.NET App Dev Hands-On Workshop

Lab 9 – MVC Pipeline Configuration, Dependency Injection

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 8.

Part 1: Configure the Application

Step 1: 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):

```
{
  "AppLoggingSettings": {
    "MSSqlServer": {
      "TableName": "SeriLogs",
      "Schema": "Logging",
      "ConnectionStringName": "AutoLot"
    },
    "File": {
      "Drive": "c",
      "FilePath": "temp",
      "FileName": "log_AutoLot.txt"
    },
    "General": {
      "RestrictedToMinimumLevel": "Information"
    }
  },
  "AppName": "AutoLot.Mvc - Dev",
  "RebuildDataBase": false,
  "ConnectionStrings": {
    //"AutoLot":
"Server=(localdb)\\MSSqlLocalDb;Database=AutoLot_Hol;Trusted_Connection=true;Encrypt=false;"
    "AutoLot": "Server=.,5433;Database=AutoLot_Hol;User ID=sa;Password=P@ssw0rd;"
  }
}
```

Step 2: Update the root AppSettings.json file

- Update the `appsettings.json` in the `AutoLot.Mvc` project to the following:

```
{
  "AllowedHosts": "*",
  "DealerInfo": {
    "DealerName": "Skimedic's Used Cars",
    "City": "West Chester",
    "State": "Ohio"
  }
}
```

Step 3: Update the Production Settings File

- Update the `appsettings.Production.json` to the following:

```
{
  "AppLoggingSettings": {
    "MSSqlServer": {
      "TableName": "SeriLogs",
      "Schema": "Logging",
      "ConnectionStringName": "AutoLot"
    },
    "File": {
      "Drive": "c",
      "FilePath": "temp",
      "FileName": "log_AutoLot.txt"
    },
    "General": {
      "RestrictedToMinimumLevel": "Information"
    }
  },
  "AppName": "AutoLot.Mvc",
  "RebuildDataBase": false,
  "ConnectionStrings": {
    "AutoLot": "It's a Secret"
  }
}
```

Part 2: Add the GlobalUsings.cs File

- Create a new file named `GlobalUsings.cs` and update the contents to the following:

```
global using AutoLot.Dal.EfStructures;
global using AutoLot.Dal.Initialization;
global using AutoLot.Dal.Repos;
global using AutoLot.Dal.Repos.Interfaces;

global using AutoLot.Mvc.Models;

global using AutoLot.Services.Logging.Configuration;
global using AutoLot.Services.Logging.Interfaces;

global using Microsoft.AspNetCore.Mvc;
global using Microsoft.AspNetCore.Mvc.Infrastructure;
global using Microsoft.EntityFrameworkCore;
global using Microsoft.Extensions.DependencyInjection.Extensions;
global using Microsoft.Extensions.Options;
global using Microsoft.EntityFrameworkCore.Diagnostics;

global using System.Diagnostics;
```

Part 3: Add the DealerInfo ViewModel

- In the AutoLot.Service project, create a new folder named ViewModels, and in that folder create a new file named DealerInfo.cs and update the contents to the following:

```
namespace AutoLot.Services.ViewModels;

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

- Add the following to the GlobalUsings.cs file:

```
global using AutoLot.Services.ViewModels;
```

Part 4: Update the Program.cs Top Level Statements

Step 1: Add Logging

- Add Serilog into the WebApplicationBuilder and add the logging interfaces to the DI container in Program.cs:

```
var builder = WebApplication.CreateBuilder(args);
builder.ConfigureSerilog();
builder.Services.RegisterLoggingInterfaces();
```

Step 2: Add Application Services to the Dependency Injection Container

- Add the repos to the DI container after the comment *//Add services to the container*:

```
//Add services to the DI container
builder.Services.AddScoped<ICarDriverRepo, CarDriverRepo>();
builder.Services.AddScoped<ICarRepo, CarRepo>();
builder.Services.AddScoped<ICreditRiskRepo, CreditRiskRepo>();
builder.Services.AddScoped<ICustomerOrderViewModelRepo, CustomerOrderViewModelRepo>();
builder.Services.AddScoped<ICustomerRepo, CustomerRepo>();
builder.Services.AddScoped<IDriverRepo, DriverRepo>();
builder.Services.AddScoped<IMakeRepo, MakeRepo>();
builder.Services.AddScoped<IOrderRepo, OrderRepo>();
builder.Services.AddScoped<IRadioRepo, RadioRepo>();
```

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

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

- Add the `IActionContextAccessor` and `HttpContextAccessor`:

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

- Add the `ApplicationDbContext`:

```
var connectionString = builder.Configuration.GetConnectionString("AutoLot");
builder.Services.AddDbContextPool<ApplicationDbContext>(
  options =>
  {
    options.ConfigureWarnings(wc => wc.Ignore(RelationalEventId.BoolWithDefaultWarning));
    options.UseSqlServer(connectionString,
      sqlOptions => sqlOptions.EnableRetryOnFailure().CommandTimeout(60));
  });
```

Step 3: Call the Data Initializer

- In the WebApplication section, flip the `IsDevelopment` if block around and add the UseDeveloperExceptionPage so the code looks like this:

```
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();

}
else
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}
```


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

```
if (app.Environment.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
  if (app.Configuration.GetValue<bool>("RebuildDataBase"))
  {
    using var scope = app.Services.CreateScope();
    var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
    SampleDataInitializer.ClearAndReseedDatabase(dbContext);
  }
}
```

- Comment out the `IncludeAssets` tag for `EntityFrameworkCore.Design` in the `AutoLot.Mvc.csproj` file:

```
<PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="7.0.1">
  <!--<IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>-->
  <PrivateAssets>all</PrivateAssets>
</PackageReference>
```

Step 4: Update the Routing for Attribute Routing

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

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

Part 5: Update the Home Controller

- 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() => 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();
}
```

- Inject the `DealerInfo OptionsMonitor` into the `Index` method, and pass the `CurrentValue` to the `View`:

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

Part 6: Test the Logging

- Update the `HomeController` Index method to log an error:

```
public IActionResult Index([FromServices]IOptionsMonitor<DealerInfo> dealerOptionsMonitor)
{
  _logger.LogAppError("Test error");
  return View(dealerOptionsMonitor.CurrentValue);
}
```

- Run the application. Since the Index method is the default entry point for the application, just running the app should create an error file as well as an entry into the SeriLogEntry table.
- Once you have confirmed that logging works, comment out the error logging code:

```
//_logger.LogAppError("Test error");
```


Part 7: Add WebOptimizer

Step 1: Add WebOptimizer to DI Container

- Update the `Program.cs` top level statements by adding the following code after adding the services but before the WebApplication is built:

```
if (builder.Environment.IsDevelopment() || builder.Environment.IsEnvironment("Local"))
{
  // builder.Services.AddWebOptimizer(false,false);
  builder.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
{
  builder.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");
  });
}
var app = builder.Build();
```

Step 2: Add WebOptimizer to HTTP Pipeline

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

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

Step 3: Update _ViewImports to enable WebOptimizer Tag Helpers

- Update the `_ViewImports.cshtml` file to enable `WebOptimizer` tag helpers:

```
@using AutoLot.Mvc
@using AutoLot.Mvc.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, WebOptimizerCore
```

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.