Logging is done by constructor injection – controller, repository … and then use extension method to log the different types of messages(warning, fatal, info, etc.)

ILogger<homecontroller> logger

It helps logger to identify the categore based on the class in the generic parameter – here homecontroller.

If we want a common controller <MVC Controller> - we can have a base class – empty with no real functionality. Messages generated by all methods in the MVC app will come under this generic parameter.

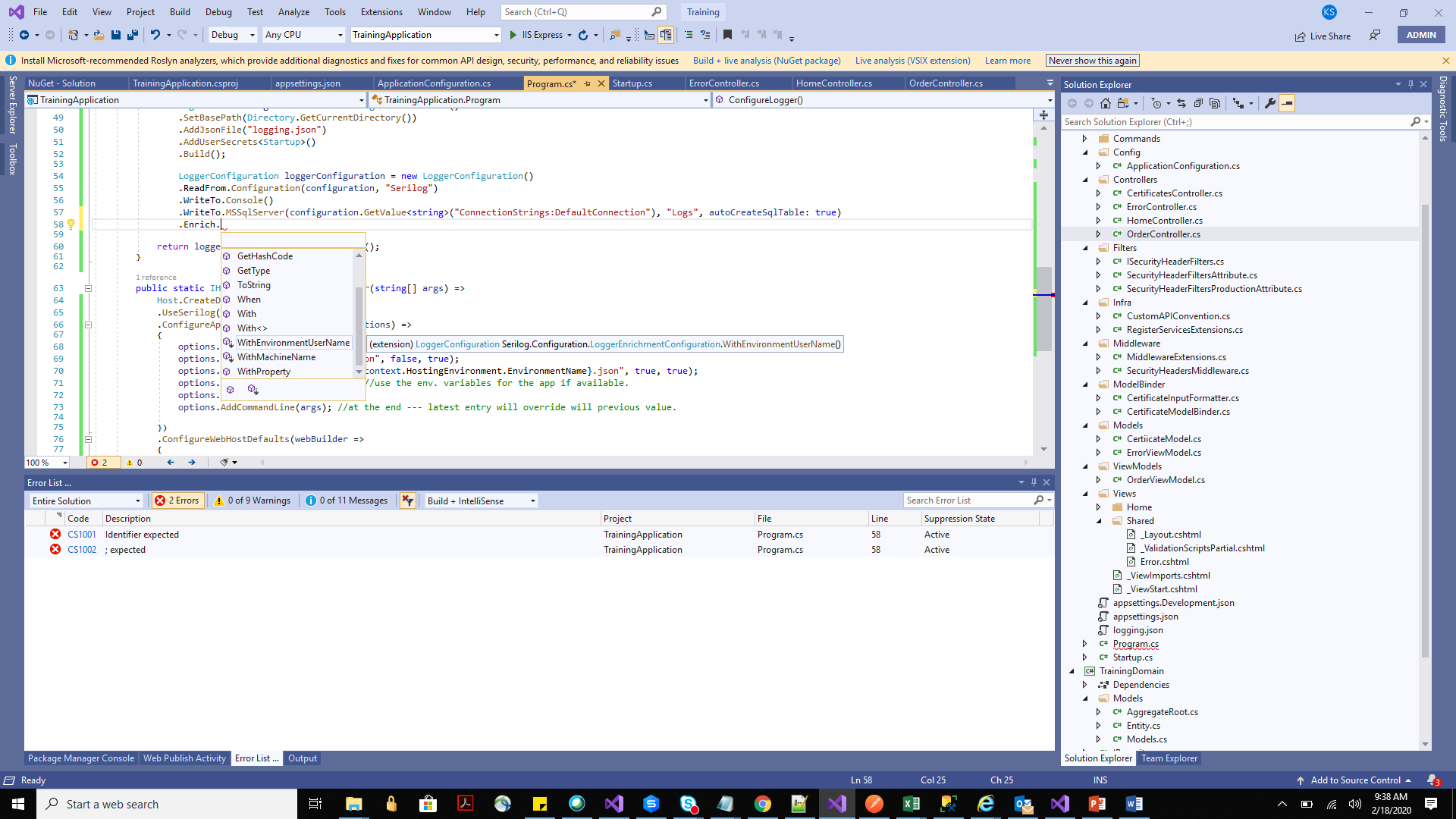
It makes it easier for us to go through the logs. Specially in environments other than the development env. In env. Where visual studio is not available.

Serilog can point to the server on which the code is running, user who is logged in, .. these can be automatically logged, name of the machine, ipaddress,

