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

Lab 8

This lab builds the shared services used by the ASP.NET Core applications. Prior to starting this lab, you must have completed Lab 6 (Lab 7 is an optional lab). The entire lab works in the `AutoLot.Services` project.

Part 1: Add Logging Support

- Delete the generated `Class1.cs` file.

Step 1: Add the Logging Interface

- Add a new folder named `Logging` in the `AutoLot.Services` project. In that folder add an interface file named `IAppLogging.cs`. Update the interface code to the following:

```
using System;
using System.Runtime.CompilerServices;

namespace AutoLot.Services.Logging
{
  public interface IAppLogging<T>
  {
    void LogAppError(Exception exception, string message,
      [CallerMemberName] string memberName = "",
      [CallerFilePath] string sourceFilePath = "",
      [CallerLineNumber] int sourceLineNumber = 0);

    void LogAppError(string message,
      [CallerMemberName] string memberName = "",
      [CallerFilePath] string sourceFilePath = "",
      [CallerLineNumber] int sourceLineNumber = 0);

    void LogAppCritical(Exception exception, string message,
      [CallerMemberName] string memberName = "",
      [CallerFilePath] string sourceFilePath = "",
      [CallerLineNumber] int sourceLineNumber = 0);

    void LogAppCritical(string message,
      [CallerMemberName] string memberName = "",
      [CallerFilePath] string sourceFilePath = "",
      [CallerLineNumber] int sourceLineNumber = 0);

    void LogAppDebug(string message,
      [CallerMemberName] string memberName = "",
      [CallerFilePath] string sourceFilePath = "",
      [CallerLineNumber] int sourceLineNumber = 0);

    void LogAppTrace(string message,
```

```
      [CallerMemberName] string memberName = "",
      [CallerFilePath] string sourceFilePath = "",
      [CallerLineNumber] int sourceLineNumber = 0);

    void LogAppInformation(string message,
      [CallerMemberName] string memberName = "",
      [CallerFilePath] string sourceFilePath = "",
      [CallerLineNumber] int sourceLineNumber = 0);

    void LogAppWarning(string message,
      [CallerMemberName] string memberName = "",
      [CallerFilePath] string sourceFilePath = "",
      [CallerLineNumber] int sourceLineNumber = 0);
  }
}
```

Step 2: Add the Logging Implementation

- In the `Logging` folder add a class file named `AppLogging.cs`. Add the following using statements to the class:

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using Serilog.Context;
```

- Make the class public and generic, and implement `IAppLogging`:

```
namespace AutoLot.Services.Logging
{
  public class AppLogging<T> : IAppLogging<T>
  {
  }
}
```

- Inject into the constructor the framework `ILogger<T>` and an instance of `IConfiguration` and create private variables for them. Also get the application name from the configuration:

```
private readonly ILogger<T> _logger;
private readonly IConfiguration _config;
private readonly string _applicationName;

public AppLogging(ILogger<T> logger, IConfiguration config)
{
  _logger = logger;
  _config = config;
  _applicationName = config.GetValue<string>("ApplicationName");
}
```

- Create an internal method to push the additional properties into the `SeriLog` context:

```
internal List<IDisposable> PushProperties(
  string memberName,
  string sourceFilePath,
  int sourceLineNumber)
{
  List<IDisposable> list = new List<IDisposable>
  {
    LogContext.PushProperty("MemberName", memberName),
    LogContext.PushProperty("FilePath", sourceFilePath),
    LogContext.PushProperty("LineNumber", sourceLineNumber),
    LogContext.PushProperty("ApplicationName", _applicationName)
  };
  return list;
}
```

- Implement the logging interface members:

```
public void LogAppError(Exception exception, string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0)
{
  var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
  _logger.LogError(exception, message);
  foreach (var item in list) { item.Dispose(); }
}
public void LogAppError(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0)
{
  var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
  _logger.LogError(message);
  foreach (var item in list) { item.Dispose(); }
}
public void LogAppCritical(Exception exception, string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0)
{
  var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
  _logger.LogCritical(exception, message);
  foreach (var item in list) { item.Dispose(); }
}
public void LogAppCritical(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0)
{
  var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
  _logger.LogCritical(message);
  foreach (var item in list) { item.Dispose(); }
}
```



```
public void LogAppDebug(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0)
{
  var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
  _logger.LogDebug(message);
  foreach (var item in list) { item.Dispose(); }
}
public void LogAppTrace(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0)
{
  var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
  _logger.LogTrace(message);
  foreach (var item in list) { item.Dispose(); }
}
public void LogAppInformation(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0)
{
  var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
    _logger.LogInformation(message);
  foreach (var item in list) { item.Dispose(); }
}
public void LogAppWarning(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0)
{
  var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
  _logger.LogWarning(message);
  foreach (var item in list) { item.Dispose(); }
}
```

Step 3: Add the Logging Configuration Extension Method

- Add a new class file named `LoggingConfiguration.cs` to the `Logging` directory. Add the following using statements to the top of the file:

```
using System;
using System.Collections.Generic;
using System.Data;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Serilog;
using Serilog.Events;
using Serilog.Sinks.MSSqlServer;
```

- Make the class public and static:

```
namespace AutoLot.Services.Logging
{
  public static class LoggingConfiguration
  {
  }
}
```

- Add variables to hold the output template (for text file logging) and the `ColumnOptions` (for SQL Server logging):

```
private static readonly string OutputTemplate =
    @"[{Timestamp:yy-MM-dd HH:mm:ss}
{Level}]{ApplicationName}:{SourceContext}{NewLine}Message:{Message}{NewLine}in method {MemberName}
at {FilePath}:{LineNumber}{NewLine}{Exception}{NewLine}";

private static readonly ColumnOptions ColumnOptions = new ColumnOptions
{
  AdditionalColumns = new List<SqlColumn>
  {
    new SqlColumn {DataType = SqlDbType.VarChar, ColumnName = "ApplicationName"},
    new SqlColumn {DataType = SqlDbType.VarChar, ColumnName = "MachineName"},
    new SqlColumn {DataType = SqlDbType.VarChar, ColumnName = "MemberName"},
    new SqlColumn {DataType = SqlDbType.VarChar, ColumnName = "FilePath"},
    new SqlColumn {DataType = SqlDbType.Int, ColumnName = "LineNumber"},
    new SqlColumn {DataType = SqlDbType.VarChar, ColumnName = "SourceContext"},
    new SqlColumn {DataType = SqlDbType.VarChar, ColumnName = "RequestPath"},
    new SqlColumn {DataType = SqlDbType.VarChar, ColumnName = "ActionName"},
  }
};
```


- Add the `IHostBuilder` extension method to register `Serilog` as the logging framework for ASP.NET Core:

```
public static IHostBuilder ConfigureSerilog(this IHostBuilder builder)
{
  builder
    .ConfigureLogging((context, logging) => { logging.ClearProviders(); })
    .UseSerilog((hostingContext, loggerConfiguration) =>
    {
      var config = hostingContext.Configuration;
      var connectionString = config.GetConnectionString("AutoLot").ToString();
      var tableName = config["Logging:MSSqlServer:tableName"].ToString();
      var schema = config["Logging:MSSqlServer:schema"].ToString();
      string restrictedToMinimumLevel =
      config["Logging:MSSqlServer:restrictedToMinimumLevel"].ToString();
      //LogEventLevel logLevel = log;
      if (!Enum.TryParse<LogEventLevel>(restrictedToMinimumLevel, out var logLevel))
      {
        logLevel = LogEventLevel.Debug;
      }
      LogEventLevel level =
        (LogEventLevel)Enum.Parse(typeof(LogEventLevel), restrictedToMinimumLevel);
      loggerConfiguration
        .Enrich.FromLogContext()
        .Enrich.WithMachineName()
        .ReadFrom.Configuration(hostingContext.Configuration)
        .WriteTo.File(
          path:"ErrorLog.txt",
          rollingInterval:RollingInterval.Day,
          restrictedToMinimumLevel:logLevel,
          outputTemplate:OutputTemplate)
        .WriteTo.Console(restrictedToMinimumLevel:logLevel)
        .WriteTo.MSSqlServer(
          connectionString: connectionString,
          new MSSqlServerSinkOptions
          {
            AutoCreateSqlTable = false,
            SchemaName = schema,
            TableName = tableName,
            BatchPeriod = new TimeSpan(0,0,0,1)
          },
          restrictedToMinimumLevel: level,
          columnOptions: ColumnOptions);
    });
  return builder;
}
```

Part 2: Add Service Wrapper and related files

The `ApiServiceWrapper` will be used by the ASP.NET Core web application to call into the ASP.NET Core service.

- Add a new folder named `ApiWrapper` in the `AutoLot.Services` project.

Step 1: Add the Settings File

- In the `ApiWrapper` folder add a class file named `ApiServiceSettings.cs`. Update the class to the following code:

```
namespace AutoLot.Services.ApiWrapper
{
  public class ApiServiceSettings
  {
    public ApiServiceSettings() { }
    public string Uri { get; set; }
    public string CarBaseUri { get; set; }
    public string MakeBaseUri { get; set; }
  }
}
```

Step 2: Add the Interface

- In the `ApiWrapper` folder add an interface file named `IApiServiceWrapper.cs`. Update the interface code to the following:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using AutoLot.Models.Entities;

namespace AutoLot.Services.ApiWrapper
{
  public interface IApiServiceWrapper
  {
    Task<IList<Car>> GetCarsAsync();
    Task<IList<Car>> GetCarsByMakeAsync(int id);
    Task<Car> GetCarAsync(int id);
    Task<Car> AddCarAsync(Car entity);
    Task<Car> UpdateCarAsync(int id, Car entity);
    Task DeleteCarAsync(int id, Car entity);
    Task<IList<Make>> GetMakesAsync();
  }
}
```


