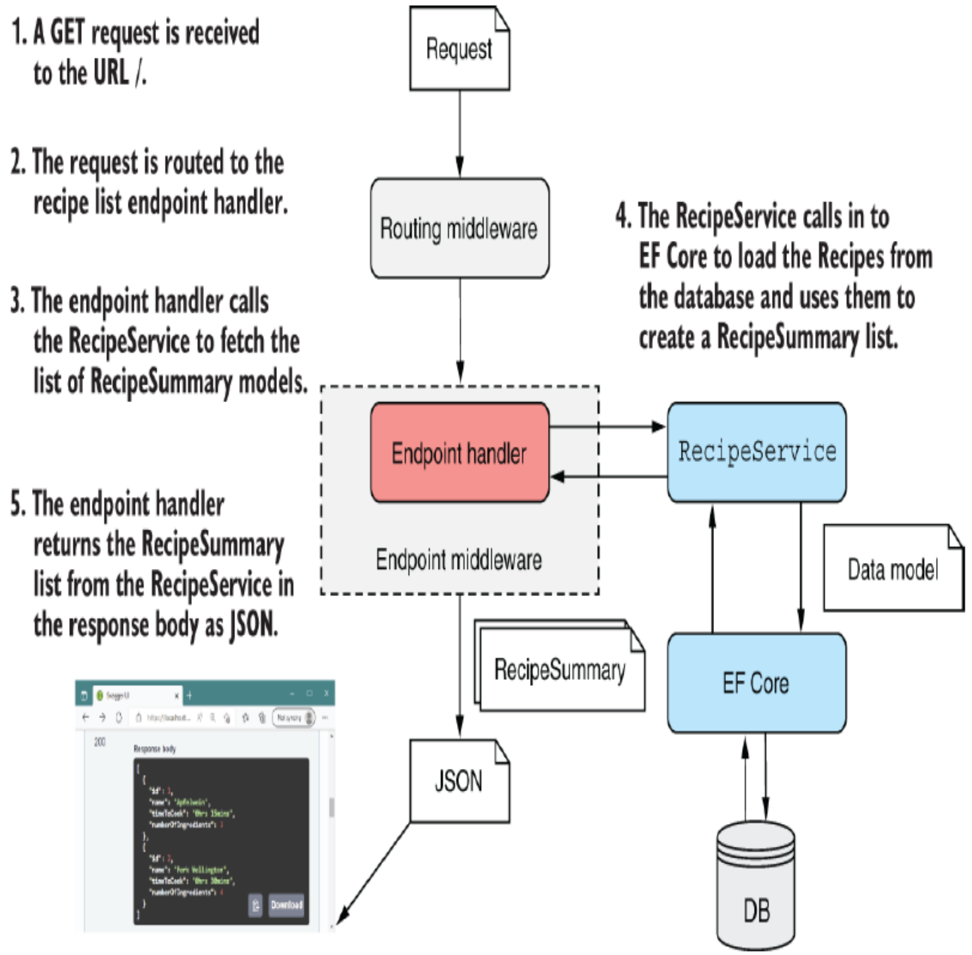
- When you interact with EF Core, you'll be using primarily
  - **POCO entities** – object-oriented representations of the tables in your database;
  - **Database context** that **inherits** from the DbContext EF Core class – to configure EF Core and access the database at runtime.
- You can potentially have **multiple DbContexts** in your application and even configure them to integrate with different databases.

MAPPING A DATABASE TO YOUR APPLICATION CODE

- When your application first uses EF Core, EF Core creates an **internal representation** of the database based on the **DbSet<T> properties** on your application's DbContext and the entity classes themselves


.NET application
AppDbContext
+ Recipes: DbSet<Recipe>
Recipe
+ RecipeId: int
+ Name: string
+ Ingredients: ICollection<Ingredient>
1
0..*
Ingredient
+ IngredientId: int
+ RecipeId: int
+ Name: string
+ Quantity: decimal
+ Unit: string
The application's DbContext serves as the entry point for all interactions with EF Core.
1. EF Core looks for any DbSet properties on the DbContext (Recipes) and adds them to its internal model.
2. It scans all the properties on known entities for linked types and adds them to its internal model.
3. EF Core uses relationships between .NET classes to model the relationship between database tables.
EF Core internal
Recipes
PK RecipeId
Name
Ingredients
PK IngredientId
RecipeId
Name
Quantity
Units

ADDING EF CORE TO AN APPLICATION


1. A GET request is received to the URL /.
2. The request is routed to the recipe list endpoint handler.
3. The endpoint handler calls the RecipeService to fetch the list of RecipeSummary models.
4. The RecipeService calls in to EF Core to load the Recipes from the database and uses them to create a RecipeSummary list.
5. The endpoint handler returns the RecipeSummary list from the RecipeService in the response body as JSON.
Request
Routing middleware
Endpoint handler
Endpoint middleware
RecipeService
Data model
EF Core
DB
RecipeSummary
JSON
Swagger UI
https://localhost...
Not syncing
200
Response body
Download

```
[
  {
    "id": 1,
    "name": "Apfelwein",
    "timeToCook": "0hrs 15mins",
    "numberOfIngredients": 3
  },
  {
    "id": 2,
    "name": "Pork Wellington",
    "timeToCook": "0hrs 30mins",
    "numberOfIngredients": 4
  }
]
```

ADDING EF CORE TO AN APPLICATION

Adding EF Core to an application is a multistep process:

1. **Choose a database provider**, such as Postgres, SQLite, or MS SQL Server.
2. **Install the EF Core NuGet packages**.
3. **Design your app's DbContext and entities** that make up your data model.
4. **Register your app's DbContext** with the ASP.NET Core DI container.
5. Use EF Core to **generate a migration** describing your data model.
6. Apply the migration to the database to **update the database's schema**.

CHOOSING A DATABASE PROVIDER AND INSTALLING EF CORE

Adding support for a given database involves adding the correct NuGet package to your .csproj file, such as the following:

- PostgreSQL—Npgsql.EntityFrameworkCore.PostgreSQL
- Microsoft SQL Server—Microsoft.EntityFrameworkCore.SqlServer
- MySQL—MySql.Data.EntityFrameworkCore
- SQLite—Microsoft.EntityFrameworkCore.SQLite

CHOOSING A DATABASE PROVIDER AND INSTALLING EF CORE

To use the SQLite database provider, so you will be using the SQLite packages:

- **Microsoft.EntityFrameworkCore.SQLite**—This package is the main database provider package for using EF Core at runtime. It also contains a reference to the main EF Core NuGet package.
- **Microsoft.EntityFrameworkCore.Design**—This package contains shared build-time components for EF Core, required for building the EF Core data model for your app.

