Build an EF and ASP.NET Core 2.1 App HOL

Lab 7

Welcome to the Build an Entity Framework Core and ASP.NET Core 2.1 Application in a Day Hands-On Lab. This lab walks you through finishing the Controllers (complete with routing) and adding the View Models.

Prior to starting this lab, you must have completed Lab 6.

Part 1: Create the MVC ViewModels

Note: The ViewModels are listed here for reference only. Copy them from Code\Completed\Lab7\SpyStore_HOL.MVC\ViewModels

Step 1: Create the base ViewModel

1) Create a new folder name ViewModels in the MVC project. Create a new folder named Base under ViewModels.
2) Add a new class named CartViewModelBase.cs.
3) Add the following using statements:

```
using System.ComponentModel.DataAnnotations;
using Newtonsoft.Json;
using SpyStore_HOL.Models.ViewModels.Base;
```

4) Update the code to the following:

```
public class CartViewModelBase : ProductAndCategoryBase
{
  public int? CustomerId { get; set; }
  [DataType(DataType.Currency), Display(Name = "Total")]
  public decimal LineItemTotal { get; set; }
  public string TimeStampString =>
    TimeStamp != null ? JsonConvert.SerializeObject(TimeStamp).Replace("\"", "") : string.Empty;
}
```

Step 2: Create the AddToCartViewModel ViewModel

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

```
using SpyStore_HOL.MVC.ViewModels.Base;
```

3) Update the code to the following:

```
public class AddToCartViewModel : CartViewModelBase
{
  public int Quantity { get; set; }
}
```

Step 3: Create the CartRecordViewModel ViewModel

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

```
using SpyStore_HOL.MVC.ViewModels.Base;
```

3) Update the code to the following:

```
public class CartRecordViewModel : CartViewModelBase
{
  public int Quantity { get; set; }
}
```

Step 4: Create the CartViewModel ViewModel

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; }
}
```

Part 2: Finish the Controllers

Step 1: Create the base controller

1) Create a new folder named Base under the Controllers folder
2) Add a class named BaseController to the Base folder
3) Add the following using statements:

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

4) Update the code to match the following. The OnActionExecuting even fires before any Action methods execute. The following code tricks out the app to provide a "logged in" user.
NOTE: Real security was not implemented in this sample application due to time constraints.

```
public class BaseController : Controller
{
  public override void OnActionExecuting(ActionExecutingContext context)
  {
    ViewBag.CustomerId = 0;
  }
}
```

Step 2: Update the ProductsController controller

1) Add the following using statements:

```
using SpyStore_HOL.MVC.Controllers.Base;
```

2) Change the base class to the Base Controller:
NOTE: The constructor and private variables were added in a previous lab.

```
public class ProductsController : BaseController
{
  //Omitted For Brevity
}
```

3) Add the Error Action method:
Note: This simple takes the place of the Error Action method on the HomeController, allowing for deletion of the HomeController:

```
[HttpGet]
public ActionResult Error()
{
  return View();
}
```

4) Add the Index Action method. This will call the Logger that was passed in by the DI container, and then redirect to the Featured Action method:

```
[HttpGet]
public ActionResult Index()
{
  Logger.LogInformation(1, "Enter About");
  return RedirectToAction(nameof(Featured));
}
```

5) Add the Details Action method. The method redirects to the AddToCart action of the CartController with the Customer ID, Product ID, and Came From Products route parameters.
NOTE: The nameof() method is used to remove magic strings, but the "Controller" extension must be removed from the returned string name.

```
public ActionResult Details(int id)
{
  return RedirectToAction(nameof(CartController.AddToCart),
       nameof(CartController).Replace("Controller", ""),
       new { customerId = ViewBag.CustomerId,
       productId = id,
       cameFromProducts = true });
}
```

6) Add the Featured Action method. This returns the ProductList.cshtml view to show just the featured products:

```
[HttpGet]
public IActionResult Featured()
{
  ViewBag.Title = "Featured Products";
  ViewBag.Header = "Featured Products";
  ViewBag.ShowCategory = true;
  ViewBag.Featured = true;
  return View("ProductList", _productRepo.GetFeaturedWithCategoryName());
}
```

7) Add the ProductList Action method which returns the ProductList.cshtml View with all of the products for a specific category. The action method gets the instance of the CategoryRepo from the DI container by using the [FromServices] attribute:
Note: This is only required for method injection. Constructor injection is the default mechanism for the ASP.NET Core DI container.

