# CSc 346
# Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 10

# Working with Data Using Entity Framework Core

- This chapter is about reading and writing to data stores, such as Microsoft SQL Server, SQLite, and Azure Cosmos DB, by using the object-to-data store mapping technology named **Entity Framework Core** (**EF Core**)
  - Understanding modern databases
  - Setting up EF Core
  - Defining EF Core models
  - Querying EF Core models
  - Loading patterns with EF Core
  - Manipulating data with EF Core
  - Working with transactions
  - Code First EF Core models

# Understanding modern databases

- Two of the most common places to store data are in a
  - **Relational Database Management System** (**RDBMS**)
    - Microsoft SQL Server
    - PostgreSQL
    - MySQL
    - SQLite
    - …
  - **NoSQL** database (**Document**)
    - Microsoft Azure Cosmos DB
    - Redis
    - MongoDB
    - Apache Cassandra
    - …

# Understanding Entity Framework Core

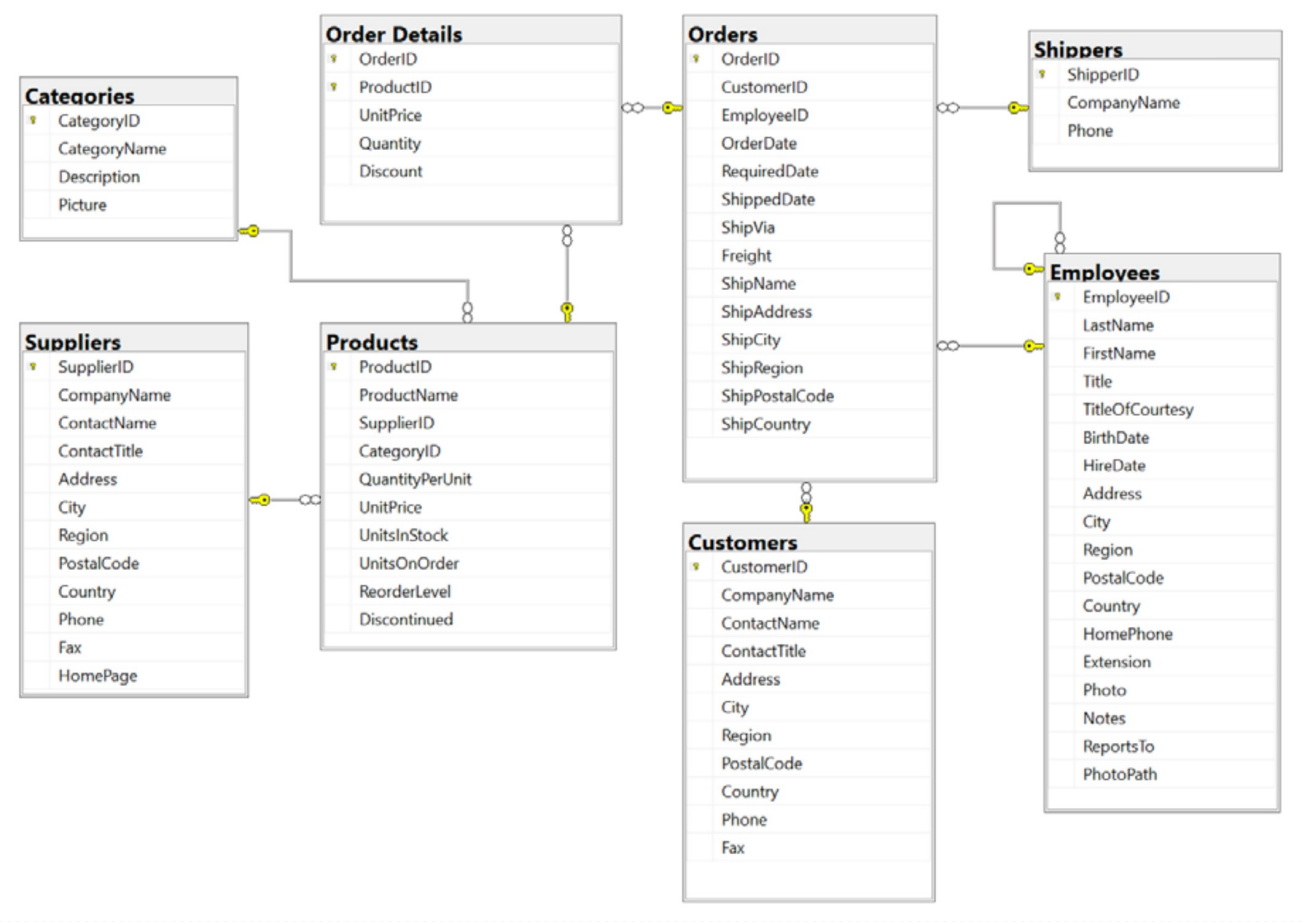
- The latest **EF Core** is version 8 to match .NET 8, EF Core supports
  - Traditional RDBMSs
  - Modern cloud-based, non-relational, schema-less data stores
- We will focus on the fundamentals that all .NET developers should know and some of the most useful new features

