Build an ASP.NET Core Service and App with .NET (Core) 5.0 Two-Day Hands-On Lab

Lab 9

This lab is the first in a series that builds the RESTful service. Prior to starting this lab, you must have completed Lab 8 (Lab 7 is an optional lab). This entire lab works on the `AutoLot.Api` project.

Part 1: Update the Launch Settings

- Update the `launchSettings.json` in the `Properties` directory to the following (shortened for brevity):

```
{
...
  "iisSettings": {
...
    "iisExpress": {
      "applicationUrl": "http://localhost:5020",
      "sslPort": 5021
    }
  },
  "profiles": {
...
    "AutoLot.Api": {
...
      "applicationUrl": "https://localhost:5021;http://localhost:5020",
...
    }
  }
}
```

Part 2: Configure the Application

Step 1: Update the base settings

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

```
{
  "Logging": {
    "MSSqlServer": {
      "schema": "Logging",
      "tableName": "SeriLogs",
      "restrictedToMinimumLevel": "Warning"
    }
  },
  "ApplicationName": "AutoLot.MVC",
  "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):

```
{
  "Logging": {
    "MSSqlServer": {
      "schema": "Logging",
      "tableName": "SeriLogs",
      "restrictedToMinimumLevel": "Warning"
    }
  },
  "RebuildDataBase": true,
  "ConnectionStrings": {
    "AutoLot": "Server=.,5433;Database=AutoLot50;User ID=sa;Password=P@ssw0rd;"  }
}
```

Step 3: 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:

```
{
  "Logging": {
    "MSSqlServer": {
      "schema": "Logging",
      "tableName": "SeriLogs",
      "restrictedToMinimumLevel": "Warning"
    }
  },
  "RebuildDataBase": false,
  "ConnectionStrings": {
    "AutoLot": "It's a secret"
  }
}
```

Part 3: Update the Program.cs class

Step 1: Convert Main to Top Level Statements

- C# 9 supports top-level statements in place of the `Program` class/`Main` method setup. Update `Program.cs` to the following to demonstrate this:

```
using AutoLot.Api;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

CreateHostBuilder(args).Build().Run();
static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder => { webBuilder.UseStartup<Startup>(); });
```

Step 2: Add Logging

- Update the `CreateHostBuilder` method by adding the extension method from the `AutoLot.Services` project:

```
static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder => { webBuilder.UseStartup<Startup>(); })
    .ConfigureSerilog();
```

Part 4: Update the Startup.cs class

Step 1: Update the using statements

- Add the following using statements to the top of the `Statup.cs` class:

```
using System;
using System.IO;
using System.Reflection;
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Initialization;
using AutoLot.Dal.Repos;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.OpenApi.Models;
```

Step 2: Add a Class Level Variable for the Environment

- Update the constructor to take an instance of `IWebHostEnvironment` and assign that injected instance to a class level variable.

```
private readonly IWebHostEnvironment _env;
public Startup(IConfiguration configuration, IWebHostEnvironment env)
{
  _env = env;
  Configuration = configuration;
}
```

Step 3: Add the Required Services to the Dependency Injection Container

- Navigate to the `ConfigureServices` method
- Use the `IConfiguration` instance to get the connection string and use the to add the ApplicationDbContext to the container:

```
var connectionString = Configuration.GetConnectionString("AutoLot");
services.AddDbContextPool<ApplicationDbContext>(options =>
  options.UseSqlServer(connectionString, sqlOptions =>
    sqlOptions.EnableRetryOnFailure().CommandTimeout(60)));
```

- Next add the repos into the container:

```
services.AddScoped<ICarRepo, CarRepo>();
services.AddScoped<ICreditRiskRepo, CreditRiskRepo>();
services.AddScoped<ICustomerRepo, CustomerRepo>();
services.AddScoped<IMakeRepo, MakeRepo>();
services.AddScoped<IOrderRepo, OrderRepo>();
```

- Finally add the logging implementation into the container:

```
services.AddScoped(typeof(IAppLogging<>), typeof(AppLogging<>));
```

Step 4: Configure JSON Casing and API Behaviors

- Change the JSON formatting to Pascal casing. Add the following code after the call to `services.AddControllers` (do not close the call with a semi colon):