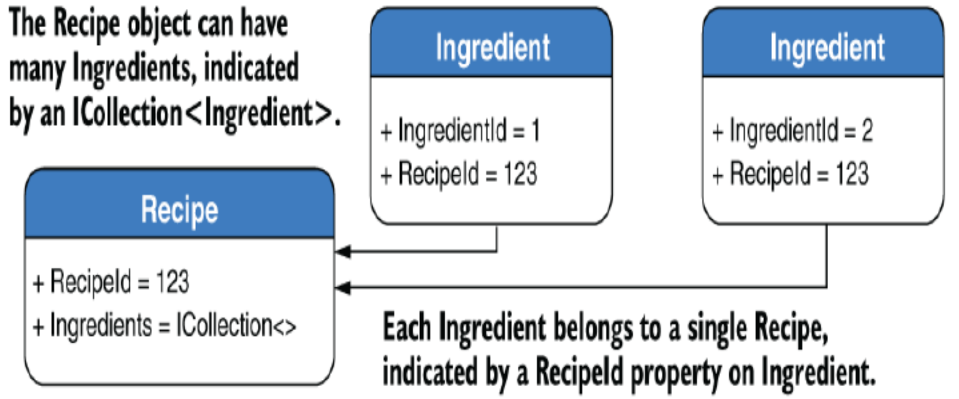
BUILDING A DATA MODEL

```
public class Recipe
{
    public int RecipeId { get; set; }
    public required string Name { get; set; }
    public TimeSpan TimeToCook { get; set; }
    public bool IsDeleted { get; set; }
    public required string Method { get; set; }
    public required ICollection<Ingredient> Ingredients { get; set; }    ❶
}
public class Ingredient
{
    public int IngredientId { get; set; }
    public int RecipeId { get; set; }
    public required string Name { get; set; }
    public decimal Quantity { get; set; }
    public required string Unit { get; set; }
}
```

❶ A Recipe can have many Ingredients, represented by ICollection.

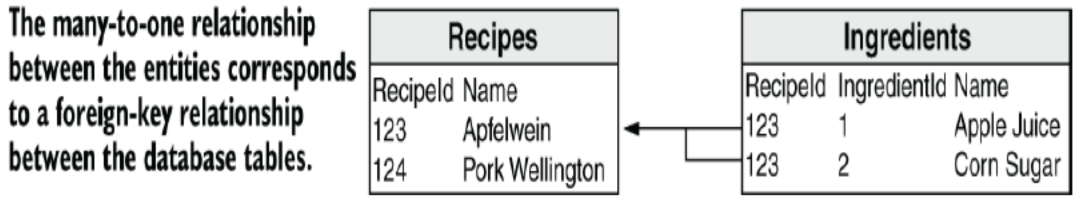
The Recipe class, has a RecipeId property, and the Ingredient class has an IngredientId property.
EF Core identifies this pattern of an Id suffix as indicating the primary key of the table.

BUILDING A DATA MODEL



The Recipe object can have many Ingredients, indicated by an ICollection<Ingredient>.

Each Ingredient belongs to a single Recipe, indicated by a RecipeId property on Ingredient.



The many-to-one relationship between the entities corresponds to a foreign-key relationship between the database tables.

Recipes

RecipeId	Name
123	Apfelwein
124	Pork Wellington

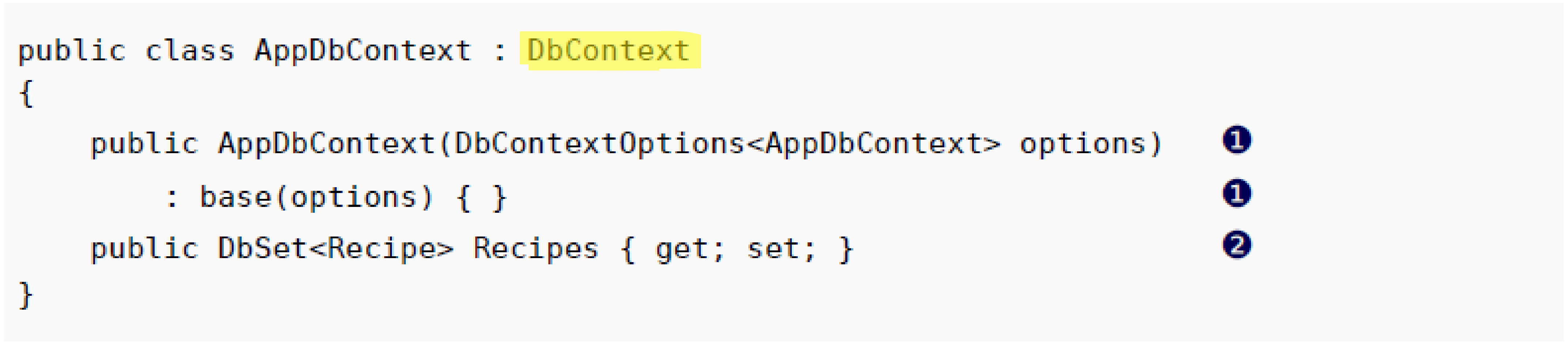
Ingredients

RecipeId	IngredientId	Name
123	1	Apple Juice
123	2	Corn Sugar

BUILDING A DATA MODEL

- You can also use **DataAnnotations** attributes to decorate your entity classes, controlling things like column naming and string length.
- EF Core will use these attributes to override the default conventions.

BUILDING A DATA MODEL



```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)    ❶
        : base(options) { }                                        ❶
    public DbSet<Recipe> Recipes { get; set; }                     ❷
}
```

❶ The constructor options object, containing details such as the connection string

❷ You'll use the Recipes property to query the database.

code-first approach

REGISTERING A DATA CONTEXT

```
using Microsoft.EntityFrameworkCore;
WebApplicationBuillder builder = WebApplication.CreateBuilder(args);
var connString = builder.Configuration                        ❶
        .GetConnectionString("DefaultConnection");            ❶

Builder.Services.AddDbContext<AppDbContext>(                  ❷
        options => options.UseSqlite(connString));            ❸

WebApplication app = builder.Build();
app.Run();
```

❶ The connection string is taken from configuration, from the ConnectionStrings section.

❷ Registers your app’s DbContext by using it as the generic parameter

❸ Specifies the database provider in the customization options for the DbContext.

MANAGING CHANGES WITH MIGRATIONS

- **Schema** refers to **how** the data is **organized** in a database, including the tables, columns, and relationships among them.
- When you deploy an app, normally you can delete the old code/executable and replace it with the new code. Job done. If you **need to roll back a change**, delete that new code, and deploy an old version of the app ?!
- EF Core provides its **own version of schema management** called **migrations**.
- **Migrations** provide a way to **manage changes** to a database schema when your EF Core data model changes.

MANAGING CHANGES WITH MIGRATIONS

- A migration is a **C# code file** in your application that defines how the data model changed—which columns were added, new entities, and so on.
- **Migrations** provide a **record over time of how your database schema evolved** as part of your application, so the **schema is always in sync with your app's data model**.
- You can use **command-line tools** to
  - **create a new database** from the migrations or to update an existing database by applying new migrations to it.
  - **to roll back a migration**, which updates a database to a previous schema.