Step 3: Add the Implementation

- Add a class file named `ApiServiceWrapper.cs`. Add the following using statements to the top of the file:

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Net.Http.Json;
using System.Text;
using System.Text.Json;
using System.Threading.Tasks;
using AutoLot.Models.Entities;
using Microsoft.Extensions.Options;
```

- Add a constructor that takes an `HttpClient` and `IOptionsMonitor<ServiceSettings>` and assigns them to private variables:

```
namespace AutoLot.Services.ApiWrapper
{
  public class ApiServiceWrapper : IApiServiceWrapper
  {
    private readonly HttpClient _client;
    private readonly ApiServiceSettings _settings;
    public ApiServiceWrapper(HttpClient client, IOptionsMonitor<ApiServiceSettings> settings)
    {
      _client = client;
      _settings = settings.CurrentValue;
      _client.BaseAddress = new Uri(_settings.Uri);
    }
  }
}
```

- Add three internal helper methods for post, put, and delete:

```
internal async Task<HttpResponseMessage> PostAsJson(string uri, string json)
{
  return await _client.PostAsync(uri, new StringContent(json, Encoding.UTF8, "application/json"));
}
internal async Task<HttpResponseMessage> PutAsJson(string uri, string json)
{
  return await _client.PutAsync(uri, new StringContent(json, Encoding.UTF8, "application/json"));
}
internal async Task<HttpResponseMessage> DeleteAsJson(string uri, string json)
{
  HttpRequestMessage request = new HttpRequestMessage
  {
    Content = new StringContent(json, Encoding.UTF8, "application/json"),
    Method = HttpMethod.Delete,
    RequestUri = new Uri(uri)
  };
  return await _client.SendAsync(request);
}
```

- Add the methods to get data from the service:

```
public async Task<IList<Car>> GetCarsAsync()
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.CarBaseUri}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadFromJsonAsync<IList<Car>>();
  return result;
}
public async Task<IList<Car>> GetCarsByMakeAsync(int id)
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.CarBaseUri}/bymake/{id}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadFromJsonAsync<IList<Car>>();
  return result;
}
public async Task<Car> GetCarAsync(int id)
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.CarBaseUri}/{id}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadFromJsonAsync<Car>();
  return result;
}
public async Task<IList<Make>> GetMakesAsync()
{
  var response = await _client.GetAsync($"{_settings.Uri}{_settings.MakeBaseUri}");
  response.EnsureSuccessStatusCode();
  var result = await response.Content.ReadFromJsonAsync<IList<Make>>();
  return result;
}
```

- Add the methods to Add, Update, and Delete a Car record:

```
public async Task<Car> AddCarAsync(Car entity)
{
  var response = await PostAsJson($"{_settings.Uri}{_settings.CarBaseUri}",
    JsonSerializer.Serialize(entity));
  if (response == null)
  {
    throw new Exception("Unable to communicate with the service");
  }
  var location = response.Headers?.Location?.OriginalString;
  return await response.Content.ReadFromJsonAsync<Car>();
}
public async Task<Car> UpdateCarAsync(int id, Car entity)
{
  var response = await PutAsJson($"{_settings.Uri}{_settings.CarBaseUri}/{id}",
    JsonSerializer.Serialize(entity));
  response.EnsureSuccessStatusCode();
  return await response.Content.ReadFromJsonAsync<Car>();
}
public async Task DeleteCarAsync(int id, Car entity)
{
  var response = await DeleteAsJson($"{_settings.Uri}{_settings.CarBaseUri}/{id}",
    JsonSerializer.Serialize(entity));
  response.EnsureSuccessStatusCode();
}
```


Step 4: Add the ApiService Configuration Extension Method

- Add a new class file named `ServiceConfiguration.cs` to the `ApiWrapper` directory. Update the code to match the following:

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
namespace AutoLot.Services.ApiWrapper
{
  public static class ServiceConfiguration
  {
    public static IServiceCollection ConfigureApiServiceWrapper(
      this IServiceCollection services, IConfiguration config)
    {
      services.Configure<ApiServiceSettings>(config.GetSection(nameof(ApiServiceSettings)));
      services.AddHttpClient<IApiServiceWrapper,ApiServiceWrapper>();
      return services;
    }
  }
}
```

Part 3: Add the String Utility Extension Method

- Add a new folder named `Utilities` and, in that folder, add a new class file named `StringExtensions.cs`. Update the code to match the following:

```
using System;
namespace AutoLot.Services.Utilities
{
  public static class StringExtensions
  {
    public static string RemoveController(this string original)
      => original.Replace("Controller", "", StringComparison.OrdinalIgnoreCase);
  }
}
```

Summary

This lab created the Services project used by the ASP.NET Core projects.

Next steps

In the next part of this tutorial series, you will start building the ASP.NET Core RESTful service.