Exception Handling Filters:

Available in .net core. Its available in the form of middleware components.

Diagnostic Middleware:

Filter logic doesn’t apply to the static files. So logic in it only or the logic on controller.

HTTP is not a very good protocol. not known for its performance. Since Microservice is available, communication becomes very fast.

GRPC is adopting the microservices.

Instead of using Controller if we use GRPC -> Service (filters won’t get applied to GRPC controller).

Instead of using Controller if we use Hub -> Service -> repo -> DB (filters not applied to Hub)

So we use middleware that gets applied to everything.

Instead of using a exception handling filter, we use the middleware ex. Handling.

One OOTB – Developer exception page. Gets added to the configure method by default. Only in development env. Then it gets used.

If error is thrown, then the request to home controller returns an HTML page. But if web api request then the output is in the form of text. (stack trace only). When using the developer exception handling middleware.

When using production env . then create a new req pipeline of exception -> add.UseExceptionHandler(“/Home/Error”);

Which part of the application caused the exception?

What is the exception?

IexceptionHandlerpathfeature (path and exception ) and I IexceptionHandlerfeature (just the exception) – they provide the information.

Error controller has more code on this.

Change the proj env to prod. And now run and check the privacy view under Home.

Now hit the api get orders (id).

Also please comment the middleware – UseSecurity – uses no sniff – content type sniffing disabled. Hence you wouldn’t see the appropriate error response.

The error is now generated in the below format:

{

    "type": "System.InvalidOperationException",

    "title": "Sequence contains no elements",

    "status": 500,

    "detail": "   at System.Linq.ThrowHelper.ThrowNoElementsException()\r\n   at System.Linq.Enumerable.Single[TSource](IEnumerable`1 source)\r\n   at Microsoft.EntityFrameworkCore.Query.Internal.QueryCompiler.Execute[TResult](Expression query)\r\n   at Microsoft.EntityFrameworkCore.Query.Internal.EntityQueryProvider.Execute[TResult](Expression expression)\r\n   at System.Linq.Queryable.Single[TSource](IQueryable`1 source)\r\n   at TrainingApplication.Controllers.OrderController.Get(Int32 id) in C:\\Users\\kasinha\\source\\repos\\TrainingApplication\\TrainingApplication\\Controllers\\OrderController.cs:line 75\r\n   at lambda\_method(Closure , Object , Object[] )\r\n   at Microsoft.Extensions.Internal.ObjectMethodExecutor.Execute(Object target, Object[] parameters)\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor.SyncActionResultExecutor.Execute(IActionResultTypeMapper mapper, ObjectMethodExecutor executor, Object controller, Object[] arguments)\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeActionMethodAsync()\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(State& next, Scope& scope, Object& state, Boolean& isCompleted)\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeNextActionFilterAsync()\r\n--- End of stack trace from previous location where exception was thrown ---\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Rethrow(ActionExecutedContextSealed context)\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(State& next, Scope& scope, Object& state, Boolean& isCompleted)\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeInnerFilterAsync()\r\n--- End of stack trace from previous location where exception was thrown ---\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeNextResourceFilter>g\_\_Awaited|24\_0(ResourceInvoker invoker, Task lastTask, State next, Scope scope, Object state, Boolean isCompleted)\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Rethrow(ResourceExecutedContextSealed context)\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Next(State& next, Scope& scope, Object& state, Boolean& isCompleted)\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.InvokeFilterPipelineAsync()\r\n--- End of stack trace from previous location where exception was thrown ---\r\n   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>g\_\_Awaited|17\_0(ResourceInvoker invoker, Task task, IDisposable scope)\r\n   at Microsoft.AspNetCore.Routing.EndpointMiddleware.<Invoke>g\_\_AwaitRequestTask|6\_0(Endpoint endpoint, Task requestTask, ILogger logger)\r\n   at Microsoft.AspNetCore.Authorization.AuthorizationMiddleware.Invoke(HttpContext context)\r\n   at Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware.<Invoke>g\_\_Awaited|6\_0(ExceptionHandlerMiddleware middleware, HttpContext context, Task task)",

    "traceId": "|d0402410-48bb9255b0e1a84d."

