

RuleEngine for IoT build using Node-Red

Yvonne

2020/5/28

Contents

1	Architecture	3
1.1	Introduction	3
2	Docker	3
2.1	Introduction	3
2.2	Concepts	4
2.3	Installation	4
2.4	How to use?	5
2.5	Docker Network	6
2.6	Dockerfile	8
3	Node-Red	10
3.1	Introduction	10
3.2	Concepts	10
3.3	Installation	13
4	InfluxDB	14
4.1	Introduction	14
4.2	Concepts	14
4.3	Installation	15
4.4	How to use?	15
4.5	In Node-Red	16
5	gRPC	18
5.1	Introduction	18
5.2	Protocol Buffers	18
5.3	How to use?	18
5.4	Protobuf 的資料類型	20
5.5	Service (以.Net Core 為例)	20

6	Grafana	23
6.1	Introduction	23
6.2	Installation	23
6.3	How to use?	23
7	Kafka	25
7.1	Introduction	25
7.2	Concepts	25
7.3	Installation	25
7.4	How to use?	28
7.5	In Node-Red	29
8	VoltDB	29
8.1	Introduction	29
8.2	Concepts	30
8.3	Installation	30
8.4	How to use?	31
9	ContainerMgr	33
9.1	Docker Engine REST API	33
9.2	NodeRed Admin API	37

