

# Observability 修煉之路

---

## 前言

---

本文為Observability的概念，並且說明如何在.Net使用OpenTelemetry來實作Observability。主要會涵蓋以下項目

## Topic List

以下為預計分享的topic 列表

- Observability 概述
  - Tracing
  - Log
  - Metrics
  - ELK
    - 現有架構
    - 結合OTel
- OpenTelemetry
  - Overview
  - Span
- Docker
  - Overview
  - Docker-compose
- ELK
  - Overview
  - Install
- OpenTelemetry on .Net
  - Auto Instrumentation
  - Manual Instrumentation
- Data visualization
  - Kibana

## 可觀察性(Observability)

可觀察性是指一個系統可透過監視和分析系統外部狀態的變化藉以追蹤內部出錯的原因及診斷這些錯誤發生的能力。白話一點的說法就是能夠藉由系統外的資訊觀察系統內部的運作狀態，將有被觀察意義的資訊給揭露出來。

以看醫生來舉例就是觀察血壓、體溫、心跳、抽血觀察等等，來判斷身體內部的狀況。

### Observability三本柱

- Log
  - 泛指系統或應用程式運行所產生用來紀錄系統或程式資訊的文字檔，像是system log、application log等。

- Metrics
  - 為系統或程式在運行時的量化數據，通常為一數值化的結果，例如CPU使用率、記憶體使用量、API被呼叫的次數等等
- Trace
  - 紀錄一個任意請求(Request)在不同應用中處裡的路徑，例如一個請求從前端進入後端，後端再呼叫其他服務，最後回傳給前端，這個過程中的每個步驟都會被記錄下來，並且可以透過`trace_id`來串聯這些步驟，作為後續的追蹤使用。通常為侵入性的作法(埋code)，而且可能會影響應用程式的作業。也有非侵入式的作法，但通常就無法觀察較細微的細節。在現今微服務的環境當中tracing的重要性大大的提升，常被用來定位問題發生的service位置以及追蹤一個request在不同service中的路徑。

## Telemetry (by chatGPT)

指的是遠端數據收集、傳輸和監控系統的過程。通常用於描述從遠端收集數據並將其傳送到集中地點進行分析、監控或記錄的過程。**Telemetry** 可以應用於各種領域，包括航空航天、醫療、工業、環境監測、運輸等等。

**Telemetry** 系統通常包括傳感器、數據收集設備、通信設備和數據處理設備。傳感器用於測量不同的參數，例如溫度、壓力、速度、位置等，然後這些數據通過通信設備傳送到中央站點。在中央站點，數據可以進行分析，以幫助監控遠程系統的狀態、性能和健康情況。

航空航天領域經常使用遙測技術，以監測和收集太空探測器、衛星和飛機的性能數據。醫療領域可以使用遙測來監控患者的生理參數，工業和環境應用中可以使用遙測來監控設備的運作和環境條件，而運輸領域則可以使用遙測來追蹤運輸工具的位置和狀態。

總之，**Telemetry** 是一個重要的技術，用於實時監測和收集遠程位置的數據，有助於提高效率、安全性和性能。

## OpenTelemetry

由字面上**Open**以及**Telemetry**可得知**OpenTelemetry** (OTel) 是一種跨平臺、開放式標準，解決系統監測時Telemetry的資料模型定義、檢測、採集、處理、輸出等標準化問題, 並提供一組標準化SDK、API與一些常用工具。