- There are two approaches to working with EF Core:
  1. **Database First**: database already exists
     - Build a model that matches its structure and features
  2. **Code First**: no database exists
     - Build a model and then use EF Core to create a database that matches its structure and features
- We will start by using EF Core with an existing database

# Using a sample relational database

- To learn how to manage an RDBMS using .NET, it would be useful to have a sample one so that you can practice on one that has a medium complexity and a decent amount of sample records
- Microsoft offers several sample databases, most of which are too complex for our needs, so instead, we will use a database that was first created in the early 1990s known as **Northwind**

# Using a sample relational database



# Choosing an EF Core database provider

- To manage data in a specific database, we need classes that know how to efficiently talk to that database
- EF Core database providers are sets of classes that are optimized for a specific data store. There is even a provider for storing the data in the memory of the current process, which can be useful for high-performance unit testing since it avoids hitting an external system

| To manage this data store | Install this NuGet package |
|---|---|
| SQL Server 2012 or later | `Microsoft.EntityFrameworkCore.SqlServer` |
| SQLite 3.7 or later | `Microsoft.EntityFrameworkCore.SQLite` |
| In-memory | `Microsoft.EntityFrameworkCore.InMemory` |
| Azure Cosmos DB SQL API | `Microsoft.EntityFrameworkCore.Cosmos` |
| MySQL | `MySQL.EntityFrameworkCore` |
| Oracle DB 11.2 | `Oracle.EntityFrameworkCore` |
| PostgreSQL | `Npgsql.EntityFrameworkCore.PostgreSQL` |

# Connecting to a database

- To connect to a SQLite database, we just need to know the
  - database filename
  - set using the parameter Filename
- We specify this information in a **connection string**

# Defining the Northwind database context class

- The Northwind class will be used to represent the database
- To use EF Core, the class must inherit from DbContext
- This class understands how to communicate with databases and dynamically generate SQL statements to query and manipulate data
- Your DbContext-derived class should have an overridden method named OnConfiguring, which will set the database connection string

# Defining the Northwind database context class

```
<ItemGroup>
    // globally and statically import for all C# files
    <Using Include="System.Console" Static="true" />
</ItemGroup>

<ItemGroup>
    // Add the following package reference
    <PackageReference
        Version="8.0.0"
        Include="Microsoft.EntityFrameworkCore.Sqlite" />
</ItemGroup>
```

# Defining the Northwind database context class

```
// NorthwindDb.cs file
using Microsoft.EntityFrameworkCore; // To use DbContext and so on
namespace Northwind.EntityModels;
// Manages interactions with the Northwind database
0 references
public class NorthwindDb : DbContext {
    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
        string databaseFile = "Northwind.db";
        string path = Path.Combine(Environment.CurrentDirectory, databaseFile);
        string connectionString = $"Data Source={path}";
        WriteLine($"Connection: {connectionString}");
        optionsBuilder.UseSqlite(connectionString);
    }
}

// Program.cs file
using Northwind.EntityModels; // To use Northwind
using NorthwindDb db = new();
0 references
WriteLine($"Provider: {db.Database.ProviderName}");
// Disposes the database context
```

# Defining EF Core models

- EF Core uses a combination of
  - **conventions**
  - **annotation attributes**
  - **Fluent API** statements

  to build an **entity model** at runtime so that any actions performed on the classes can later be automatically translated into actions performed on the actual database
- An entity class represents the structure of a table, and an instance of the class represents a row in that table
- First, we will review the three ways to define a model, with code examples, and then we will create some classes that implement those techniques

# Using EF Core conventions to define the model

- The name of a table is assumed to match the name of a DbSet<T> property in the DbContext class
  - E.g., Products
- The names of the columns are assumed to match the names of properties in the entity model class
  - E.g., ProductId
- The **string** .NET type is assumed to be a **nvarchar** type in the database
- The **int** .NET type is assumed to be an **int** type in the database
- The **primary key** is assumed to be a property that is named **Id** or **ID**
  - When the entity model class is named **Product**, then the property can be named **ProductId** or **ProductID**
- If this property is an **integer** type or the **Guid** type, then it is also assumed to be an **IDENTITY** column
  - A column type that automatically assigns a value when inserting

