

## Introduction

This document will provide a walk-through of the database first approach using Entity Framework Core. The other document that had a walk through of the code first approach used a multi-layer architecture. In this walk-through we will flatten the architecture by using only two projects with the understanding that we could use the same four project solution. The database we will use for this walk-through is the Inventory database but any database can be used, just change the name in the connection string when scaffolding the database.

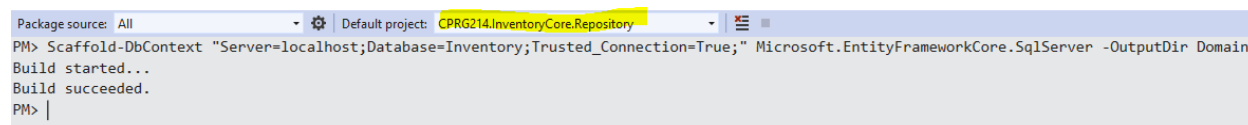
## Step 1 - Setup the Data Project

The solution to be created will start with the data project, a .NET Core class library called **CPRG214.InventoryCore.Repository** and you can rename the solution **CPRG214.Inventory**

An application project will be added later for testing.

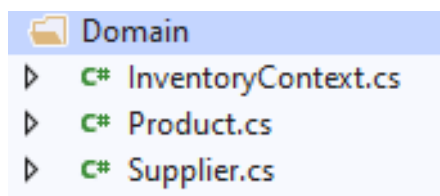
## Step 2 – Scaffold the Entity Data Model

1. Using the Nuget Package Manager (Tools menu), find and install the “Microsoft.EntityFrameworkCore.SqlServer” package to the Repository class library project. Use the same version as you .NET Core version.
2. In the Package Manager Console (Tools menu), set the Repository project as the default project.
3. Install the EF Core tools so run this command at the PM prompt (same .NET Core version!):  
`Install-Package Microsoft.EntityFrameworkCore.Tools -Version 3.1.2`
4. In the Package Manager Console, run the Scaffold-DbContext command. The connection string may be different if you are running this in the lab or on your own laptop and using SQLEXPRESS.



```
Package source: All | Default project: CPRG214.InventoryCore.Repository
PM> Scaffold-DbContext "Server=localhost;Database=Inventory;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Domain
Build started...
Build succeeded.
PM> |
```

5. A folder called Domain will be added to the project containing 3 classes: InventoryContext, Supplier and Product. View the generated code on the following page.



The supplier Code:

```
public partial class Supplier
{
    0 references
    public Supplier()
    {
        Product = new HashSet<Product>();
    }

    1 reference
    public int Id { get; set; }
    1 reference
    public string Name { get; set; }

    2 references
    public virtual ICollection<Product> Product { get; set; }
}
```

The Product Code:

```
public partial class Product
{
    1 reference
    public int Id { get; set; }
    1 reference
    public string Name { get; set; }
    0 references
    public int Quantity { get; set; }
    1 reference
    public decimal Price { get; set; }
    2 references
    public int SupplierId { get; set; }

    1 reference
    public virtual Supplier Supplier { get; set; }
}
```

