.NET 10 App Dev Hands-On Lab

Razor Pages Lab 3 –Pipeline Configuration, Dependency Injection

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

Part 1: Configure the Application

Step 1: Update the Development App Settings

- Update the `appsettings.Development.json` in the `AutoLot.Web` project to the following (adjusted for your connection string and ports):
  Note the comma added after `"AutoLot.Web - Dev"`

```
{
//omitted for brevity
  "AppName": "AutoLot.Web - Dev",
  "RebuildDataBase": true,
  "ConnectionStrings": {
    "AutoLot": "Server=(localdb)\\MSSQLLocalDB;Database=AutoLot_Hol;Trusted_Connection=True;"
  },
  "DealerInfo": {
    "DealerName": "Skimedic's Used Cars Development Razor Pages Site",
    "City": "West Chester",
    "State": "Ohio"
  }
}
```

Step 2: Update the Staging Settings File

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

```
{
//omitted for brevity
  "AppName": "AutoLot.Web - Staging",
  "RebuildDataBase": false,
  "ConnectionStrings": {
    "AutoLot": "Server=(localdb)\\MSSQLLocalDB;Database=AutoLot_Hol;Trusted_Connection=True;"
  },
  "DealerInfo": {
    "DealerName": "Skimedic's Used Cars Staging Razor Pages Site",
    "City": "West Chester",
    "State": "Ohio"
  }
}
```

Step 3: Update the Production Settings File

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

```
{
//omitted for brevity
  "AppName": "AutoLot.Web",
  "RebuildDataBase": false,
  "ConnectionStrings": {
    "AutoLot": "[its-a-secret]"
  },
  "DealerInfo": {
    "DealerName": "Skimedic's Used Cars Production Razor Pages Site",
    "City": "West Chester",
    "State": "Ohio"
  }
}
```

Step 4: Update the AppSettings.json file

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

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

Step 5: Update the Project File

- If you updated the tables as temporal tables (EF Core Lab 8), comment out the `IncludeAssets` tag for **`EntityFrameworkCore.Design`** in the `AutoLot.Web.csproj` file:

```
<PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="[10.0.*,11.0)">
<!--<IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>-->
<PrivateAssets>all</PrivateAssets>
</PackageReference>
```

Part 2: Add the GlobalUsings.cs File

- Create a new file named `GlobalUsings.cs` in the `AutoLot.Web` 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.Services.Logging.Configuration;
global using AutoLot.Services.Logging.Interfaces;
global using AutoLot.Services.Simple;
global using AutoLot.Services.Simple.Interfaces;
global using AutoLot.Services.Utilities;
global using AutoLot.Services.ViewModels;
global using Microsoft.AspNetCore.Http.Features;
global using Microsoft.AspNetCore.Mvc;
global using Microsoft.AspNetCore.Mvc.Infrastructure;
global using Microsoft.AspNetCore.Mvc.RazorPages;
global using Microsoft.EntityFrameworkCore;
global using Microsoft.EntityFrameworkCore.Diagnostics;
global using Microsoft.Extensions.DependencyInjection.Extensions;
global using Microsoft.Extensions.Options;
global using System.Diagnostics;
global using System.Text.Json.Serialization;
```

Part 3: Update the Program.cs Top Level Statements

Copilot Agent Mode

The following prompt covers Steps 1-5

```
Prompt: Always use file scoped namespaces. Always combine attributes on a single line when
possible. The project does not use nullable reference types. There is a GlobalUsings.cs file that
includes common usings, don't include using statements in new files if they are already in the
globalusings.cs file. I prefer expression bodied members when possible. Single line if statements
should still use braces. Use ternary operators when appropriate. Use internal over private. All
classes and methods are public unless told otherwise. Don't add a constructor unless instructed to
do so. Use primary constructors when possible and don't declare a class level variable if the
parameter from the constructor can be used. Don't initialize properties unless instructed to do
so. All work is to be done in the Autolot.Web project.