# Using EF Core annotation attributes to define the model

- Conventions often aren't enough to map the classes to the database objects
- A simple way of adding more smarts to your model is to apply annotation attributes

| Attribute | Description |
|---|---|
| `[Required]` | Ensures the value is not null. In .NET 8, it has a `DisallowAllDefaultValues` parameter to prevent value types having their default value. For example, an `int` cannot be `0`. |
| `[StringLength(50)]` | Ensures the value is up to 50 characters in length. |
| `[Column(TypeName = "money", Name = "UnitPrice")]` | Specifies the column type and column name used in the table. |

# Using EF Core annotation attributes to define the model

| Attribute | Description |
|---|---|
| `[RegularExpression(expression)]` | Ensures the value matches the specified regular expression. |
| `[EmailAddress]` | Ensures the value contains one `@` symbol, but not as the first or last character. It does not use a regular expression. |
| `[Range(1, 10)]` | Ensures a `double`, `int`, or `string` value within a specified range. New in .NET 8 are the parameters `MinimumIsExclusive` and `MaximumIsExclusive`. |
| `[Length(10, 20)]` | Ensures a string or collection is within a specified length range, for example, minimum 10 characters or items, maximum 20 characters or items. |
| `[Base64String]` | Ensures the value is a well-formed Base64 string. |
| `[AllowedValues]` | Ensures value is one of the items in the `params` array of objects. For example, `"alpha"`, `"beta"`, `"gamma"`, or `1`, `2`, `3`. |
| `[DeniedValues]` | Ensures value is not one of the items in the `params` array of objects. For example, `"alpha"`, `"beta"`, `"gamma"`, or `1`, `2`, `3`. |

# Using EF Core annotation attributes to define the model

```
CREATE TABLE Products (
    ProductId           INTEGER         PRIMARY KEY,
    ProductName         NVARCHAR (40)   NOT NULL,
    SupplierId          "INT",
    CategoryId          "INT",
    QuantityPerUnit     NVARCHAR (20),
    UnitPrice           "MONEY"         CONSTRAINT DF_Products_UnitPrice DEFAULT (0),
    UnitsInStock        "SMALLINT"      CONSTRAINT DF_Products_UnitsInStock DEFAULT (0),
    UnitsOnOrder        "SMALLINT"      CONSTRAINT DF_Products_UnitsOnOrder DEFAULT (0),
    ReorderLevel        "SMALLINT"      CONSTRAINT DF_Products_ReorderLevel DEFAULT (0),
    Discontinued        "BIT"           NOT NULL
                                        CONSTRAINT DF_Products_Discontinued DEFAULT (0),
    CONSTRAINT FK_Products_Categories FOREIGN KEY (CategoryId)
    REFERENCES Categories (CategoryId),
    CONSTRAINT FK_Products_Suppliers FOREIGN KEY (SupplierId)
    REFERENCES Suppliers (SupplierId),
    CONSTRAINT CK_Products_UnitPrice CHECK (UnitPrice >= 0),
    CONSTRAINT CK_ReorderLevel CHECK (ReorderLevel >= 0),
    CONSTRAINT CK_UnitsInStock CHECK (UnitsInStock >= 0),
    CONSTRAINT CK_UnitsOnOrder CHECK (UnitsOnOrder >= 0)
);
```

# Using EF Core annotation attributes to define the model

- In a Product class, we could apply attributes to specify this

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

- An attribute can be use when there isn't an obvious map between .NET types and database types
- E.g., column type of UnitPrice for the Products table is money
  - .NET does not have a money type, so it should use decimal instead

```
[Column(TypeName = "money")]
public decimal? UnitPrice { get; set; }
```

# Using the EF Core Fluent API to define the model

- The last way that the model can be defined is by using the Fluent API
- This API can be used instead of attributes, as well as being used in addition to them
- E.g., to define the ProductName property, instead of decorating the property with two attributes, an equivalent Fluent API statement could be written in the OnModelCreating method of the database context class

```
modelBuilder.Entity<Product>()
    0 references
    .Property(product => product.ProductName)
    0 references
    .IsRequired()
    0 references
    .HasMaxLength(40);
```

# Understanding data seeding with the Fluent API

- Another benefit of the Fluent API is to provide initial data to populate a database
- EF Core automatically works out what insert, update, or delete operations must be executed
- E.g., if we wanted to make sure that a new database has at least one row in the Product table, then we would call the HasData method

```
modelBuilder.Entity<Product>()
    0 references
    .HasData(new Product {
        ProductId = 1,
        ProductName = "Chai",
        UnitPrice = 8.99M
    });
```

