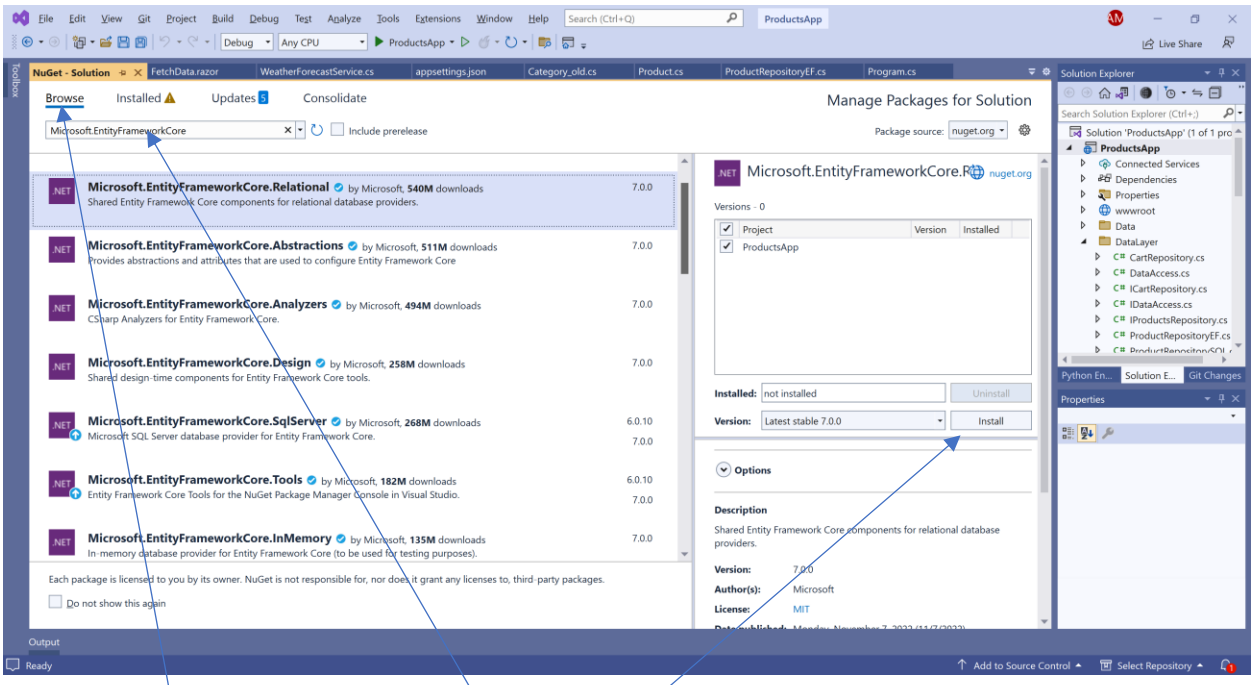
CPSC 555 – Assignment #5 – Fall 2022

The purpose of this assignment is to understand the database programming in server-side Blazor applications using EntityFramework Core .Net 7. As explained in the lectures, Entity Framework (EF) makes the interaction with the database extremely easy and at an object-oriented level. The key concepts is that there is a database context class that is generated through which we can query data, and do insert, updates and deletes as if we are operating on lists. Behind the scenes, the EF generates the proper SQL to issue to the database. To demonstrate the use of EF, you will be recreating the SQL based ProductsApp by instead using EF.

Version 7 of EF has been released, so first upgrade your Visual Studio to the latest version by checking the Help menu and selecting “Check for Updates”. If it indicates a new version is available, update your Visual Studio.

To be able to use the entity framework in the ProductsApp, you will need to install three Nuget packages. These are:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools



From the Tools menu, select “Nuget Package Manager -> Manage Nuget Packages for Solution”, then click on the browse link and type EntityFrameworkCore in the search box. Select the package needed and check the check box for the project to install it.

The creation of automatically generated classes (via the scaffold-dbcontext command) corresponding to the database and the database context class requires that there should be no errors in the project.