1 Architecture

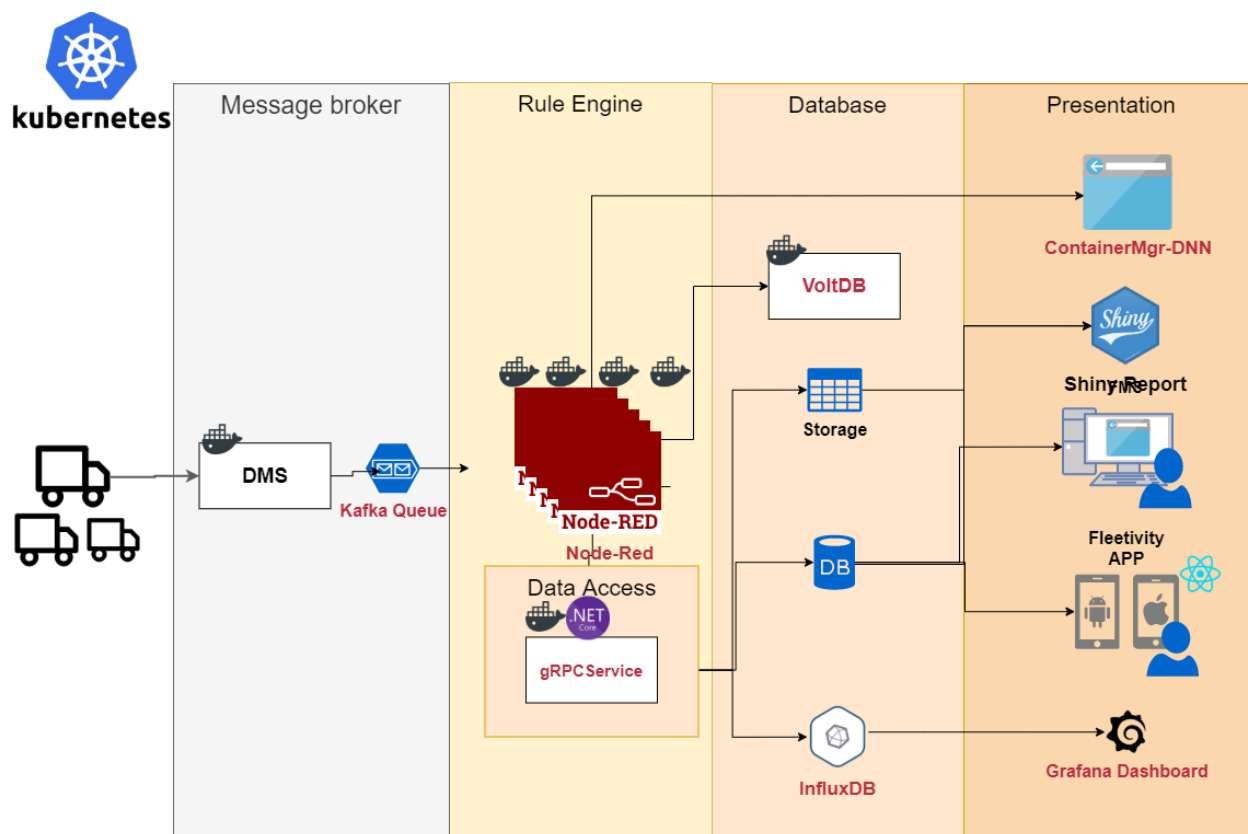


Figure 1: EDR Node Red Architecture

1.1 Introduction

這次 RuleEngine 的架構應用容器化架構及微服務架構來進行，微服務架構將單一的應用程式拆分成多個小型服務的方式，將每一個服務包成 Container 獨立運作，利用 kafka Queue、gRPC 或是 HTTP 等輕量級通訊機制使服務之間能互相溝通，因為服務皆是獨立運行，所以能夠達到獨立部屬、擴展、替換且跨平台的效果，希望解決現行架構過度耦合，開發成本高及可用性低的問題。

不過因為服務數量的增加，其實整體的架構是複雜化的，維運管理難度也相應提升，因此後續需要導入類似 Kubernetes 等工具，協助我們進行微服務的管理、監控與維運。

架構圖中紅色的部分為此次文件包含的範圍，這次我們選擇使用 Node-Red 視覺化程式作為整個 Rule-Engine 資料流程處理的工具，由 Kafka 擔任 Adapter 與 Rule-Engine 中傳遞資訊的角色，其中搭配 gRPC Service 作為服務運算邏輯，VoltDB 作為 Cache 資料庫，Influxdb 這個時間序列資料庫儲存 RawData 等時間序列資料，Grafana 作為儀錶板視覺化的呈現工具。

2 Docker

2.1 Introduction

Docker 是一個開源專案，提供開發、轉移和執行應用程式。透過映像檔，將作業系統核心外，運作應用程式所需要的系統環境，由下而上打包，達到應用程式跨平臺無縫接軌運作。

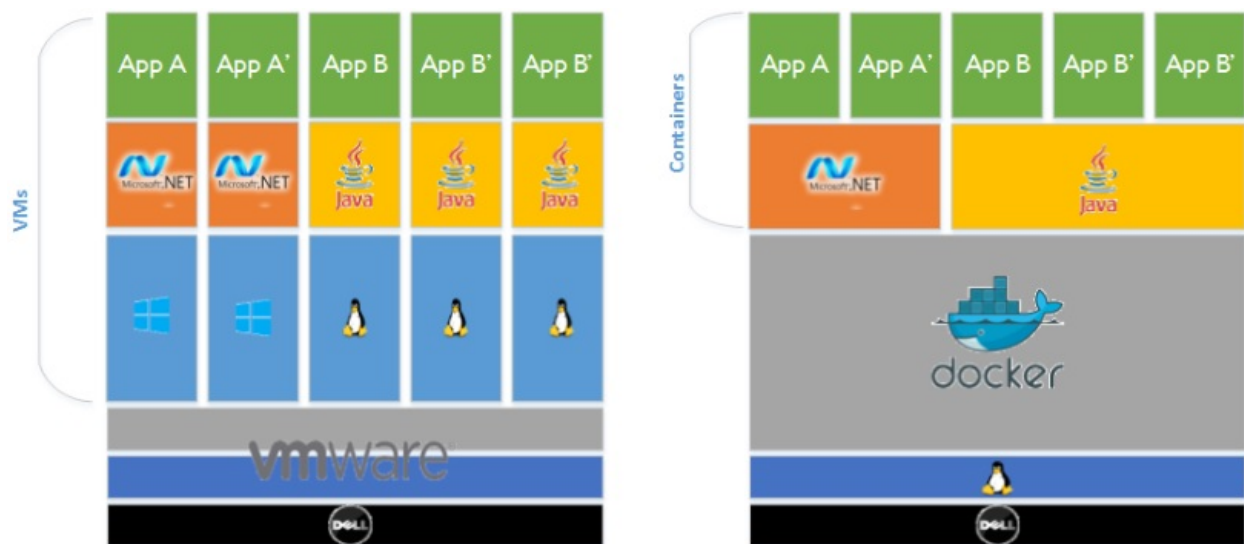


Figure 2: VM vs Docker

1. 環境一致性：能夠解決開發、測試、正式環境的差異性，使運行環境能夠一致且搬移彈性便捷。
2. 快速部署：快速地建立、測試和部署應用程式至各種環境。
3. 獨立運作與隔離：運行在同一個實體機中的各個 Container 資源獨立互不影響
4. 搬遷、擴展更容易：透過映像檔容易將服務進行搬遷、複製、擴展。

