.NET 10 App Dev Hands-On Lab

MVC Lab 3 –Pipeline Configuration, Dependency Injection

This lab updates the HTTP pipeline, 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"

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

Step 2: Update the Staging Settings File

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

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

Step 3: 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"

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

Step 4: 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 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.Api.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

Manual

- 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;
global using System.Text.Json.Serialization;
```


Part 3: Update the Program.cs Top Level Statements

Copilot Agent Mode

The following prompt covers Steps 1-7

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

Add the following after 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>();
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 AddControllersWithViews to add DI validation and anti-forgery token
globally.
builder.Services.AddControllersWithViews(options =>
{
  options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());
})
    .AddControllersAsServices()
    .AddViewComponentsAsServices()
    .AddTagHelpersAsServices();
builder.Host.UseDefaultServiceProvider(o =>
{
  o.ValidateOnBuild = true;
  o.ValidateScopes = true;
});
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();
}

Add the call to map controllers and comment out the call to MapControllerRoute so the code looks
like this:
app.MapControllers();
//app.MapControllerRoute(
//        name: "default",
//        pattern: "{controller=Home}/{action=Index}/{id?}")
//    .WithStaticAssets();


Prompt: The following is to be completed in the AutoLot.MVC.csproj file:
Comment out the IncludeAssets tag for EntityFrameworkCore.Design like this:
<PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version=" [9.0.*,10.0)">
  <!--<IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>-->
  <PrivateAssets>all</PrivateAssets>
</PackageReference>
```

Manual

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

Step 5: Add Anti Forgery Token to the Entire Application

- Update the AddControllersWithViews call to the following (changes in bold):

```
builder.Services.AddControllersWithViews(options =>
{
  options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());
})
```

Step 6: Call the Data Initializer

- 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 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);
  }
}
```

Step 7: 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.

Copilot Agent Mode

The following prompt covers Steps 1-4

```
Prompt: The following work is done in Program.cs of the AutoLot.Mvc 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();

Prompt: 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 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 4: 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]")]
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("/")]
[Route("/[controller]")]
[Route("/[controller]/[action]")]
[HttpGet]
public IActionResult Index()
{
  return View();
}
```

Step 2: Add a primary constructor that gets IAppLogging and test it

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

```
public class HomeController(IAppLogging logger) : Controller
```

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

```
public IActionResult Index()
{
  logger.LogAppError("Test error");
  return View();
}
```

- 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.LogAppError("Test error");
```

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

```
public IActionResult Index([FromServices]IOptionsSnapshot<DealerInfo> dealerOptionsSnapshot)
{
  return View(dealerOptionsSnapshot. Value);
}
```

- 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 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.