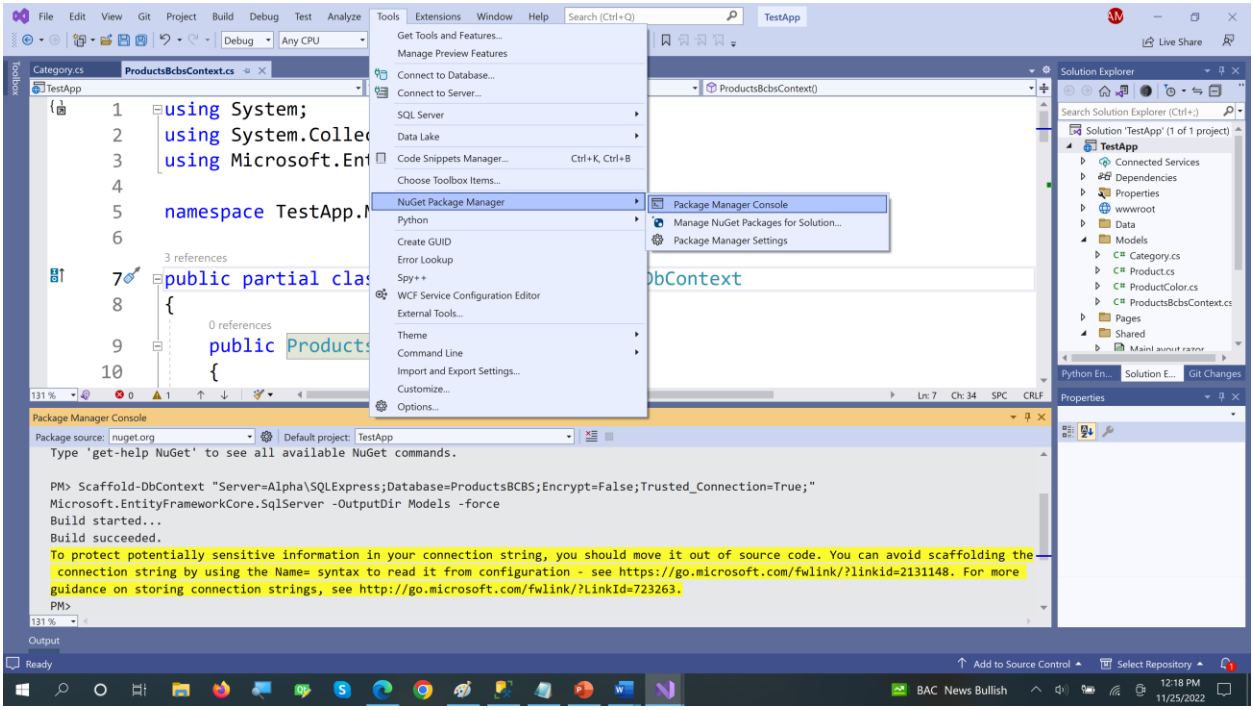
Some times as we modify the fields in the database, and we need to recreate the database related classes, the scaffold-dbcontext will not run successfully if the project code has errors because of field name mismatches. Thus it is some times easier to create an additional temporary Blazor server side project just for the purpose of generating EF classes via the scaffold-dbcontext command. Once the classes are generated in the temporary project, we can copy these to our ProductsApp project.

Create a new Blazor server side project called TestAPP (e.g., in the temp folder). Add the three packages to it.

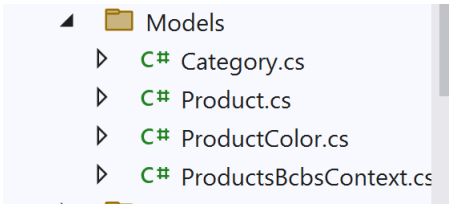
- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools

Then add a Models folder to the TestApp project. From the Tools menu, choose Nuget Package Manager and then “Package Manager Console”. From the package manager console prompt, issues the following command (all on one line). Replace the name of SQL server with your SQL server.:

Scaffold-DbContext
"Server=Alpha\SQLExpress;Database=ProductsBCBS;Encrypt=False;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -force



This will end up generating the following classes in the Models folder of the TestApp project.



Once you hit enter, it will generate the EF related database classes including a class mapping each table and a database context class. If the name of your database is ProductsBcbs, the database context class will be called ProductsBcbsContext. The classes generated in the Models folder belong to a namespace TestApp.Models. As we copy these to the ProductsApp proect we will need to change the namespace in these four files to ProductsApp.Models.