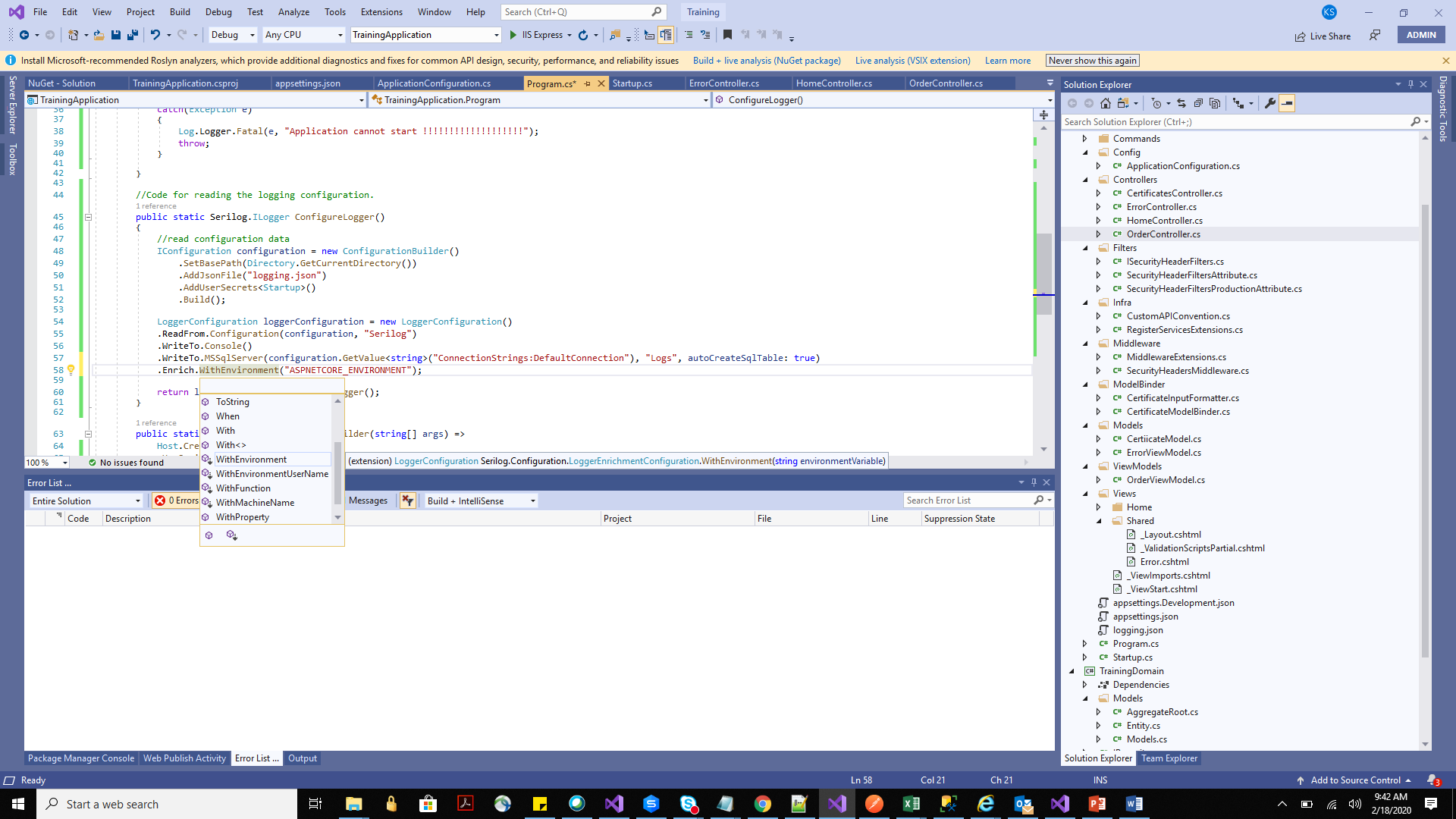
This is where enrichers come into picture.. we needn’t supply all of that information.

Serilog.Enrichers.Environment install and go to the method where you are configuring the serilog.

See with properties.



Serilog.Enrichers.Context – install



You can do this using the configuration file as well .. not just with the code.

<properties><property key='EndpointName'>TrainingApplication.Controllers.HomeController.Index (TrainingApplication) – fully qualified name of the class from twhere the log has happened – hierarchical string – training application – root </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'>8000001a-0003-fd00-b63f-84710c7967bb</property><property key='RequestPath'>/</property><property key='SpanId'>|b5ccd05b-41666700f662c273.</property><property key='TraceId'>b5ccd05b-41666700f662c273</property><property key='ParentId'></property><property key='ASPNETCORE\_ENVIRONMENT'>Prod</property></properties>

Now an additional property in the properties field.

Log config says the minimum level of messages that should be logged.

Override section overrides information at the minimum level. For category Microsoft – log messages with minimum level warning. System – info and above

We can define the categories like for root category – TrainingApplication – we can decide the default and above in the config. TrainingApplication.Controllers ---