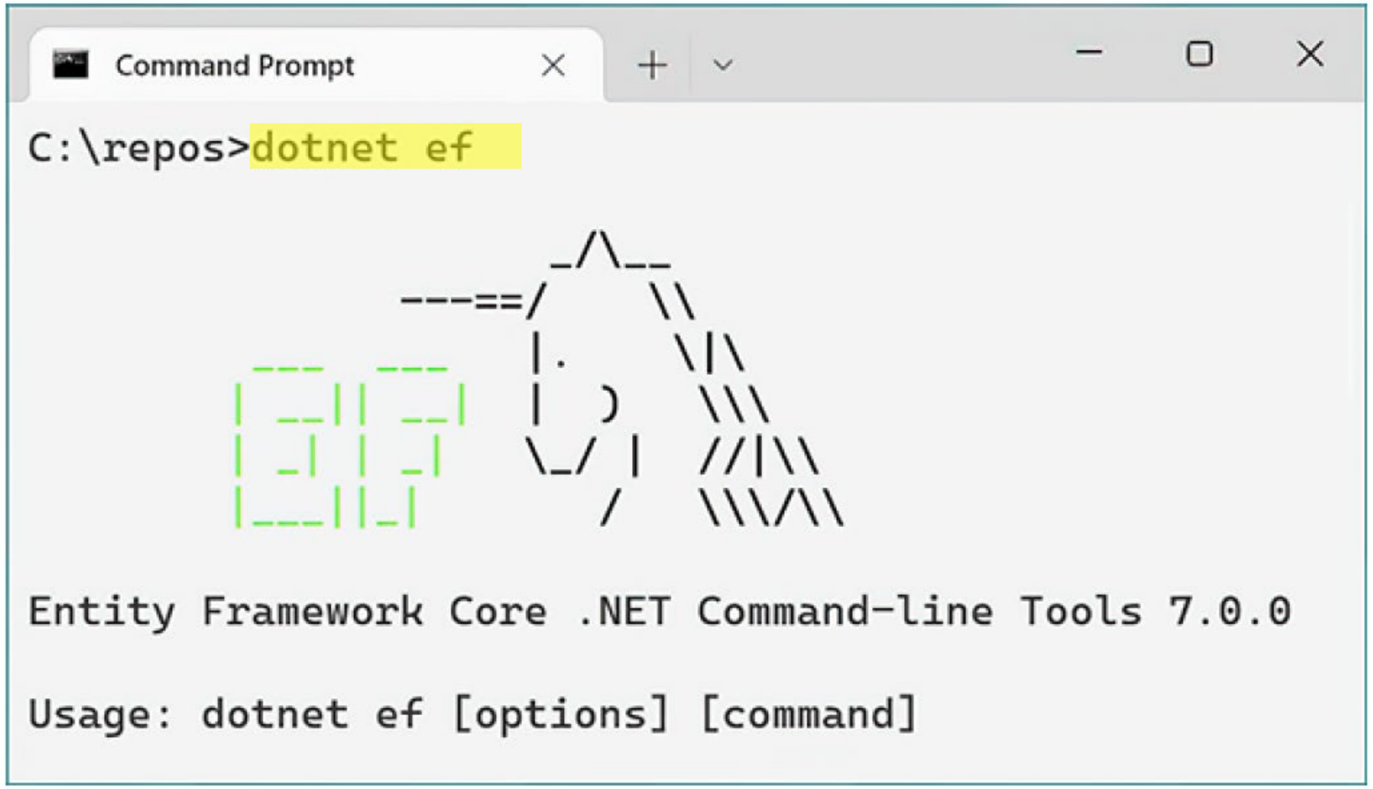
CREATING YOUR FIRST MIGRATION

You need to install the necessary tooling. You have two primary ways to do this:

- **Package manager console**—You can use PowerShell cmdlets inside Visual Studio's Package Manager Console (PMC). You can install them directly from the PMC or by adding the Microsoft.EntityFrameworkCore.Tools package to your project.
- **.NET tool**—You can use cross-platform, command line tooling that extends the .NET SDK. You can install the EF Core .NET tool globally for your machine by running

dotnet tool install -- global dotnet-ef.

CREATING YOUR FIRST MIGRATION



```
C:\repos>dotnet ef

                     _/\__
               ---==/    \\
         ___  ___   |.    \|\
        | __|| __|  |  )    \\\
        | _| | _|   \_/ |  //|\\
        |___||_|       /   \\\/\\

Entity Framework Core .NET Command-line Tools 7.0.0

Usage: dotnet ef [options] [command]
```

CREATING YOUR FIRST MIGRATION

- You can create your first migration by running the following command from inside your web project folder **(in which you registered your AppDbContext)** and providing a name for the migration (in this case, InitialSchema):

```
dotnet ef migrations add InitialSchema
```

CREATING YOUR FIRST MIGRATION

This command creates three files in the Migrations folder in your project:

- **Migration file**—This file, with the **Timestamp_MigrationName.cs** format, describes the actions to take on the database, such as creating a table or adding a column.
- **Migration designer.cs** file—This file describes **EF Core's internal model** of your data model **at the point in time when the migration** was generated.
- **AppDbContextModelSnapshot.cs**—This file describes EF Core's current internal model.
  - This file is **updated** when you add another migration.
  - It should always be the same as the current (latest) migration.
  - EF Core can use AppDbContextModelSnapshot.cs to **determine a database's previous state** when creating a new migration without interacting with the database directly.

CREATING YOUR FIRST MIGRATION

You can apply migrations in any of four ways:

- Using the .NET tool

  dotnet ef database update

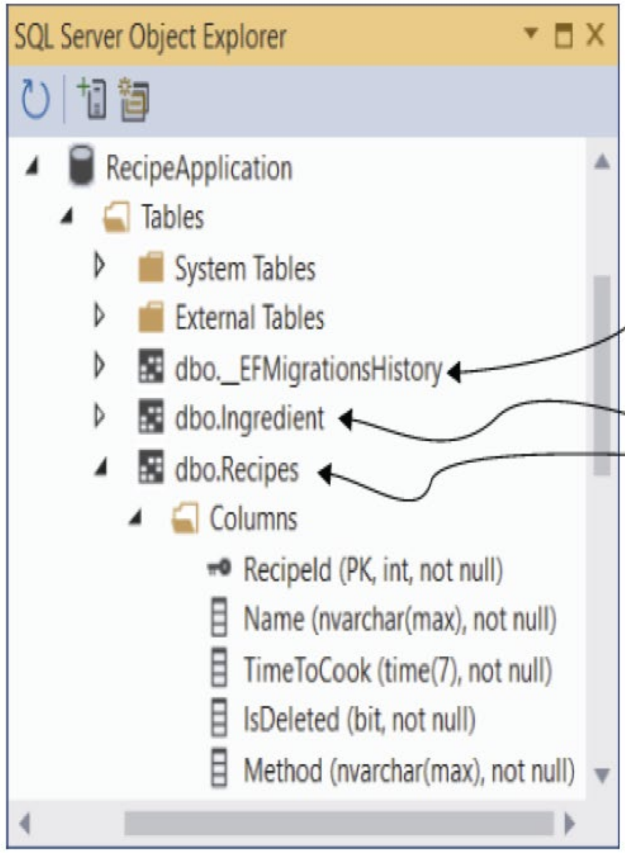
- Using the Visual Studio PowerShell cmdlets
- In code, by obtaining an instance of your AppDbContext from the DI container and calling **context.Database.Migrate()**
- By generating a migration bundle application

CREATING YOUR FIRST MIGRATION

Updating data base performs four steps:

1. Builds your application
2. Loads the services configured in your app's Program.cs, including AppDbContext.
3. Checks whether the database in the AppDbContext connection string exists and if not, creates it
4. Updates the database by applying any unapplied migrations