# Building EF Core models for the Northwind tables

- Now that you've learned about ways to define EF Core models, let's build models to represent two of the tables in the Northwind database

# Defining the Category and Product entity classes

```
CREATE TABLE Categories (
    CategoryId   INTEGER       PRIMARY KEY,
    CategoryName NVARCHAR (15) NOT NULL,
    Description  "NTEXT",
    Picture      "IMAGE"
);
```

# Defining the Category and Product entity classes

```
using System.ComponentModel.DataAnnotations.Schema; // To use [Column]
namespace Northwind.EntityModels;
0 references
public class Category {
    // These properties map to columns in the database
    0 references
    public int CategoryId { get; set; } // The primary key
    0 references
    public string CategoryName { get; set; } = null!;
    [Column(TypeName = "ntext")]
    0 references
    public string? Description { get; set; }
    // Defines a navigation property for related rows
    0 references
    public virtual ICollection<Product> Products { get; set; }
    // To enable developers to add products to a Category, we must
    // initialize the navigation property to an empty collection
    // This also avoids an exception if we get a member like Count
    = new HashSet<Product>();
}
```

# Defining the Category and Product entity classes

- The Category class will be in the Northwind.EntityModels namespace
- The CategoryId property follows the primary key naming convention, so it will be mapped to a column marked as the primary key with an index
- The CategoryName property maps to a column that does not allow database NULL values so it is a non-nullable string, and to disable nullability warnings, we have assigned the null-forgiving operator
- The Description property maps to a column with the ntext data type instead of the default mapping for string values to nvarchar
- We initialize the collection of Product objects to a new, empty HashSet
- A hash set is more efficient than a list because it is unordered
- If you do not initialize Products, then it will be null and if you try to get its Count then you will get an exception

# Defining the Category and Product entity classes

```
using System.ComponentModel.DataAnnotations; // To use [Required]
using System.ComponentModel.DataAnnotations.Schema; // To use [Column]
namespace Northwind.EntityModels;
0 references
public class Product {
    0 references
    public int ProductId { get; set; } // The primary key
    [Required]
    [StringLength(40)]
    0 references
    public string ProductName { get; set; } = null!;
    // Property name is different from the column name
    [Column("UnitPrice", TypeName = "money")]
    0 references
    public decimal? Cost { get; set; }
    [Column("UnitsInStock")]
    0 references
    public short? Stock { get; set; }
    0 references
    public bool Discontinued { get; set; }
    // These two properties define the foreign key relationship
    // to the Categories table
    0 references
    public int CategoryId { get; set; }
    0 references
    public virtual Category Category { get; set; } = null!;
}
```

# Defining the Category and Product entity classes

- The Product class is used to represent a row in the Products table
- You do not need to include all columns from a table as properties of a class
- Columns that are not mapped to properties cannot be read or set using the class instances
- If the class is used to create new objects, then the new row in the table will have NULL, some default value, for the unmapped column values in that row
- Make sure that those missing columns are optional or have default values set by the database or an exception will be thrown at runtime
- A column may be renamed by defining a property with a different name, like Cost, and then decorating the property with the [Column] attribute and specifying its column name, like UnitPrice
- The final property, CategoryId, is associated with a Category property that will be used to map each product to its parent category

# Adding tables to the Northwind database context class

- Inside your DbContext-derived class, define at least one property of the DbSet<T> type
  - These properties represent the tables
- To tell EF Core what columns each table has, the DbSet<T> properties use generics to specify a class that represents a row in the table
  - That entity model class has properties that represent its columns
- The DbContext-derived class can optionally have an overridden method named OnModelCreating
  - This is where you can write Fluent API statements as an alternative to decorating your entity classes with attributes

