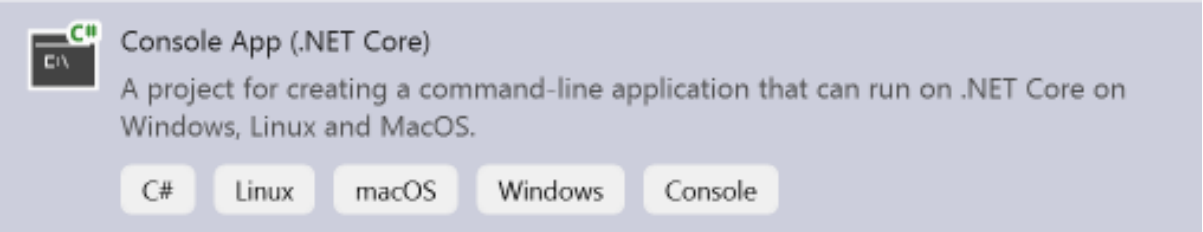
Using Entity Framework (Code-First)
.NET Console Application

Entity Framework is useful when our data is stored in a database and we want to use this data in our .NET Application. It allows us to connect to databases and manipulate our data without writing much database code.

Create an empty console project.

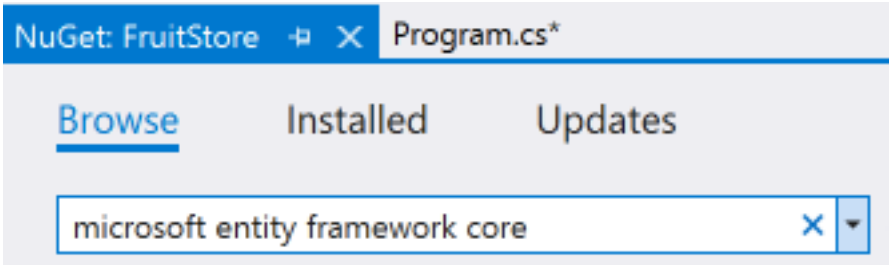


We need to include the EF Core package and use an SQL Server database.

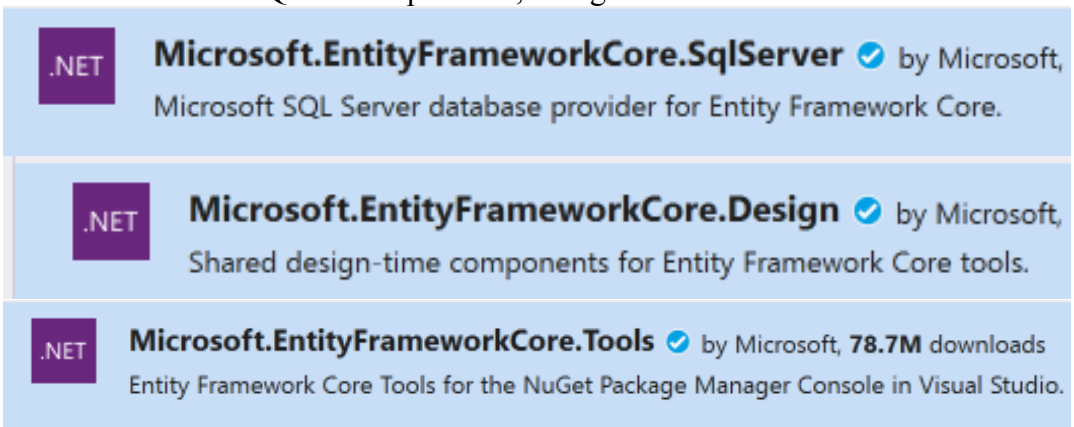
Next Steps:

- Understand a little about EF and how it works
- Understand the concepts of a relational database such as SQL Server

Solution Explorer → Manage NuGet Packages



We will install the SQL Server provider, Design and Core Tools.