There are some system categories like

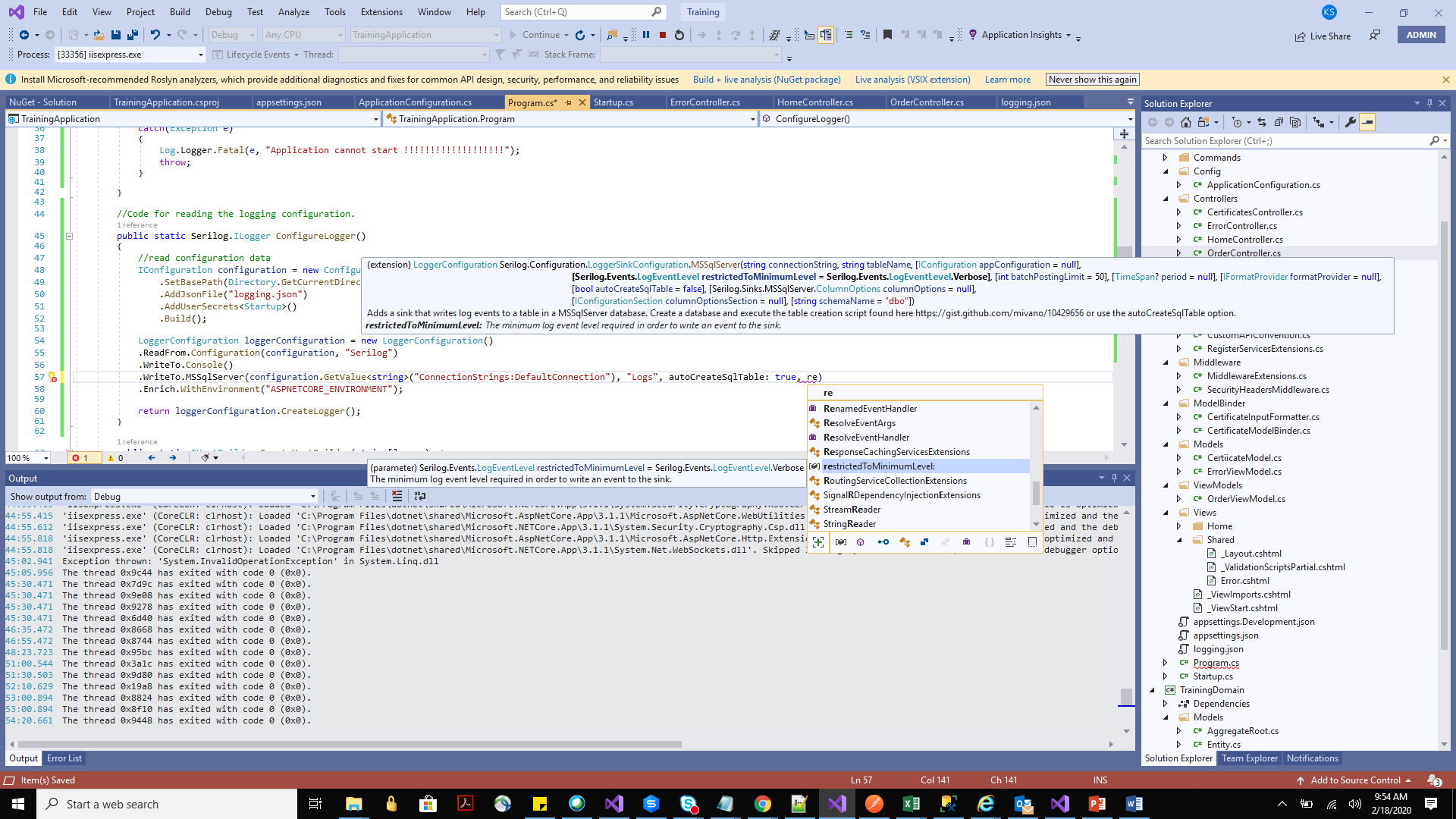
"Microsoft": "Warning",

       "Microsoft.AspNetCore.Routing": "Information",

       "Microsoft.AspNetCore.Mvc": "Information",

       "System": "Information"

We can define the levels at the sync level – for DB logging we can have information (in one db), warning anf above in another. Or you can’t log info at all.



Restricting logs capabilities – related to semantic logging, or filtering loggings – is a framework level feature itself (not as good as enricher)

We can use different logging.JSON for different environment. So that we have cotrol over what gets logged in different environments.

STATE MANAGEMENT:

* Cookies – HTTP Cookies
* Session – HTTP Cookies and app code
* TempData – HTTP cookies or session state
* HttpContext.Items - App code
* Cache – App code

Cookie are supposed to store very little information. There is a valid 4K bytes limits.

If you use .net core cookie then you can use a larger cookie. If the size of the cookies change then they make chunks of it each of 4k bytes and what info is stored in which cookie – there would be another cookie.

Cookies – 4k or 6k or 8k but then don’t make it 4mb. Technically possible but DON’T.

**Cache**.

Caching is the most feature rich in the above list. In core caching, it tries to respect the HTTP caching.

