

.NET 9 App Dev Hands-On Lab

MVC Lab 3 –Pipeline Configuration, Dependency Injection

This lab configures the HTTP pipeline and configures the configuration and dependency injection. Before starting this lab, you must have completed MVC Lab 2b.

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): Note the comma added after "AutoLot.Mvc - Dev"

```
{
  "AppLoggingSettings": {
    "MSSqlServer": {
      "TableName": "SeriLogs",
      "Schema": "Logging",
      "ConnectionStringName": "AutoLot"
    },
    "File": {
      "Drive": "c",
      "FilePath": "temp",
      "FileName": "log_AutoLot_Debug.txt"
    },
    "General": {
      "RestrictedToMinimumLevel": "Information"
    }
  },
  "AppName": "AutoLot.Mvc - Dev",
  "RebuildDataBase": true,
  "ConnectionStrings": {
    //SQL Server Local Db
    "AutoLot": "Server=(localdb)\\MSSQLLocalDB;Database=AutoLot_Hol;Trusted_Connection=True;"
    //"AutoLot": "Server=(localdb)\\ProjectModels;Database=AutoLot_Hol;Trusted_Connection=True;"
    //Docker
    //"AutoLot": "Server=.,5433;Database=AutoLot_Hol;User ID=sa;Password=P@ssw0rd;"
  },
  "DealerInfo": {
    "DealerName": "Skimedic's Used Cars Development MVC Site",
    "City": "West Chester",
    "State": "Ohio"
  }
}
```

Step 2: Add the Staging Settings File

- Add a new file named `appsettings.Staging.json` to the root of the `AutoLot.Mvc` project and update it to the following:

```
{
  "AppLoggingSettings": {
    "MSSqlServer": {
      "TableName": "SeriLogs",
      "Schema": "Logging",
      "ConnectionStringName": "AutoLot"
    },
    "File": {
      "Drive": "c",
      "FilePath": "temp",
      "FileName": "log_AutoLot_Staging.txt"
    },
    "General": {
      "RestrictedToMinimumLevel": "Information"
    }
  },
  "AppName": "AutoLot.Mvc - Staging",
  "RebuildDataBase": false,
  "ConnectionStrings": {
    //SQL Server Local Db
    "AutoLot": "Server=(localdb)\\MSSQLLocalDB;Database=AutoLot_Hol;Trusted_Connection=True;"
    //"AutoLot": "Server=(localdb)\\ProjectModels;Database=AutoLot_Hol;Trusted_Connection=True;"
    //Docker
    //"AutoLot": "Server=.,5433;Database=AutoLot_Hol;User ID=sa;Password=P@ssw0rd;"
  },
  "DealerInfo": {
    "DealerName": "Skimedic's Used Cars Staging MVC Site",
    "City": "West Chester",
    "State": "Ohio"
  }
}
```

Step 3: Update the AppSettings.json file

- Update the `appsettings.json` in the `AutoLot.Mvc` project to the following:
Note the added comma after `"*"`:

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

Step 4: Update the Production Settings File

- Update the appsettings.Production.json in the AutoLot.Mvc project to the following:
Note the comma added after "AutoLot.Mvc"

```
{
  "AppLoggingSettings": {
    "MSSqlServer": {
      "TableName": "SeriLogs",
      "Schema": "Logging",
      "ConnectionStringName": "AutoLot"
    },
    "File": {
      "Drive": "c",
      "FilePath": "temp",
      "FileName": "log_AutoLot.txt"
    },
    "General": {
      "RestrictedToMinimumLevel": "Error"
    }
  },
  "AppName": "AutoLot.Mvc",
  "RebuildDataBase": false,
  "ConnectionStrings": {
    "AutoLot": "[its-a-secret]"
  }
}
```

Part 2: Add the GlobalUsings.cs File

- Create a new file named GlobalUsings.cs in the AutoLot.Mvc project 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 AutoLot.Services.Simple;
global using AutoLot.Services.Simple.Interfaces;
global using AutoLot.Services.ViewModels;

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: Update the Program.cs Top Level Statements

Step 1: Add Logging

- Add Serilog to the WebApplicationBuilder and the logging interfaces to the DI container in Program.cs in the AutoLot.Mvc project:

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

Step 2: Update WebHost for CSS Isolation

- The CSS Isolation file is created in the development environment or when an app is published. To create the CSS file in other environments, update the web host to use static web assets:

```
builder.Services.RegisterLoggingInterfaces();
if (!builder.Environment.IsDevelopment())
{
    builder.WebHost.UseStaticWebAssets();
}
```