2.2 Concepts

- Docker Image (映像檔)
Docker Image 是唯獨的環境模板，包括應用程式、函式庫、環境設定檔...，用來建立 Docker Container。
- Docker Container (容器)
Docker Container 是根據 Docker Image 所建立的 instance Process。Docker Image 是唯獨的，而 container 是可寫層。同一個 Docker Image 可以啟動多個 Docker Container，每個 **Container** 之間都是互相獨立的個體。
- Docker Registry (倉庫)
Docker Registry 是一個線上伺服器，儲存及管理各式的 Docker Image。Registry 分成 Public 和 Private，目前最大的 Public Registry 是 Docker Hub。
- Volume
當你使用 volume 時，Docker 會在你的本機上隨機新增一個資料夾，並可以設定讓這個資料夾跟 Container 裡面指定的資料夾互通。所以當你 Container 裡面那個資料夾有任何變更時，Host 的資料夾也會跟著變，且 Container 被刪掉時那個資料夾並不會被刪掉，可以永久保存 Container 的資料。依據 Volume Driver 的設定，可以指定本機內部儲存空間或是外部儲存空間。

2.3 Installation

Docker 本質上是基於 Linux 核心來建立的技術，因此 Linux 對於 Docker 支援是最完善的。

以下範例為在 Ubuntu 上安裝 Docker：

- 進行 Docker 的安裝

```
sudo apt-get install docker.io
```

- 啟動 docker 服務

```
sudo service docker start
```

- 查看 docker version

```
sudo docker version
```

正常的情況應該要看到這樣

```
Client:
Version:      18.09.5
API version:  1.39
Go version:   go1.10.8
Git commit:   e8ff056dbc
Built:        Thu Apr 11 04:44:28 2019
OS/Arch:      linux/amd64
Experimental: false
Server: Docker Engine - Community
Engine:
Version:      18.09.5
API version:  1.39 (minimum version 1.12)
Go version:   go1.10.8
Git commit:   e8ff056
Built:        Thu Apr 11 04:10:53 2019
OS/Arch:      linux/amd64
Experimental: false
```

2.4 How to use?

- 執行 Docker Container

```
docker run -d -p 8080:80 --restart=always --name myinfluxdb influxdb
```

- -d: 把 container 執行在背景裡
- -p: 將 Container 裡的 80 port mapping 到 host 的 8080 port
- --restart=always: 如果 container 遇到例外的情況被停止，docker 會試著重新啟動這個 Container
- --name=myinfluxdb: 設定 container 的名稱為 myinfluxdb
- 最後一個參數 influxdb 是 Docker Image 的名稱