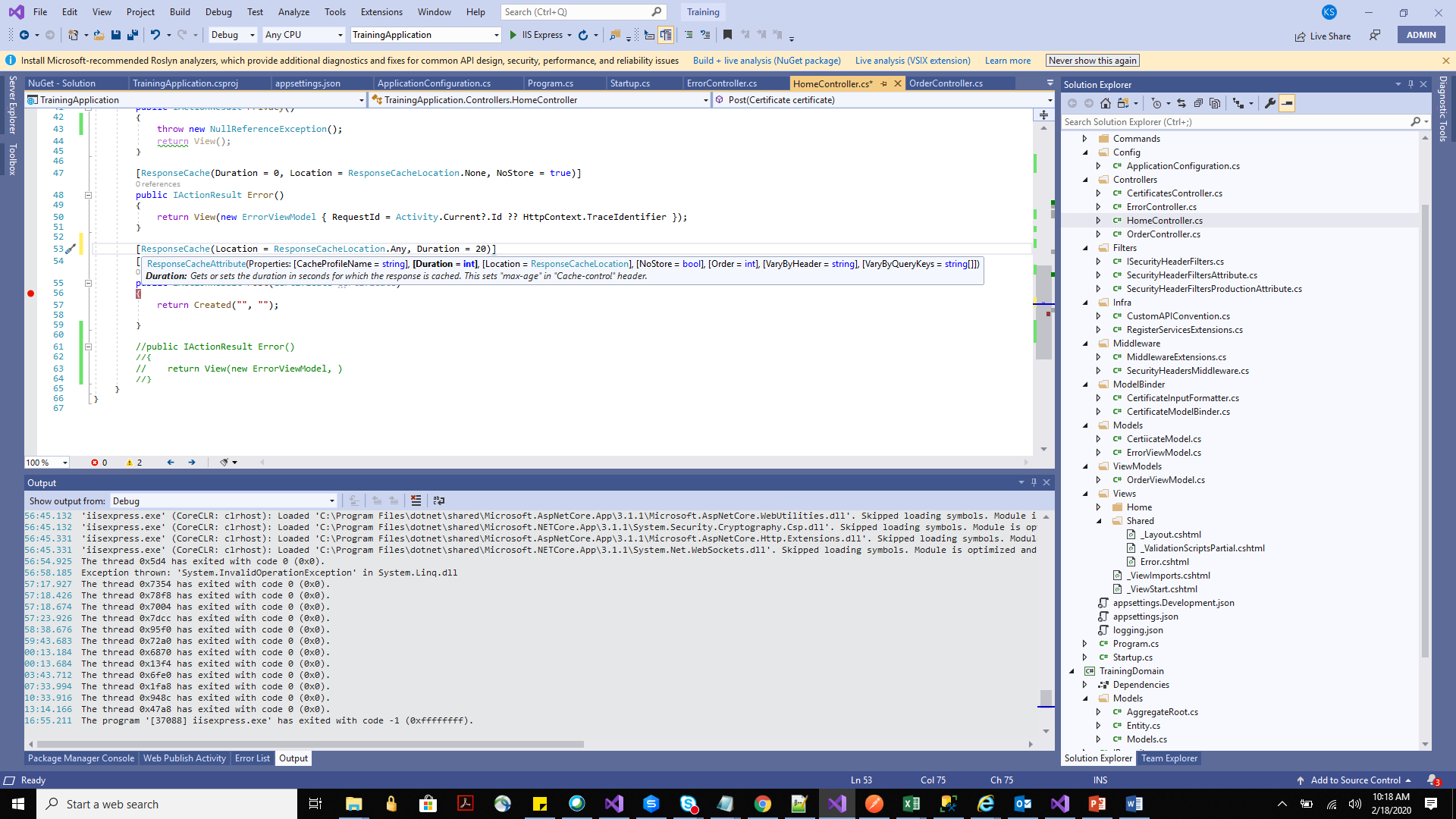
Public cache – load balancer, API g/w, Proxy

Private cache – browser.

No-store and No-cache

Server -> load bal -> API g/w -> Proxy -> forward -> browser -> request to server.

In Asp.net there is a ResponseCache Attribute. Add it on home and order.



If breakpoint is hit in 20 seconds then caching is not working.

You can check response headers on the browswer – it says max age 20.

[ResponseCache(Location = ResponseCacheLocation.Any, Duration = 20)]

       public IActionResult Privacy()

       {

           //throw new NullReferenceException();

           return View();

       }

Also if you hit refresh button then as per the client (browser that you are using it may or may not hit the breakpoint.

Cache control header - public,max-age=20 in the postman as well.

In postman everyttime we click on send then the breakpoint gets hit… postman just shows the headers but doesn’t really implement the caching as a browser would.

Browsers respect these headers.

Vary by header –

GET api/order/1

GET api/order/2

Cache is not used.

[ResponseCache(Location = ResponseCacheLocation.Any, Duration = 20, VaryByHeader = "Accept", VaryByQueryKeys = new[]{"\*"})]

VaryByQueryKeys = requires response cache middleware other wise error on postman. Also register the service.

Data that requires authorization should never be cached. Cannot.

If authorize attribute added to the action then response cache control header to no cache.