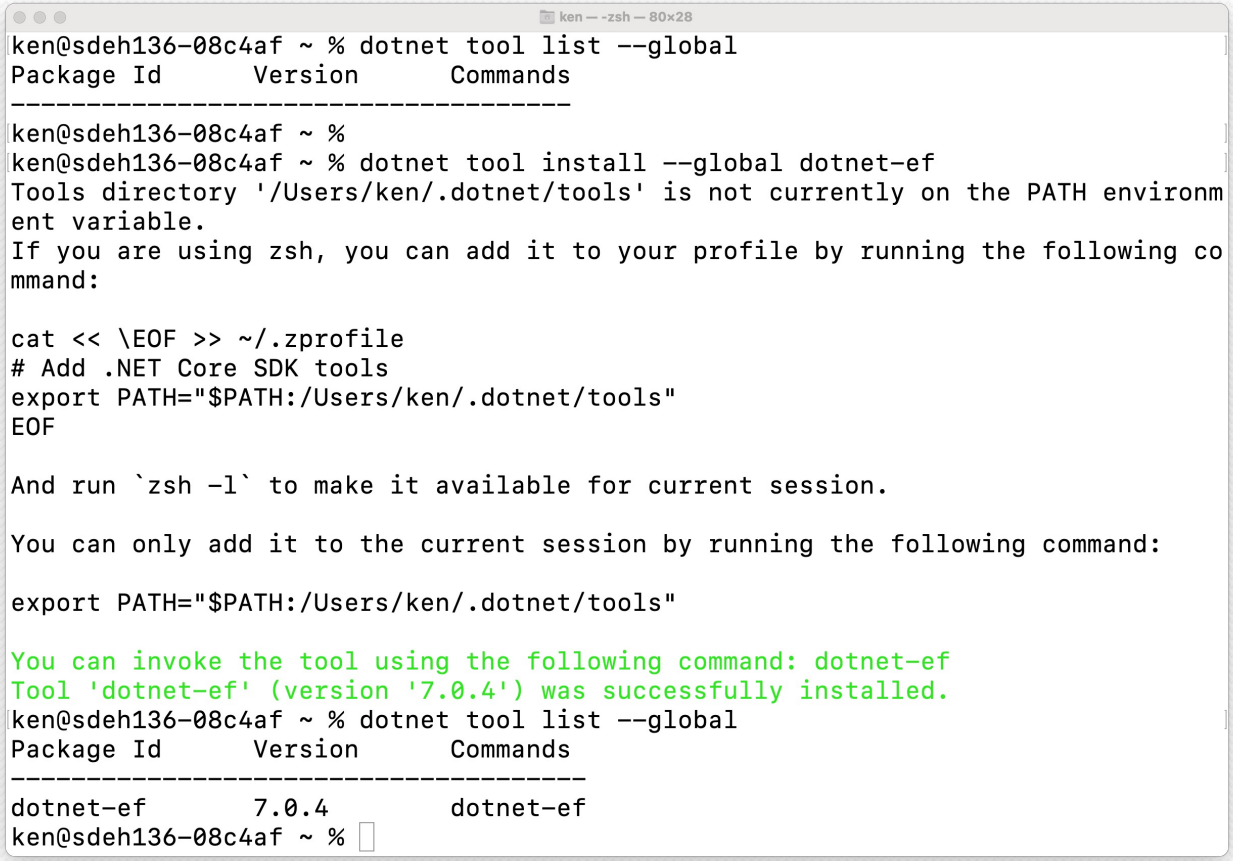
# Adding tables to the Northwind database context class

```
0 references
public class NorthwindDb : DbContext {
    // These two properties map to tables in the database
    0 references
    public DbSet<Category>? Categories { get; set; }
    0 references
    public DbSet<Product>? Products { get; set; }
    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {...}
    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
        // Example of using Fluent API instead of attributes
        // to limit the length of a category name to 15
        modelBuilder.Entity<Category>()
            .Property(category => category.CategoryName)
            .IsRequired() // NOT NULL
            .HasMaxLength(15);
        // Some SQLite-specific configuration
        if (Database.ProviderName?.Contains("Sqlite") ?? false) {
            // To "fix" the lack of decimal support in SQLite
            modelBuilder.Entity<Product>()
                .Property(product => product.Cost)
                .HasConversion<double>();
        }
    }
}
```

# Setting up the dotnet-ef tool

- The .NET CLI tool named dotnet can be extended with capabilities useful for working with EF Core
- It can perform design-time tasks like creating and applying migrations from an older model to a newer model and generating code for a model from an existing database
- The dotnet ef command-line tool is not automatically installed
  - You must install this package as either a global or local tool

# Setting up the dotnet-ef tool



# Scaffolding models using an existing database

- Scaffolding is the process of using a tool to create classes that represent the model of an existing database using reverse engineering
- A good scaffolding tool allows you to extend the automatically generated classes and then regenerate those classes without losing your extended classes
- If you know that you will never regenerate the classes using the tool, then feel free to change the code for the automatically generated classes as much as you want
- The code generated by the tool is just the best approximation

# Scaffolding models using an existing database

```
dotnet ef dbcontext scaffold "Filename=Northwind.db" Microsoft.
EntityFrameworkCore.Sqlite --table Categories --table Products --output-
dir AutoGenModels --namespace WorkingWithEFCore.AutoGen --data-
annotations --context Northwind
```