- 查看 Docker Container 清單

```
docker ps -a
```

- -a: 顯示所有 Docker Container，如果沒有，只會顯示正在執行中的 container

- 啟動 Container

```
docker start myinfluxdb
```

– start 後面可以接 ContainerName 或是 ContainerID

- 停止 Container

```
docker stop nginx
```

– stop 後面可以接 ContainerName 或是 ContainerID

- 刪除 Container

```
docker rm -f myinfluxdb
```

– -f: 強制刪除 Container

– 最後可加 ContainerName 或是 ContainerID

- 進入 Container

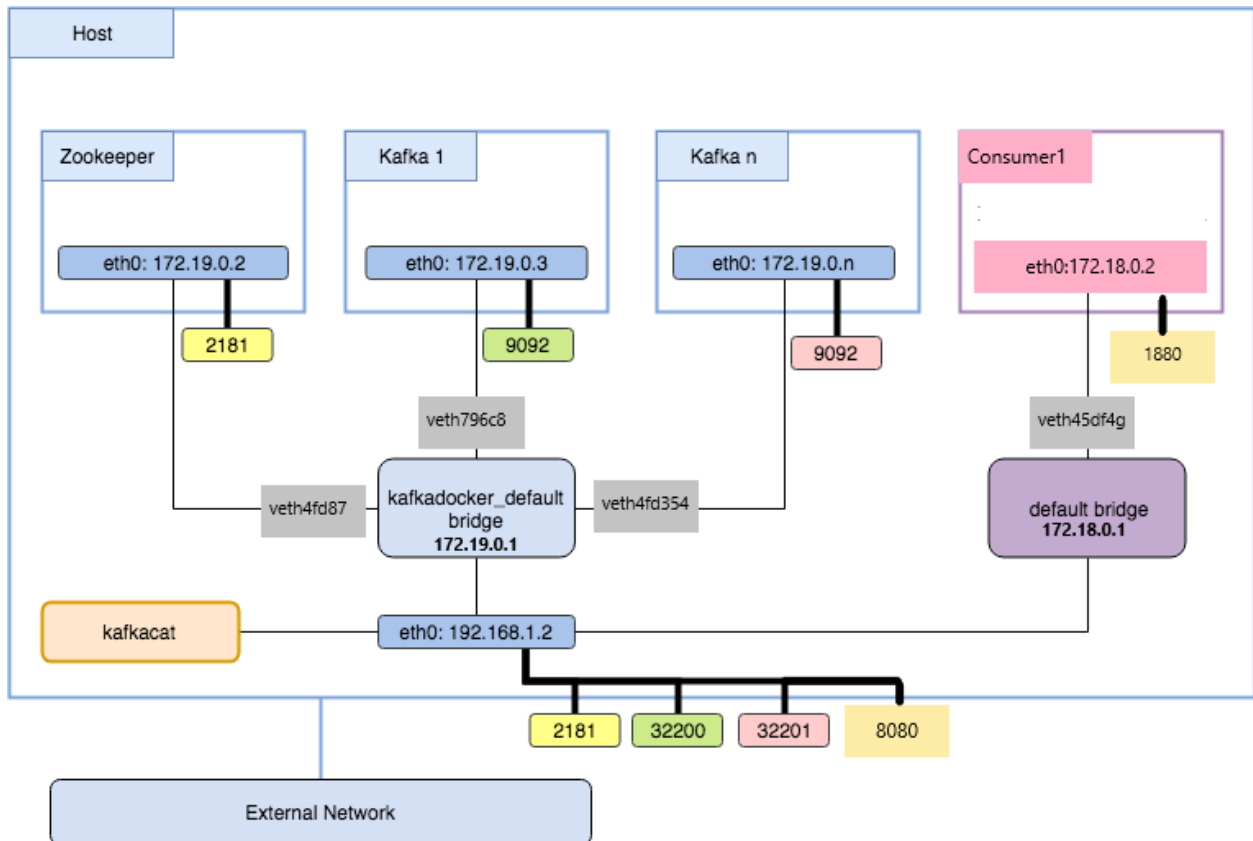
```
docker exec -it myinfluxdb /bin/bash
```

- 查看 Container 詳細資訊

```
docker inspect myinfluxdb
```

2.5 Docker Network

Docker 會產生 Linux bridge 作為 Container 與 Host 網路的橋樑，並透過此 bridge 可以讓同一個 bridge 的 Container 之間相互進行通訊。



上圖所示，Zookeeper、kafka1、kafkan，這三個 Container 在同一個 Docker Network(bridge) 中，各自透過虛擬網路 veth 連接至 kafkadocker_default bridge 中，而透過 kafkadocker_default bridge 連接至實體網路。

不同 bridge 上 Container 的通訊測試：

```
# 測試與 kafka1 通訊
$ docker exec -it Zookeeper ping -c 2 kafka1
PING kafka1 (172.19.0.3): 56 data bytes
64 bytes from 172.19.0.3: seq=0 ttl=64 time=0.095 ms
64 bytes from 172.19.0.3: seq=1 ttl=64 time=0.076 ms

# 測試與 consumer1 通訊
$ docker exec -it Zookeeper ping -c 2 Consumer1
ping: bad address 'Consumer1'

# 以 IP 的方式測試與 Consumer1 通訊
$ docker exec -it Zookeeper ping -c 2 172.18.0.2
PING 172.18.0.2 (172.18.0.2): 56 data bytes

# 以 IP 的方式測試與 default_bridge 通訊
$ docker exec -it Zookeeper ping -c 2 172.18.0.1
PING 172.18.0.1 (172.18.0.1): 56 data bytes
64 bytes from 172.18.0.1: seq=1 ttl=64 time=0.089 ms
64 bytes from 172.18.0.1: seq=2 ttl=64 time=0.101 ms

# 測試連外能力
$ docker exec -it Zookeeper ping -c 2 www.google.com
PING www.google.com (172.217.160.68): 56 data bytes
```

```
64 bytes from 172.217.160.68: seq=0 ttl=53 time=7.252 ms
64 bytes from 172.217.160.68: seq=1 ttl=53 time=7.329 ms
```

結論：

- 在同一個 Docker network (bridge) 中的 Container 可以透過 Name 或 IP 的方式通訊
- 不同 Docker network (bridge) 之間的 Container，無論是透過 Name 或 IP 的都無法直接相互通訊的，
- 不同 Docker network (bridge) 之間的 Container 可以透過 bridge Gateway IP 相互通訊。

