Build an EF and ASP.NET Core 3.0 App HOL

Lab 7

This lab walks you through configuring the pipeline, setting up configuration, the dependency injection, and creating the CarsController. Prior to starting this lab, you must have completed Lab 6.

Part 1: Configure the Application

Step 1: Update the application settings JSON

1) Update the `appsettings.json` to the following (this data will be used later in the lab):

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "DealerInfo": {
    "DealerName": "Skimedic's Used Cars",
    "City": "West Chester",
    "State": "Ohio"
  }
}
```

Step 2: Update the development settings JSON

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

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "System": "Warning",
      "Microsoft": "Information"
    }
  },
  "ConnectionStrings": {
    "Autolot": "Server=.,5433;Database=AutoLotWorkshop;User ID=sa;Password=P@ssw0rd; "
  }
}
```

3) Add a Boolean flag to let the app know if the database should be rebuilt on each run:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "RebuildDatabase" :  true,
  "ConnectionStrings": {
    "Autolot": "Server=.,5433;Database=AutoLotWorkshop;User ID=sa;Password=P@ssw0rd; "
  }
}
```

Step 3: Create the production settings file

1) Add a new JSON file to the `AutoLot.Web` 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": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "RebuildDatabase" :  false,
  "ConnectionStrings": {
    "Autolot": "It's a secret "
  }
}
```

Part 2: Create the DealerInfo class

1) Add a new folder named `ConfigSettings` in the AutoLot.Web project. In that folder, add a new class named `DealerInfo.cs`. This file will be used to hold the information in the appsettings.json configuration file.

```
public class DealerInfo
{
  public string DealerName { get; set; }
  public string City { get; set; }
  public string State { get; set; }
}
```

Part 3: Update the Startup.cs class

Step 1: Update the using statements

1) Update the using statements to the following:

```
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Initialization;
using AutoLot.Dal.Repos;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Web.ConfigSettings;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection.Extensions;
using Microsoft.Extensions.Hosting;
```

Step 2: Update the Constructor

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

```
private readonly IWebHostEnvironment _env;
public Startup(IConfiguration configuration, IWebHostEnvironment 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) Use the `IConfiguration` instance to get the connection string:

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

3) EF Core support is added to the ASP.NET Core DI Container using the built-in `AddDbContextPool` method. Add the following code into the `ConfigureService` method:

```
services.AddDbContextPool<ApplicationDbContext>(options => options
  .UseSqlServer(connectionString,o=>o.EnableRetryOnFailure()));
```

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

```
services.AddScoped<ICarRepo, CarRepo>();
services.AddScoped<ICreditRiskRepo, CreditRiskRepo>();
services.AddScoped<ICustomerRepo, CustomerRepo>();
services.AddScoped<IMakeRepo, MakeRepo>();
services.AddScoped<IOrderRepo, OrderRepo>();
```

5) Add the following code that uses the configuration file to create the `CustomSettings` class when requested by another class:

```
services.Configure<DealerInfo>(Configuration.GetSection(nameof(DealerInfo)));
```

6) Finally, add the following code to add the ActionContextAccessor and HttpContextAccessor into the DI container. This will be used later in the lab.

```
services.TryAddSingleton<IActionContextAccessor, ActionContextAccessor>();
services.AddHttpContextAccessor();
```

Step 4: 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())
{
  app.UseDeveloperExceptionPage();
  using var serviceScope =
    app.ApplicationServices.GetRequiredService<IServiceScopeFactory>().CreateScope();
  var context = serviceScope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
  if (Configuration.GetValue<bool>("RebuildDataBase"))
  {
    SampleDataInitializer.InitializeData(context);
  }
}
else
{
  app.UseExceptionHandler("/Home/Error");
}
```

Step 5: Remove the Routing Table

1) In the Configure method, change the call to app.UseEndpoints to not create a route table entry. All controller will be updated to use AttributeRouting:

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapControllers();
});
```

Part 4: Update the HomeController and Index View

Step 1: Add the Routing Attributes

1) In the Controllers directory of the AutoLot.Web project, open the home controller and add the following attribute to the class:
Note: The reserved keywords are in square brackets "[]", any custom route variables are in braces "{}"

```
[Route("[controller]/[action]")]
public class HomeController : Controller
{
  //omitted for brevity
}
```

2) Update the Index action method for the default routes:

```
[Route("/")]
[Route("/[controller]")]
[Route("/[controller]/[action]")]
[HttpGet]
public IActionResult Index()
{
  return View();
}
```

3) Add the HttpGet attribute to the Privacy and Error action methods.

Step 2: Use the DI Container to Get the Dealer Information

1) Add the following using statement to the HomeController:

```
using System.Diagnostics;
using Microsoft.Extensions.Options;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Web.ConfigSettings;
```

2) Update the Index action method to the following:

```
public IActionResult Index([FromServices] IOptionsMonitor<DealerInfo> dealerMonitor)
{
  var vm = dealerMonitor.CurrentValue;
  return View(vm);
}
```

Step 3: Update the Index View

1) In the Views\Home directory of the AutoLot.Web project, open the Index.cshtml view. Update the markup to the following:

```
@model AutoLot.Web.ConfigSettings.DealerInfo
@{
    ViewData["Title"] = "Home Page";
}