```
[HttpGet]
public IActionResult ProductList([FromServices]ICategoryRepo categoryRepo,int id)
{
  var cat = categoryRepo.Find(id);
  ViewBag.Title = cat?.CategoryName;
  ViewBag.Header = cat?.CategoryName;
  ViewBag.ShowCategory = false;
  ViewBag.Featured = false;
  return View(_productRepo.GetProductsForCategory(id));
}
```

8) Add the Search Action method which returns the PrpductList.cshtml view with the Products that match the search criteria:

```
[Route("[controller]/[action]")]
[HttpPost("{searchString}")]
public IActionResult Search(string searchString)
{
  ViewBag.Title = "Search Results";
  ViewBag.Header = "Search Results";
  ViewBag.ShowCategory = true;
  ViewBag.Featured = false;
  return View("ProductList", _productRepo.Search(searchString));
}
```

Step 3: Update the OrdersController controller

1) Add the following using statement to the class:

```
using SpyStore_HOL.MVC.Controllers.Base;
```

2) Change the base class to BaseController. Add the Attribute route for Controller/Action/CustomerId:
Note: The reserved keywords are in square brackets "[]" and the custom route variables are in braces "{}"

```
[Route("[controller]/[action]/{customerId}")]
public class OrdersController : BaseController
{

  //Omitted for brevity
}
```

3) Add the Index Action method:

```
[HttpGet]
public  IActionResult Index(int customerId)
{
  ViewBag.Title = "Order History";
  ViewBag.Header = "Order History";
  IList<Order> orders = _orderRepo.GetOrderHistory(customerId).ToList();
  return View(orders);
}
```

4) Add the Details Action method:

```
[HttpGet("{orderId}")]
public IActionResult Details(int customerId, int orderId)
{
  ViewBag.Title = "Order Details";
  ViewBag.Header = "Order Details";
  OrderWithDetailsAndProductInfo orderDetails = _orderRepo.GetOneWithDetails(customerId, orderId);
  if (orderDetails == null) return NotFound();
  return View(orderDetails);
}
```

Step 4: Update the CartController controller

1) Add the following using statements to the class:

```
using SpyStore_HOL.MVC.Controllers.Base;
using SpyStore_HOL.MVC.ViewModels;
```

2) Change the base class to BaseController. Add the Attribute route for Controller/Action/CustomerId:

```
[Route("[controller]/[action]/{customerId}")]
public class CartController : BaseController
{
  //Omitted for brevity
}
```

3) Update the constructor to create an AutoMapper configuration. When converting from a AddToCartViewModel to a ShoppingCart record, the Id should be set to 0 (zero) and the Timestamp value set to null. Additionally, map the CartRecordViewModel to the ShoppingCartRecord, the CartRecordWithProductInfo to the CartRecordViewModel and the ProductAndCategoryBase to the AddToCartViewModel:

```
public CartController(IShoppingCartRepo shoppingCartRepo)
{
  _shoppingCartRepo = shoppingCartRepo;
  _config = new MapperConfiguration(
    cfg =>
    {
      cfg.CreateMap<AddToCartViewModel, ShoppingCartRecord>()
         .AfterMap((s, t) =>
           {
             t.Id = 0;
             t.TimeStamp = null;
           });
      cfg.CreateMap<CartRecordViewModel, ShoppingCartRecord>();
      cfg.CreateMap<CartRecordWithProductInfo, CartRecordViewModel>();
      cfg.CreateMap<ProductAndCategoryBase, AddToCartViewModel>();
  });
}
```

4) Create the Index method. Add the [HttpGet] Route Attribute accept an ICustomerRepo from the DI container with [FromServices]. In the action method, create a CartViewModel from the Customer and ShoppingCart repos. Return the default View with the ViewModel as the model:

```
[HttpGet]
public IActionResult Index([FromServices] ICustomerRepo customerRepo, int customerId)
{
  ViewBag.Title = "Cart";
  ViewBag.Header = "Cart";
  var cartItems = _shoppingCartRepo.GetShoppingCartRecords(customerId);
  var customer = customerRepo.Find(customerId);
  var mapper = _config.CreateMapper();
  var viewModel = new CartViewModel
  {
    Customer = customer,
    CartRecords = mapper.Map<IList<CartRecordViewModel>>(cartItems)
  };
  return View(viewModel);
}
```

5) The AddToCart HttpGet method builds an AddToCartViewModel than returns the default View. Inject a ProductRepo with the [FromServices] attribute. Update the code to match the following:

```
[HttpGet("{productId}")]
public IActionResult AddToCart([FromServices] IProductRepo productRepo,
  int customerId, int productId, bool cameFromProducts = false)
{
  ViewBag.CameFromProducts = cameFromProducts;
  ViewBag.Title = "Add to Cart";
  ViewBag.Header = "Add to Cart";
  ViewBag.ShowCategory = true;
  var prod = productRepo.GetOneWithCategoryName(productId);
  if (prod == null) return NotFound();
  var mapper = _config.CreateMapper();
  var cartRecord = mapper.Map<AddToCartViewModel>(prod);
  cartRecord.Quantity = 1;
  return View(cartRecord);
}
```

6) The HttpPost version of the AddToCart method uses Model Binding to accept an AddToCartViewModel. The ValidateAntiForgeryToken is used in conjunction with the Form TagHelper (covered later). If the Model Binding is successful, the AddToCartViewModel is converted to a ShoppingCartRecord class, then saved to the database. If anything fails on the save, the exception is added to the ModelState and the user gets to try again. Add the following code:

```
[HttpPost("{productId}"),ValidateAntiForgeryToken]
public IActionResult AddToCart(int customerId, int productId, AddToCartViewModel item)
{
  if (!ModelState.IsValid) return View(item);
  try
  {
    var mapper = _config.CreateMapper();
    var cartRecord = mapper.Map<ShoppingCartRecord>(item);
    cartRecord.DateCreated = DateTime.Now;
    cartRecord.CustomerId = item.CustomerId ?? 0;
    _shoppingCartRepo.Add(cartRecord);
  }
  catch (Exception)
  {
    ModelState.AddModelError(string.Empty, "There was an error adding the item to the cart.");
    return View(item);
  }
  return RedirectToAction(nameof(CartController.Index), new { customerId });
}
```

7) The Update HttpPost method follows the same pattern as the AddToCart HttpPost method. A CartRecordViewModel is reconstituted using Model Binding, then the TimeStamp string is converted back to a byte array. If the ModelState is valid, the record is updated using the repo. If anything fails, the error is added to the ModelState and the user gets another chance at updating the records.

```
[HttpPost("{id}"), ValidateAntiForgeryToken]
public IActionResult Update(int customerId, int id, int quantity, string timeStampString)
{
  ShoppingCartRecord item = _shoppingCartRepo.Find(id);
  item.TimeStamp = JsonConvert.DeserializeObject<byte[]>($"\"{timeStampString}\"");
  item.Quantity = quantity;
  var mapper = _config.CreateMapper();
  try
  {
    item.DateCreated = DateTime.Now;
    _shoppingCartRepo.Update(item);
    var updatedItem = _shoppingCartRepo.GetShoppingCartRecord(customerId, item.ProductId);
    var newItem = mapper.Map<CartRecordViewModel>(updatedItem);
    return PartialView(newItem);
  }
  catch (Exception)
  {
    ModelState.AddModelError(string.Empty, "An error occurred updating the cart.  Please reload
the page and try again.");
    var updatedItem = _shoppingCartRepo.GetShoppingCartRecord(customerId, item.ProductId);
    var newItem = mapper.Map<CartRecordViewModel>(updatedItem);
    return PartialView(newItem);
  }
}
```

8) Add the Delete Action method:

```
[HttpPost("{id}"),ValidateAntiForgeryToken]
public IActionResult Delete(int customerId, int id, ShoppingCartRecord item)
{
  _shoppingCartRepo.Delete(id, item.TimeStamp);
  return RedirectToAction(nameof(Index), new { customerId });
}
```

Part 3: Update the App for the Products Controllers

Step 1: Change the Default Route in the Startup.cs Configure() method

1) Update the default route to the following:

```
routes.MapRoute(
  name: "default",
  template: "{controller=Products}/{action=Index}/{id?}");
```

Step 2: Change the error handler in the Startup.cs Configure() method

1) Update the UseExceptionHandler method to the following:
   NOTE: Until the view is created, the and application errors in non-development environments will cause additional issue due to the missing view.

```
app.UseExceptionHandler("/Products/Error");
```

Step 3: Update the Layout

1) Open the _Layout.cshtml file in Views\Shared

1) Change the controller for the home link to Products from Home as follows (make this change twice):

```
<a asp-area="" asp-controller="Products" asp-action="Index" class="navbar-
brand">SpyStore_HOL.MVC</a>
<!-- omitted for brevity -->
<li><a asp-area="" asp-controller="Products" asp-action="Index">Home</a></li>
```

2) Delete the other two menu links

Summary

In this lab you finished the Controllers, added attribute routing, and added the ViewModels. It won't properly run until creating the next lab, which adds the views.

Next steps

In the next part of this tutorial series, you will create the Views for the application.