Build an ASP.NET Core Service, and App with Core 2.2 Two-Day Hand-On Lab

Lab 11

This lab is the first in a series that creates the ASP.NET Core web application. This lab walks you through configuring the pipeline, creating the view models, and setting up the configuration, and dependency injection. Prior to starting this lab, you must have completed Lab 10.

Part 1: Configure the Application

Step 1: Add the Service End Points to the Development Settings

1) Update the `appsettings.Development.json` in the `SpyStore.Hol.Mvc` project to the following (adjusted for your `docker-compose.yml` port):

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "ServiceSettings": {
    "Uri": "http://localhost:38080/",
    "CartBaseUri": "api/ShoppingCart",
    "CartRecordBaseUri": "api/ShoppingCartRecord",
    "CategoryBaseUri": "api/Category",
    "CustomerBaseUri": "api/Customer",
    "ProductBaseUri": "api/Product",
    "SearchBaseUri": "api/Search",
    "OrdersBaseUri": "api/Orders",
    "OrderDetailsBaseUri": "api/OrderDetails"
  }
}
```

Step 2: Add the CustomerId to the AppSetting.json File

1) Update the appsettings.json file for the default CustomerId (this might need to be adjusted based on your SQL Server edition):

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "CustomerId" :  1
```

```
}
```

Step 3: Add the Service End Points 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 Uri string is invalid):
2) Update the `appsettings.Production.json` to the following:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning",
      "System": "Warning",
      "Microsoft": "Warning"
    }
  },
  "ServiceSettings": {
    "Uri": "",
    "CartBaseUri": "api/ShoppingCart",
    "CartRecordBaseUri": "api/ShoppingCartRecord",
    "CategoryBaseUri": "api/Category",
    "CustomerBaseUri": "api/Customer",
    "ProductBaseUri": "api/Product",
    "SearchBaseUri": "api/Search",
    "OrdersBaseUri": "api/Orders",
    "OrderDetailsBaseUri": "api/OrderDetails"
  }
}
```

Part 2: Create the ServiceSettings class

1) Add a new folder named `Support` in the `SpyStore.Hol.MVC` project. In that folder, add a new class named `ServiceSettings.cs`. This file will be used to hold configuration information for the services. The `ServiceSettings` class is populated using the “ServiceSettings” configuration section.

```
public class ServiceSettings
{
  public ServiceSettings() { }
  public string Uri { get; set; }
  public string CartBaseUri { get; set; }
  public string CartRecordBaseUri { get; set; }
  public string CategoryBaseUri { get; set; }
  public string CustomerBaseUri { get; set; }
  public string ProductBaseUri { get; set; }
  public string SearchBaseUri { get; set; }
  public string OrdersBaseUri { get; set; }
  public string OrderDetailsBaseUri { get; set; }
}
```

Part 3: Create the ViewModels

1) Create a new folder name `ViewModels` under the `Models` folder in the `SpyStore.Hol.Mvc` project.

Step 1: Create the AddToCartViewModel Class

This view model extends the `CartRecordWithProductInfo` view model. The `Quantity` property is hidden in the base class and replaced with a new `Quantity` in this view model. When the validations are added later in this workshop, this new property will be used.

1) Add a new class to the `ViewModels` folder named `AddToCartViewModel.cs`.
2) Add the following using statements:

```
using System.ComponentModel.DataAnnotations;
using SpyStore.Hol.Models.ViewModels;
```

3) Update the code to the following:

```
public class AddToCartViewModel : CartRecordWithProductInfo
{
  [Required]
  public new int Quantity { get; set; }
}
```

Step 2: Create the CartRecordViewModel Class

This view model extends the `CartRecordWithProductInfo` view model. The `Quantity` property is hidden in the base class and replaced with a new `Quantity` in this view model. When the validations are added later in this workshop, this new property will be used.

1) Add a new class named `CartRecordViewModel.cs` to the `ViewModels` folder. Add the following using statements:

```
using System.ComponentModel.DataAnnotations;
using SpyStore.Hol.Models.ViewModels;
```

2) Update the code to the following:

```
public class CartRecordViewModel : CartRecordWithProductInfo
{
  [Required]
  public new int Quantity { get; set; }
}
```

Step 3: Create the CartViewModel Class

1) Add a new to the `ViewModels` folder class named `CartViewModel.cs`.
2) Add the following using statements:

```
using System.Collections.Generic;
using SpyStore.Hol.Models.Entities;
```

3) Update the code to the following:

```
public class CartViewModel
{
  public Customer Customer { get; set; }
  public IList<CartRecordViewModel> CartRecords { get; set; }
}
```

Step 4: Create the ProductViewModel

1) Add a new to the `ViewModels` folder class named `ProductViewModel.cs`.
2) Add the following using statements:

```
using SpyStore.Hol.Models.Entities.Base;
```

3) Update the code to the following:

```
  public class ProductViewModel : EntityBase
  {
    public ProductDetails Details { get; set; } = new ProductDetails();
    public bool IsFeatured { get; set; }
    public decimal UnitCost { get; set; }
    public decimal CurrentPrice { get; set; }
    public int UnitsInStock { get; set; }
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
  }
```

Part 4: Create the SpyStoreServiceWrapper Interface and Class

The `HttpClientFactory` uses the DI container to create `HttpClient` instances and manages disposal. It also allows for configuration of strongly typed classes that encapsulate `HttpClient` calls.

Step 1: Create the ISpyStoreServiceWrapper

1) In the `Support` folder of the `SpyStore.Hol.Mvc` project, add a new interface named `ISpyStoreServiceWrapper.cs`.
2) Update the using statements to the following:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using SpyStore.Hol.Models.Entities;
using SpyStore.Hol.Models.ViewModels;
using SpyStore.Hol.Mvc.Models.ViewModels;
```

3) Update the code to the following:

```
public interface ISpyStoreServiceWrapper
{
  //CategoryController
  Task<IList<Category>> GetCategoriesAsync();
  Task<Category> GetCategoryAsync(int id);
  Task<IList<ProductViewModel>> GetProductsForACategoryAsync(int categoryId);
  //Orders Controller
```

```
  Task<IList<Order>> GetOrdersAsync(int customerId);
  Task<OrderWithDetailsAndProductInfo> GetOrderDetailsAsync(int orderId);
  //Product Controller
  Task<ProductViewModel> GetOneProductAsync(int productId);
  Task<IList<ProductViewModel>> GetFeaturedProductsAsync();
  //Search Controller
  Task<IList<ProductViewModel>> SearchAsync(string searchTerm);
  //Shopping Cart Record Controller
  Task<IList<CartRecordWithProductInfo>> GetCartRecordsAsync(int id);
  Task AddToCartAsync(int customerId, int productId, int quantity);
  Task<CartRecordWithProductInfo> UpdateShoppingCartRecord(int recordId, ShoppingCartRecord item);
  Task RemoveCartItemAsync(int id, ShoppingCartRecord item);
  //Shopping Cart Controller
  Task<CartWithCustomerInfo> GetCartAsync(int customerId);
  Task<string> PurchaseAsync(int customerId, Customer customer);
  //Customer Controller
  Task<Customer> GetCustomerAsync(int customerId);
  Task<IList<Customer>> GetCustomersAsync();
}
```

Step 2: Create the SpyStoreServiceWrapper class

1) In the `Support` folder of the `SpyStore.Hol.Mvc` project, add a new class named `SpyStoreServiceWrapper.cs`.
2) Update the using statements to the following:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using SpyStore.Hol.Models.Entities;
using SpyStore.Hol.Models.ViewModels;
using SpyStore.Hol.Mvc.Models.ViewModels;
```

3) Make sure the class is public and implements ISpyStoreServiceWrapper:

```
public class SpyStoreServiceWrapper : ISpyStoreServiceWrapper
{
}
```

4) Add a constructor that takes an instance of HttpClient and an IOptionsSnapshot<ServiceSettings> and private variables to hold the instances. Both of these will be injected into the class by the DI container:

```
private HttpClient _client;
private ServiceSettings _settings;
public SpyStoreServiceWrapper(HttpClient client, IOptionsSnapshot<ServiceSettings> settings)
{
  _client = client;
  _settings = settings.Value;
  _client.BaseAddress = new Uri(_settings.Uri);
}
```

5) Add the GetCalls from the interface. These use the GetAsync method of the HttpClient and returns Entities (or lists of Entities):