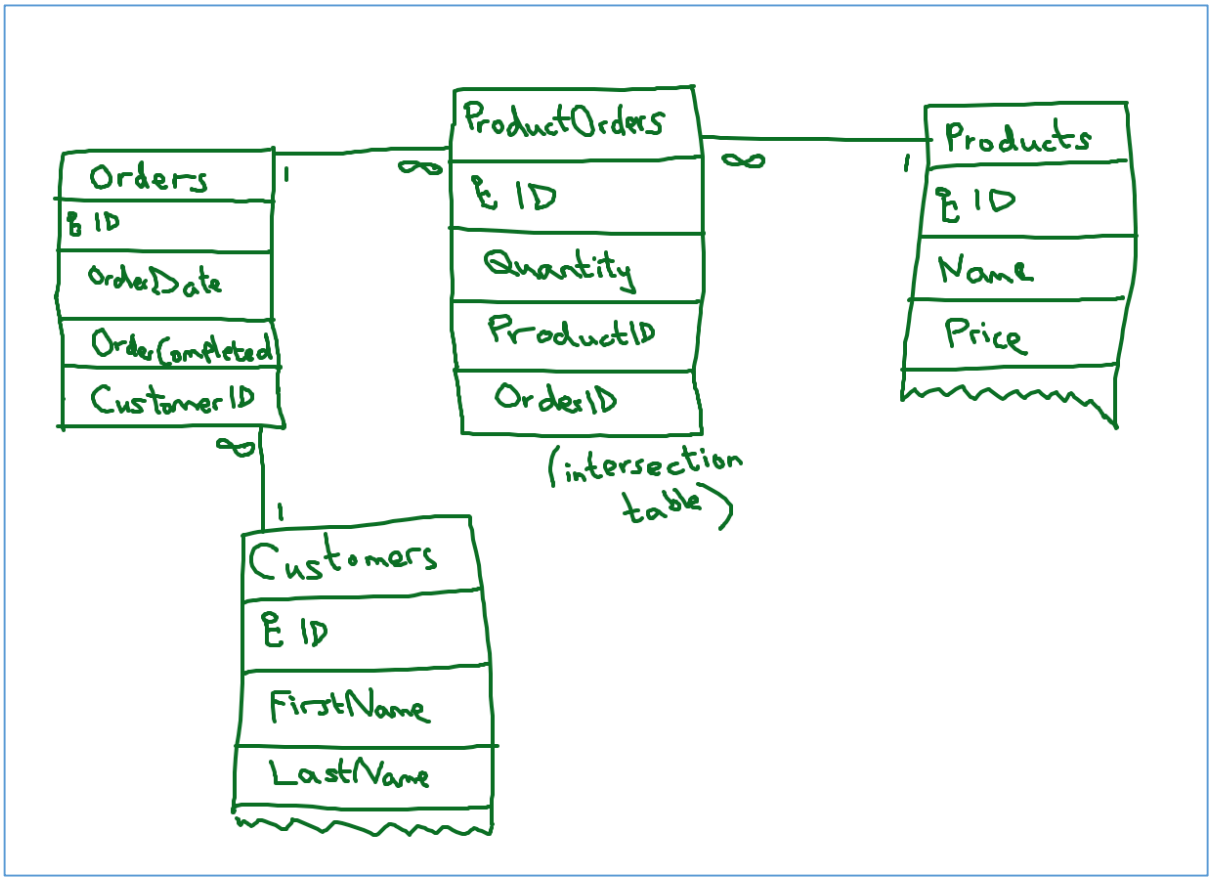


Understanding the Database Structure

We need to understand the database structure that will be used by our application.

We are using a basic Product / Customer / Order database with an intersection table between Products and Orders (Products and Orders will have a many-to-many relationship.

The database structure as discussed has been drawn as follows:



Next Steps:

- Understand relational database concepts.
- Refresh your memory on creating classes and objects in C# with properties.
- Recognise one-to-many and many-to-many relationships.
- Understand the need to create intersection tables in relational databases.
- Be able to use SQL Server to navigate a database (view design, data, etc.

Now we need to create an entity model (the classes that map to the database tables).

Create a Models folder.
Create a Product class and a Customer class in the Models folder.

Product.cs

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;

namespace FruitStore.Models
{
    public class Product
    {
        // Proerties

        // Id or ProductID is the Key [Key]
        public int Id { get; set; }

        [Required]
        public string Name { get; set; }

        [Column(TypeName = "decimal(18,2)")]
        public decimal Price { get; set; }
    }
}
```

Customer.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace FruitStore.Models
{
    public class Customer
    {
        // Properties
        public int ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        // Navigation property (a customer can have zero or more orders)
        public ICollection<Order> Orders { get; set; }

    }
}
```

Next Steps:

- Understand Data Annotations and how they work.

Order.cs

```
using System;
using System.Collections.Generic;

namespace FruitStore.Models
{
    public class Order
    {
        // Properties
        public int ID { get; set; }
        public DateTime OrderDate { get; set; }
        public DateTime? OrderCompleted { get; set; }

        // Foreign key relationship to the Customer table
        public int CustomerID { get; set; }

        // Navigation properties (one customer per order)
        public Customer Customer { get; set; }
        // Navigation properties (to out intersection table)
        public ICollection<ProductOrder> ProductOrders { get; set; }
    }
}
```

ProductOrder.cs (intersection table)

```
namespace FruitStore.Models
{
    public class ProductOrder
    {
        // Properties
        public int Id { get; set; }
        public int Quantity { get; set; }

        // Represent the foreign key relationships
        public int OrderID { get; set; }
        public int ProductID { get; set; }

        // Navigation properties (order and product)
        public Order Order { get; set; }
        public Product Product { get; set; }
    }
}
```

Next Steps:

- Understand the use of nullable fields (such as OrderCompleted above)

Create a Data folder to store our database context class.
A DBContext is like a session to the database.
A DBSet represents a table in the database.