要產生Telemetry主要有兩種方式，一個為**Auto Instrumentation**，另一個為**Manual Instrumentation**。

**Auto Instrumentation**為透過**自動**的方式，將Telemetry資料從系統中擷取出來，並且將其送到OTel的後端，這種方式不需要修改原始碼，但是會有一些限制，例如無法自訂資料的內容等。

**Manual Instrumentation**為透過**手動**的方式，將Telemetry資料從系統中擷取出來，並且將其送到OTel的後端，這種方式需要修改原始碼，但是可以自訂資料的內容等。

## Automatic v.s Manual Instrumentation

- Automatic 會tracing RESTful API 那層，功能內部元件的呼叫並無法tracing到。

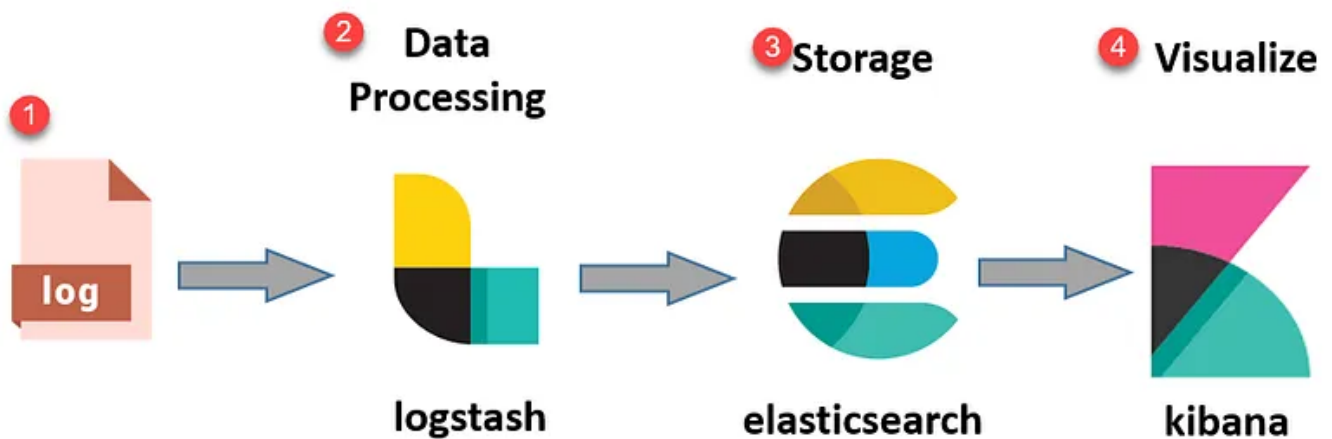
- Manual 則是自訂細部的tracing以及metrics量測，缺點就是要改code,

## EL(F)K 簡介

EL(F)K為一套由Elastic所提供的開源軟體組合，包含Elasticsearch, Logstash(FluentD), Kibana三個軟體

- Elasticsearch為一個分散式搜尋引擎，也是個TSDB(Time Series Database)，來儲存以及搜尋資料。
- Logstash(FluentD)為一個資料收集引擎，建立data process pipeline,
- Kibana為一個資料視覺化工具。

### ELK流程圖



### 安裝ELK

透過下方docker-compose內容，並開啟CMD後輸入以下指令

```
docker-compose pull #下載image  
docker-compose up #啟動container
```

docker-compose.yml內容如下

```
version: "3.8" # docker-compose 使用的版本  
services:  
  elasticsearch:  
    image: elasticsearch:8.10.2 # 指定使用的docker image  
    environment: # 環境變數  
      - discovery.type=single-node # 使用single-node模式  
      - network.host=0.0.0.0 # 開放所有網路  
      - http.host=0.0.0.0  
      - xpack.security.enabled=true # 需打開才能連線  
      - xpack.security.authc.api_key.enabled=true  
      - ELASTIC_PASSWORD=changeme # 預設密碼
```

```

ports:
  - 9200:9200 # elastic search 使用的port
  - 9300:9300
healthcheck:
  test: nc -z localhost 9200 || exit 1 # 檢查是否有開啟port
  interval: 5s
  timeout: 10s
  retries: 100
kibana:
  image: kibana:8.10.2
  ports:
    - 5601:5601
  environment:
    - ELASTICSEARCH_USERNAME="kibana_system"
    - ELASTICSEARCH_PASSWORD="kibana_system"
  healthcheck:
    test: ["CMD-SHELL", "curl -u kibana_system:kibana_system -s
http://localhost:5601/api/status"]
    interval: 5s
    timeout: 10s
    retries: 120
  depends_on:
    elasticsearch:
      condition: service_healthy
fleet-server:
  image: elastic/elastic-agent:8.10.2
  container_name: fleet-server
  user: root
  ports:
    - 8220:8220
  environment:
    - FLEET_SERVER_ENABLE=1
    - FLEET_SERVER_ELASTICSEARCH_HOST=http://elasticsearch:9200
    -
FLEET_SERVER_SERVICE_TOKEN=AAEAaWVsYXN0aWMvZmxlZXQtc2VydmcVvL3Rva2VuLTE2OTc0MjQ0NjE
2NTQ6cGZDLVIydnhUbW01WXN2VHlzT3I15QQ
    - FLEET_SERVER_POLICY_ID=fleet-server-policy
    - FLEET_SERVER_ELASTICSEARCH_USERNAME=elastic
    - FLEET_SERVER_ELASTICSEARCH_PASSWORD=elastic
    - p 8220:8220
  healthcheck:
    test: ["CMD-SHELL", "curl -u elastic:elastic -s
http://localhost:5601/api/status"]
    depends_on:
      kibana:
        condition: service_healthy
agent01:
  image: elastic/elastic-agent:8.10.2
  container_name: agent01
  user: root
  environment:
    -
FLEET_ENROLLMENT_TOKEN=aFFiZE40c0IyYTk30TFNM0prS3M6UDVmcFVwLVRRbzJEZWprVnF0c3JKQQ=
=