2.6 Dockerfile

Dockerfile 是用來撰寫 Docker Image 的建置腳本，使用 Docker build 即可執行腳本。

Docker 17.05+ 以後提供名為「多階段構建 (multi-stage builds)」的新功能，將多個映像檔整合進同個 Dockerfile 內，而後續的映像檔可透過指令取得中間映像檔所產生的檔案，如此能讓整個過程更為簡單。

常用指令：

- FROM：使用到的 Docker Image 名稱
- WORKDIR：設定工作路徑
- RUN：RUN 指令後面放 Linux 指令，用來執行安裝和設定這個 Image 需要的東西
- EXPOSE：向外部開放 Port
- ENV：用來設定環境變數
- CMD：描述 Container 啟動後執行的程序
- 多階段建構 Dockerfile 範例 (.NetCore)：

```
# Stage 0
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

# Stage 1
# copy csproj and restore its dependencies
FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster AS build
WORKDIR /src
COPY ["GrpcAppService/GrpcAppService.csproj", "GrpcAppService/"]
RUN dotnet restore "GrpcAppService/GrpcAppService.csproj"

# Stage 2
# copy everything else and build app
COPY . .
WORKDIR "/src/GrpcAppService"
RUN dotnet build "GrpcAppService.csproj" -c Release -o /app/build
```

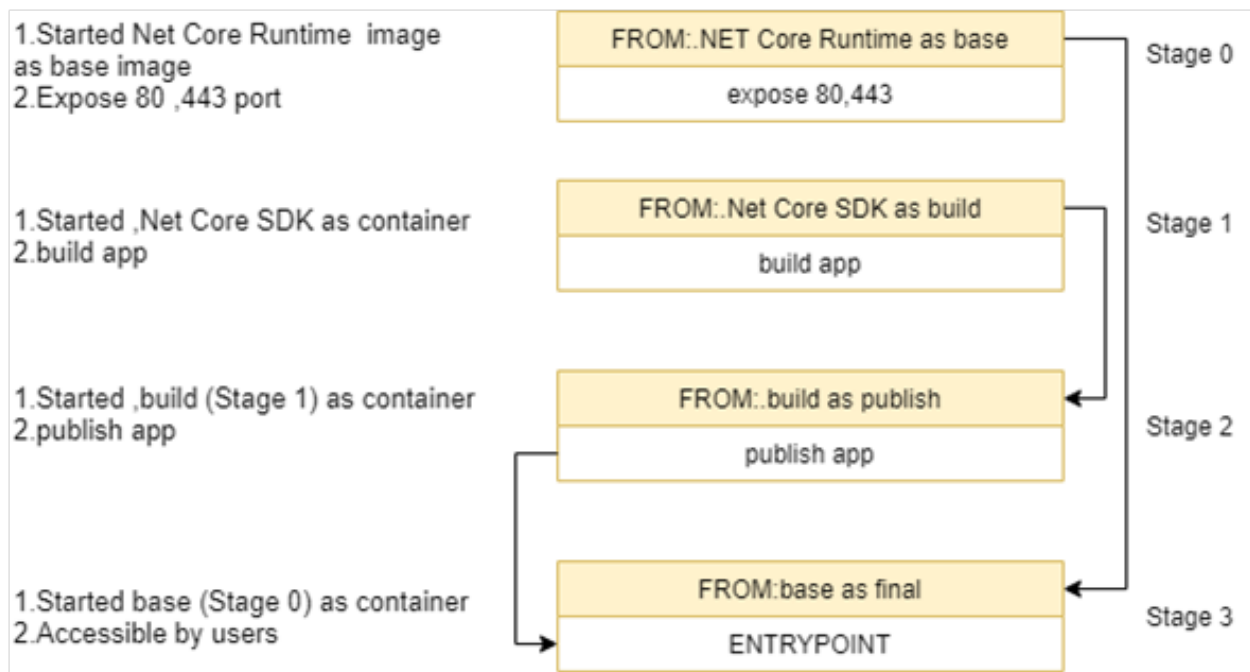



Figure 3: 多階段建構

```
# Publish app and its dependencies to a folder for deployment to a hosting system.
FROM build AS publish
RUN dotnet publish "GrpcAppService.csproj" -c Release -o /app/publish

# Stage 3

# copy the published app and set the entrypoint
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "GrpcAppService.dll"]
```

- 分階段講解

- Stage 0

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443
```

- * 以 mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim(ASP.Net Core Runtime) 為 base image，命名為 base
- * 指定工作路徑為 /app
- * 把 PORT 80 和 443 EXPOSE 出來

- Stage 1

```
# copy csproj and restore
FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster AS build
```

```

WORKDIR /src
COPY ["GrpcAppService/GrpcAppService.csproj", "GrpcAppService/"]
RUN dotnet restore "GrpcAppService/GrpcAppService.csproj"