{

    "type": "System.InvalidOperationException",

    "title": "'VaryByQueryKeys' requires the response cache middleware.",

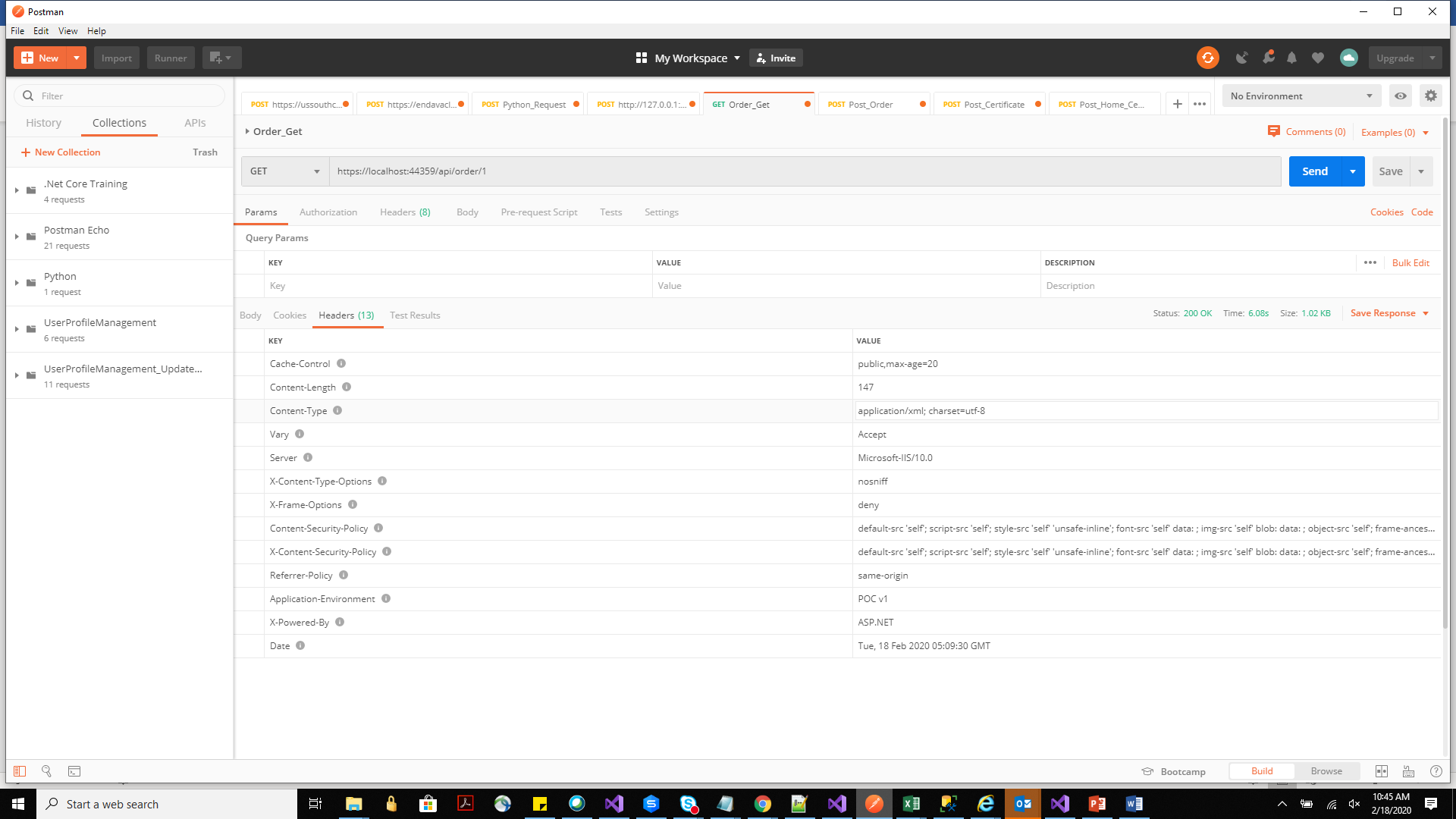
    "status": 500,

    "detail": "   at Microsoft.AspNetCore.Mvc.Filters.ResponseCacheFilterExecutor.Execute(FilterContext context)\r\n   at Microsoft.AspNetCore.Mvc.Filters.ResponseCacheFilter.OnActionExecuting(ActionExecutingContext 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.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\_\_Logged|17\_1(ResourceInvoker invoker)\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": "|f3a99266-4c309139dbd7bba1."

}

See vary header in the response this was added on the attribute for caching.



Cache-control header should be provided on the request – postman is making a request. But breakpoint is not hit. Client side control of cache.

Client side or server side caching….. not both as debugging might become an issue. Not a clean design.

In-memory caching – server side caching .. allows authorization – where client side caching is not applicable.

Asp.net allows to explicitly manage the cache data on the server.- using Imemory Cache. Cache policy, size control… if we want to implement caching in a predictable way. Like postman doesn’t adhere to and response the cache control.

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

whereever you want to do tha caching, in trhe controller inject Imemery Cache

Already registered in the DI container by default.

var order = \_cache.Get<Order>($"Order-{id}");

            if (order == null)

            {

                order = \_unitOfWork.GetRepository<Order>().Get(o => o.ID == id).SingleOrDefault();

                if (order == null)

                    return NotFound();

                \_cache.Set(id, order);

            }

            return Ok(order);

//\_cache.Set(id, order);

\_cache.Set($"Order-{id}", order);

Nw it doesn’t hit the DB.

Order order;

          order = \_cache.GetOrCreate<Order>($"Order-{id}", cacheEntry =>

          {

              order = \_unitOfWork.GetRepository<Order>().Get(o => o.Id == id).SingleOrDefault();

              if (order == null)

                  throw new ArgumentException("Provided order id is invalid");

              cacheEntry.AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(1);

              cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(10);

              cacheEntry.Priority = CacheItemPriority.Normal;

              return order;

          });

Does the same thing as earlier. But we are adding some more properties like expiry time. We don’t want the data to be cached for indefinitely.

Using the prefix for defining the cache key - ($"Order-{id}", because tomorrow we can add product with the same id.

This service doesn’t llow us to define the size limit.

If the size is huge then there would be an issue with the application because not enough size for the application to run.