}

**Configuration Providers:**

* Azure Key Vault,………..
* Prev our application ootb supported was web.config. We needn’t write a plumbing logic to fetch the data from the we.config.
* Now we have a very rich configuration system. Variety of formats, variety of location.
* Each configuration provider targets a diff location.
* Default config file is a JSON.. we can use XML or INI (really old) as well.
* We can create a custom configuration provider. YAML is the newest one available. There is not YAML configuration provider but in nuget lot of them available.
* Environment variable are a common way to provide config values to app running in a container.
* Instead of storing sensitive info in the plain text – use user Secrets manager.
* JSON – appsettings, appsettings.Development.JSON.
* Env variables, command line (for console based app) configuration provider. We can create another JSON file for logging.

CreateHostBuilder

= configures all the configuration providers and makes them available to our host.

If we want to override, use the below code under CreateHostBuilder

.ConfigureAppConfiguration((context, options) =>

          {

              options.Sources.Clear(); //first clear the existing config providers

              options.AddJsonFile("appSettings.json", false, true);

              options.AddJsonFile($"appSettings.{context.HostingEnvironment.EnvironmentName}.json", true, true);

              options.AddEnvironmentVariables(); //use the env. variables for the app if available.

              options.AddCommandLine(args); //at the end --- latest entry will override will previous value.

          });

See below. In production env. Bool reloadOnCHange -> true (if contents of the file change, then the application reloads the file without shutting down the apication.