```

```

- FLEET_ENROLL=1
- FLEET_URL=https://fleet-server:8220
- FLEET_INSECURE=true
- p 8200:8200
ports:
- 8200:8200
depends_on:
  fleet-server:
    condition: service_healthy

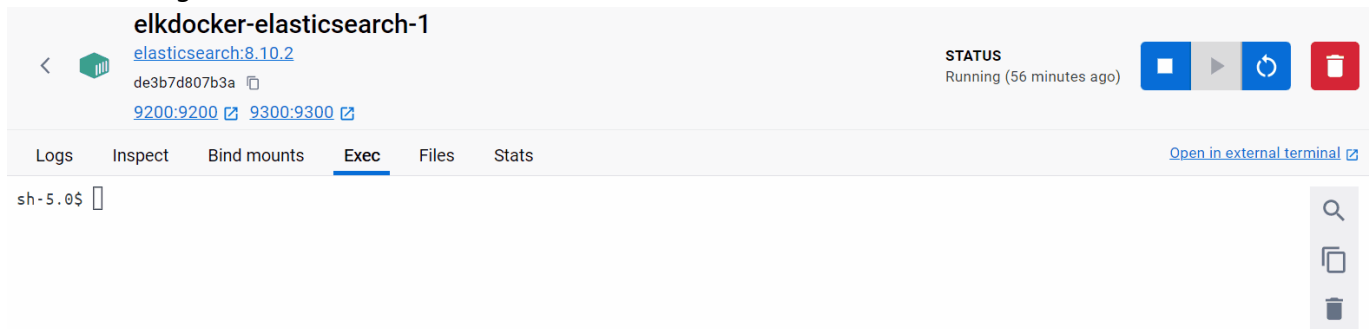
```

下載後container並未全部啟動，須完成下方流程。

1. 進入ElasticSearch設定kibana的帳號密碼，首先進入elasticsearch container輸入以下指令

```
./bin/elasticsearch-setup-passwords interactive
```

可參考下方的gif圖，設定的密碼之後會用到，因此需要先記下來。



設定完kibana\_system的密碼後，需要把密碼設定回docker-compose.yml/kibana.ELASTICSEARCH\_PASSWORD的欄位