# copy everything else and build app
COPY . .
WORKDIR "/src/GrpcAppService"
RUN dotnet build "GrpcAppService.csproj" -c Release -o /app/build

```

- * 以 `mcr.microsoft.com/dotnet/core/sdk:3.1-buster` (.NetCore SDK) 為 base image，命名為 `build`
- * 複製 “GrpcAppService/GrpcAppService.csproj” 到 GrpcAppService/ 資料夾中
- * `dotnet restore`：用 NuGet 來還原相依性以及專案檔中指定的專案特定工具
- * `COPY . .` 複製資料夾中所有檔案
- * `dotnet build`：建置專案和其所有相依性

– Stage 2

```

FROM build AS publish
RUN dotnet publish "GrpcAppService.csproj" -c Release -o /app/publish

```

- * 利用 stage 1 建立好的中間映像檔 (`build`) 作為 parent image，命名為 `publish`
- * `dotnet publish`：將應用程式和所有相依套件封裝到 `publish` 資料夾，以準備進行部署

– Stage 3

```

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "GrpcAppService.dll"]

```

- * 利用 stage 1 建立好的中間映像檔 (`base`) 作為 parent image，命名為 `final`
- * 利用 `COPY` 取得剛剛在 stage 2 產生的 artifacts
- * 設定 `ENTRYPOINT` 指定容器的啟動命令。

3 Node-Red

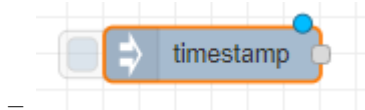
3.1 Introduction

Node-RED 是由 IBM 開發的，基於 Web 瀏覽器的流程可視化程式開發工具，運行在 Node.js 上，可用於創建 JavaScript 函數，應用程式的元素可以保存或共享以供重複使用。

3.2 Concepts

- Node