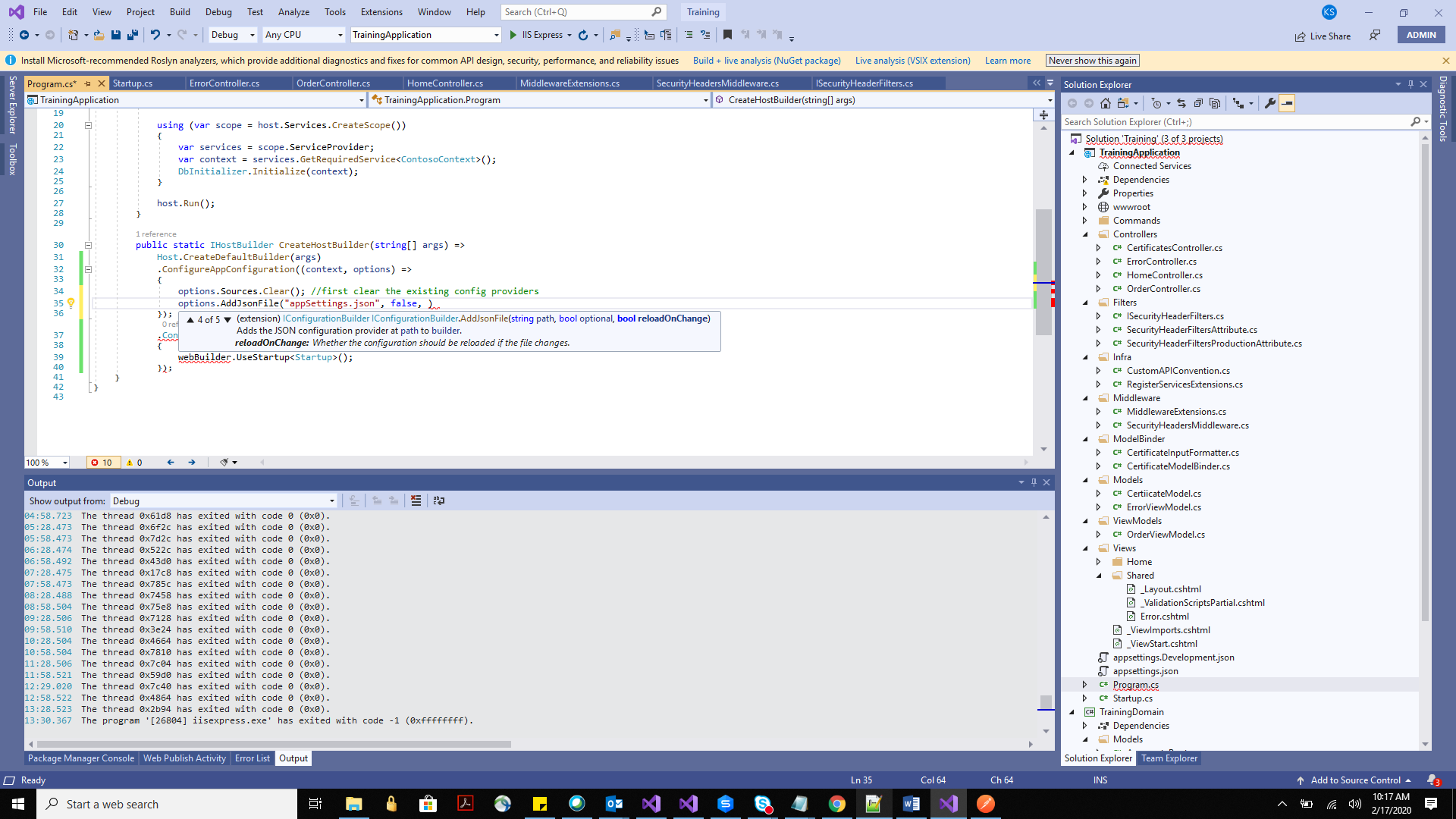
Mandatory file

Thhird file is optional – and env specific file.

The value in the env based JSON file will override the one in the orig file.

Command line configuration provider in the end, if I pass some information in the command line, it should supercedes everything.

If you want to add user secret or azure key vault or blah – add everything explicitly.



Web.config if you change --- application in asp.net would shut down .. and a new app domain would get created. – recycling of app domain. Production users impacted. Because it used to contain version and pipeline etc. information of the application.

**Accessing configuration:**

* Inject IConfiguration service anywhere in the application.
* GetValue, GetSection etc. to read configuration information.

Configuration.GetValue<string>(“ConnectionString:DefaultConnection”);

Configuration.GetValue<string>(“env variable…. blah”);

Needn’t mention the source of the key… we can just put the name and we get the value,

**Option Pattern:**

We can forget the name of the keys while getting the values. So we create an options class. ApplicationConfiguration in our application we are creating.

Register this options class (ApplicationConfiguration )with the DI.

services.Configure<ApplicationConfiguration>(Configuration);

Configuration is the Iconfiguration variable which is already injected in the startup constructor.

Keys provided on these options class are supposed to be exactly the same as the ones present on either of the configuration provider.

Now you were initiallu using the IConfiguration configuration

In the home controller constructor directly… inject IOptions<ApplicationConfiguration> options instead.

public HomeController(ILogger<HomeController> logger, IWebHostEnvironment environment, IConfiguration configuration,

    IOptions<ApplicationConfiguration> options)

You can compare the configuration and the options object to see the different.

And you could access the information via options.Value…

Whatever it is not able to find in any of the configuration provider, default values are assigned.

Recommended that multiple options class should be used – interface segregation principle – not every property should be made available to all parts of the application.

Email only sending should be allowed the uses of SMTP.

UoW only should allowed the connection strings.

Also we can instead of IOtions, we can use IOptionsMonitor or IOptionsSnapshot (available only in above version 3).

IOPtions is available for the backward compatibility. Use the newer things please.

In case of OptionsMonitor…

Options.CurrentValue, Options.onchange event handler ??????????// read about this – google //// didn’t hear. (if you are not really interested in the change monitoring via the event handler, use the ISnapshotMonitor – find and replace IOptions in your existing app to IsnapshotsMonitor)

IOptionsSnapshot. – simimar to IOptions. 0 No onchange event.