```
public async Task<IList<Category>> GetCategoriesAsync()
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.CategoryBaseUri}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<IList<Category>>();
  return result;
}
public async Task<Category> GetCategoryAsync(int id)
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.CategoryBaseUri}/{id}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<Category>();
  return result;
}
public async Task<IList<ProductViewModel>> GetProductsForACategoryAsync(int categoryId)
{
  var response =
    await _client.GetAsync($"{_settings.Uri}{_settings.CategoryBaseUri}/{categoryId}/products");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<IList<ProductViewModel>>();
  return result;
}
public async Task<IList<Order>> GetOrdersAsync(int customerId)
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.OrdersBaseUri}/{customerId}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<IList<Order>>();
  return result;
}
public async Task<OrderWithDetailsAndProductInfo> GetOrderDetailsAsync(int orderId)
{
  var response = await
    _client.GetAsync($"{_settings.Uri}{_settings.OrderDetailsBaseUri}/{orderId}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<OrderWithDetailsAndProductInfo>();
  return result;
}
public async Task<ProductViewModel> GetOneProductAsync(int productId)
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.ProductBaseUri}/{productId}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<ProductViewModel>();
  return result;
}
public async Task<IList<ProductViewModel>> GetFeaturedProductsAsync()
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.ProductBaseUri}/featured");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<IList<ProductViewModel>>();
  return result;
}
```

```
public async Task<IList<ProductViewModel>> SearchAsync(string searchTerm)
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.SearchBaseUri}/{searchTerm}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<IList<ProductViewModel>>();
  return result;
}
public async Task<IList<CartRecordWithProductInfo>> GetCartRecordsAsync(int id)
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.CartRecordBaseUri}/{id}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<IList<CartRecordWithProductInfo>>();
  return result;
}
public async Task<CartWithCustomerInfo> GetCartAsync(int customerId)
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.CartBaseUri}/{customerId}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<CartWithCustomerInfo>();
  return result;
}
public async Task<Customer> GetCustomerAsync(int customerId)
{
  var response = await
_client.GetAsync($"{_settings.Uri}{_settings.CustomerBaseUri}/{customerId}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<Customer>();
  return null;
}
public async Task<IList<Customer>> GetCustomersAsync()
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.CustomerBaseUri}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadAsAsync<IList<Customer>>();
  return result;
}
```

6) The AddToCart process takes in a customer id, product id, and quantity, creates a new ShoppingCartRecord, and then calls into the service using the PostAsJsonAsync extension method:

```
public async Task AddToCartAsync(int customerId, int productId, int quantity)
{
  var record = new ShoppingCartRecord
  {
    ProductId = productId,
    CustomerId = customerId,
    Quantity = quantity
  };
  var response = await _client
    .PostAsJsonAsync($"{_settings.Uri}{_settings.CartRecordBaseUri}/{customerId}", record);
  response.EnsureSuccessStatusCode();
}
```

7) The UpdateShoppingCartRecord takes in a ShoppingCartRecord and sends it to the service with the PutAsJsonAsync extension method. If the call is successful, the updated record is retrieved using the service and then sent back to the caller:

```
public async Task<CartRecordWithProductInfo> UpdateShoppingCartRecord(
    int recordId, ShoppingCartRecord item)
{
  var response = await
_client.PutAsJsonAsync($"{_settings.Uri}{_settings.CartRecordBaseUri}/{recordId}", item);
  response.EnsureSuccessStatusCode();
  var location = response.Headers.Location.OriginalString;
  var updatedResponse = await _client.GetAsync(location);
  if (updatedResponse.StatusCode == HttpStatusCode.NotFound)
  {
    return null;
  }
  return await updatedResponse.Content.ReadAsAsync<CartRecordWithProductInfo>();
}
```

8) The purchase process uses the PostAsJsonASync extension method to trigger the fake purchase process:

```
public async Task<string> PurchaseAsync(int customerId, Customer customer)
{
  var response = await _client
    .PostAsJsonAsync($"{_settings.Uri}{_settings.CartBaseUri}/{customerId}/buy", customer);
  response.EnsureSuccessStatusCode();
  return response.Headers.Location.Segments[3];
}
```

9) The final method is to delete a cart item. There isn't an extension method to call Delete with JSON content, so the message must be built up from scratch and the call SendAsync:

```
protected StringContent CreateStringContent(string json)
{
  return new StringContent(json, Encoding.UTF8, "application/json");
}
public async Task RemoveCartItemAsync(int id, ShoppingCartRecord item)
{
  var json = JsonConvert.SerializeObject(item);
  HttpRequestMessage request = new HttpRequestMessage
  {
    Content = CreateStringContent(json),
    Method = HttpMethod.Delete,
    RequestUri = new Uri($"{_settings.Uri}{_settings.CartRecordBaseUri}/{id}")
  };
  var response = await _client.SendAsync(request);
  response.EnsureSuccessStatusCode();
}
```

Part 5: Update the Startup.cs class

Step 1: 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 2: 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 3: Add Services to the Dependency Injection Container

1) Open the `Startup.cs` file and navigate to the `ConfigureServices` method
2) Add the HttpClientFactory that will create the SpyStoreServiceWrapper when requested by another class:

```
services.AddHttpClient<SpyStoreServiceWrapper>();
```

3) Finally, add the following code that uses the configuration file to create the `ServiceSettings` class when requested by another class:

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

Step 4: Add Local to the Development Processing

1) Open the `Startup.cs` file and navigate to the `ConfigureServices` method
2) In the line that tests for `IsDevelopment`, add another check for Local:

```
if (env.IsDevelopment() || env.IsEnvironment("Local"))
{
  app.UseDeveloperExceptionPage();
}
else
{
  app.UseExceptionHandler("/Home/Error");
}
```

Part 6: Use the DI Container

Step 1: Add the Base controller

1) In the Controllers directory of the `SpyStore.Hol.Mvc` project, 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;
using Microsoft.Extensions.Configuration;
```

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

```
public class BaseController : Controller
{
  private IConfiguration _configuration;
  public BaseController(IConfiguration configuration) => this._configuration = configuration;
  public override void OnActionExecuting(ActionExecutingContext context)
  {
    ViewBag.CustomerId = _configuration.GetValue<int>("CustomerId");
  }
}
```

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 Microsoft.Extensions.Configuration;
using SpyStore.Hol.Mvc.Controllers.Base;
using SpyStore.Hol.Mvc.Support;
```

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

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

3) Add a constructor that takes instances of the `SpyStoreServiceWrapper` and `IConfiguration`. Add a private variable to hold the `SpyStoreServiceWrapper` instance and pass the `IConfiguration` instance to the base class.

```
public class CartController : BaseController
{
  private readonly SpyStoreServiceWrapper _serviceWrapper;
  public CartController(SpyStoreServiceWrapper serviceWrapper, IConfiguration configuration)
    : base(configuration)
  {
    _serviceWrapper = serviceWrapper;
  }
}
```

Step 4: Update the OrdersController

1) Update the using statements to the following:

```
using Microsoft.Extensions.Configuration;
using SpyStore.Hol.Mvc.Controllers.Base;
using SpyStore.Hol.Mvc.Support;
```

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

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

3) Add a constructor that takes instances of the `SpyStoreServiceWrapper` and `IConfiguration`. Add a private variable to hold the `SpyStoreServiceWrapper` instance and pass the `IConfiguration` instance to the base class.

```
public class OrdersController : BaseController
{
  private readonly SpyStoreServiceWrapper _serviceWrapper;
  public OrdersController(SpyStoreServiceWrapper serviceWrapper, IConfiguration configuration)
    : base(configuration)
  {
    _serviceWrapper = serviceWrapper;
  }
}
```

Step 5: Update the ProductsController

1) Update the using statements to the following:

```
using Microsoft.Extensions.Configuration;
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>`:

```
public class ProductsController : BaseController
{
  private readonly SpyStoreServiceWrapper _serviceWrapper;
  public ProductsController(SpyStoreServiceWrapper serviceWrapper, IConfiguration configuration)
    : base(configuration)
  {
    _serviceWrapper = serviceWrapper;
  }
}
```

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.