t

The InventoryContext class is shown on the following page. Note the connection string in the OnConfiguring method. This can be deleted as we will put the connection string in the appsettings.json file of the UI project that uses this class library.

```
public partial class InventoryContext : DbContext
{
    0 references
    public InventoryContext()
    { }

    0 references
    public InventoryContext(DbContextOptions<InventoryContext> options)
        : base(options)
    { }

    0 references
    public virtual DbSet<Product> Product { get; set; }
    0 references
    public virtual DbSet<Supplier> Supplier { get; set; }

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            optionsBuilder.UseSqlServer("Server=localhost;Database=Inventory;Trusted_Connection=True;");
        }
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>(entity =>
        {
            entity.Property(e => e.Id).HasColumnName("ID");

            entity.Property(e => e.Name)
                .IsRequired()
                .HasMaxLength(50)
                .IsUnicode(false);

            entity.Property(e => e.Price).HasColumnType("decimal(8, 2)");

            entity.Property(e => e.SupplierId).HasColumnName("SupplierID");

            entity.HasOne(d => d.Supplier)
                .WithMany(p => p.Product)
                .HasForeignKey(d => d.SupplierId)
                .OnDelete(DeleteBehavior.ClientSetNull)
                .HasConstraintName("FK_Product_Supplier");
        });

        modelBuilder.Entity<Supplier>(entity =>
        {
            entity.Property(e => e.Id).HasColumnName("ID");

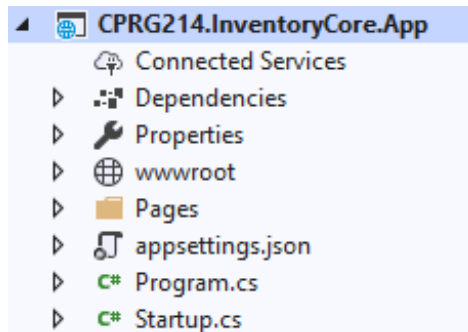
            entity.Property(e => e.Name)
                .IsRequired()
                .HasMaxLength(50)
                .IsUnicode(false);
        });

        OnModelCreatingPartial(modelBuilder);
    }

    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}
```

## Step 3 - Setup the UI Project (Web Application)

1. Start by adding an ASP.NET Core Web Application called CPRG214.InventoryCore.App to the solution. Choose Web Application (using Razor pages) and set it as the startup project. The process is similar if using the Web Application (Model-View-Controller) project.



2. Add the connection string to the appsettings.json:

```
"ConnectionStrings": {  
  "InventoryConnection": "Server=localhost;Database=Inventory;Trusted_Connection=True;"  
}
```

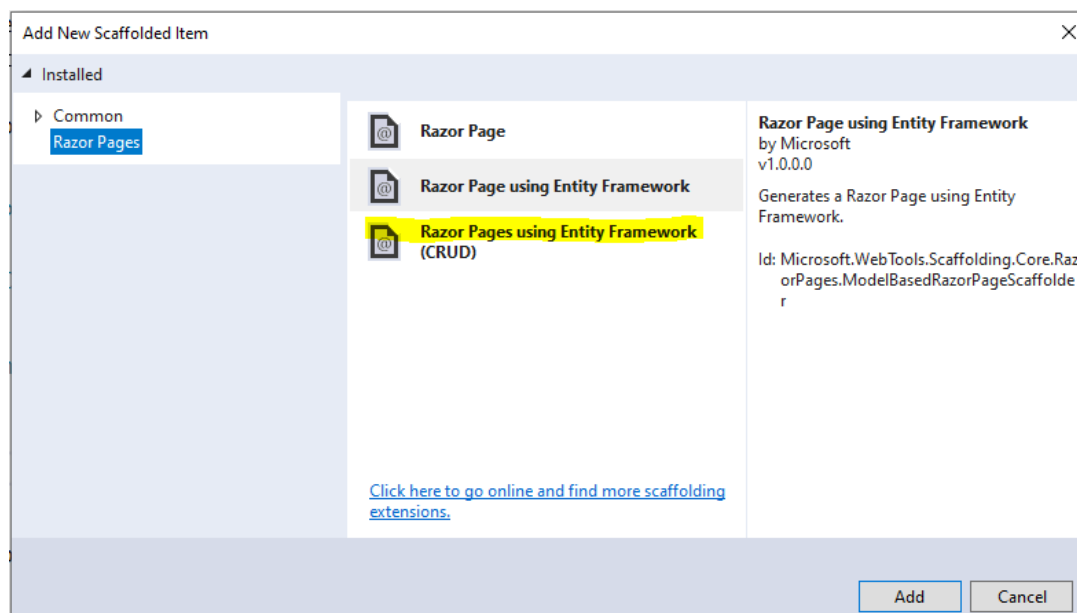
3. Right click "Dependencies" in the UI project and add project reference to the Data project.
4. Add the DbContext service in the ConfigureServices method of the Startup class. This is going to enable dependency injection (DI).