CREATING YOUR FIRST MIGRATION


SQL Server Object Explorer

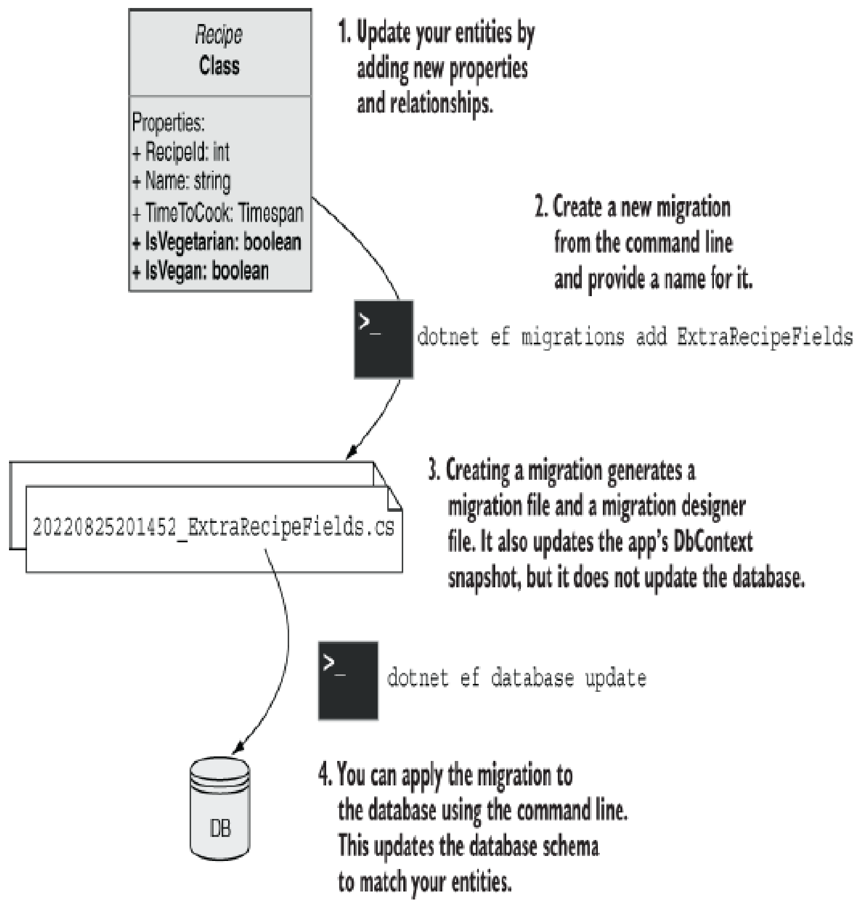
- RecipeApplication
  - Tables
    - System Tables
    - External Tables
    - dbo.__EFMigrationsHistory
    - dbo.Ingredient
    - dbo.Recipes
      - Columns
        - RecipeId (PK, int, not null)
        - Name (nvarchar(max), not null)
        - TimeToCook (time(7), not null)
        - IsDeleted (bit, not null)
        - Method (nvarchar(max), not null)

The __EFMigrationsHistory table contains a list of all the migrations that have been applied to the database.

The entities in our data model, Recipe and Ingredient, correspond to tables in the database.

The properties on the Recipe entity correspond to the columns in the Recipes table.

ADDING A SECOND MIGRATION



Recipe
Class
Properties:
+ RecipeId: int
+ Name: string
+ TimeToCook: Timespan
+ IsVegetarian: boolean
+ IsVegan: boolean
1. Update your entities by adding new properties and relationships.
2. Create a new migration from the command line and provide a name for it.
dotnet ef migrations add ExtraRecipeFields
20220825201452_ExtraRecipeFields.cs
3. Creating a migration generates a migration file and a migration designer file. It also updates the app's DbContext snapshot, but it does not update the database.
dotnet ef database update
DB
4. You can apply the migration to the database using the command line. This updates the database schema to match your entities.

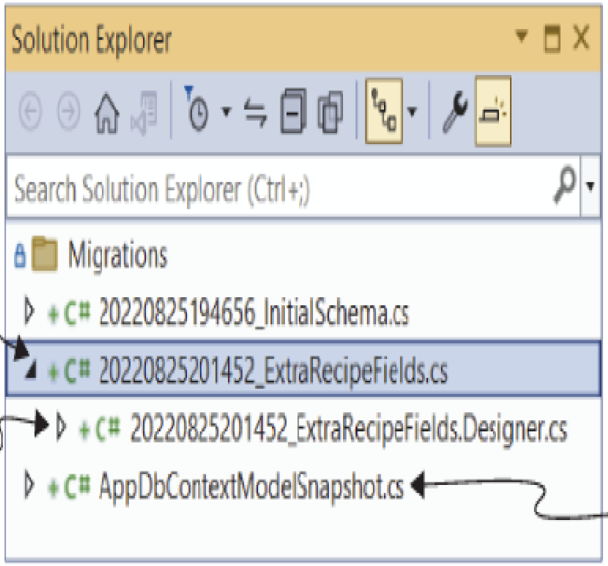
```
public class Recipe
{
    public int RecipeId { get; set; }

    public required string Name { get; set; }
    public TimeSpan TimeToCook { get; set; }
    public bool IsDeleted { get; set; }
    public required string Method { get; set; }
    public bool IsVegetarian { get; set; }
    public bool IsVegan { get; set; }
    public required ICollection<Ingredient> Ingredients { get; set; }
}
```

ADDING A SECOND MIGRATION

```
dotnet ef migrations add ExtraRecipeFields
```

Creating a migration adds a cs file to your solution with a timestamp and the name you gave the migration.

It also adds a Designer.cs file that contains a snapshot of EF Core's internal data model at the point in line.



Solution Explorer

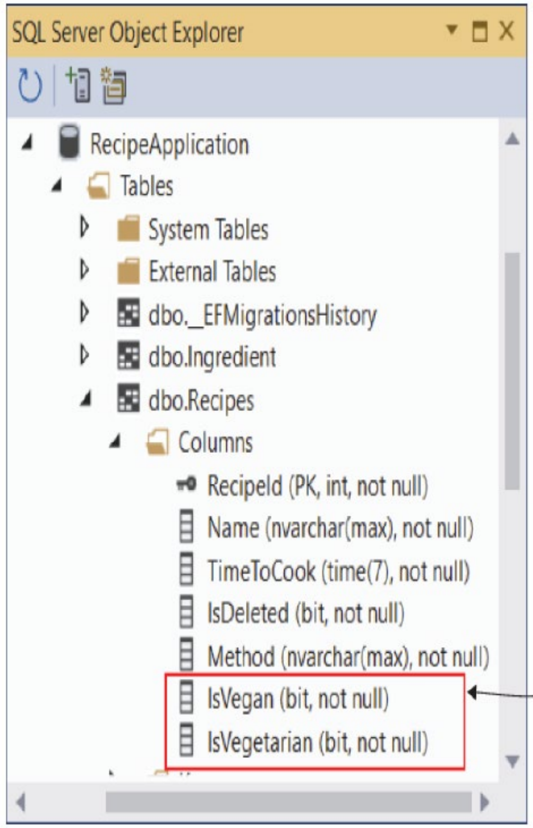
Search Solution Explorer (Ctrl+;)

```
Migrations
  20220825194656_InitialSchema.cs
  20220825201452_ExtraRecipeFields.cs
    20220825201452_ExtraRecipeFields.Designer.cs
  AppDbContextModelSnapshot.cs
```

The AppDbContextModelSnapshot is updated to match the snapshot for the new migration.