Register the OrderCache

\_ordersCache.Cache instead of \_cache.

cacheEntry.Size = 1; can now be defined. Till the size limit is met the cache would be added. Once it reaches the max limit (as defined in the orders cache), it deletes the older data in the cache.

**Distributed Cache:**

Shared by multiple add server

Advantages:

* Consistent across request to multiple servers
* Survives server restart
* Doesn’t use local memory. (in memory uses the same memory that is given to our web application) physically distributed to other servers. Physically separated from the server space assigned to the server space.

OOTB implementation of IDistributedCache. In your constructor use this instead of Imememy cache. IDis allows you to add the add or remove data from the cache only in the form of byte array.

Imemery – allows any thing – like object.

Technically speaking it still stores in the in-memory.

Google more about it.

Usage scenario:

* Development env. This is like a dummy service and to only simulate the paid disturbed cache for production. Cloud based are cheap and most companies don’t mind it.
* Single server deployment

Workng with the background Task in the Asp.net core.

Background tasks can be implementes as hosted services

* Class with background task logic that implements IHostedService
* Activated once at app startup and shut down at app shutdown.

IHostedService.StartAsync

* Contains logic to start the background task (make sure that you create a new thread and on that thread.. run the business logic)
* By default call before the app request processing pipeline is configured.

IHostedService.StopAsync

* Contains Logic to end the background task
* Called when the host is performing a graceful shutdown
* Cancellation token has 5 seconds (by default) timeout. It will wait for 5 seconds for the Stop Async to perform any work.

**Hangfire for performing background task. automatic retries, muli server. Open source component.**

Create a class and it should inherit from the IHsotedService. or instead

There is a class that asp.net core provides – background service. It gives basic infrastructure for threading related logic. Recommended for newbies on Threading.

TrainingWorkerService – this is a project of worker service – scaled down templated of Asp.net. core project. No start up but program.cs and appsettings.json .. in addition it has a worker.cs. services.AddHostedService<Worker>(); this is the service that is registered.

There is not middleware pipeline that is getting populated.

In the worker.cs -> its inheriting from backgroundservice class which inherits from IHostedService.

ExecuteAsync is exposed by BGService. .. this is where we write the business logic.

Below logic -> do something and delay by 1000 ms and then do something…. This loops runs until the cancellation token is invoked. Cancellation tokens are objects that invoke the cancellation.

while (!stoppingToken.IsCancellationRequested)

           {

               \_logger.LogInformation("Worker running at: {time}", DateTimeOffset.Now);

               await Task.Delay(1000, stoppingToken);

           }

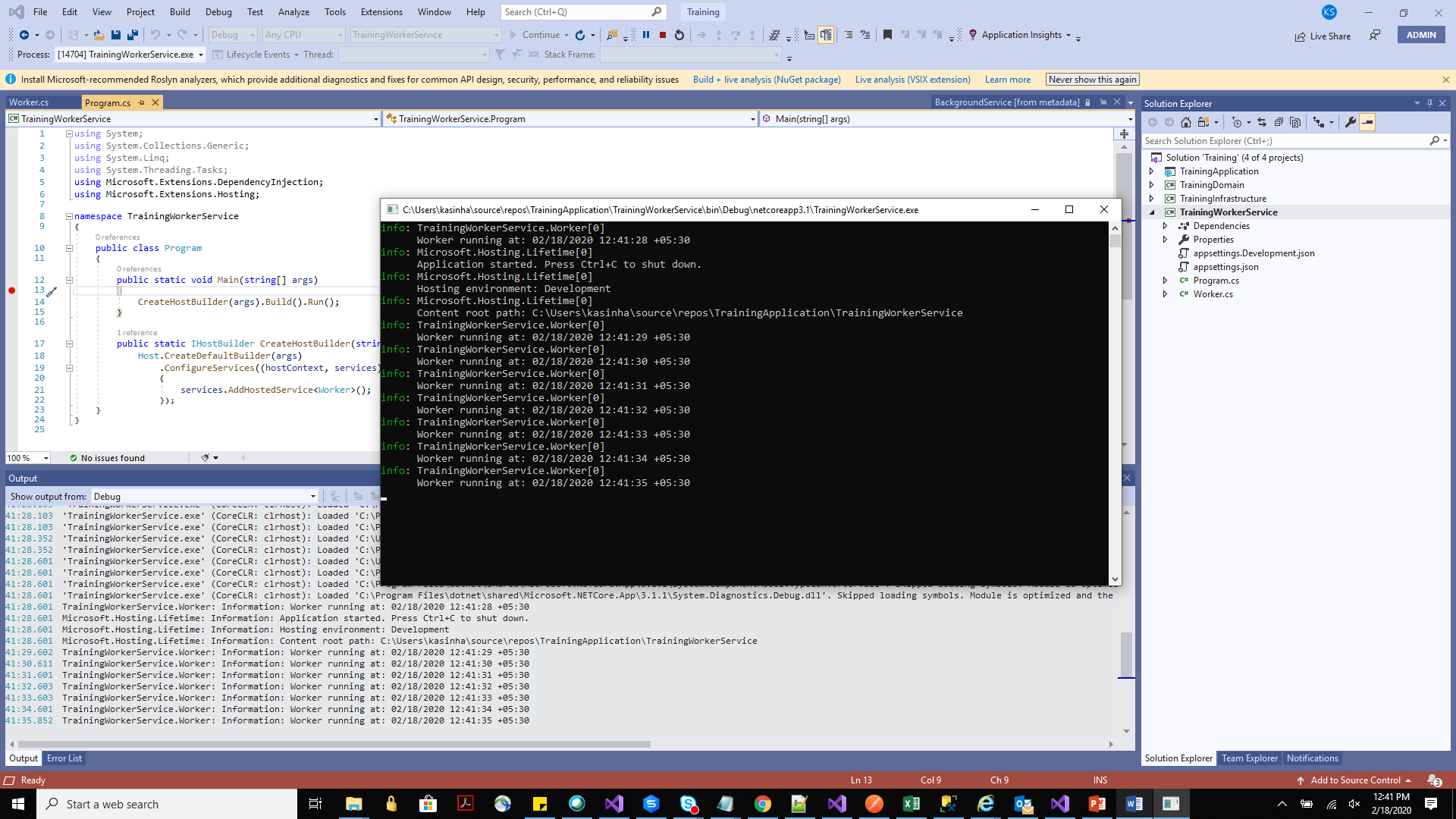
Lets make it do something useful.