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<InventoryContext>(options =>  
        options.UseSqlServer(Configuration.GetConnectionString("InventoryConnection")));  
    services.AddRazorPages();  
}
```

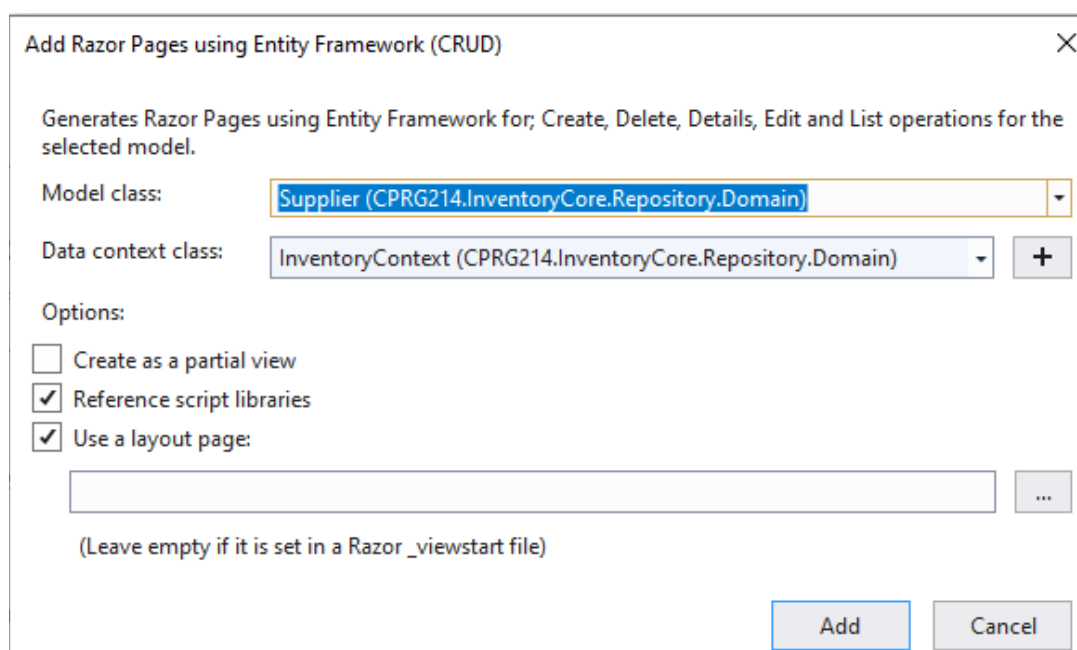
5. Import 2 namespaces to resolve the "red squiggles"