ADDING A SECOND MIGRATION

```
dotnet ef database update
```

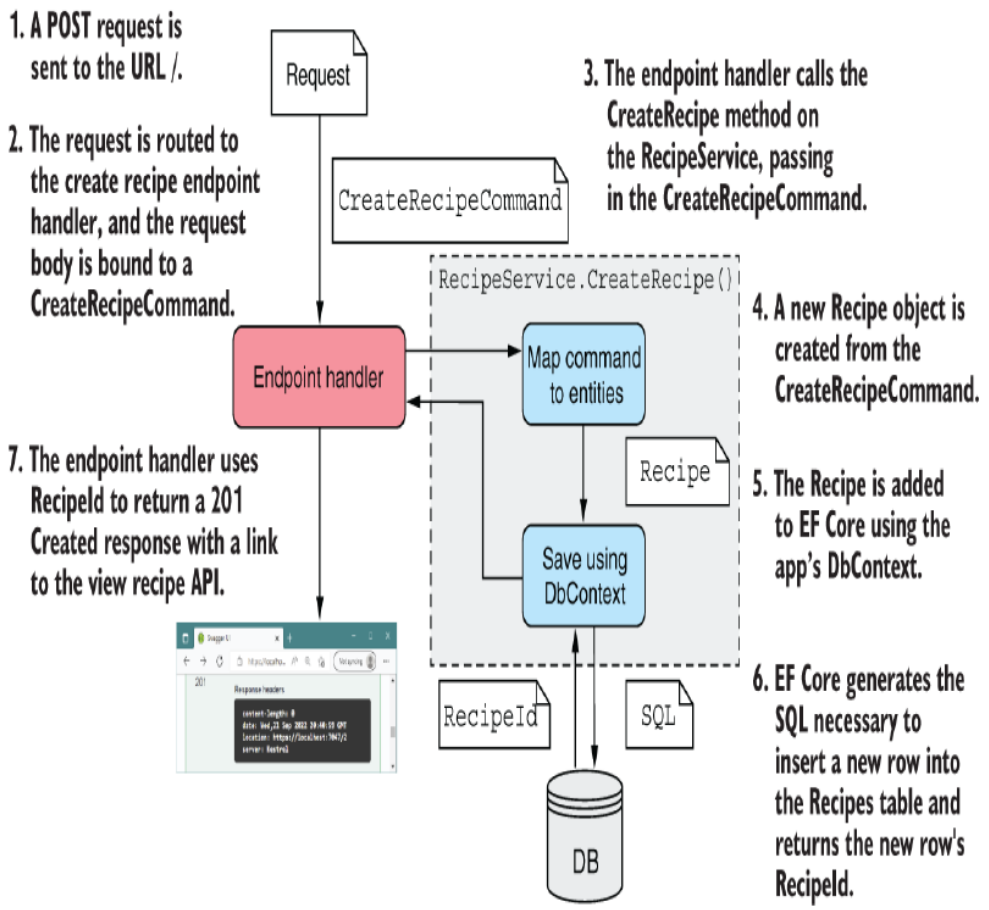


SQL Server Object Explorer

```
RecipeApplication
  Tables
    System Tables
    External Tables
    dbo.__EFMigrationsHistory
    dbo.Ingredient
    dbo.Recipes
      Columns
        RecipeId (PK, int, not null)
        Name (nvarchar(max), not null)
        TimeToCook (time(7), not null)
        IsDeleted (bit, not null)
        Method (nvarchar(max), not null)
        IsVegan (bit, not null)
        IsVegetarian (bit, not null)
```

Applying the second migration to the database adds the new fields to the Recipes table.

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - CREATING A RECORD


1. A POST request is sent to the URL /.
Request
2. The request is routed to the create recipe endpoint handler, and the request body is bound to a CreateRecipeCommand.
CreateRecipeCommand
3. The endpoint handler calls the CreateRecipe method on the RecipeService, passing in the CreateRecipeCommand.
RecipeService.CreateRecipe()
Endpoint handler
Map command to entities
4. A new Recipe object is created from the CreateRecipeCommand.
Recipe
Save using DbContext
5. The Recipe is added to EF Core using the app's DbContext.
7. The endpoint handler uses RecipeId to return a 201 Created response with a link to the view recipe API.
RecipeId
SQL
DB
6. EF Core generates the SQL necessary to insert a new row into the Recipes table and returns the new row's RecipeId.
Swagger UI
201
Response headers

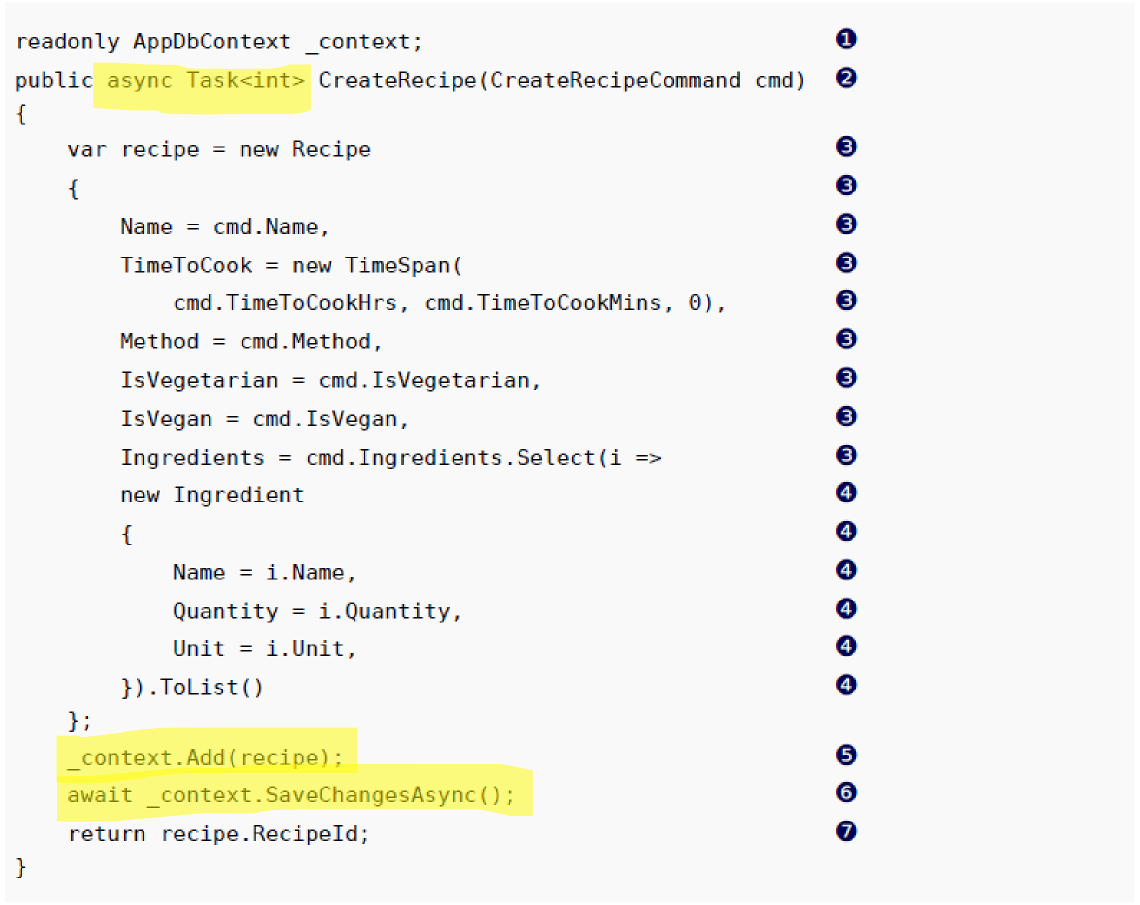
```
content-length: 0
date: Wed,21 Sep 2022 20:40:55 GMT
location: https://localhost:7047/2
server: Kestrel
```

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - CREATING A RECORD

Creating a new entity requires three steps:

1. Create the Recipe and Ingredient entities.
2. Add the entities to EF Core's list of tracked entities using **_context.Add**(entity).
3. Execute the **SQL INSERT statements** against the database, adding the necessary rows to the Recipe and Ingredient tables, by calling **_context.SaveChangesAsync()**.

