

.NET App Dev Hands-On Workshop

API Lab 2b –Pipeline, Dependency Injection

This lab configures the RESTful service. Prior to starting this lab, you must have completed API Lab 2a.

Part 1: Add the Global Usings File

- Create a new file named `GlobalUsings.cs` in the root directory of the `AutoLot.Api` project. Update it to the following:

```
global using Asp.Versioning;
global using Asp.Versioning.ApiExplorer;
global using AutoLot.Dal.EfStructures;
global using AutoLot.Dal.Exceptions;
global using AutoLot.Dal.Exceptions.Base;
global using AutoLot.Dal.Initialization;
global using AutoLot.Dal.Repos;
global using AutoLot.Dal.Repos.Base;
global using AutoLot.Dal.Repos.Interfaces;
global using AutoLot.Dal.Repos.Interfaces.Base;
global using AutoLot.Models.Entities;
global using AutoLot.Models.Entities.Base;
global using AutoLot.Services.Logging.Configuration;
global using AutoLot.Services.Logging.Interfaces;
global using AutoLot.Services.Utilities;
global using Microsoft.AspNetCore.Authentication;
global using Microsoft.AspNetCore.Authorization;
global using Microsoft.AspNetCore.Mvc;
global using Microsoft.AspNetCore.Mvc.ApiExplorer;
global using Microsoft.AspNetCore.Mvc.Authorization;
global using Microsoft.AspNetCore.Mvc.Filters;
global using Microsoft.EntityFrameworkCore;
global using Microsoft.EntityFrameworkCore.Diagnostics;
global using Microsoft.Extensions.Options;
global using Microsoft.OpenApi.Any;
global using Microsoft.OpenApi.Models;
global using Swashbuckle.AspNetCore.Annotations;
global using Swashbuckle.AspNetCore.SwaggerGen;
global using System.Net.Http.Headers;
global using System.Reflection;
global using System.Security.Claims;
global using System.Text;
global using System.Text.Encodings.Web;
global using System.Text.Json;
global using System.Text.Json.Serialization;
```

Part 2: Update the Launch Settings

- Remove the http profile and update the ports in the launchSettings.json file to be 5011 for https and 5010 for http (only relevant parts shown here):

```
"iisExpress": {
  "applicationUrl": "http://localhost:5010",
  "sslPort": 5011
}
"https": {
  "applicationUrl": "https://localhost:5011;http://localhost:5010",
},
```

Part 3: Configure the Application

Step 1: Update the Main Settings File

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

```
{
  "AllowedHosts": *
}
```

Step 2: Update the Development Settings File

- Update the appsettings.Development.json in the AutoLot.Api project to the following (adjust the connection string for your machine's setup):

```
{
  "AppLoggingSettings": {
    "MSSqlServer": {
      "TableName": "SerilogLogs",
      "Schema": "Logging",
      "ConnectionStringName": "AutoLot"
    },
    "File": {
      "Drive": "c",
      "FilePath": "temp",
      "FileName": "log_AutoLot.txt"
    },
    "General": {
      "RestrictedToMinimumLevel": "Warning"
    }
  },
  "ConnectionStrings": {
    "AutoLot": "Server=(localdb)\\MSSQLLocalDB;Database=AutoLot_Hol;Trusted_Connection=True;"
  },
  "RebuildDataBase": false,
  "AppName": "AutoLot.Api - Dev"
}
```

Step 3: Add the Staging Settings File

- Add a new JSON file to the AutoLot.Api project named appsettings.Staging.json and update the file to the following:

```
{
  "AppLoggingSettings": {
    "MSSqlServer": {
      "TableName": "SeriLogs",
      "Schema": "Logging",
      "ConnectionStringName": "AutoLot"
    },
    "File": {
      "Drive": "c",
      "FilePath": "temp",
      "FileName": "log_AutoLot.txt"
    },
    "General": {
      "RestrictedToMinimumLevel": "Warning"
    }
  },
  "ConnectionStrings": {
    "AutoLot": "Server=(localdb)\\MSSQLLocalDB;Database=AutoLot_Hol;Trusted_Connection=True;"
  },
  "RebuildDataBase": false,
  "AppName": "AutoLot.Api - Staging"
}
```

Step 4: Add the Production Settings File

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

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

Step 3: Update the Project File

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

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

Part 4: Update the Program.cs Top Level Statements

Step 1: Add services to the DI service collection

- Add Serilog support into the `WebApplicationBuilder` and add the logging interfaces to the DI container in `Program.cs` (updates in bold):

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

- 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));
    });
```

- Add the repos:

```
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>();
```

- Change the JSON formatting to Pascal casing, ignore case on incoming JSON, output JSON indented, and ignore reference cycles when serializing. Add the following code after the call to `services.AddControllers` (do not close the call with a semi colon):

```
builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.PropertyNamingPolicy = null;
        options.JsonSerializerOptions.PropertyNameCaseInsensitive = true;
        options.JsonSerializerOptions.WriteIndented = true;
        options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;
    })
```

- Configure the `ApiController` behavior by adding the following immediately after `AddJsonOptions`:

```
.ConfigureApiBehaviorOptions(options =>
```

```
{
  //suppress automatic model state binding errors
  options.SuppressModelStateInvalidFilter = true;
  //suppress all binding inference
  //options.SuppressInferBindingSourcesForParameters= true;
  //suppress multipart/form-data content type inference
  //options.SuppressConsumesConstraintForFormFileParameters = true;
  //Don't create a problem details error object if set to true
  options.SuppressMapClientErrors = false;
  options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
"https://httpstatuses.com/404";
  options.ClientErrorMapping[StatusCodes.Status404NotFound].Title = "Invalid location";
});
```

- Add the CORS policy to the services collection:

NOTE: Production applications need to be locked down.