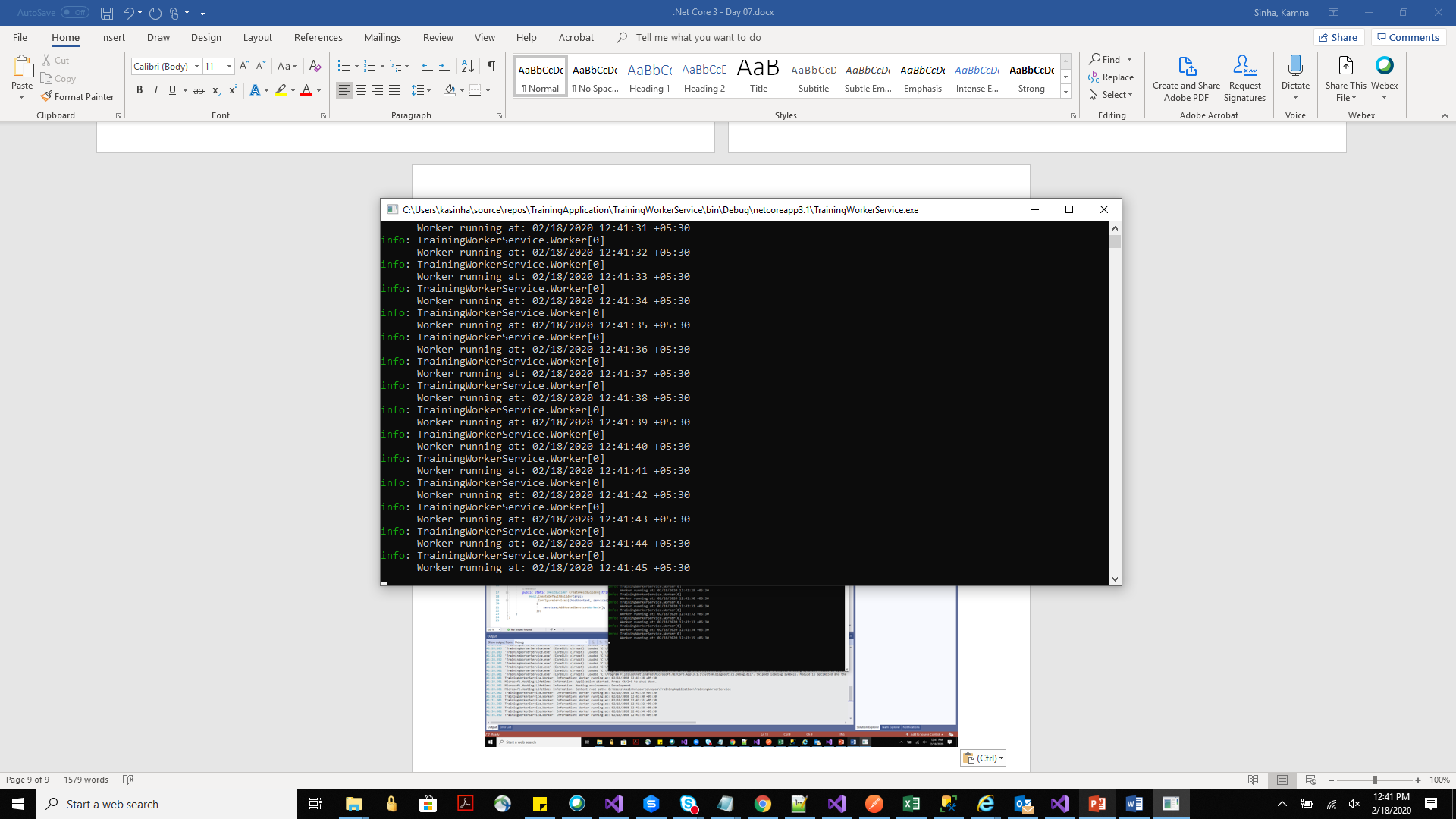
Make the project as startup project. (there is a service for healthcheck in ASP.net core --- use that not this—like pinging an IPAdress).