之後重跑一次docker-compose up


啟動後可以看到docker container如下，並且可以透過localhost:5601進入Kibana



## Welcome to Elastic

**Username**

**Password**



**Log in**

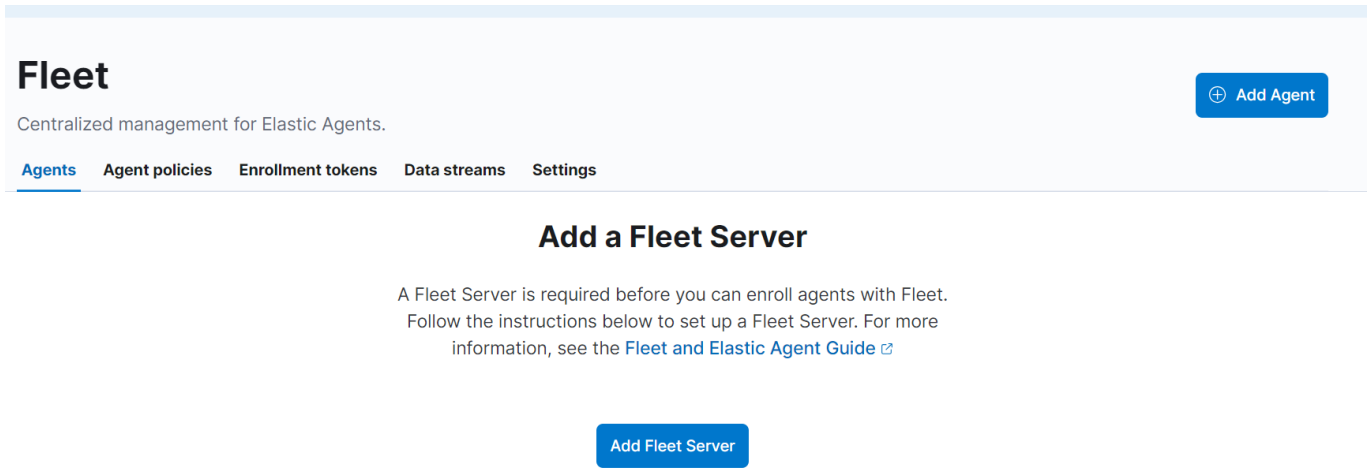
一進到頁面後需要輸入`elastic`的帳號才能進入，密碼為剛剛設定的密碼。

若遇到無法登入時需要進入到ES的container內修改密碼，指令如下

```
./elasticsearch-reset-password -u elastic
```

此指令代表要重設`elastic`這user的密碼，重設完後會得到新的密碼，輸入即可登入。

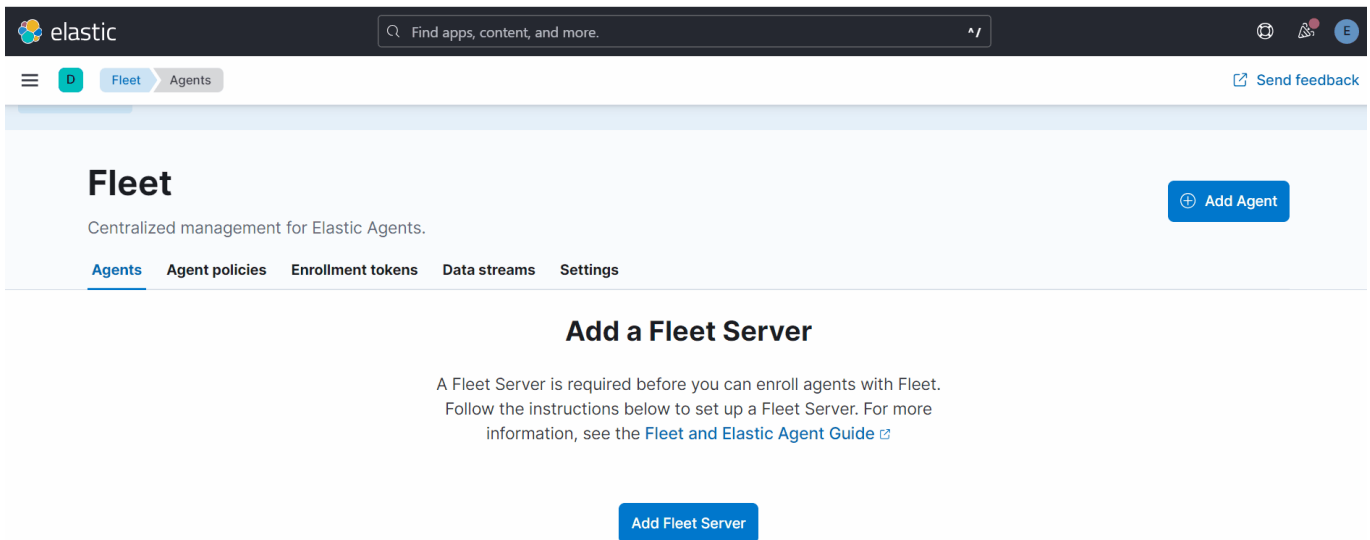
登入之後需要先設定`fleet-server`的policy，點選左方的menu選擇Management選區的`Fleet`



接著請照以下流程設定

1. 選擇 **Add Fleet Server**
2. 選擇 **Advance**
3. 然後點選 **Create Polity**
4. 當出現 **Agent Policy Created** 出現時，代表有成功。
5. 產生 **service-token**
6. 將 **token** 貼回 **docker-compose.yml/fleet-server.FLEET\_SERVER\_SERVICE\_TOKEN** 的欄位
7. 重啟 **docker-compose up**
8. 回到網頁確認是否有連上 **fleet-server**

流程可參考以下示範



接著需要設定agent的政策,

# Fleet

Centralized management for Elastic Agents.