```
builder.Services.AddCors(options =>
{
  options.AddPolicy("AllowAll", builder =>
  {
    builder
      .AllowAnyHeader()
      .AllowAnyMethod()
      .AllowAnyOrigin();
  });
});
```

Step 2: Test the Updated API Client Error Mapping Behavior

- Add a new API Controller named `ValuesController` to the `Controllers` folder, and update the code to the following:

```
namespace AutoLot.Api.Controllers;
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
  [HttpGet("/problem")]
  public IActionResult Problem() => return NotFound();
}
```

- Run the application and use SwaggerUI to test the method. You will get a result as follows:

```
{
  "type": "https://httpstatuses.com/404",
  "title": "Invalid location",
  "status": 404,
  "traceId": "00-5e3d69f9dc1c2bf9fb8b2d15374da657-3799f31b2cfd6def-00"
}
```

Step 3: Test the Logging

- Open the ValuesController and inject the logging interface into a primary constructor:

```
public class ValuesController(IAppLogging<ValuesController> logger) : ControllerBase
{
    //omitted for brevity
}
```

- Add a new Get method:

```
[HttpGet("/logging")]
public IActionResult TestLogging()
{
    logger.LogAppError("Test error");
    return Ok();
}
```

- Run the application and use the SwaggerUI to execute the end point. You will see a new record in the database and a new file created in the root of the AutoLot.Api project. Comment out the logging call when done testing.

Step 4: Configure the HTTP Pipeline

This code must be placed after the app is built with the call to `builder.Build()`:

- Add the CORS policy to the Application:

```
app.UseCors("AllowAll");
```

- 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.UseSwagger();
    app.UseSwaggerUI();
    //Initialize the database
    if (app.Configuration.GetValue<bool>("RebuildDataBase"))
    {
        using var scope = app.Services.CreateScope();
        var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
        SampleDataInitializer.ClearAndReseedDatabase(dbContext);
    }
}
```

Part 5: Create and Apply the Exception Filter

Exception filters come into play when an unhandled exception is thrown in an action method (or bubbles up to an action method).

Step 1: Create the Exception Filter

- Add a new folder named Filters into the AutoLot.Api project. Add a new class named CustomExceptionHandlerAttribute.cs in the Filters directory. Make the class public, inherit ExceptionFilterAttribute, and add a primary constructor that takes in an instance of the hosting environment, as shown here:

```
namespace AutoLot.Api.Filters;
public class CustomExceptionHandlerAttribute(IWebHostEnvironment hostEnvironment) :
    ExceptionFilterAttribute
{
    //add code into here
}
```

- The ExceptionFilter has only one method to be implemented, OnException. It takes one parameter, an ExceptionContext. The ExceptionContext provides the ActionContext, the Exception thrown, the HttpContext, ModelState, and RouteData. Use the Exception information to build a custom Response message. If the environment is development, include the stack trace. Override this from the base class as follows:

```
public override void OnException(ExceptionContext context)
{
    var ex = context.Exception;
    string stackTrace =
        hostEnvironment.IsDevelopment() ? context.Exception.StackTrace : string.Empty;
    string message = ex.Message;
    string error;
    IActionResult actionResult;
    switch (ex)
    {
        case DbUpdateConcurrencyException ce:
            //Returns a 400
            error = "Concurrency Issue.";
            actionResult = new BadRequestObjectResult(
                new { Error = error, Message = message, StackTrace = stackTrace });
            break;
        default:
            error = "General Error.";
            actionResult = new ObjectResult(
                new { Error = error, Message = message, StackTrace = stackTrace })
            {
                StatusCode = 500
            };
            break;
    }
    //context.ExceptionHandled = true; //If this is uncommented, the exception is swallowed
    context.Result = actionResult;
}
```

- Add the following global using statement to `GlobalUsings.cs`:

```
global using AutoLot.Api.Filters;
```

Step 2: Apply the Exception Filter

- Open the `Program.cs` class add the configuration to the `AddControllers` method:

```
builder.Services.AddControllers(
    config => config.Filters.Add(new CustomExceptionHandlerAttribute(builder.Environment))
)
.AddJsonOptions(options => { ... })
.ConfigureApiBehaviorOptions(options => { ... });
```

Step 3: Test the Exception Filter

- Open the `ValuesController` and create a new `HttpGet` method that throws an exception, like this:

```
[HttpGet("/error")]
public IActionResult TestExceptionHandling()
{
    throw new Exception("Test Exception");
}
```

- Run the application and use SwaggerUI to test the method. You will get a result as follows (stack trace omitted here):

```
{
  "Error": "General Error.",
  "Message": "Test Exception",
  "StackTrace": "<omitted for brevity>"
}
```

Summary

This lab configured the DI container and the HTTP Pipeline.

Next steps

In the next part of this series, you will add the controllers.