- The command action: dbcontext scaffold
- The connection string: "Filename=Northwind.db"
- The database provider: Microsoft.EntityFrameworkCore.Sqlite
- The tables to generate models for: --table Categories --table Products
- The output folder: --output-dir AutoGenModels
- The namespace: --namespace WorkingWithEFCore.AutoGen
- To use data annotations as well as the Fluent API: --data-annotations
- To rename the context from [database_name]Context: --context Northwind

# Scaffolding models using an existing database

```
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string,
you should move it out of source code. You can avoid scaffolding the
connection string by using the Name= syntax to read it from configuration
- see https://go.microsoft.com/fwlink/?linkid=2131148. For more
guidance on storing connection strings, see http://go.microsoft.com/
fwlink/?LinkId=723263.
Skipping foreign key with identity '0' on table 'Products' since
principal table 'Suppliers' was not found in the model. This usually
happens when the principal table was not included in the selection set.
```

# Scaffolding models using an existing database

```
namespace WorkingWithEFCore.AutoGen;
[Index("CategoryName", Name = "CategoryName")]
0 references
public partial class Category {
    [Key]
    0 references
    public int CategoryId { get; set; }
    [Column(TypeName = "nvarchar (15)")]
    0 references
    public string CategoryName { get; set; } = null!;
    [Column(TypeName = "ntext")]
    0 references
    public string? Description { get; set; }
    [Column(TypeName = "image")]
    0 references
    public byte[]? Picture { get; set; }
    [InverseProperty("Category")]
    0 references
    public virtual ICollection<Product> Products { get; set; } = new List<Product>();
}
```

# Scaffolding models using an existing database

- It decorates the entity class with the [Index] attribute
- This indicates properties that should have an index when using the Code First approach to generate a database at runtime
- Since we are using Database First with an existing database, this is not needed
- The table name in the database is Categories but the dotnet-ef tool uses the **Humanizer** third-party library to automatically singularize the class name to Category, which is a more natural name when creating a single entity
- The entity class is declared using the partial keyword so that you can create a matching partial class for adding additional code
- This allows you to rerun the tool and regenerate the entity class without losing that extra code

# Scaffolding models using an existing database

- The CategoryId property is decorated with the [Key] attribute to indicate that it is the primary key for this entity
- The data type for this property is int for SQL Server and long for SQLite
- We did not do this because we followed the naming primary key convention
- The CategoryName property is decorated with the [Column(TypeName = "nvarchar (15)")] attribute
  - Only needed if you want to generate a database from the model
- We chose not to include the Picture column as a property because this is a binary object that we will not use in our console app
- The Products property uses the [InverseProperty] attribute to define the foreign key relationship to the Category property on the Product entity class
  - It initializes the collection to a new empty list

# Scaffolding models using an existing database

```
using Microsoft.EntityFrameworkCore;
namespace WorkingWithEFCore.AutoGen;
3 references
public partial class NorthwindDb : DbContext {
    0 references
    public NorthwindDb() { }
    0 references
    public NorthwindDb(DbContextOptions<NorthwindDb> options) : base(options) { }
    0 references
    public virtual DbSet<Category> Categories { get; set; }
    0 references
    public virtual DbSet<Product> Products { get; set; }
    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        #warning To protect potentially sensitive information in your connection string, you should move it out
        of source code. You can avoid scaffolding the connection string by using the Name= syntax to read it
        from configuration - see https://go.microsoft.com/fwlink/?linkid=2131148. For more guidance on storing
        connection strings, see http://go.microsoft.com/ fwlink/?LinkId=723263.
        => optionsBuilder.UseSqlite("Data Source=Northwind.db");
    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
        modelBuilder.Entity<Category>(entity => {
            entity.Property(e => e.CategoryId).ValueGeneratedNever();
        });
        modelBuilder.Entity<Product>(entity =>   {
            entity.Property(e => e.ProductId).ValueGeneratedNever();
            entity.Property(e => e.Discontinued).HasDefaultValueSql("0");
            entity.Property(e => e.ReorderLevel).HasDefaultValueSql("0");
            entity.Property(e => e.UnitPrice).HasDefaultValueSql("0");
            entity.Property(e => e.UnitsInStock).HasDefaultValueSql("0");
            entity.Property(e => e.UnitsOnOrder).HasDefaultValueSql("0");
        });
        OnModelCreatingPartial(modelBuilder);
    }
    1 reference
    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}
```

# Scaffolding models using an existing database

- The NorthwindDb data context class is partial to allow you to extend it and regenerate it in the future
- It has two constructors: a default parameter-less one and one that allows options to be passed in (i.e., connection string)
- In the OnConfiguring method, if options have not been specified in the constructor, then it defaults to using a connection string that looks for the database file in the current folder
- It has a compiler warning to remind you that you should not hardcode security information in this connection string
- In the OnModelCreating method, the Fluent API is used to configure the two entity classes, and then a partial method named OnModelCreatingPartial is invoked
- Allows you to implement that partial method in your own partial Northwind class to add your own Fluent API configuration that will not be lost if you regenerate the model classes