Suppose at midnight we want to run something to check for new orders and then send welcome emails.

Perform some back up.

Perform some clean up (some archiving for data older than an year suppose).





OrdersNotificationService added to the proj.

It had backgrounfd processing logic so inherit from that class.

Reference to infrastructure and domain – because DB we need to connect to.

We need UoW and we need the logging .. so we inject them

**ISSUE 1**

await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);

is the graceful stop point.

Before we pick up the next order in the foreach loop .. because eimagine if that foreach is going to take 2 hours but if our application sends a cancellation token in 30 min then it wouldn’t be a graceful exit.

So we add the below line- this checks if the next unit of work should be started or now depending on if we have received the cancellation token.

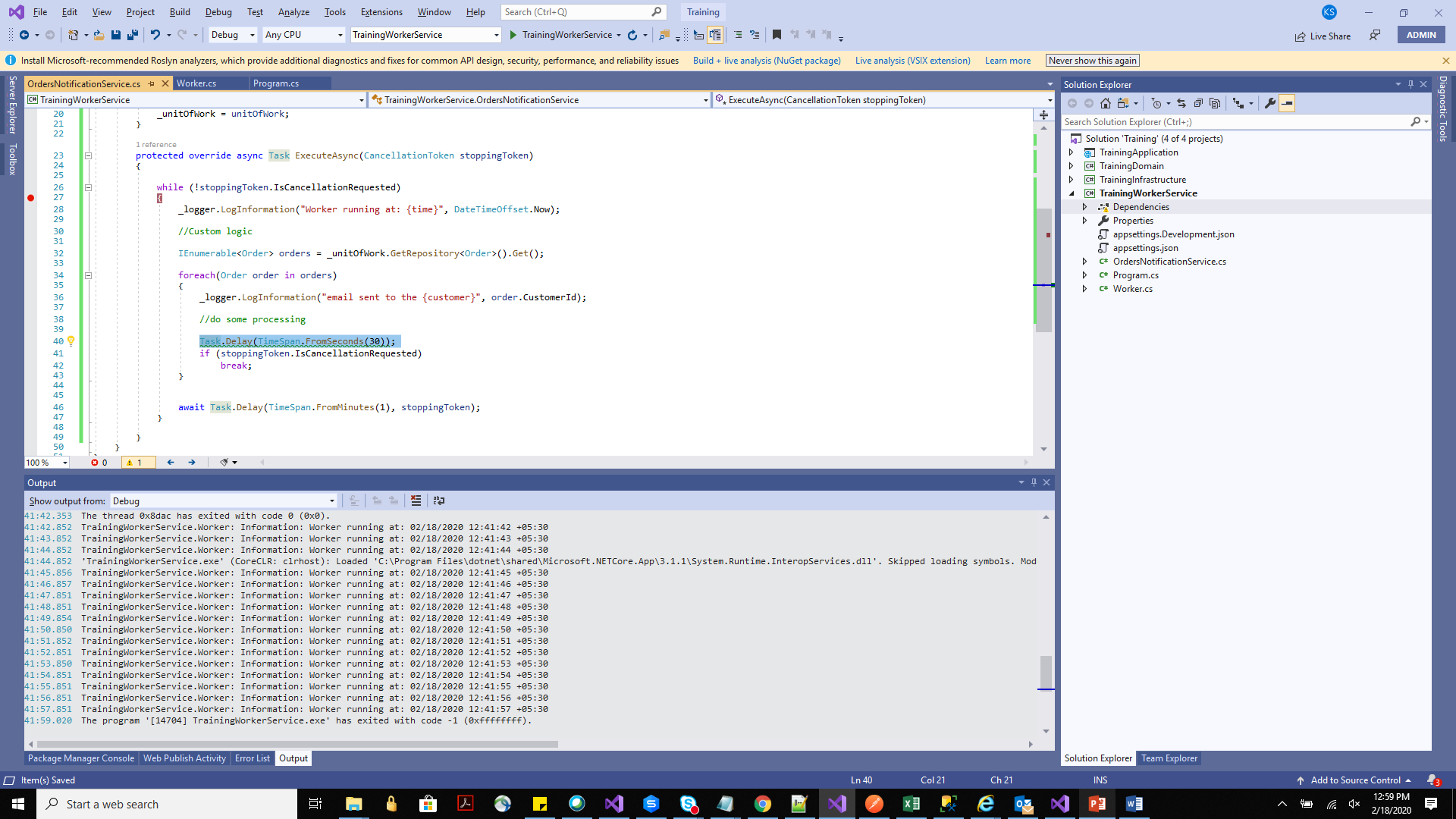
if (stoppingToken.IsCancellationRequested)

                       break;

**ISSUE 2**

No further services are started until ExecuteAsync becomes asynchronous.

Lets simulate this issue (above line)



Until our code doesn’t reach the await, it doesn’t become asynchronous. It will block any other work to happen.

**Issue 3**

UnitofWork is dependent on DB context. Everything is scoped. But our services have singleton lifetime. So captive dependency.

The dependency injection will not get resolved.

First time run - code will work. Next day when the code gets triggered after await, the DBcontext won’t be available and new connection won’t be provided because it would be expecting the singleton dbcontext.

The DB connection will get forced stopped after its scope is over.