Node 是 Node-Red 處理流程 (Flow) 中的最小單位。通過從流程 (Flow) 中的上一個 Node 接收消息或通過等待某些外部事件 (例如傳入的 HTTP 請求，計時器或 GPIO 硬件更改) 來觸發節點。他們處理該消息或事件，然後可以將消息發送到流程 (Flow) 中的下一個 Node。一個 Node 最多可以具有一個輸入端口和所需的多個輸出端口。以下介紹幾種內建常用的 Node：



Inject：Inject Node 可用來手動觸發流程 (點擊按鈕) 或是設定定時自動觸發流程，可以設定發送訊

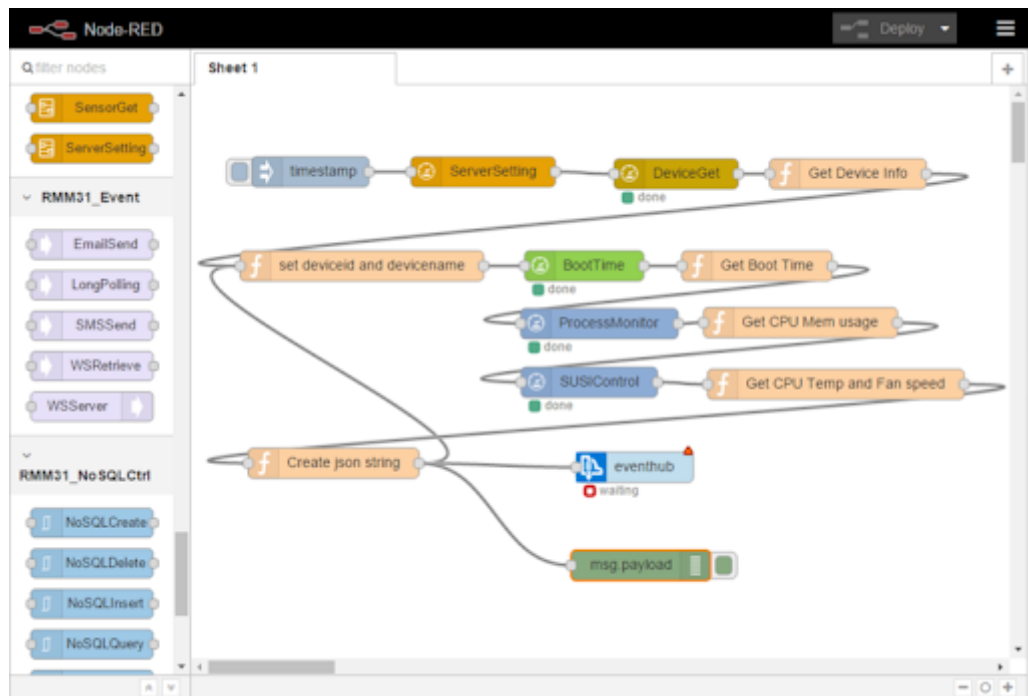


Figure 4: Nodered Editor

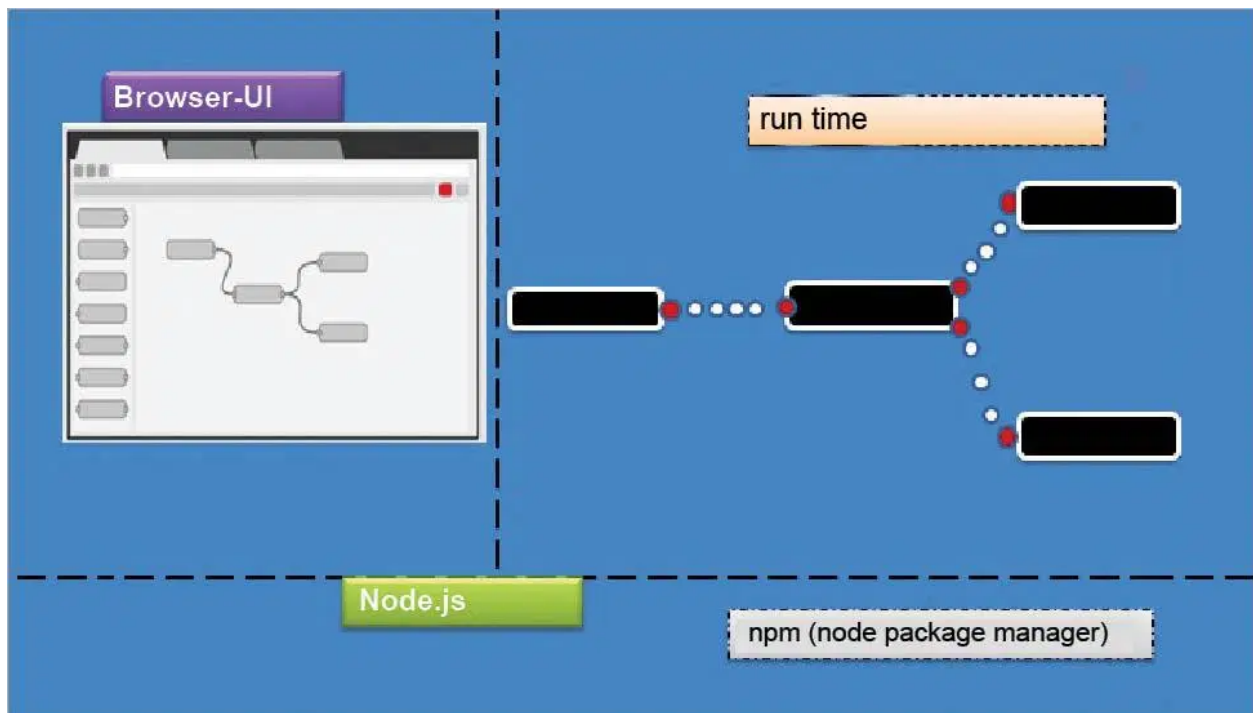
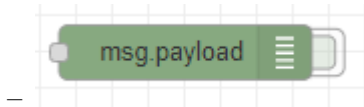


Figure 5: Node Red Architecture

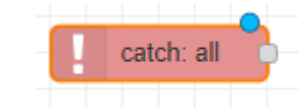
息的 payload，預設是當下的 TimeStamp，Unix epoch time in milliseconds，從 UTC1970 年 1 月 1 日 0 時 0 分 0 秒起至現在的總毫秒數。



Debug：Debug Node 可以將指定訊息顯示至側邊欄 Debug Tab 中。右側的按鈕可以啟用或禁用。



Function：Function Node 可以在其中編寫 JavaScript 程式碼運行。



Catch：Catch Node 可以抓取所有 Nodes 或是指定 Node 所產生的 Exception。

- Configuration node

Configuration (config) Node 負責設定其他 Node 的 Config，無法在流程中串接其他 Node。

- Flow

在 Node-Red 中，編輯器裡的一個 Tab 或者是由 Node 串接起來的處理流程都稱為 Flow。

- Context

Context 是 Node-Red 中存取資訊的地方，在 Node-Red 的 config 檔中可以設定 context 所使用的儲存方式，可設定一至多種，預設儲存在 Memory 中，所以並不是永久保存會隨著 Node-Red 關閉而消失，另外也可以設定儲存在 Local File System 中或是利用 API 自定義儲存，以做永久性資料保存之用。