[Agents](#) [Agent policies](#) [Enrollment tokens](#) [Data streams](#) [Settings](#)

🕒 Agent activity

Add Fleet Server

Add agent

🔍 Filter your data using KQL syntax

Status 4

Tags 0

Agent policy 2

Upgrade available

Showing 1 agent

[Clear filters](#)

● Healthy 1 ● Unhealthy 0 ● Updating 0 ● Offline 0

<input type="checkbox"/>	Status	Host	Agent policy	CPU ⓘ	Memory ⓘ	Last activity	Version	Actions
<input type="checkbox"/>	Healthy	c4b6b808c5b7	Fleet Server policy 1 rev. 2	N/A ⓘ	N/A ⓘ	39 seconds ago	8.10.2	...

Rows per page: 20

< 1 >

- 點選 **Add Agent**
- Create Policy**
- 在下方的Enroll找到**FLEET\_ENROLLMENT\_TOKEN**並貼回docker-compose裡面  
**agent01.FLEET\_ENROLLMENT\_TOKEN**的欄位
- 重啟 **docker-compose up**
- 回到網頁確認是否有連上agent

可參考以下流程

The screenshot shows the Elastic Fleet management interface. At the top, there's a search bar and navigation tabs for Fleet and Agents. The Fleet tab is active. Below the navigation, there's a summary of the fleet: 1 Healthy agent, 0 Unhealthy, 0 Updating, and 0 Offline. A table lists the agents, showing one agent with ID c4b6b808c5b7, status Healthy, and last activity 47 seconds ago. The table has columns for Status, Host, Agent policy, CPU, Memory, Last activity, Version, and Actions.

接著要設定APM server，點選左方的menu選擇Management選區的**APM**

- 選擇 **Add Data**
- 選擇 **Manage APM integration in Fleet**
- Add Elastic APM**
- 需要將Host欄位的**localhost**改為 **0.0.0.0**
- Save and Continue**
- 成功後選擇 **Add Elastic Agent**
- 一樣複製**ENROLLMENT\_TOKEN**並貼回docker-compose裡面**agent01.FLEET\_ENROLLMENT\_TOKEN**的欄位
- 重啟 **docker-compose**
- 回到網頁確認是否有連上agent
- 接著點進Agent裡面頁面旁邊的Setting，需要把elasticsearch的網址從**localhost**改為**elasticsearch**



到此即設定完成，接著進management申請完API Key後，完成 [安裝OpenTelemetry 於.net](#)的步驟，即能看到APM資料，可參考以下流程。

## OpenTelemetry on .Net

### Auto Instrumentation

以下安裝流程為參考Otel官方的[教學文件](#)

#### 1. 先建置測試專案

```
dotnet new web
```

#### 2. 專案Program.cs內容

```
using System.Globalization;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

var logger = app.Logger;

int RollDice()
{
    return Random.Shared.Next(1, 7);
}

string HandleRollDice(string? player)
{
    var result = RollDice();

    if (string.IsNullOrEmpty(player))
    {
        logger.LogInformation("Anonymous player is rolling the dice: {result}",
result);
    }
    else
    {
        logger.LogInformation("{player} is rolling the dice: {result}", player,
result);
    }

    return result.ToString(CultureInfo.InvariantCulture);
}

app.MapGet("/rolldice/{player?}", HandleRollDice);

app.Run();
```

### 3. 修改 properties/launchSetting.json

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:8080",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

### 4. 使用auto-instrumentation的方式置入instrumentation

開啟power-shell (需要管理員權限) · 並執行以下指令

```
$module_url = "https://github.com/open-telemetry/opentelemetry-dotnet-instrumentation/releases/latest/download/OpenTelemetry.DotNet.Auto.psm1"
$download_path = Join-Path $env:temp "OpenTelemetry.DotNet.Auto.psm1"
Invoke-WebRequest -Uri $module_url -OutFile $download_path -UseBasicParsing
Import-Module $download_path
Install-OpenTelemetryCore
$env:OTEL_TRACES_EXPORTER="none"
$env:OTEL_METRICS_EXPORTER="none"
$env:OTEL_LOGS_EXPORTER="none"
$env:OTEL_DOTNET_AUTO_TRACES_CONSOLE_EXPORTER_ENABLED="true"
$env:OTEL_DOTNET_AUTO_METRICS_CONSOLE_EXPORTER_ENABLED="true"
$env:OTEL_DOTNET_AUTO_LOGS_CONSOLE_EXPORTER_ENABLED="true"
Register-OpenTelemetryForCurrentSession -OTelServiceName "RollDiceService"
```