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - CREATING A RECORD

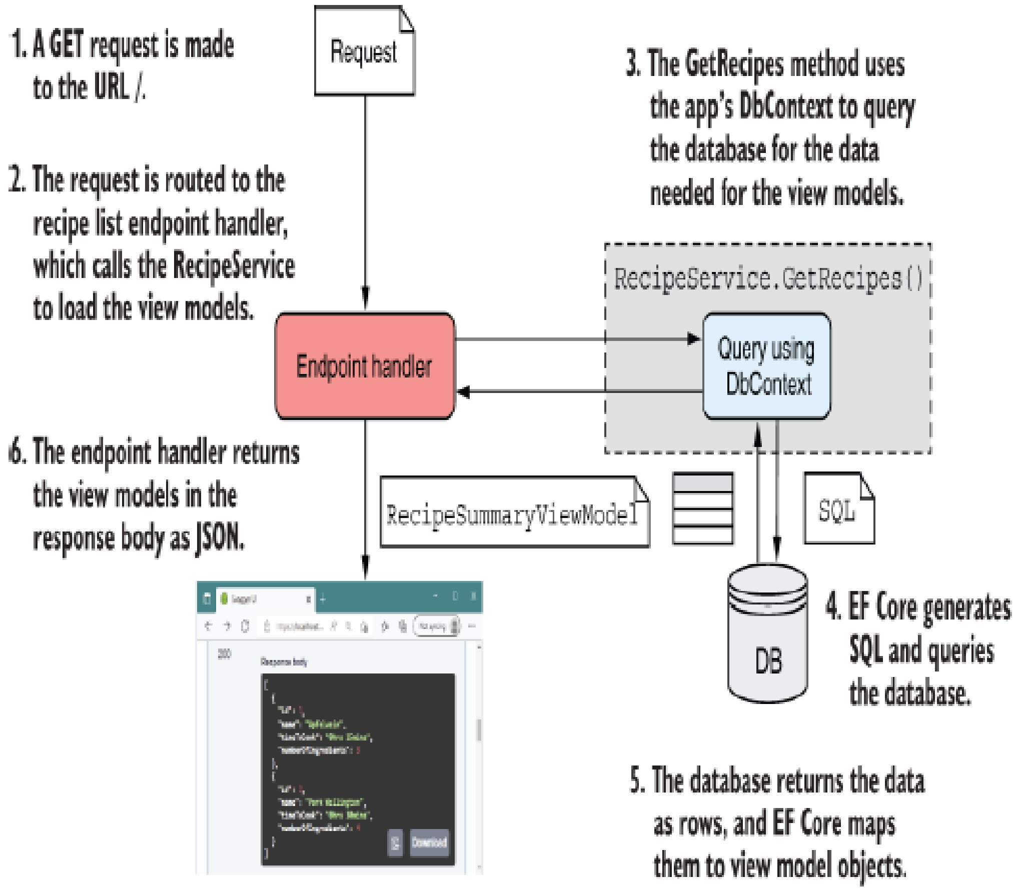


```
readonly AppDbContext _context;                                    ❶
public async Task<int> CreateRecipe(CreateRecipeCommand cmd)       ❷
{
    var recipe = new Recipe                                        ❸
    {                                                              ❸
        Name = cmd.Name,                                           ❸
        TimeToCook = new TimeSpan(                                 ❸
            cmd.TimeToCookHrs, cmd.TimeToCookMins, 0),             ❸
        Method = cmd.Method,                                       ❸
        IsVegetarian = cmd.IsVegetarian,                           ❸
        IsVegan = cmd.IsVegan,                                     ❸
        Ingredients = cmd.Ingredients.Select(i =>                  ❸
        new Ingredient                                             ❹
        {                                                          ❹
            Name = i.Name,                                         ❹
            Quantity = i.Quantity,                                 ❹
            Unit = i.Unit,                                         ❹
        }).ToList()                                                ❹
    };
    _context.Add(recipe);                                          ❺
    await _context.SaveChangesAsync();                             ❻
    return recipe.RecipeId;                                        ❼
}
```

❶ An instance of the AppDbContext is injected in the class constructor using DI.

❷ CreateRecipeCommand is passed in from the endpoint handler.

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - LOADING A LIST OF RECORDS


1. A GET request is made to the URL /.
Request
3. The GetRecipes method uses the app's DbContext to query the database for the data needed for the view models.
2. The request is routed to the recipe list endpoint handler, which calls the RecipeService to load the view models.
RecipeService.GetRecipes()
Endpoint handler
Query using DbContext
6. The endpoint handler returns the view models in the response body as JSON.
RecipeSummaryViewModel
SQL
DB
4. EF Core generates SQL and queries the database.
5. The database returns the data as rows, and EF Core maps them to view model objects.
200
Response body
Download

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - LOADING A LIST OF RECORDS

```
public async Task<ICollection<RecipeSummaryViewModel>> GetRecipes()
{
    return await _context.Recipes                              ❶
        .Where(r => !r.IsDeleted)
        .Select(r => new RecipeSummaryViewModel                ❷
        {                                                      ❷
            Id = r.RecipeId,                                   ❷
            Name = r.Name,                                     ❷
            TimeToCook = $"{r.TimeToCook.TotalMinutes}mins"    ❷
        })
        .ToListAsync();                                        ❸
}
```

