Build an EF and ASP.NET Core 2.2 App HOL

Lab 6

This lab walks you through configuring the pipeline, setting up configuration, and dependency injection. Prior to starting this lab, you must have completed Lab 5.

Part 1: Configure the Application

Step 1: Add the connection string to the development settings

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

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "ConnectionStrings": {
    "SpyStore": "Server=.,6433;Database=SpyStoreHol;User
ID=sa;Password=P@ssw0rd;MultipleActiveResultSets=true;"
  }
}
```

2) [SQL 2017 LocalDb] Update the `appsettings.Development.json` to the following (adjusted for your machine's setup):

```
  "ConnectionStrings": {
    "SpyStore": "Data
Source=(localdb)\\mssqllocaldb;Database=SpyStoreHOL;AttachDbFileName={path}\\SpyStoreHOL.mdf;Trust
ed_Connection=True;MultipleActiveResultSets=true",
  }
}
```

Step 2: Add the connection string to the production settings

1) Add a new JSON file to the `SpyStore.Hol.Mvc` project named `appsettings.Production.json`. Update the file to the following (this will cause the app to fail in production since the connection string is invalid):

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

Step 3: Add the Custom Settings to AppSettings.json

1) Open `appsettings.json` and update it to the following:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "CustomSettings": {
    "MySetting1": "Foo",
    "MySetting2": 5
  }
}
```

Part 2: 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. The `CustomSettings` class is populated using the "CustomSettings" configuration section.

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

Part 3: Update the Startup.cs class

Step 1: Update the using statements

1) Update the using statements to the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using SpyStore.Hol.Dal.EfStructures;
using SpyStore.Hol.Dal.Initialization;
using SpyStore.Hol.Dal.Repos;
using SpyStore.Hol.Dal.Repos.Interfaces;
using SpyStore.Hol.Mvc.Support;
```

Step 2: Comment out the GDPR Cookie Code in Startup.cs

Without authentication, the cookie window will pop up every time the app is started.

1) Comment out the following in the `ConfigureServices` method:

```
services.Configure<CookiePolicyOptions>(options =>
{
  // This lambda determines whether user consent for non-essential cookies is needed for a given
request.
  options.CheckConsentNeeded = context => true;
  options.MinimumSameSitePolicy = SameSiteMode.None;
});
```

2) Comment out the following in the `Configure` method:

```
app.UseCookiePolicy();
```

Step 3: Add a Class Level Variable for the Environment

1) Open the `Startup.cs` file and navigate to the constructor. The constructor by default takes in an instance of `IConfiguration`, but it can also take in `IHostingEnvironment` and `ILoggerFactory` instances. Update the constructor to take an instance of `IHosingEnvironment`, and assign that injected instance to a class level variable.

```
private readonly IHostingEnvironment _env;
public Startup(IConfiguration configuration, IHostingEnvironment env)
{
  Configuration = configuration;
  _env = env;
}
```

Step 4: Add Services to the Dependency Injection Container

1) Open the `Startup.cs` file and navigate to the `ConfigureServices` method
2) Use the `IConfiguration` instance to get the connection string:

```
var connectionString = Configuration.GetConnectionString("SpyStore");
```

   a) [SQL 2017 LocalDb] Use the `IConfiguration` instance to get the connection string and modify based on the APPDATA user directory (only if the environment is `Development`):

```
var connectionString = Configuration.GetConnectionString("SpyStore");
if (_env.IsDevelopment())
{
  var path = Environment.GetEnvironmentVariable("APPDATA");
  connectionString = connectionString.Replace("{path}", path);
}
```

3) EF Core support is added to the ASP.NET Core DI Container using the built-in `AddDbContextPool` method. This method constructs the derived `DbContext` using the constructor that takes an instance of the `StoreContext` class. Add the following code into the `ConfigureService` method:

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

4) Next add all of the repos into the DI container by adding these lines into the `ConfigureServices` method:

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

5) 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 5: Call the Data Initializer in the Configure method

1) Navigate to the Configure method and update the code block in the `IsDevelopment` if block:

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

Part 4: Use the DI Container

Step 1: Add the Base controller

1) In the Controllers directory of the MVC app, create a new folder named Base. Add a class named BaseController.cs. Update the using statements to match the following:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
```

2) Update the code to match the following. This is the fake security for the sample app:

```
public class BaseController : Controller
{
  public override void OnActionExecuting(ActionExecutingContext context)
  {
    ViewBag.CustomerId = 1;
  }
}
```

Step 2: Add the Controllers

1) Add three new controller classes into the `Controllers` directory:
   CartController.cs
   OrdersController.cs
   ProductsController.cs

Step 3: Update the CartController

1) Update the using statements to the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using SpyStore.Hol.Dal.Repos.Interfaces;
using SpyStore.Hol.Models.Entities;
using SpyStore.Hol.Models.Entities.Base;
using SpyStore.Hol.Models.ViewModels;
using SpyStore.Hol.Mvc.Controllers.Base;
```

2) Make the class public and inherit from `BaseController`:

```
public class CartController : BaseController { }
```

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

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

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

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

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

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

Step 4: 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;
using SpyStore.Hol.Mvc.Controllers.Base;
```

2) Make the class public and inherit from `BaseController`:

```
public class OrdersController : BaseController { }
```

3) Add a constructor that takes an instance of `IOrdersRepo` and a private variable to hold the instance.

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

Step 5: 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.Controllers.Base;
using SpyStore.Hol.Mvc.Support;
```

2) Make the class public and inherit from `BaseController`:

```
public class ProductsController : BaseController { }
```

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

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

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 fully implement the Controllers.