<div class="text-center">
    <h1 class="display-4">Welcome to @Model.DealerName</h1>
    <p class="lead">Located in @Model.City, @Model.State</p>
</div>
```

You can now run the app and see the dealer information on the home page.

Step 4: Add the RazorSyntax action method

1) Add the following action method to the HomeController:

```
[HttpGet]
public IActionResult RazorSyntax([FromServices] ICarRepo carRepo)
{
  var car = carRepo.Find(1);
  return View(car);
}
```

Part 5: Add the CarsController

If you are using Visual Studio, you can use the Controller scaffolding to add the Cars Controller. In this lab, you will add the controllers and views by hand.

Step 1: Add the controller, routings, and the constructor

1) In the Controllers directory of the AutoLot.Web project, add a class named `CarsController.cs`. Update the using statements to match the following:

```
using System.Threading.Tasks;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Models.Entities;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
```

2) Update the code to match the following:

```
[Route("[controller]/[action]")]
public class CarsController : Controller
{
}
```

3) Add the constructor the code to match the following:

```
[Route("[controller]/[action]")]
public class CarsController : Controller
{
  private readonly ICarRepo _repo;
  public CarsController(ICarRepo repo)
  {
    _repo = repo;
  }
}
```

Step 2: Add the helper methods

1) Add a method to get the Makes as a select list:

```
internal SelectList GetMakes(IMakeRepo makeRepo)
  => new SelectList(makeRepo.GetAll(), nameof(Make.Id), nameof(Make.Name));
```

2) Add a method to get a Car instance:

```
internal Car GetOneCar(int? id)
{
  return id == null ? null :  _repo.Find(id.Value);
}
```

Step 3: Add the Index, ByMake, and Details action methods

1) Add the `Index` action method. This redirects to the `Featured` action method and also serves as the entry point into the application and the default action for the controller:

```
[Route("/[controller]")]
[Route("/[controller]/[action]")]
public IActionResult Index()
{
  return View(_repo.GetAllIgnoreQueryFilters());
}
```

2) Add the `ByMake` action method:
NOTE: The `{makeId} and {makeName}` parameters in the `HttpGet` attribute are appended to the route

```
[HttpGet("{makeId}/{makeName}")]
public IActionResult ByMake(int makeId, string makeName)
{
  ViewBag.MakeName = makeName;
  return View(_repo.GetAllBy(makeId));
}
```

3) Add the `Details` action method:

```
// GET: Cars/Details/5
[HttpGet("{id?}")]
public IActionResult Details(int? id)
{
  var car = GetOneCar(id);
  if (car == null)
  {
    return NotFound();
  }
  return View(car);
}
```

Step 4: Add the Create action methods

The Create, Edit, and Delete processes use two action methods, HttpGet and HttpPost. They all use the Post-Redirect-Get pattern to prevent double submits.

1) Add the HttpGet Create action method:

```
[HttpGet]
public IActionResult Create([FromServices] IMakeRepo makeRepo)
{
  ViewData["MakeId"] = GetMakes(makeRepo);
  return View();
}
```

2) The HttpPost version leverages ASP.NET Core's anti-forgery token. It also checks the entity for errors generated during the model binding or validation process. If the entity is valid, it gets saved to the database. Otherwise the user gets to try again.

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create([FromServices] IMakeRepo makeRepo, Car car)
{
  if (ModelState.IsValid)
  {
    _repo.Add(car);
    return RedirectToAction(nameof(Index));
  }
  ViewData["MakeId"] = GetMakes(makeRepo);
  return View(car);
}
```

Step 5: Add the Edit action methods

1) Add the HttpGet Edit action method:

```
[HttpGet("{id?}")]
public IActionResult Edit([FromServices] IMakeRepo makeRepo, int? id)
{
  var car = GetOneCar(id);
  if (car == null)
  {
    return NotFound();
  }
  ViewData["MakeId"] = GetMakes(makeRepo);
  return View(car);
}
```

2) Add the HttpPost Edit action method:

```
// POST: Cars/Edit/5
[HttpPost("{id}")]
[ValidateAntiForgeryToken]
public IActionResult Edit([FromServices] IMakeRepo makeRepo, int id, Car car)
{
  if (id != car.Id)
  {
    return BadRequest();
  }
  if (ModelState.IsValid)
  {
    _repo.Update(car);
    return RedirectToAction(nameof(Index));
  }
  ViewData["MakeId"] = GetMakes(makeRepo);
  return View(car);
}
```

Step 6: Add the Delete action methods

1) Add the HttpGet Delete action method:

```
 [HttpGet("{id?}")]
public IActionResult Delete(int? id)
{
  var car = GetOneCar(id);
  if (car == null)
  {
    return NotFound();
  }

  return View(car);
}
```

2) Add the HttpPost Delete action method:

```
// POST: Cars/Delete/5
[HttpPost("{id}")]
[ValidateAntiForgeryToken]
public IActionResult Delete(int id, Car car)
{
  _repo.Delete(car);
  return RedirectToAction(nameof(Index));
}
```

Summary

This lab configured the application, used the Options pattern to get the dealer information, added the data access library classes into the DI container, and added the CarsController.

Next steps

In the next part of this tutorial series, you will start working on the user interface.