最後執行專案

```
dotnet run
```

執行完後可以看到console內有openTelemetry的log

```

Export process.runtime.dotnet.gc.duration, The total amount of time paused in GC since the process start., Unit: ns, Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2037674Z, 2023-10-02T03:40:45.2515569Z] LongSum
Value: 0

Export process.runtime.dotnet.jit.il.compiled.size, Count of bytes of intermediate language that have been compiled since the process start., Unit: bytes, Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2037833Z, 2023-10-02T03:40:45.2515573Z] LongSum
Value: 840102

Export process.runtime.dotnet.jit.methods.compiled.count, The number of times the JIT compiler compiled a method since the process start. The JIT compiler may be invoked multiple times for the same method to compile with different generic parameters, or because tiered compilation requested different optimization settings., Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2038076Z, 2023-10-02T03:40:45.2515577Z] LongSum
Value: 12450

Export process.runtime.dotnet.jit.compilation.time, The amount of time the JIT compiler has spent compiling methods since the process start., Unit: ns, Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2038220Z, 2023-10-02T03:40:45.2515581Z] LongSum
Value: 2529930900

Export process.runtime.dotnet.monitor.lock_contention.count, The number of times there was contention when trying to acquire a monitor lock since the process start. Monitor locks are commonly acquired by using the lock keyword in C#, or by calling Monitor.Enter() and Monitor.TryEnter()., Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2038356Z, 2023-10-02T03:40:45.2515585Z] LongSum
Value: 24

Export process.runtime.dotnet.thread_pool.threads.count, The number of thread pool threads that currently exist., Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2038507Z, 2023-10-02T03:40:45.2515588Z] LongSumNonMonotonic
Value: 5

Export process.runtime.dotnet.thread_pool.completed_items.count, The number of work items that have been processed by the thread pool since the process start., Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2038641Z, 2023-10-02T03:40:45.2515592Z] LongSum
Value: 85

Export process.runtime.dotnet.thread_pool.queue.length, The number of work items that are currently queued to be processed by the thread pool., Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2038766Z, 2023-10-02T03:40:45.2515598Z] LongSumNonMonotonic
Value: 0

Export process.runtime.dotnet.timer.count, The number of timer instances that are currently active. Timers can be created by many sources such as System.Threading.Timer, Task.Delay, or the timeout in a CancellationTokenSource. An active timer is registered to tick at some point in the future and has not yet been canceled., Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2038961Z, 2023-10-02T03:40:45.2515604Z] LongSumNonMonotonic
Value: 1

Export process.runtime.dotnet.assemblies.count, The number of .NET assemblies that are currently loaded., Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2039125Z, 2023-10-02T03:40:45.2515608Z] LongSumNonMonotonic
Value: 124

Export process.runtime.dotnet.exceptions.count, Count of exceptions that have been thrown in managed code, since the observation started. The value will be unavailable until an exception has been thrown after OpenTelemetry.Instrumentation.Runtime initialization., Meter: OpenTelemetry.Instrumentation.Runtime/1.5.1.0
2023-10-02T03:39:21.2039277Z, 2023-10-02T03:40:45.2515612Z] LongSum
Value: 7

Export process.memory.usage, The amount of physical memory allocated for this process., Unit: By, Meter: OpenTelemetry.Instrumentation.Process/0.5.0.3
2023-10-02T03:39:21.2039416Z, 2023-10-02T03:40:45.2515618Z] LongSumNonMonotonic
Value: 65191936

```

## Manual Instrumentation

在 .net framework 中，已提供 logging，metrics 以及 activity APIs 實作 OTel 的標準，代表 OTel 不用再額外提供 APIs，只要使用原生的 APIs 即可。 .Net OTel 用下列的方式實作了 OTel 的標準

- Microsoft.Extensions.Logging.ILogger (Logging)
- System.Diagnostics.Metrics.Meter (Metrics)
- System.Diagnostics.ActivitySource and System.Diagnostics.Activity (Tracing)

OpenTelemetry in .NET is implemented as a series of NuGet packages that form a couple of categories:

- Core API
- Instrumentation - these packages collect instrumentation from the runtime and common libraries.
- Exporters - these interface with APM systems such as Prometheus, Jaeger, and OTLP.