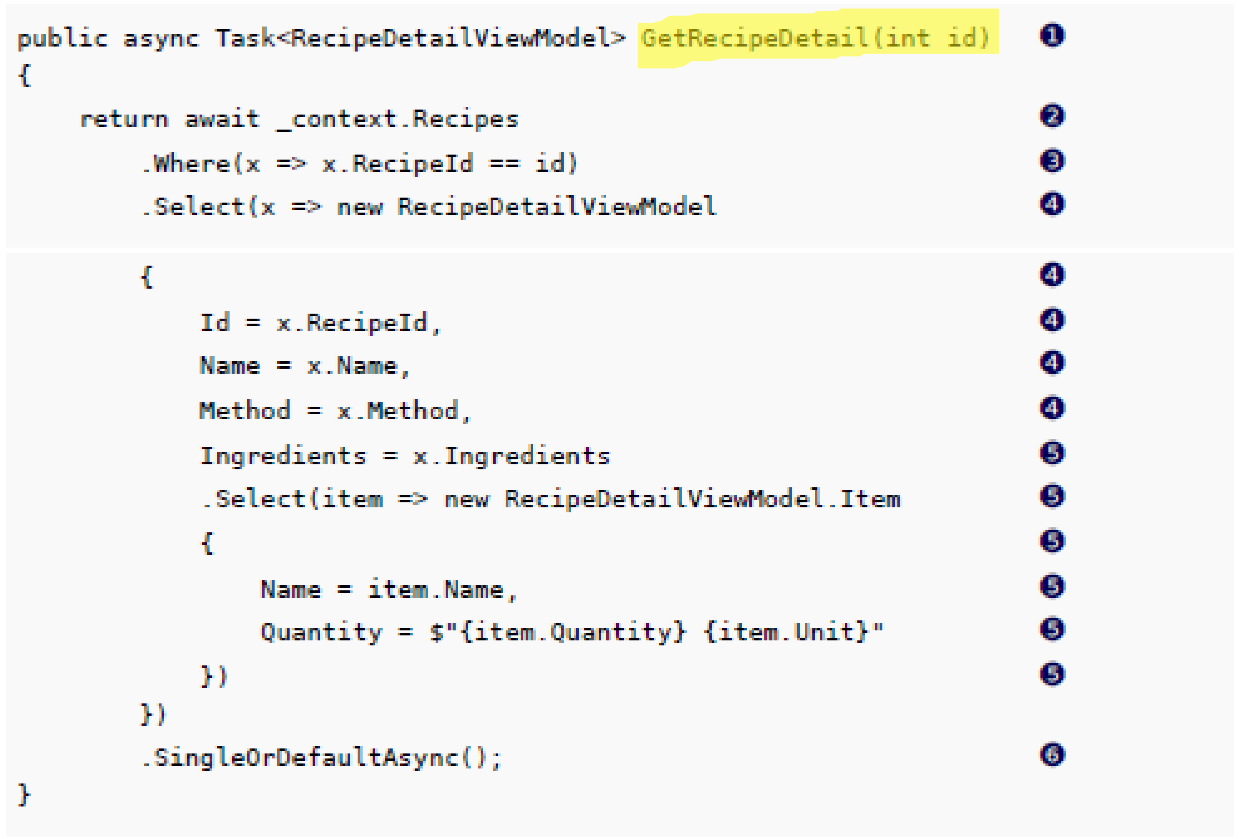
❶ A query starts from a DbSet property.

❷ EF Core queries only the Recipe columns it needs to map the view model correctly.

❸ Executes the SQL query and creates the final view models

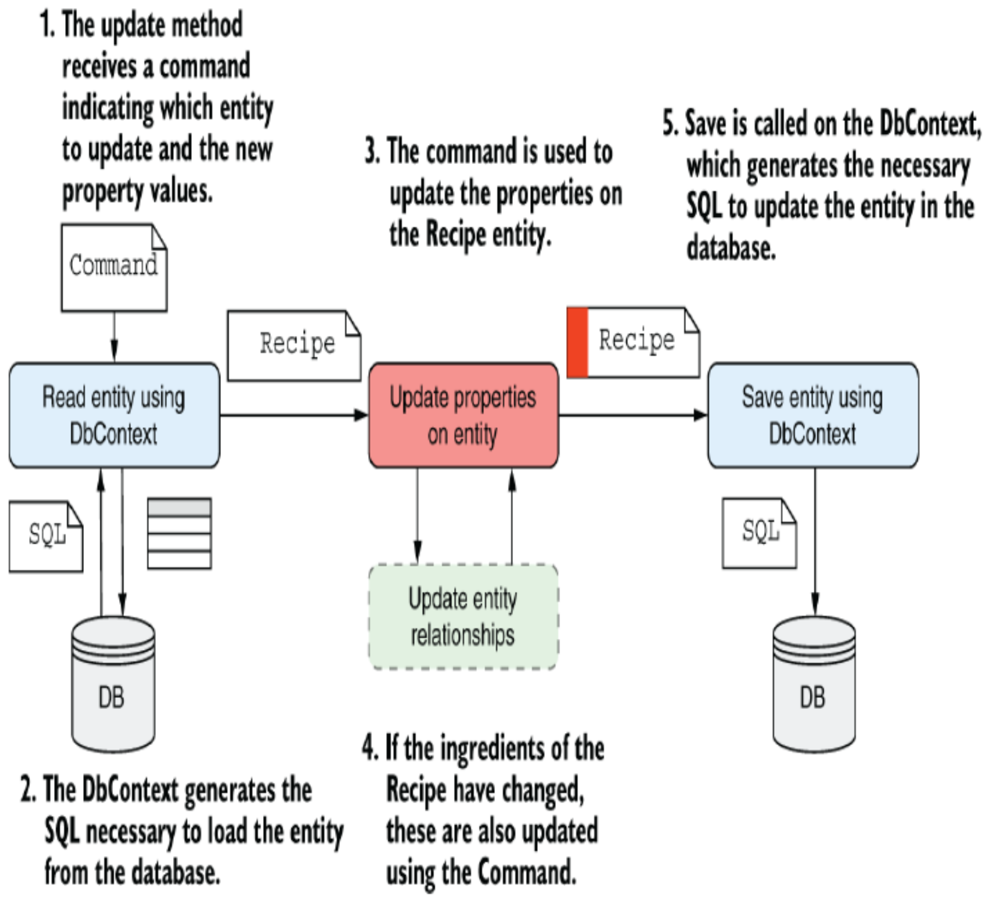
$$\underbrace{\texttt{\_context.Recipes}}_{\text{AppDbContext DbSet Property access}}\texttt{.}\underbrace{\texttt{Where(r => !r.IsDeleted)}}_{\text{LINQ commands to modify data returned}}\texttt{.}\underbrace{\texttt{ToListAsync()}}_{\text{Execute query command}}$$