Snapshot an monitor difference - > a particular httprequest will get a single snapshot of the config values. A new instance of this service is created for every HTTP request. Different snapshot to different request. Significance of this is

IoptionSnapshot is scoped. Ioptionsmonitor -> singleton

**App Secrets** – Anything that should not be visible to the user (clear text), we use App Secrets (sensitive inf)

Store the information outside our code repository. AppSettings.Json is the part of the code repository.

AppSettings is a mandatory file. Has to be always present on the application.

Use the feature

* Secrets Manager
* Environment Variable
* Azure Key Vault. (paid service – requires internet and a paid subscription.

Secrets Manager and Environment Variable still store the data in plain text but they are not a part of the code repository.

Azure key vault allows encryption and out of the code repository storing.

Assigned your project a secret ID – GUID

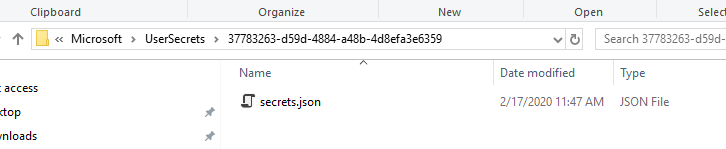
A system protected user specific folder created – its created the specific user’s profile

Right click and manage user secrets on the project. A secret Json would be created.

Any other user (not admin) would not have the access to this file. If you close this file by mistake you will find it in the below path.

C:\Users\kasinha\AppData\Roaming\Microsoft\UserSecrets

%AppData%\Roaming\......



The csproj file contains the same GUID and it is registered in the csproj file.

The same guid can be shared across the 10 csproj. They will refer to the same secrets.json



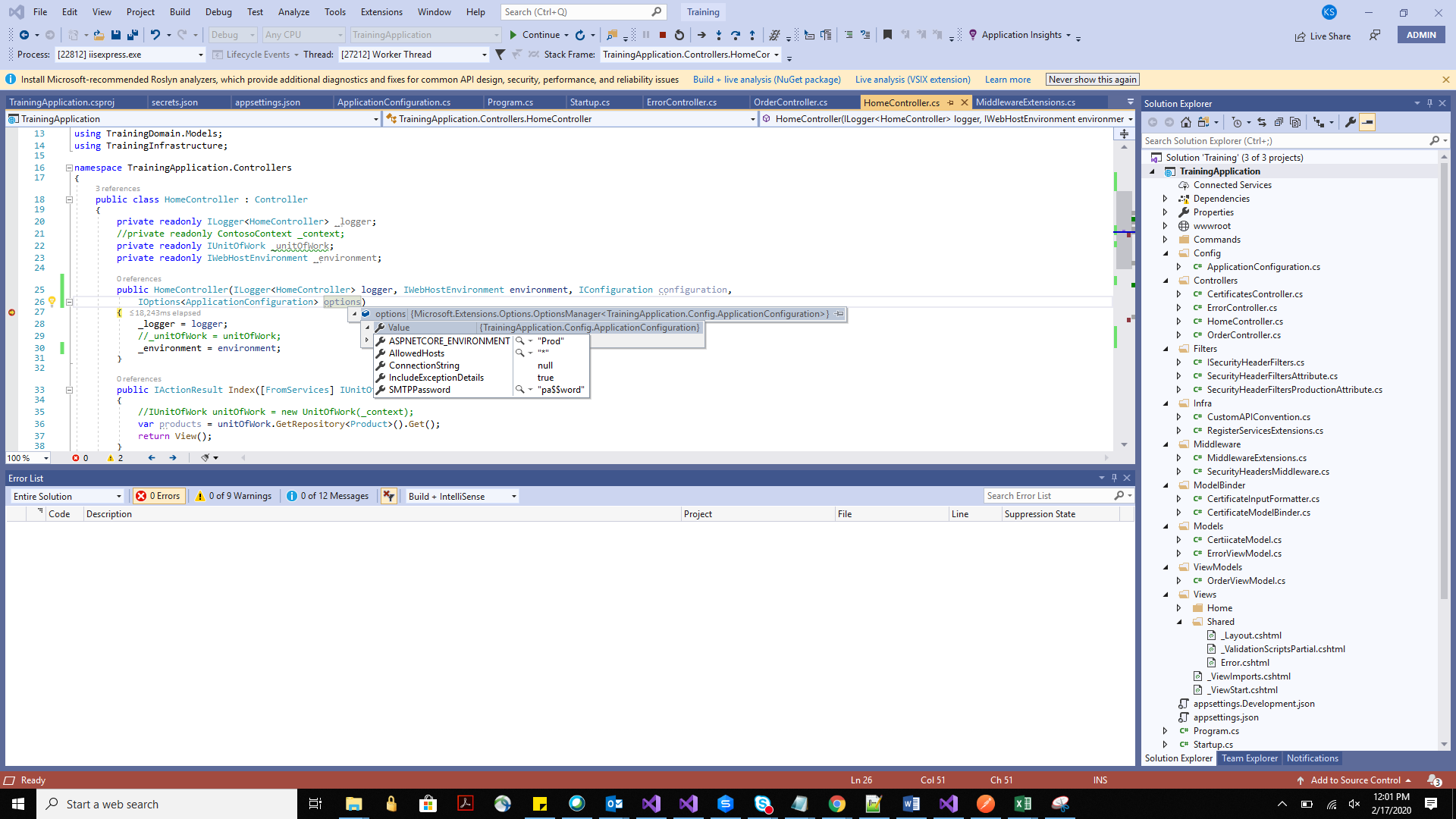
To read from the secrets.json, we need to add this configuration provider.

