Build an EF and ASP.NET Core 2 App HOL

Welcome to the Build an Entity Framework Core and ASP.NET Core 2 Application in a Day Hands On Lab. This lab walks you through configuring the pipeline, setting up configuration, and dependency injection.

Prior to starting this lab, you must have completed Lab 2.

All labs and files are available at https://github.com/skimedic/dotnetcore_hol.

Part 1: Create the CustomSettings class

1) Add a new folder named Support in the MVC project. In that folder, add a new class named CustomSettings.cs. This file will be used to hold configuration information.

```
public class CustomSettings
{
  public CustomSettings() { }
  public string MySetting1 { get; set; }
  public int MySetting2 { get; set; }
}
```

Part 2: Update the Startup.cs class

Step 1: Update the using statements

2) Update the using statements to the following:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using SpyStore_HOL.DAL.EfStructures;
using SpyStore_HOL.DAL.EfStructures.Initialization;
using SpyStore_HOL.DAL.Repos;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.MVC.Support;
```

Step 2: Add Items to the Dependency Injection Container

1) Open the Startup.cs file and navigate to the ConfigureServices method

2) EF Core support is added to the ASP.NET Core DI Container using the built-in AddDbContextPool method. This method constructs the DbContext using the constructor that takes an instance of DbContextOptions. Add the following code after the services.MVC() call:

```
services.AddDbContextPool<StoreContext>(options => options
  .UseSqlServer(Configuration.GetConnectionString("SpyStore"),o=>o.EnableRetryOnFailure())
  .ConfigureWarnings(warnings=>warnings.Throw(RelationalEventId.QueryClientEvaluationWarning)));
```

3) Next add all of the repos into the DI container:

```
  services.AddScoped<ICategoryRepo, CategoryRepo>();
  services.AddScoped<IProductRepo, ProductRepo>();
  services.AddScoped<ICustomerRepo, CustomerRepo>();
  services.AddScoped<IShoppingCartRepo, ShoppingCartRepo>();
  services.AddScoped<IOrderRepo, OrderRepo>();
  services.AddScoped<IOrderDetailRepo, OrderDetailRepo>();
```

4) Finally, add the following code that uses the configuration file to create the CustomSettings class when requested by another class:

```
services.Configure<CustomSettings>(Configuration.GetSection("CustomSettings"));
```

Step 3: Call the Data Initializer in the Configure method

1) Navigate to the Configure method.
2) Update the code block in the `IsDevelopment` if block:

```
if (env.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
  app.UseBrowserLink();
  using (var serviceScope = app.ApplicationServices
    .GetRequiredService<IServiceScopeFactory>().CreateScope())
  {
    StoreDataInitializer
      .InitializeData(serviceScope.ServiceProvider.GetRequiredService<StoreContext>());
  }
}
else
{
  app.UseExceptionHandler("/Home/Error");
}
```

Part 3: Use the DI Container

Step 1: Add the controllers

1) In the Controllers directory of the MVC app, add three new controller classes:
   CartController.cs
   OrdersController.cs
   ProductsController.cs
   NOTE: You can use the Add Controller wizard or just add regular classes.

Step 2: Update the CartController

1) Update the using statements to the following:

```
using System;
using System.Collections.Generic;
using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels;
using SpyStore_HOL.Models.ViewModels.Base;
```

2) Make the class public and inherit from Controller:

```
public class CartController : Controller
{
}
```

3) Add a constructor that takes an instance of IShoppingCartRepo and a private variable to hold the instance.
   NOTE: This will be automatically populated by the DI container. The StoreContext is also automatically populated by the DI container.

```
private readonly IShoppingCartRepo _shoppingCartRepo;
readonly MapperConfiguration _config = null;
public CartController(IShoppingCartRepo shoppingCartRepo)
{
  _shoppingCartRepo = shoppingCartRepo;
}
```

4) Create a method named Index that takes an ICustomerRepo. When leveraging the DI container in a method, you use the FromServices attribute:

```
public IActionResult Index([FromServices] ICustomerRepo customerRepo, int customerId)
{
 return null;
}
```

5) Create a method named AddToCart that takes an IProductRepo.

```
public IActionResult AddToCart([FromServices] IProductRepo productRepo,int customerId, int
productId, bool cameFromProducts = false)
{
  return null;
}
```

Step 3: Update the OrdersController

1) Update the using statements to the following:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels;
```

2) Make the class public and inherit from Controller:

```
public class OrdersController : Controller
{
}
```

3) Add a constructor that takes an instance of IOrdersRepo and a private variable to hold the instance. NOTE: This will be automatically populated by the DI container

```
private readonly IOrderRepo _orderRepo;
public OrdersController(IOrderRepo orderRepo)
{
  _orderRepo = orderRepo;
}
```

Step 4: Update the ProductsController

1) Update the using statements to the following:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.MVC.Support;
```

2) Make the class public and inherit from Controller:

```
public class ProductsController : Controller
{
}
```

3) Add a constructor that takes an instance of IProductRepo, IOptionsSnapShot<CustomSettings>, and ILogger:

```
private readonly IProductRepo _productRepo;
private readonly CustomSettings _settings;
private ILogger Logger { get; }

public ProductsController(
  IProductRepo productRepo,
  IOptionsSnapshot<CustomSettings> settings,
  ILogger<ProductsController> logger)
{
  _settings = settings.Value;
  _productRepo = productRepo;
  Logger = logger;
}
```

Part 4: Configure the Application

Step 1: Add the connection string to the development settings

4) Update the `appsettings.Development.json` to the following (adjusted for your machine's setup):

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "ConnectionStrings": {
    "SpyStore":
"Server=(localdb)\\mssqllocaldb;Database=SpyStore_HOL2;Trusted_Connection=True;MultipleActiveResul
tSets=true"
  }
}
```

Step 2: Add the connection string to the production settings

1) Add a new json file named appsettings.Production.json.
   Note: You can right click on the project – select Add -> New Item -> JSON File.

2) Update the file to the following:

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "None"
    }
  },
  "ConnectionStrings": {
    "SpyStore": "Production connection string"
  }
}
```

3) If you change the environment to Production, the app will fail on the data initialization since the connection string is not valid.

Step 3: Add the Custom Settings to AppSettings.JSON

The CustomSettings class is populated using a configuration section named “CustomSettings”.

1) Open appsettings.json.
2) Update the file to the following:

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "CustomSettings": {
    "MySetting1": "Foo",
    "MySetting2": 5
  }
}
```

Summary

This lab added the necessary classes into the DI container and modified the application configuration.

Next steps

In the next part of this tutorial series, you will add the ViewModels and Controllers.