FruitStoreContext.cs

```
using Microsoft.EntityFrameworkCore;
using FruitStore.Models;

using System;
using System.Collections.Generic;
using System.Text;

namespace FruitStore.Data
{
    class FruitStoreContext: DbContext
    {
        // Inherits from DbContext class (represents session with the database)

        // Properties (represent the tables in the database)
        public DbSet<Customer> Customers { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<Product> Products { get; set; }
        public DbSet<ProductOrder> ProductOrders { get; set; }

        // Override this method to use the SQL Server extension method (hard code the db location for now)
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=FruitStoreContext");
        }
    }
}
```

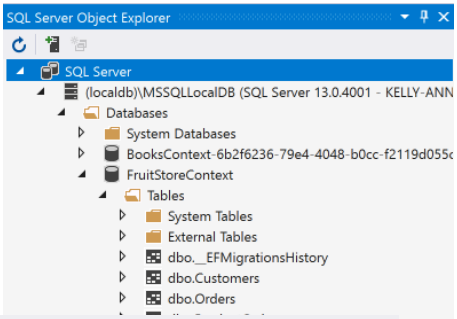
Now we handle the migration to create the database.

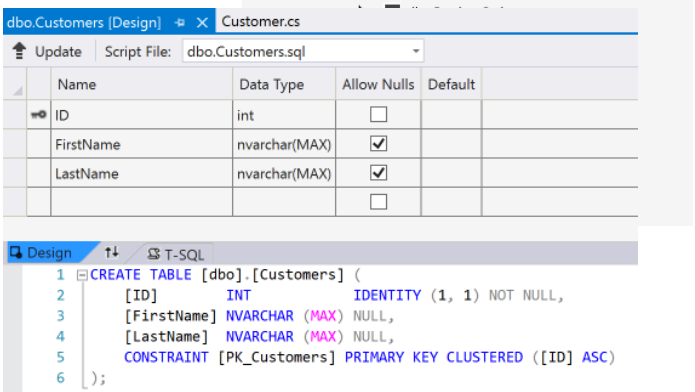
In our Package Manager Console we create the database using Add-Migration
We give our initial migration a name (InitialCreate is what is normally used).
PM> Add-Migration InitialCreate

We can see the code that has been generated in the Migrations folder. Then we can run that migration using the Update-Database command.

PM> Update-Database
This will create the database for us.

We can see this using
SQL Server Object Explorer





Next Steps:

- Understand Migration with EF Code-First

Making A Change To Our Database Model

Now we have the database, and the classes that map to them.
So the problem is if we have to make a change to the structure, where do we handle that?

Let's modify the Customer class.
We forgot to add fields for the customer's address.

```
    public class Customer
    {
#nullable enable
        // Properties
        public int ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        // Adding these properties that we missed earlier
        public string? HomeAddress { get; set; }
        public string? EmailAddress { get; set; }
#nullable disable
        // Navigation property (a customer can have zero or more orders)
        public ICollection<Order> Orders { get; set; }

    }
```

Now we can create a new migration for these changes and update the database

PM> Add-Migration AddressAndEmail
PM> Update-Database

Next Steps:

- Understand how nullable properties/fields work.

The next set of notes will show how we can add, edit and retrieve data from the database.

Adding Data To Our Database

For now we will just write some code in our Program.cs to add some products to the database, so these products are added when we run our application.

Program.cs

```
Program.cs
FruitStore                    FruitStore.Program                    Main(string[] args)
 1  using FruitStore.Data;
 2  using FruitStore.Models;
 3  using System;
 4
 5  namespace FruitStore
 6  {
 7      class Program
 8      {
 9          static void Main(string[] args)
10          {
11              Console.WriteLine("Fabulous Fruits");
12              // Adding some products to the database table
13
14              // This makes sure that the context is disposed off when we finish with it
15              using FruitStoreContext context = new FruitStoreContext();
16
17              // Create a product
18              Product galaApple = new Product()
19              {
20                  Name = "Gala apple",
21                  Price = 0.95M
22              };
23              // Add to the products table
24              context.Products.Add(galaApple);
25
26              // Create another product
27              Product banana = new Product()
28              {
29                  Name = "Banana",
30                  Price = 0.50M
31              };
32              // Add it to the database (EF knows it is a product)
33              context.Add(banana);
34
35              // Save the changes
36              context.SaveChanges();
37
38              Console.WriteLine("Saved products to the database as required.");
39          }
40      }
41  }
```