Open the ProductsApp project in Visual Studio. Rename the Category, Product files in the Models folder to Category_old, Product_old and comment out the entire contents of these two files. Then right click on the Models folder in the ProductsApp proect and choose “Add Existing Item”. Then browse to the Models folder of the TestApp, and by holding the control key, select the four files of Category, Product, ProductColor and ProductsBcbsContext and select Add. Change the namespace of each of these four files from TestApp.Models to ProductsAp.Models e.g., the Category.cs file should appear as:

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

namespace ProductsApp.Models;

public partial class Category
{
    public int CategoryId { get; set; }

    public string CategoryName { get; set; } = null!;

    public virtual ICollection<Product> Products { get; } = new List<Product>();
}
```

Now we will create the ProductRepositoryEF class that implement the same interface as before i.e., IProductsRepository. Right click on the DataLayer folder and add a class called ProductRepositoryEF with the following code in it.

```
using Microsoft.EntityFrameworkCore;
using ProductsApp.Models;

namespace ProductsApp.DataLayer
{
    public class ProductRepositoryEF : IProductsRepository
    {
        ProductsBcbsContext _context;
        public ProductRepositoryEF(ProductsBcbsContext context)
        {
            _context = context;
        }

        public bool AddProduct(Product prod)
        {
            Product pr = _context.Products.FirstOrDefault(p => p.ProductId ==
prod.ProductId);
            if (pr == null)
            {
                _context.Products.Add(prod);
                _context.SaveChanges();
```

```
                return true;
            }
            else
                return false;
        }

        public bool ApplyDiscount(int prodid, double percentDiscount)
        {
            Product pr = _context.Products.FirstOrDefault(p => p.ProductId ==
prodid);
            if (pr != null)
            {
                pr.Price = pr.Price - pr.Price * (decimal)percentDiscount /
100.0m;
                _context.Entry(pr).State =
Microsoft.EntityFrameworkCore.EntityState.Modified;
                _context.SaveChanges();
                return true;
            }
            else
                return false;
        }

        public bool DeleteProduct(int prodid)
        {
            Product pr = _context.Products.FirstOrDefault(p => p.ProductId ==
prodid);
            if (pr != null)
            {
                _context.Products.Remove(pr);
                _context.SaveChanges();
                return true;
            }
            else
                return false;
        }

        public List<Category> GetCategories()
        {
            return _context.Categories.ToList<Category>();
        }

        public Product GetProductById(int prodid)
        {
            return _context.Products.Include(p => p.PcolorNavigation).Where(p
=> p.ProductId == prodid).FirstOrDefault<Product>();
        }

        public List<Product> GetProductsByCatId(int catid)
        {
            return _context.Products.Include(p => p.PcolorNavigation).Where(p
=> p.CategoryId == catid).ToList<Product>();

        }
```

```
        public bool UpdateProduct(Product prod)
        {
            Product pr = _context.Products.FirstOrDefault(p => p.ProductId ==
prod.ProductId);
            if (pr != null)
            {
                _context.Entry(pr).State = EntityState.Detached;
                _context.Products.Update(prod);
                _context.SaveChanges();
                return true;
            }
            else
                return false;
        }
    }
}
```

Modify the Program.cs file to appear as (modifications and lines to be added are shown in bold):

```
var builder = WebApplication.CreateBuilder(args);
var connstr = builder.Configuration.GetConnectionString("ProductsDBConn");
ConnectionStringHelper.CONNSTR = connstr;

builder.Services.AddDbContext<ProductsBcbsContext>(options =>
options.UseSqlServer(builder.Configuration["ConnectionStrings:ProductsDBConn"])
);
builder.Services.AddScoped<IProductsRepository, ProductRepositoryEF>();
//builder.Services.AddScoped<IProductsRepository, ProductRepositorySQL>();
// this will make ProductRepositorySQL available to all pages via @inject statement
// or any class via constructor injection
```

The above provides dependency injection for ProductRepositoryEF.

Build and test the application. All pages will behave as before but now they use the Entity Framework based repository to talk to the database.