```
using CPRG214.InventoryCore.Repository.Domain;  
using Microsoft.EntityFrameworkCore;
```

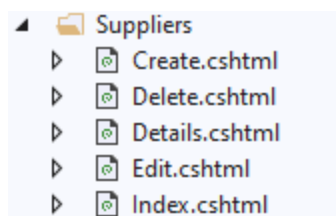
6. The Context is now ready to be used in the application and can be injected into any constructor in the UI project. The easy way to demonstrate this is to add a folder and name it with the name of the model class you are going to create pages for (e.g. Suppliers). Add a new item to this folder and select Razor Page. In the following dialog, you have options of just a Razor Page or the Page with Entity Framework enabled or all CRUD pages created. Select the latter and click "Add".



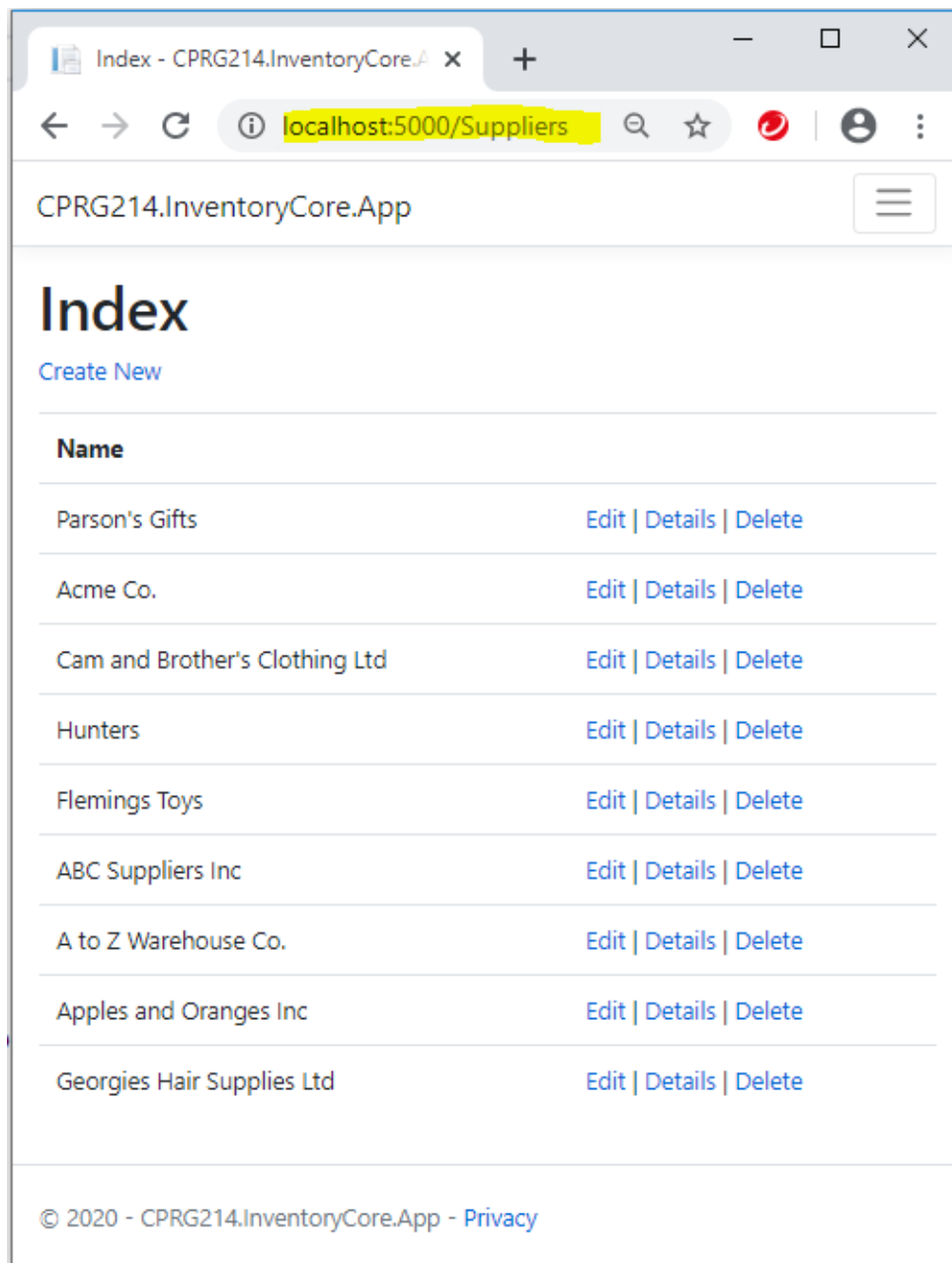
7. Select the Model class (e.g. Supplier) and select InventoryContext as the Data context class if it is not already displaying and click “Add”. If it asks you if you want to replace Index file as it already exists just answer Yes. It is only replacing the existing Home page.




Razor pages are created:



8. Run the application (note the URL – it may be different if using IIS Express):



The create page for the Supplier:

CPRG214.InventoryCore.App 

---

## Create

### Supplier

---

Name

Create

[Back to List](#)

**NOTE:** You need to know what version of .NET Core you are using because different versions can run side-by-side. If you are using version 3.1.2 then install version 3.1.2 of other .NET Core assemblies. When a class library is version 3.1.2 then use version 3.1.2 for the other projects.