options.AddUserSecrets<Startup>();

why did we provide the Startup class to the extension method?

You can give the name of any class which id present on the csproj (the assembly of this class as in the csproj underwhich this class is present – )

During compile time the manifest(dll) will contain the guid for secret JSOn.



See IOptions pulled it from the secret JSON.

How do I deploy it on another environment. Because secret json is not a part of the code repository.

Steps need to be taken to ensure the variables are present in the other env. (like env variable, secret folder, key vault)

CLI available to script this using powershell or dotnet CLI.

Right click and click open command line -> default(cmd)

dotnet user-secret init

dotnet user-secret list -> list all the secrets stored.

Using this dotnet CLI we can add the key on an environment.

If we are working with docker or a new env, tell the admin to run these commands, or create a power shell script to be executed on the new env.

**Logging Providers:**

Below are the OOTB logging providers available. – Ilogger

* Console, (not very useful for visual studio – good in dev- when application stops, it goes away)
* Debug, (not very useful for visual studio – good in dev- when application stops, it goes away)
* EventSource, ---- Best one of these.
* EventLog(only windows – hence useless – because today the trend Is to use a container – it is generally a Linux not windows container)

Multiple logging providers can be registered.

nLog, SeriLog – popular third party logging libraries.

SeriLog is the most feature rich library.

nLog is the oldest.

**Creating Logs:**

Don’t write any business logic in the startup.Configure as the information wouldn’t be available in the logs. Reason is as below:

Before .net core 3.0,

Host (creates a DI container for startup) -> Startup creates a DI container for the application but now there are 2 DI containers.

Now Host -> Startup -> this only creates one DI container (at this stage Ilogger service is configured in the DI container) -> Only after this, the application can use the DI container and then the Ilogger.

Using a 3rd party Logging

* Read and Apply the logger configuration (filesystem logs or blah….) create a dedicated JSON for logging
* Create a logger component – this uses the third party logging library.
* Register it with the DI container.

<http://quartzsystems.com/downloads/core3/logging.txt>

Serilog – 80M downloads – add this nuget (there are Serilog sinks and enricher)

Serilog.AspNetCore – this one as well

Serilog.Sinks is a logging provider which decides where to log the messages (in DB, filesystem .. blah).

