Build an ASP.NET Core Service and App with .NET (Core) 5.0 Two-Day Hands-On Lab

Lab 13

This lab is the first in a series that creates the ASP.NET Core web application using the Model-View-Controller pattern. This lab walks you through configuring the pipeline, setting up the configuration, and dependency injection. Prior to starting this lab, you must have completed Lab 11 (Lab 12 was optional). All work in this lab is in the AutoLot.Mvc project.

Part 1: Configure the Application

Step 1: Update the root AppSettings file

- Update the `appsettings.json` in the `AutoLot.Mvc` project to the following (adjusted for your connection string and ports):

```
{
  "Logging": {
    "MSSqlServer": {
      "schema": "Logging",
      "tableName": "SeriLogs",
      "restrictedToMinimumLevel": "Warning"
    }
  },
  "ApplicationName": "AutoLot.MVC",
  "AllowedHosts": "*",
  "DealerInfo": {
    "DealerName": "Skimedic's Used Cars",
    "City": "West Chester",
    "State": "Ohio"
  }
}
```

Step 2: Update the Development App Settings

- Update the `appsettings.Development.json` in the `AutoLot.Mvc` project to the following (adjusted for your connection string and ports):

```
{
  "Logging": {
    "MSSqlServer": {
      "schema": "Logging",
      "tableName": "SeriLogs",
      "restrictedToMinimumLevel": "Warning"
    }
  },
  "RebuildDataBase": false,
  "ConnectionStrings": {
    //SQL Server Local Db
    //"AutoLot": "Server=(localdb)\\MSSQLLocalDB;Database=AutoLot50;Trusted_Connection=True;"
    //Docker-Compose
    //"AutoLot": "Server=db;Database=AutoLot50;User Id=sa;Password=P@ssw0rd;"
    //Docker
    "AutoLot": "Server=.,5433;Database=AutoLot50;User ID=sa;Password=P@ssw0rd;"
  },
  "ApiServiceSettings": {
    "Uri": "https://localhost:5021/",
    "CarBaseUri": "api/Cars",
    "MakeBaseUri": "api/Makes"
  }
}
```

Step 3: Add the Production Settings File

- Add a new JSON file to the `AutoLot.Mvc` project named `appsettings.Production.json` and update the file to the following:

```
{
  "Logging": {
    "MSSqlServer": {
      "schema": "Logging",
      "tableName": "SeriLogs",
      "restrictedToMinimumLevel": "Warning"
    }
  },
  "RebuildDataBase": false,
  "ConnectionStrings": {
    "AutoLot": "It's a Secret"
  },
  "ApiServiceSettings": {
    "Uri": "https://localhost:5021/",
    "CarBaseUri": "api/Cars",
    "MakeBaseUri": "api/Makes"
  }
}
```


Part 2: Update the Program.cs class

- Update the using statements to the following:

```
using AutoLot.Mvc;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
```

- Convert Program/Main to top-level statements and add SeriLog configuration:

```
CreateHostBuilder(args).Build().Run();

static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder => { webBuilder.UseStartup<Startup>(); })
        .ConfigureSerilog();
```

Part 3: Update the Startup.cs class

Step 1: Update the using statements

- Add the following using statements to the top of the `Statup.cs` class:

```
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Initialization;
using AutoLot.Dal.Repos;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Models.ViewModels;
using AutoLot.Services.ApiWrapper;
using AutoLot.Services.Logging;
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: Add a Class Level Variable for the Environment

- 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)
{
  _env = env;
  Configuration = configuration;
}
```

Step 3: Add Application Services to the Dependency Injection Container

- Open the `Startup.cs` file and navigate to the `ConfigureServices` method
- Add the repos to the DI container:

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

- Add the `HttpClientFactory` by calling the `ConfigureApiServiceWrapper` extension method:

```
services.ConfigureApiServiceWrapper(Configuration);
```

- Add the following code to populate the `DealerInfo` class from the configuration file:

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

- Add the `IActionContextAccessor` and `HttpContextAccessor`:

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

- Add the `ApplicationDbContext`:

```
var connectionString = Configuration.GetConnectionString("AutoLot");
services.AddDbContextPool<ApplicationDbContext>(
  options => options.UseSqlServer(connectionString,
    sqlOptions => sqlOptions.EnableRetryOnFailure().CommandTimeout(60)));
```

Step 4: Call the Data Initializer in the Configure method

- Navigate to the Configure method and update the method signature to inject in an `ApplicationDbContext`:

```
public void Configure(
  IApplicationBuilder app,
  IWebHostEnvironment env,
  ApplicationDbContext context)
{ ... }
```

- In the `IsDevelopment` if block, check the settings to determine if the database should be rebuilt, and it yet, call the data initializer:

```
if (env.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
  //Initialize the database
  if (Configuration.GetValue<bool>("RebuildDataBase"))
  {
    SampleDataInitializer.ClearAndReseedDatabase(context);
  }
}
```


Step 5: Update the Routing for Attribute Routing

- Update the call to `UseEndpoints` to the following:

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapControllers();
  //endpoints.MapControllerRoute(
  //    name: "default",
  //    pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Part 4: Update the Home Controller

- Add the following using statement to the `HomeController`:

```
using AutoLot.Services.Logging;
```

- Replace the default `ILogger` with the `IAppLogging`:

```
private readonly IAppLogging<HomeController> _logger;
public HomeController(IAppLogging<HomeController> logger)
{
  _logger = logger;
}
```

- Add the Controller level route to the `HomeController`:

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

- Add HttpGet attribute to all Get action methods:

```
[HttpGet]
public IActionResult Index() => return View();
[HttpGet]
public IActionResult Privacy()
{
  return View();
}
```

- Update the Index method to the default, controller only, and controller/action routes:

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

Part 5: Add WebOptimizer

Step 1: Add WebOptimizer to DI Container

- Update the `ConfigureServices` method by adding the following code:

```
if (_env.IsDevelopment() || _env.IsEnvironment("Local"))
{
  //services.AddWebOptimizer(false,false);
  services.AddWebOptimizer(options =>
  {
    options.MinifyCssFiles(); //Minifies all CSS files
    //options.MinifyJsFiles(); //Minifies all JS files
    //options.MinifyJsFiles("js/site.js");
    options.MinifyJsFiles("lib/**/*.js");
    //options.AddJavaScriptBundle("js/validations/validationCode.js", "js/validations/**/*.js");
    //options.AddJavaScriptBundle("js/validations/validationCode.js",
    //  "js/validations/validators.js", "js/validations/errorFormatting.js");
  });
}
else
{
  services.AddWebOptimizer(options =>
  {
    options.MinifyCssFiles(); //Minifies all CSS files
    //options.MinifyJsFiles(); //Minifies all JS files
    options.MinifyJsFiles("js/site.js");
    options.AddJavaScriptBundle("js/validations/validationCode.js", "js/validations/**/*.js");
    //options.AddJavaScriptBundle("js/validations/validationCode.js",
    //  "js/validations/validators.js", "js/validations/errorFormatting.js");
  });
}
```

Step 2: Add WebOptimizer to HTTP Pipeline

- Update the `Configure` method by adding the following code (**before** `app.UseStaticFiles()`):

```
app.UseHttpsRedirection();
app.UseWebOptimizer();
app.UseStaticFiles();
```

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 support for client-side libraries, update the layout, and add GDPR Support.