# Querying EF Core models

```
0 references
partial class Program {
    0 references
    private static void ConfigureConsole(string culture = "en-US", bool useComputerCulture = false) {
        // To enable Unicode characters like Euro symbol in the console
        OutputEncoding = System.Text.Encoding.UTF8;
        if (!useComputerCulture) {
            CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo(culture);
        }
        WriteLine($"CurrentCulture: {CultureInfo.CurrentCulture.DisplayName}");
    }
    3 references
    private static void WriteLineInColor(string text, ConsoleColor color) {
        ConsoleColor previousColor = ForegroundColor;
        ForegroundColor = color;
        WriteLine(text);
        ForegroundColor = previousColor;
    }
    0 references
    private static void SectionTitle(string title) {
        WriteLineInColor($"*** {title} ***", ConsoleColor.DarkYellow);
    }
    0 references
    private static void Fail(string message) {
        WriteLineInColor($"Fail > {message}", ConsoleColor.Red);
    }
    0 references
    private static void Info(string message) {
        WriteLineInColor($"Info > {message}", ConsoleColor.Cyan);
    }
}
```

# Querying EF Core models

```
using Microsoft.EntityFrameworkCore; // To use Include method
using Northwind.EntityModels; // To use Northwind, Category, Product
1 reference
partial class Program {
    0 references
    private static void QueryingCategories() {
        using NorthwindDb db = new();
        SectionTitle("Categories and how many products they have");
        // A query to get all categories and their related products
        IQueryable<Category>? categories = db.Categories?.Include(c => c.Products);
        if (categories is null || !categories.Any()) {
            Fail("No categories found.");
            return;
        }
        // Execute query and enumerate results.
        foreach (Category c in categories) {
            WriteLine($"{c.CategoryName} has {c.Products.Count} products.");
        }
    }
}
```

# Querying EF Core models

```
Beverages has 12 products.
Condiments has 12 products.
Confections has 13 products.
Dairy Products has 10 products.
Grains/Cereals has 7 products.
Meat/Poultry has 6 products.
Produce has 5 products.
Seafood has 12 products.
```

# Modifying data with EF Core

- Inserting, updating, and deleting entities using EF Core is an easy task to accomplish. This is often referred to as **CRUD**, an acronym that includes the following operations:
  - **C** for **Create**
  - **R** for **Retrieve** (or **Read**)
  - **U** for **Update**
  - **D** for **Delete**
- By default, DbContext maintains change tracking automatically, so the local entities can have multiple changes tracked, including adding new entities, modifying existing entities, and removing entities
- When you are ready to send those changes to the underlying database, call the SaveChanges method
- The number of entities successfully changed will be returned

# Inserting entities

```
using Microsoft.EntityFrameworkCore; // To use ExecuteUpdate, ExecuteDelete
using Microsoft.EntityFrameworkCore.ChangeTracking; // To use EntityEntry<T>
using Northwind.EntityModels; // To use Northwind, Product
2 references
partial class Program {
    0 references
    private static void ListProducts(int[]? productIdsToHighlight = null) {
        using NorthwindDb db = new();
        if (db.Products is null || !db.Products.Any()) {
            Fail("There are no products.");
            return;
        }
        WriteLine("| {0,-3} | {1,-35} | {2,8} | {3,5} | {4} |",
                  "Id", "Product Name", "Cost", "Stock", "Disc.");
        foreach (Product p in db.Products) {
            ConsoleColor previousColor = ForegroundColor;
            if (productIdsToHighlight is not null && productIdsToHighlight.Contains(p.ProductId)) {
                ForegroundColor = ConsoleColor.Green;
            }
            WriteLine("| {0:000} | {1,-35} | {2,8:$#,##0.00} | {3,5} | {4} |",
                      p.ProductId, p.ProductName, p.Cost, p.Stock, p.Discontinued);
            ForegroundColor = previousColor;
        }
    }
}
```

# Inserting entities

```
0 references
private static (int affected, int productId) AddProduct(
    int categoryId, string productName, decimal? price, short? stock) {
    using NorthwindDb db = new();
    if (db.Products is null) return (0, 0);
    Product p = new() {
        CategoryId = categoryId,
        ProductName = productName,
        Cost = price,
        Stock = stock
    };
    // Set product as added in change tracking
    EntityEntry<Product> entity = db.Products.Add(p);
    WriteLine($"State: {entity.State}, ProductId: {p.ProductId}");
    // Save tracked change to database
    int affected = db.SaveChanges();
    WriteLine($"State: {entity.State}, ProductId: {p.ProductId}");
    return (affected, p.ProductId);
}
```