## Register Customer Signals to Auto Instrumentation

使用方式為在使用 Automatic 的情況下塞入自定義的 Instrumentation，這種方式可以在僅修改少量原始碼的情況下，將自定義的 Telemetry 塞入到 Auto Instrumentation 中。

接續先前 rollingDice 的範例，首先在專案中加入 System.Diagnostics.DiagnosticSource 的 Nuget Package

```
<PackageReference Include="System.Diagnostics.DiagnosticSource" Version="7.0.2" />
```

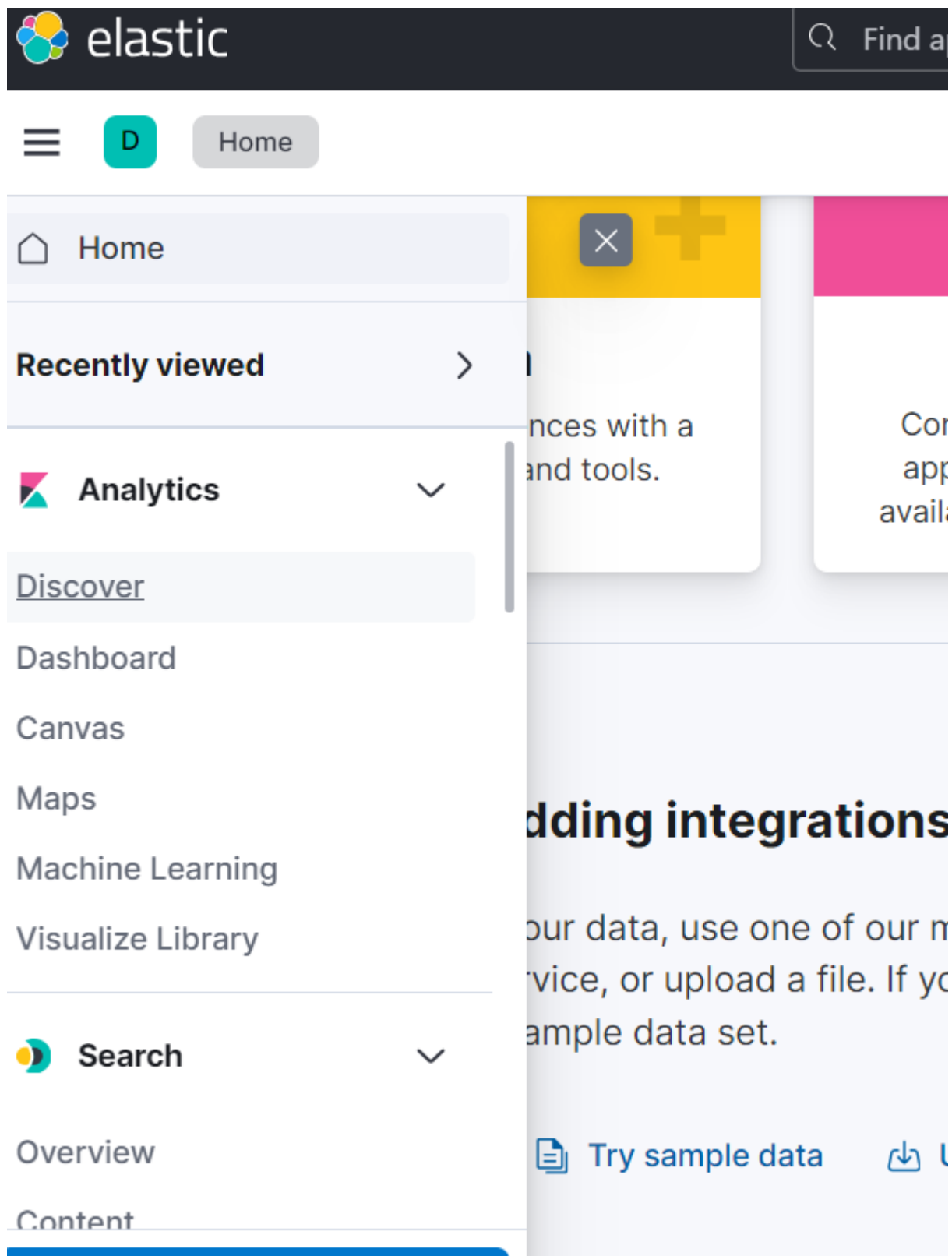
接著新增一個名為 source 的 ActivitySource，使用的名稱為 Sample.DistributedTracing

```
ActivitySource source = new ActivitySource("Sample.DistributedTracing", "1.1.0");
```

