.NET 10 App Dev Hands-On Workshop

API Lab 2b –Pipeline, Dependency Injection

This lab configures the RESTful service. Before starting this lab, you must have completed MVC/RP/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.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.Models.Exceptions.Base;

global using AutoLot.Services.Logging.Configuration;
global using AutoLot.Services.Logging.Interfaces;

global using Microsoft.AspNetCore.Mvc;
global using Microsoft.AspNetCore.Mvc.Filters;
global using Microsoft.AspNetCore.Mvc.Formatters;

global using Microsoft.EntityFrameworkCore;
global using Microsoft.EntityFrameworkCore.Diagnostics;

global using System.Dynamic;
global using System.Reflection;
global using System.Text;
global using System.Text.Json;
global using System.Text.Json.Serialization;
```

Part 2: 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": "SeriLogs",
      "Schema": "Logging",
      "ConnectionStringName": "AutoLot"
    },
    "File": {
      "Drive": "c",
      "FilePath": "temp",
      "FileName": "log_AutoLot_Development.txt"
    },
    "General": {
      "RestrictedToMinimumLevel": "Warning"
    }
  },
  "ConnectionStrings": {
    "AutoLot": "Server=(localdb)\\MSSQLLocalDB;Database=AutoLot_Hol;Trusted_Connection=True;"
  },
  "RebuildDataBase": true,
  "AppName": "AutoLot.Api - Dev"
}
```


Step 3: Update 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_Staging.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 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 3: Update the Program.cs Top Level Statements

Copilot Agent Mode

The following prompts cover Steps 1 and 2:

```
Setup 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.Api project.

Prompt: The following is all done in Program.cs

Add the following after the call to CreateBuilder:

builder.ConfigureSerilog();
builder.Services.RegisterLoggingInterfaces();

Add the following after AddControllers():
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>();

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 AddControllers to add customize the JSON handling, add DI validation,
and customize the API behavior:
  builder.Services.AddControllers()
  .AddJsonOptions(options =>
  {
    options.JsonSerializerOptions.PropertyNamingPolicy = null;
    options.JsonSerializerOptions.PropertyNameCaseInsensitive = true;
    options.JsonSerializerOptions.WriteIndented = true;
    options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;
  })
  .AddControllersAsServices()
  .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";
  });

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

Prompt: Add the CORS policy to the services collection:
builder.Services.AddCors(options =>
{
  options.AddPolicy("AllowAll", pb =>
  {
    pb
      .AllowAnyHeader()
      .AllowAnyMethod()
      .AllowAnyOrigin();
  });
});
```

```
Prompt: In the section after builder.Build() in the IsDevelopment if block add the code to
initialize the database like this:
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
  if (app.Configuration.GetValue<bool>("RebuildDataBase"))
  {
    using var scope = app.Services.CreateScope();
    var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
    SampleDataInitializer.ClearAndReseedDatabase(dbContext);
  }
}

Add the code to UseCors:
app.UseCors("AllowAll");
```

Manual

Step 1: Add services to the DI service collection and configure the Controllers

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

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

- 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 since you will add to this block in the next step):

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

```
builder.Services.AddControllers()
  .AddJsonOptions(//omitted)
  .AddControllersAsServices()
  .ConfigureApiBehaviorOptions(//omitted);

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

- Add the `ApplicationDbContext` after the call to `AddControllers`:

```
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>();
```

- Add the CORS policy to the services collection (it will be added to the HTTP pipeline later in this lab):

NOTE: Production applications must be locked down and not wide open like this example.

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

Step 2: Configure the HTTP Pipeline

This code must be placed after the call to `builder.Build()`. I usually place it right before the call to `app.UseHttpsRedirection()`:

- 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.MapOpenApi();
  //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 4: Create and Apply the Exception Filter

Exception filters are invoked when an unhandled exception is thrown in an action method (or bubbles up to one).

Copilot Agent Mode

The following prompts cover Steps 1 and 2:

```
Prompt: All work is to be done in the Autolot.Api project.
Add a new folder named Filters. Add a new class named CustomExceptionFilterAttribute.cs in the
Filters directory and update the code to match the following:
namespace AutoLot.Api.Filters;
public class CustomExceptionFilterAttribute(IWebHostEnvironment hostEnvironment) :
ExceptionFilterAttribute
{
  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.Result = actionResult;
  }
}

Add the following global usings to the GlobalUsings.cs file if it does not already exist (sorted
alphabetically. Don't remove any existing global using statements).
global using AutoLot.Api.Filters;

The following work is to be done in the Program.cs class. Update the AddControllers method to add
the filter:
builder.Services.AddControllers(
  config => config.Filters.Add(new CustomExceptionFilterAttribute(builder.Environment))
)
```

Manual

Step 1: Create the Exception Filter

- Add a new folder named Filters into the `AutoLot.Api` project. Add a new class named `CustomExceptionFilterAttribute.cs` in the Filters directory and update the code to match the following:

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

- The Exception Filter has only one method to be implemented, `OnException`. 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 CustomExceptionFilterAttribute(builder.Environment))
)
.AddControllersAsServices()
.AddJsonOptions(//omitted)
.ConfigureApiBehaviorOptions(//omitted);
```

Part 5: Add the ValuesController

Copilot Agent Mode

```
Prompt: 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(IAppLogging appLogging) : ControllerBase
{
  [HttpGet("problem")]
  public IActionResult Problem() => NotFound();
  [HttpGet("logging")]
  public IActionResult TestLogging()
  {
    logger.LogAppError("Test error");
    return Ok();
  }
  [HttpGet("error")]
  public IActionResult TestExceptionHandling() => throw new Exception("Test Exception");
}
```

Manual

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

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

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

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


Part 6: Test Logging and Exception Handling

Step 1: Test the Updated API Client Error Mapping Behavior

- Run the application and use Bruno or CURL to test the `Problem` endpoint. Either way, execute the following endpoint:

```
https://localhost:5011/api/Values/problem
```

- You will get a result as follows (your `traceId` value will be different):

```
{
  "type": "https://httpstatuses.com/404",
  "title": "Invalid location",
  "status": 404,
  "traceId": "00-3658bf06721852174d5e3476d1c48e21-8717962ad3817271-00"
}
```

Step 2: Test the Logging

- Run the application and use Bruno or CURL to execute the `logging` endpoint:

```
https://localhost:5011/api/Values/logging
```

- You will see a new record in the database and a new file created in the root of the `AutoLot.Api` project. When you are done testing, comment out the logging call.

Step 3: Test the Exception Filter

- Run the application and use Bruno or CURL to test the method:

```
https://localhost:5011/api/Values/error
```

- You will get the 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.