Serilog.Sinks.ColoredConsole – different colors on the console .. this as well.

Serilog.Sinks.MSSqlServer – this as well.

Serilog.Sinks.Seq – doesn’t just collects the information, helps us analyse the log messages – because it segregates the messages. When we are writing logs over a file, we need some assistance.

* We needn’t use it now because to use this we should install Seq in our system.
* When using the container – we can just copy the SEQ in our container.

Serilog.enrichers.

----- See we created Logging.Json – that’s read and apply.

Then we wrote the code to read the configuration in program.cs and then created a logger component.

(This is why we should keep the logging config in a separate file)

And now whaterver code in program.cs for application initialization that can be put in the try and catch and the logging mechanism can detect errors when we are actually moving to a new env, deployment, application initialization, during data seeding… right at the beginning we can catch the errors.

Different levels can be assigned to the log messages – fatal/critical, information, warning …. Blah .. check the Slide.

public static Serilog.ILogger ConfigureLogger()

     {

         //read configuration data

         IConfiguration configuration = new ConfigurationBuilder()

             .SetBasePath(Directory.GetCurrentDirectory())

             .AddJsonFile("logging.json")

             .AddUserSecrets<Startup>()

             .Build();

We have added the user secrets for getting the conn string because we want to store the log messages in the db.

The below is the code for telling where to log -> write to console, write to SQL. We can do this via JSOn as well. Here via code.

LoggerConfiguration loggerConfiguration = new LoggerConfiguration()

.ReadFrom.Configuration(configuration, "Serilog")

.WriteTo.Console()

.WriteTo.MSSqlServer(configuration.GetValue<string>("ConnectionStrings:DefaultConnection"), "Logs", autoCreateSqlTable: true);

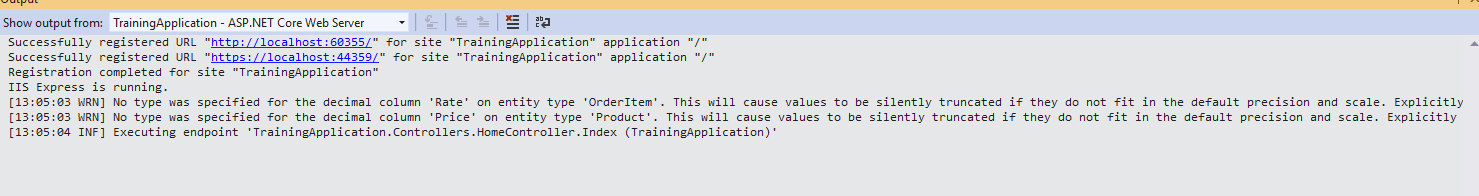
We are doing a options.Sources.Clear();

So the scope of configuration builder then this logger is very local to program.cs.. before CreateHostBuilder is called.

.UseSerilog() register inside the CreateHostBuilder

Now we can use Ilogger in the Configure method of Startup.

Check output window – different kind of logging now.



Check whether the messages have gone into DB.

Logs table should be created. And a bunch of entries should be added as well.

Logs table structure is created by the Serilog itself.

Properties Column in the Logs table is like a trace.

<properties><property key='EndpointName'>TrainingApplication.Controllers.HomeController.Index (TrainingApplication)</property><property key='EventId'><structure type=''><property key='Id'>1</property><property key='Name'>ExecutedEndpoint</property></structure></property><property key='SourceContext'>Microsoft.AspNetCore.Routing.EndpointMiddleware</property><property key='RequestId'>8000003a-0004-ff00-b63f-84710c7967bb</property><property key='RequestPath'>/</property><property key='SpanId'>|92fadbc4-4c2139056ccd0ac3.</property><property key='TraceId'>92fadbc4-4c2139056ccd0ac3</property><property key='ParentId'></property></properties>