Step 3: Add Application Services to the Dependency Injection Container

- Add the repos to the DI container after the comment *//Add services to the container* and after the call to `AddControllersWithViews()`:

```
//Add services to the DI container
builder.Services.AddControllersWithViews();
builder.Services.AddScoped<ICarDriverRepo, CarDriverRepo>();
builder.Services.AddScoped<ICarRepo, CarRepo>();
builder.Services.AddScoped<IDriverRepo, DriverRepo>();
builder.Services.AddScoped<IMakeRepo, MakeRepo>();
builder.Services.AddScoped<IRadioRepo, RadioRepo>();
```

- Add the keyed services into the DI container:

```
builder.Services.AddKeyedScoped<ISimpleService, SimpleServiceOne>(nameof(SimpleServiceOne));
builder.Services.AddKeyedScoped<ISimpleService, SimpleServiceTwo>(nameof(SimpleServiceTwo));
```

- 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 4: Add DI Validation

- Add the code to validate DI services and scopes on building the web app:

```
builder.Services.AddControllersWithViews()
    .AddControllersAsServices()
    .AddViewComponentsAsServices()
    .AddTagHelpersAsServices();
builder.Host.UseDefaultServiceProvider(o =>
{
    o.ValidateOnBuild = true;
    o.ValidateScopes = true;
});
```

Step 5: Call the Data_INITIALIZER and Update the Project File

- In the section after `builder.Build()`, 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");
    // The default HSTS value is 30 days.
    //You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
```

- In the `IsDevelopment` if block, check the settings to determine if the database should be rebuilt, and if 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);
    }
}
```

```
}
```

- If you converted the tables to be temporal (EF Core Lab 8), comment out the IncludeAssets tag for EntityFrameworkCore.Design in the AutoLot.Mvc.csproj file:

```
<PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version=" [9.0.*,10.0) ">
  <!--<IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>-->
  <PrivateAssets>all</PrivateAssets>
</PackageReference>
```

Step 6: Update the Routing for Attribute Routing

- Back in Program.cs in the AutoLot.Mvc project, comment out the call to MapControllerRoute and add the MapControllers call:

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

Part 4: Add WebOptimizer

This section shows how to use WebOptimizer for bundling, minification, and caching. The MapStaticFiles method (introduced in ASP.NET Core 9) and WebOptimizer do not work together yet, so it must be commented out.

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 call to builder.Build():

```
if (builder.Environment.IsDevelopment() || builder.Environment.IsEnvironment("Local"))
{
    builder.Services.AddWebOptimizer(false, false);
}
else
{
    builder.Services.AddWebOptimizer(options =>
    {
        //options.MinifyCssFiles(); //Minifies all CSS files
        options.MinifyCssFiles("css/**/*.css");
        //options.MinifyJsFiles(); //Minifies all JS files
        options.MinifyJsFiles("js/site.js");
        //options.MinifyJsFiles("js/**/*.js");
    });
}
var app = builder.Build();
```

Step 2: Add UseStaticFiles to the HTTP Pipeline

- Add in the call to UseStaticFiles and comment out the MapStaticAssets code in Program.cs as shown here:

```
app.UseHttpsRedirection();  
app.UseStaticFiles();  
app.UseRouting();  
app.UseAuthorization();  
//app.MapStaticAssets();  
app.MapControllers();
```

Step 3: 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 *, WebOptimizer.Core
```

Part 5: Update the Home Controller

Step 1: Update the Controller and Action method routing

- Add the Controller level route to the HomeController (the commented-out route shows the equivalent route using a literal instead of a token):

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

- Add `HttpGet` attribute to all Get action methods:

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

- Add the default, controller only, and controller/action routes to the Index action method (the commented-out route shows a route using just a literal):

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

Step 2: Update the constructor to a primary constructor, then add and test logging

- Remove the constructor and the logger field. Add a primary constructor to the class and inject `IAppLogging` into the constructor:

```
public class HomeController(IAppLogging<HomeController> logger) : Controller
{
private readonly ILogger<HomeController> _logger = logger;
    //omitted for brevity
}
```


- Inject the DealerInfo OptionsMonitor into the Index method and pass the CurrentValue to the View (the view will be updated in a later lab):

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

- Update the HomeController Index method to log an error:

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

- Run the application and make sure to launch a browser. Since the Index method is the default entry point for the application, just running the app should create an error file and an entry into the SeriLog table. Once you have confirmed that logging works, comment out the error logging code:

```
//logger.LogError("Test error");
```

- Inject the SimpleService into two new action methods and pass the message from the service to the SimpleService view (the view will be created in the next lab):

```
[HttpGet]
public IActionResult GetServiceOne(
    [FromKeyedServices(nameof(SimpleServiceOne))] ISimpleService service)
{
    return View("SimpleService",service.SayHello());
}
```

```
[HttpGet]
public IActionResult GetServiceTwo(
    [FromKeyedServices(nameof(SimpleServiceTwo))] ISimpleService service)
{
    return View("SimpleService",service.SayHello());
}
```

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.