```
services.AddControllers()
  .AddJsonOptions(options =>
  {
    options.JsonSerializerOptions.PropertyNamingPolicy = null;
    options.JsonSerializerOptions.WriteIndented = true;
  })
```

- Add the methods to change the `ApiController` behavior. They are currently all commented out to leave the default behavior in place. Update the `AddControllers` method to the following:

```
services.AddControllers()
  .AddJsonOptions(options =>
  {
    options.JsonSerializerOptions.PropertyNamingPolicy = null;
    options.JsonSerializerOptions.WriteIndented = true;
  })
  .ConfigureApiBehaviorOptions(options =>
  {
    //options.SuppressConsumesConstraintForFormFileParameters = true;
    //options.SuppressInferBindingSourcesForParameters = true;
    //options.SuppressModelStateInvalidFilter = true;
    //options.SuppressMapClientErrors = true;
    //options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
"https://httpstatuses.com/404";
  });
```


Step 5: Configure CORS

This policy lets any application call the methods on the service. **NOTE:** Production applications need to be more locked down.

- Add the CORS policy in the `ConfigureService` method:

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

- Add CORS support to the Application (using the policy created in `ConfigureServices` method) in the `Configure` method:

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

Step 6: Call the Data Initializer in the Configure method

- Navigate to the Configure method and update the method signature to inject in an `ApplicationDbContext`:

```
public void Configure(
  IApplicationBuilder app,
  IWebHostEnvironment env,
  ApplicationDbContext context)
{ ... }
```

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

```
if (env.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
  app.UseSwagger();
  app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "AutoLot.Api v1"));
  //Initialize the database
  if (Configuration.GetValue<bool>("RebuildDataBase"))
  {
    SampleDataInitializer.ClearAndReseedDatabase(context);
  }
}
```

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 `CustomExceptionFilter.cs` in the Filters directory. Add the following using statements to the top of the file:

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Hosting;
```

- Make the class public and inherit `ExceptionFilterAttribute`, as shown here:

```
public class CustomExceptionFilter : ExceptionFilterAttribute
{
}
```

- Add a constructor that takes an instance of `IWebHostEnvironment` and assigns it to a private class variable:

```
private readonly IWebHostEnvironment _hostEnvironment;
public CustomExceptionFilter(IWebHostEnvironment hostEnvironment)
{
  _hostEnvironment = hostEnvironment;
}
```

- The `ExceptionFilter` has only one method to be implemented, `OnException`. Override this from the base class:

```
public override void OnException(ExceptionContext context)
{
}
```

- 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. Add the following to the `OnException` method:

```
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;
}
```

Step 2: Apply the Exception Filter

- Open the Startup.cs class and add the following using statement:

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

- Navigate to the `ConfigureServices` method. Add the configuration to the `AddControllers` method:

```
services.AddControllers(
  config => config.Filters.Add(new CustomExceptionFilter(_env))
)
.AddJsonOptions(options => { ... })
.ConfigureApiBehaviorOptions(options => { ... });
```


Step 3: Test the Exception Filter

- Open the `WeatherForecastController` and navigate to the Get method. Add an exception to the action method, like this:

```
[HttpGet]
public IEnumerable<WeatherForecast> Get()
{
  throw new Exception("Test Exception");
  var rng = new Random();
  return Enumerable.Range(1, 5).Select(index => new WeatherForecast
  {
    Date = DateTime.Now.AddDays(index),
    TemperatureC = rng.Next(-20, 55),
    Summary = Summaries[rng.Next(Summaries.Length)]
  })
  .ToArray();
}
```

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

```
{
  "Error": "General Error.",
  "Message": "Test Exception",
  "StackTrace": "   at AutoLot.Api.Controllers.WeatherForecastController.Get() in
D:\\Projects\\dotnetcore_hol\\TwoDayHandsOnLabFiles\\5.0\\Labs\\Lab9\\AutoLot.Api\\Controllers\\We
atherForecastController.cs:line 29\r\n   at lambda_method279(Closure , Object , Object[] )\r\n   "
}
```

Summary

This lab configured the DI container and the HTTP Pipeline.

Next steps

In the next part of this tutorial series, you will add Swagger support to the services project.