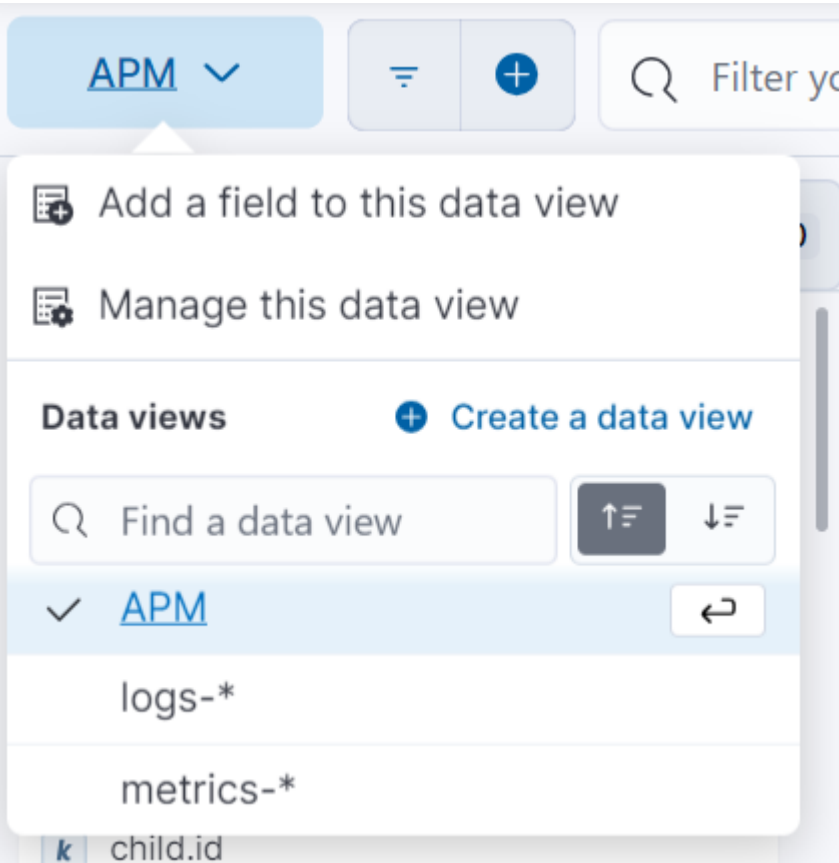
接著幫此Activity加入key為foo的tag, value為bar1

```
using (var activity = source.StartActivity("Main"))
{
    activity?.SetTag("foo", "bar1"); //加入此trace要使用的log
}
```

接著就是開始建置後使用ELK觀察結果，首先到左方的menu選擇discover



接著左方的filter選擇APM



接著在右方可看到剛剛傳送上來的traces，點選其中一個並查看其內容。



可以看到剛剛設的tag已經被傳送上了。



以上就成功埋入相關的tag到automatic instrumentation中了。

接著為嘗試加入metrics

首先一樣在專案中加入System.Diagnostics.DiagnosticSource的Nuget Package

接著在code裡加入以下的程式碼

```
var meter = new Meter("Sample.Service", "1.0");
var successCounter = meter.CreateCounter<long>("srv.successes.bing",
description: "Number of successful responses");
successCounter.Add(1, new KeyValuePair<string, object?>("tagName",
"tagValue"));
```

命名一個名為Sample.Service的Meter，並且建立一個名為srv.successes.bing的counter，並且加入一個tag為tagName，value為tagValue的counter。

接著就是開始建置後使用ELK觀察結果，首先一樣到左方的menu選擇discover，並且左方的filter選擇APM 接著在右方的filter輸入data\_stream.type:"metrics"，可看到下方出現相關的record，點選其中一條即可看到剛剛傳送上的srv.successes.bing，並且可以看到剛剛設的tag已經被傳送上了。



```
],  
  "event.ingested": [  
    "2023-10-12T09:20:37.000Z"  
  ],  
  "srv.successes.bing": [  
    41  
  ],  
  "@timestamp": [  
    "2023-10-12T09:20:36.801Z"  
  ],  
],
```

 Copy value

ps. `metrics`預設為每分鐘一筆，所以要等一分鐘才會出現。`traces`則是每個operation的當下產生並送出。

## Demo

### resource

<https://cloud.google.com/architecture/devops>