# Inserting entities

```
State: Added, ProductId: 0
dbug: 05/03/2022 14:21:37.818 RelationalEventId.CommandExecuting[20100]
(Microsoft.EntityFrameworkCore.Database.Command)
      Executing DbCommand [Parameters=[@p0='6', @p1='500' (Nullable =
true), @p2='False', @p3='Bob's Burgers' (Nullable = false) (Size = 13), @
p4=NULL (DbType = Int16)], CommandType='Text', CommandTimeout='30']
      INSERT INTO "Products" ("CategoryId", "UnitPrice", "Discontinued",
"ProductName", "UnitsInStock")
      VALUES (@p0, @p1, @p2, @p3, @p4);
      SELECT "ProductId"
      FROM "Products"
      WHERE changes() = 1 AND "rowid" = last_insert_rowid();
State: Unchanged, ProductId: 78
Add product successful with ID: 78.
| Id  | Product Name                          |     Cost | Stock | Disc. |
| 001 | Chai                                  |   $18.00 |    39 | False |
| 002 | Chang                                 |   $19.00 |    17 | False |
...
| 078 | Bob's Burgers                         |  $500.00 |    72 | False |
```

# Updating entities

```
0 references
private static (int affected, int productId) IncreaseProductPrice(
    string productNameStartsWith, decimal amount) {
    using NorthwindDb db = new();
    if (db.Products is null) return (0, 0);
    // Get the first product whose name starts with the parameter value
    Product updateProduct = db.Products.First(
        p => p.ProductName.StartsWith(productNameStartsWith));
    updateProduct.Cost += amount;
    int affected = db.SaveChanges();
    return (affected, updateProduct.ProductId);
}
0 references
var resultUpdate = IncreaseProductPrice(productNameStartsWith: "Bob", amount: 20M);
if (resultUpdate.affected == 1) {
    0 references
    WriteLine($"Increase price success for ID: {resultUpdate.productId}.");
}
ListProducts(productIdsToHighlight: new[] { resultUpdate.productId });
```

# Updating entities

```
dbug: 05/03/2022 14:44:47.024 RelationalEventId.CommandExecuting[20100]
(Microsoft.EntityFrameworkCore.Database.Command)

      Executing DbCommand [Parameters=[@__productNameStartsWith_0='Bob'
(Size = 3)], CommandType='Text', CommandTimeout='30']
      SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
      FROM "Products" AS "p"
      WHERE NOT ("p"."Discontinued") AND (@__productNameStartsWith_0 =
'' OR (("p"."ProductName" LIKE @__productNameStartsWith_0 || '%') AND
substr("p"."ProductName", 1, length(@__productNameStartsWith_0)) = @__
productNameStartsWith_0) OR @__productNameStartsWith_0 = '')
      LIMIT 1
dbug: 05/03/2022 14:44:47.028 RelationalEventId.CommandExecuting[20100]
(Microsoft.EntityFrameworkCore.Database.Command)

      Executing DbCommand [Parameters=[@p1='78', @p0='520' (Nullable =
true)], CommandType='Text', CommandTimeout='30']
      UPDATE "Products" SET "UnitPrice" = @p0
      WHERE "ProductId" = @p1;
      SELECT changes();
Increase price success for ID: 78.
| Id  | Product Name                          |     Cost | Stock | Disc. |
| 001 | Chai                                  |   $18.00 |    39 | False |
...
| 078 | Bob's Burgers                         |  $520.00 |    72 | False |
```

# Deleting entities

```
private static int DeleteProducts(string productNameStartsWith) {
    using NorthwindDb db = new();
    IQueryable<Product>? products = db.Products?.Where(
        p => p.ProductName.StartsWith(productNameStartsWith));
    if (products is null || !products.Any()) {
        WriteLine("No products found to delete.");
        return 0;
    } else {
        if (db.Products is null) return 0;
        db.Products.RemoveRange(products);
    }
    int affected = db.SaveChanges();
    return affected;
}

WriteLine("About to delete all products whose name starts with Bob.");
Write("Press Enter to continue or any other key to exit: ");
if (ReadKey(intercept: true).Key == ConsoleKey.Enter) {
    int deleted = DeleteProducts(productNameStartsWith: "Bob");
    WriteLine($"{deleted} product(s) were deleted.");
} else {
    WriteLine("Delete was canceled.");
}
```

# Deleting entities

```
1 product(s) were deleted.
```

# Acknowledgements