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - LOADING A SINGLE RECORD

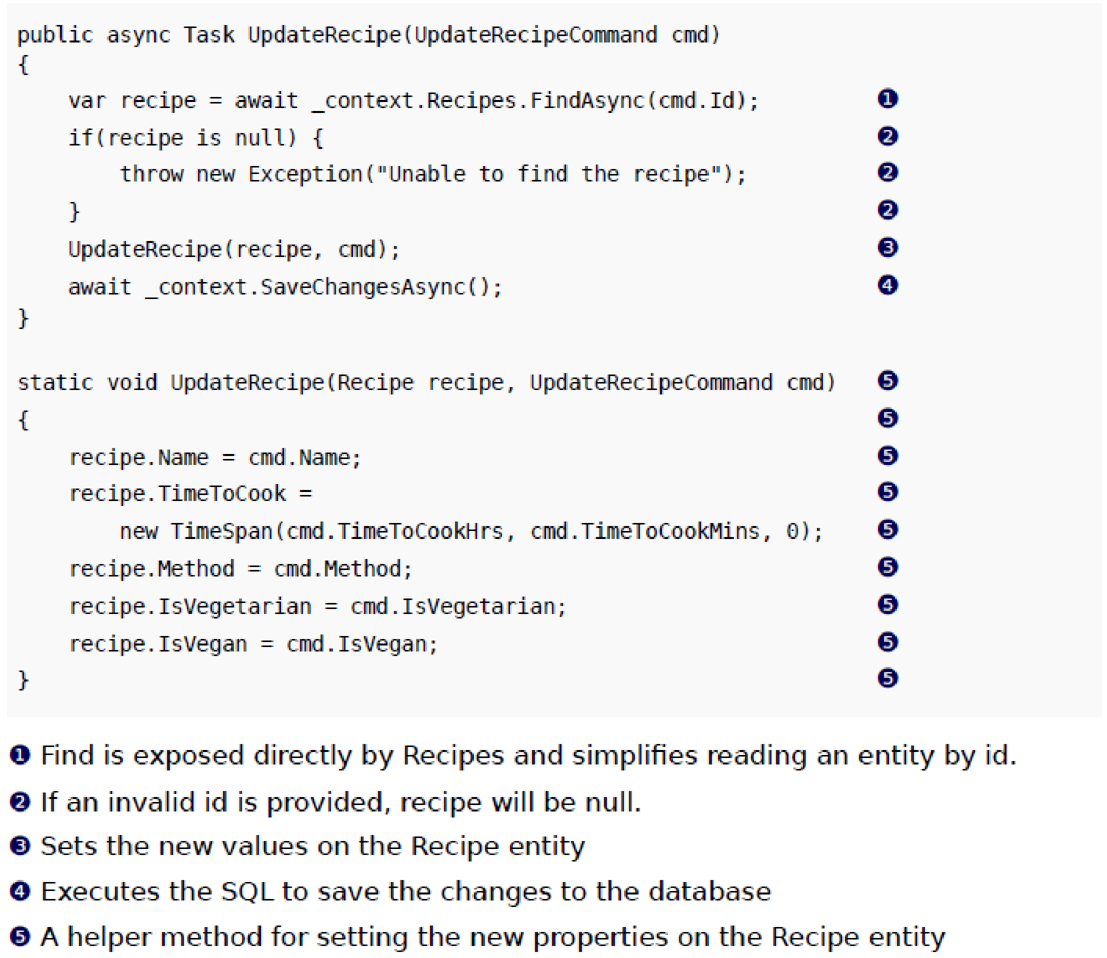


```
public async Task<RecipeDetailViewModel> GetRecipeDetail(int id)    ❶
{
    return await _context.Recipes                                   ❷
        .Where(x => x.RecipeId == id)                               ❸
        .Select(x => new RecipeDetailViewModel                      ❹
        {                                                           ❹
            Id = x.RecipeId,                                        ❹
            Name = x.Name,                                          ❹
            Method = x.Method,                                      ❹
            Ingredients = x.Ingredients                             ❺
            .Select(item => new RecipeDetailViewModel.Item          ❺
            {                                                       ❺
                Name = item.Name,                                   ❺
                Quantity = $"{item.Quantity} {item.Unit}"           ❺
            })                                                      ❺
        })
        .SingleOrDefaultAsync();                                    ❻
}
```

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - UPDATING A MODEL WITH CHANGES


1. The update method receives a command indicating which entity to update and the new property values.
Command
3. The command is used to update the properties on the Recipe entity.
5. Save is called on the DbContext, which generates the necessary SQL to update the entity in the database.
Recipe
Recipe
Read entity using DbContext
Update properties on entity
Save entity using DbContext
SQL
SQL
Update entity relationships
DB
DB
2. The DbContext generates the SQL necessary to load the entity from the database.
4. If the ingredients of the Recipe have changed, these are also updated using the Command.

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - UPDATING A MODEL WITH CHANGES



```
public async Task UpdateRecipe(UpdateRecipeCommand cmd)
{
    var recipe = await _context.Recipes.FindAsync(cmd.Id);          ❶
    if(recipe is null) {                                            ❷
        throw new Exception("Unable to find the recipe");           ❷
    }                                                               ❷
    UpdateRecipe(recipe, cmd);                                      ❸
    await _context.SaveChangesAsync();                              ❹
}

static void UpdateRecipe(Recipe recipe, UpdateRecipeCommand cmd)    ❺
{                                                                   ❺
    recipe.Name = cmd.Name;                                         ❺
    recipe.TimeToCook =                                             ❺
        new TimeSpan(cmd.TimeToCookHrs, cmd.TimeToCookMins, 0);     ❺
    recipe.Method = cmd.Method;                                     ❺
    recipe.IsVegetarian = cmd.IsVegetarian;                         ❺
    recipe.IsVegan = cmd.IsVegan;                                   ❺
}                                                                   ❺
```

❶ Find is exposed directly by Recipes and simplifies reading an entity by id.

❷ If an invalid id is provided, recipe will be null.

❸ Sets the new values on the Recipe entity

❹ Executes the SQL to save the changes to the database

❺ A helper method for setting the new properties on the Recipe entity

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - DELETING A MODEL WITH CHANGES

- EF Core can easily handle these true deletion scenarios for you with the DbContext .Remove(entity) command, but often what you mean when you find a need to delete data is to archive it or hide it from the UI.

```
public bool IsDeleted { get; set; }
```

QUERYING DATA FROM AND SAVING DATA TO THE DATABASE - DELETING A MODEL WITH CHANGES

```
public async Task DeleteRecipe(int recipeId)
{
    var recipe = await _context.Recipes.FindAsync(recipeId);   ❶
    if(recipe is null) {                                       ❷
        throw new Exception("Unable to find the recipe");      ❷
    }                                                          ❷
    recipe.IsDeleted = true;                                   ❸


    await _context.SaveChangesAsync();                         ❹
}
```

❶ Fetches the Recipe entity by id

❷ If an invalid id is provided, recipe will be null.

❸ Marks the Recipe as deleted

❹ Executes the SQL to save the changes to the database

USING EF CORE IN PRODUCTION APPLICATIONS

set of things to consider before you dive into production:

- **Scaffolding of columns**—EF Core uses conservative values for things like string columns by allowing strings of large or unlimited length. In practice, you may want to restrict these and other data types to sensible values.
- **Validation**—You can decorate your entities with DataAnnotations validation attributes, but EF Core won't validate the values automatically before saving to the database. This behavior differs from EF 6.x behavior, in which validation was automatic.

USING EF CORE IN PRODUCTION APPLICATIONS

- **Handling concurrency**—EF Core provides a few ways to handle concurrency, which occurs when multiple users attempt to update an entity at the same time. One partial solution is to use Timestamp columns on your entities.
- **Handling errors**—Databases and networks are inherently flaky, so you'll always have to account for transient errors. EF Core includes various features to maintain connection resiliency by retrying on network failures.
- **Synchronous vs. asynchronous**—EF Core provides both synchronous and asynchronous commands for interacting with the database. Often, async is better for web apps, but this argument has nuances that make it impossible to recommend one approach over the other in all situations.

USING EF CORE IN PRODUCTION APPLICATIONS

The following problems are likely to affect ASP.NET Core developers at some point:

- **Automatic migrations**—If you deploy your app to production automatically as part of some sort of DevOps pipeline, you'll inevitably need some way to apply migrations to a database automatically. You can tackle this situation in several ways, such as scripting the .NET tool, applying migrations in your app's startup code, using EF Core bundles, or using a custom tool. Each approach has its pros and cons.
- **Multiple web hosts**—One specific consideration is whether you have multiple web servers hosting your app, all pointing to the same database. If so, applying migrations in your app's startup code becomes harder, as you must ensure that only one app can migrate the database at a time.

.

USING EF CORE IN PRODUCTION APPLICATIONS

- **Making backward-compatible schema changes**—A corollary of the multiple-web-host approach is that you'll often be in a situation in which your app accesses a database that has a newer schema than the app thinks. Normally, you should endeavor to make schema changes backwardcompatible wherever possible.
- **Storing migrations in a different assembly**—In this chapter I included all my logic in a single project, but in larger apps, data access is often in a different project from the web app. For apps with this structure, you must use slightly different commands when using .NET CLI or PowerShell cmdlets.
- **Seeding data**—When you first create a database, you often want it to have some initial seed data, such as a default user. EF 6.x had a mechanism for seeding data built in, whereas EF Core requires you to seed your database explicitly yourself