Prompt: The following work is done in the Program.cs file.
Add the following after the call to CreateBuilder:
builder.ConfigureSerilog();
builder.Services.RegisterLoggingInterfaces();
if (!builder.Environment.IsDevelopment())
{
  builder.WebHost.UseStaticWebAssets();
}
```

```
Prompt: Add the following after AddRazorPages():
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>();
builder.Services.AddKeyedScoped<ISimpleService, SimpleServiceOne>(nameof(SimpleServiceOne));
builder.Services.AddKeyedScoped<ISimpleService, SimpleServiceTwo>(nameof(SimpleServiceTwo));
builder.Services.Configure<DealerInfo>(builder.Configuration.GetSection(nameof(DealerInfo)));
builder.Services.AddHttpContextAccessor();
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));
  });

Prompt: Update the call to AddRazorPages to add DI validation (remove the original call to
AddRazorPages).
builder.Services.AddRazorPages()
    .AddControllersAsServices()
    .AddViewComponentsAsServices()
    .AddTagHelpersAsServices();
builder.Host.UseDefaultServiceProvider(o =>
{
  o.ValidateOnBuild = true;
  o.ValidateScopes = true;
});
Prompt: 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();
  if (app.Configuration.GetValue<bool>("RebuildDataBase"))
  {
    using var scope = app.Services.CreateScope();
    var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
    SampleDataInitializer.ClearAndReseedDatabase(dbContext);
  }
}
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();
}
```


Manual

Step 1: Add Logging

- Add `Serilog` to the `WebApplicationBuilder` and the logging interfaces to the DI container in `Program.cs` in the `AutoLot.Web` 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 `AddRazorPages()`:

```
//Add services to the DI container
builder.Services.AddRazorPages();
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.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.AddRazorPages()
  .AddControllersAsServices()
  .AddViewComponentsAsServices()
  .AddTagHelpersAsServices();

builder.Host.UseDefaultServiceProvider(o =>
{
  o.ValidateOnBuild = true;
  o.ValidateScopes = true;
});
```

Step 5: Call the Data Initializer

- In the section after `builder.Build()`, flip the `IsDevelopment` if block around, add the `UseDeveloperExceptionPage` and check the settings to determine if the database should be rebuilt, and if yes, call the data initializer so the code looks like this:

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


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.

Copilot Agent Mode

The following prompt covers Steps 1-4

```
Prompt: The following work is done in Program.cs of the AutoLot.Web project:
Add the following 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");
  });
}

Prompt: Add app.UseWebOptmizer() before UseHttpsRedirection()
Add app.UseStaticFiles() after UseHttpsRedirection()
Comment out app.MapStaticAssets();
Comment out the call WithStaticAsets() from the call to MapRazorPages()

Prompot: The following is done in _ViewImports.cshtml
Add the following line to the end of the file:
@addTagHelper *, WebOptimizer.Core
```


Manual

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);
}
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()` and `WithStaticAssets()` calls in `Program.cs` as shown here:

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

Step 3: Add WebOptimizer to the HTTP Pipeline

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

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

Step 4: Update _ViewImports to enable WebOptimizer Tag Helpers

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

```
@using AutoLot.Web
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, WebOptimizer.Core
```


Part 5: Update the Index.cshtml Page

Step 1: Update the Code Behind

- Add a primary constructor in `Index.cshtml.cs` that takes `IAppLogging` and update the `OnGet` method to log an error:

```
public class IndexModel(IAppLogging logger) : PageModel
{
  public void OnGet()
  {
    logger.LogAppError("Test Error");
  }
}
```

- Run the application and launch a browser. Since the Index page is the application's default entry point, just running the app should create an error file and an entry in the `SeriLog` table.
- Once you have confirmed that logging works, comment out the error logging code:

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

- Inject the `DealerInfo OptionsMonitor` into the primary `Constructor`, add a public `DealerInfo` property, and set the value in the constructor:

```
public class IndexModel(IAppLogging logger,
 IOptionsMonitor<DealerInfo> dealerOptionsSnapshot) : PageModel
{
  [BindProperty]
  public DealerInfo Entity { get; } = dealerOptionsSnapshot.Value;

  public void OnGet()
  {
    //logger.LogAppError("Test Error");
  }
}
```


Step 2: Update the View

- Replace the HTML in `Index.cshtml` with the following:

```
@page
@model IndexModel
@inject IServiceProvider serviceProvider
@{
  ViewData["Title"] = "Home page";
  var service = serviceProvider.GetKeyedService<ISimpleService>(nameof(SimpleServiceOne));
}
<div class="text-center">
  <h1 class="display-4">Welcome to @Model.Entity.DealerName</h1>
  <p class="lead">Located in @Model.Entity.City, @Model.Entity.State</p>
</div>
<div>
  @if (service != null)
  {
    <p>@service.SayHello()</p>
  }
</div>
```

Summary

This lab added the necessary classes to 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.