After we run our application we can use SQL Server Object Explorer to view the Products table, and we can see that the products were added to the correct database table.

dbo.Products [Data] | dbo.Products [Design] | Program.cs

Max Rows: 1000

Id	Name	Price
1	Gala apple	0.95
2	Banana	0.50

Reading Data From Our Database

Assuming we have products in our database table as shown,
We can use LINQ to query the database and a foreach loop to print the data.

dbo.Products [Data] | Program.cs

Max Rows: 1000

Id	Name	Price
1	Gala apple	0.95
2	Banana	0.50
3	Orange	0.70
4	Granny Smith a...	0.50
5	Fuji apple	0.65
7	Barlett pear	0.82

```
using FruitStore.Data;
using FruitStore.Models;
using System;
using System.Linq;

namespace FruitStore
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Fabulous Fruits!");
            // Retrieving data from our database tables

            using FruitStoreContext context = new FruitStoreContext();

            // Using LINQ (query syntax) to query the products
            var cheapProducts = from product in context.Products
                                where product.Price < 0.75m
                                orderby product.Name
                                select product;

            // Using LINQ (method syntax) to do the same thing
            // var cheapProducts = context.Products.Where(p => p.Price <= 0.75M).OrderBy(p => p.Name);

            Console.WriteLine("Fruits Costing Less Than 0.75 Are :");

            foreach (Product p in cheapProducts)
            {
                // Print the data from the
                Console.WriteLine($"ID:  { p.Id}");
                Console.WriteLine($"Name:  { p.Name}");
                Console.WriteLine($"Price:  { p.Price}");
                Console.WriteLine();
            }
        }
    }
}
```

Microsoft Visual Studio Debug Console

```
Fabulous Fruits!
Fruits Costing Less Than 0.75 Are :
ID:  2
Name:  Banana
Price:  0.50

ID:  5
Name:  Fuji apple
Price:  0.65

ID:  4
Name:  Granny Smith apple
Price:  0.50

ID:  3
Name:  Orange
Price:  0.70
```

Editing Data In Our Database

We can use LINQ to query the database and return the product we want to edit.
We can then edit the data required, and save the changes.
Now our query on products costing 0.75 or less will include the Gala apple.

```
Console.WriteLine("Fabulous Fruits!");
// Editing data in our database tables

using FruitStoreContext context = new FruitStoreContext();

// Using LINQ (method syntax) to get the product that matches this name (or null).
var galaApple = context.Products.Where(p => p.Name == "Gala apple").FirstOrDefault();
// Edit the data stored in the database table (if a product was returned)
if (galaApple is Product)
    galaApple.Price = 0.65M;
// Save changes
context.SaveChanges();
```

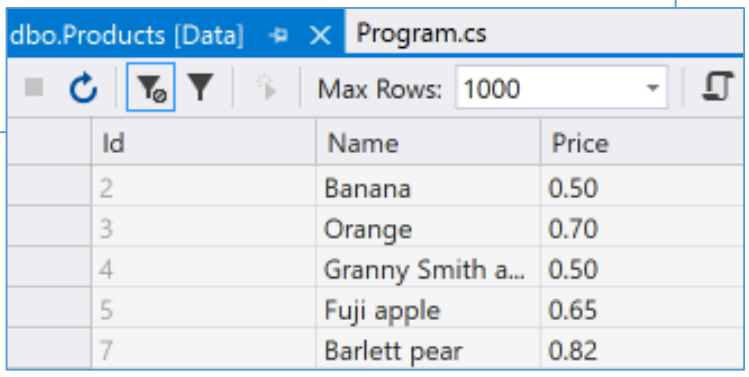
Deleting A Product From Our Database

Again we can use LINQ to query the database and return the product we want to delete.
Then we use the Remove method on the context.
We can see that the Gala apple product is no longer in our database table.

```
using FruitStoreContext context = new FruitStoreContext();

// Using LINQ (method syntax) to get the product that matches this name (or null).
var galaApple = context.Products.Where(p => p.Name == "Gala apple").FirstOrDefault();

// Delete the product from the database table (if a product was returned)
if (galaApple is Product)
    context.Remove(galaApple);
// Save changes
context.SaveChanges();
```



Id	Name	Price
2	Banana	0.50
3	Orange	0.70
4	Granny Smith a...	0.50
5	Fuji apple	0.65
7	Barlett pear	0.82