```
contextStorage: {
  default: "Inmemory", //預設儲存
  //在 Node-Red 中指定"Inmemory_store_name"的存取方式，會使用 memory 方式存取
  Inmemory_store_name: { module: 'memory' },
  //在 Node-Red 中指定"file_store_name"的存取方式，會使用 localfilesystem 方式存取
  file_store_name: { module: 'localfilesystem' }
}
```

在 Node-Red 中的 Function Node 可以用以下方式操作存取 Context 及指定存取方式。

```
// Get value_sync (指定 Inmemory_store_name 的存取方式)
var myCount = flow.get("count", "Inmemory_store_name");

// Set value_async (指定 file_store_name 的存取方式)
flow.set("count", 123, "file_store_name", function(err) { ... })
```

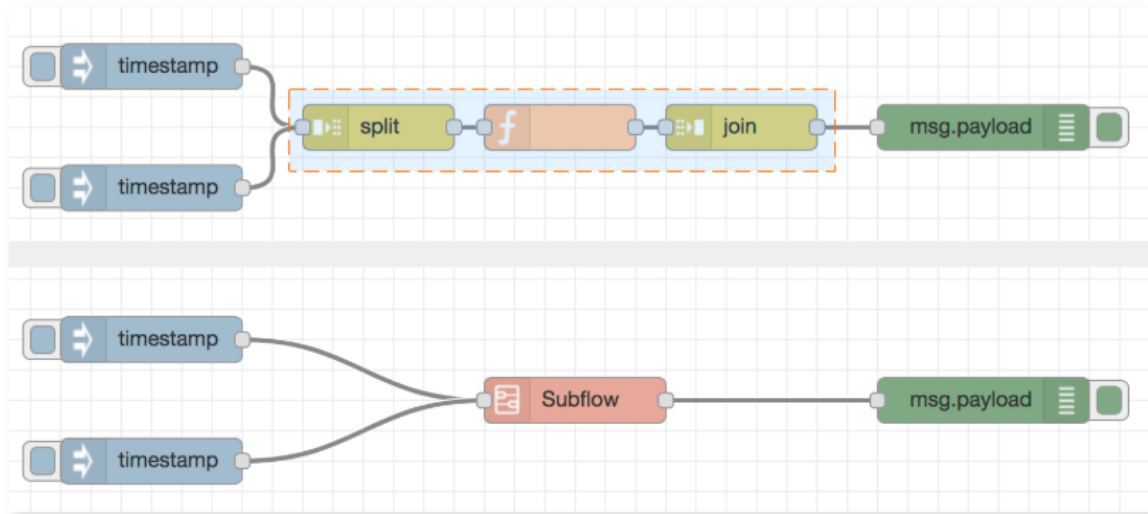
Context 依照存取範圍有分為以下三種：

- Node - only visible to the node that set the value
- Flow - visible to all nodes on the same flow (or tab in the editor)
- Global - visible to all nodes

- Message(msg)

Messages 是一個用來在流程中傳遞資訊的 javascript 物件。msg.payload 通常用來儲存最常用的資訊，大部分 Node 預設的輸入及輸出也都是 msg.payload。

- Subflow



Subflow 可以將 nodes 打包成一個 Subflow Node，方便作為一個可重複利用的元件，也可以讓 workspace 看起來不那麼複雜。

- Input/Output：Subflow 可以設定輸入及輸出端口數量，輸入只能最多 1 個，輸出則不限可自行設定。
- Properties：Properties 中可以設定 Subflow 中會使用的環境變數，

- Palette

Palette 中可以看到所有可用的 Node，也可以在此安裝需要的 Node 套件。

Example：



上圖這個 flow 即表示將從 kafka 收到的 queue，進行 Json Parser 成 Javascript Object，接著分別進行 BackUpRawData 即 IdleAnalyzer 的 subflow 流程。

3.3 Installation

```
docker run -it -p 1880:1880 -v <Host-Volume-Path>:/data
--name <docker-name> nodered/node-red
```

- -it 參數代表在執行 Docker 虛擬容器環境時，開啟虛擬終端機，以互動的模式執行。
- -p 參數將 Docker 容器內部的連接埠 mapping 到實體機器 (VM) 的連接埠，例如：-p 1880:8080 即為將 Docker 容器的 8080 連接埠對應到實體機的 1880 連接埠，所以輸入 localhost:1880 即可造訪容器中的 8080，
- -v 參數，可以指定 volume 要跟容器內哪一個資料夾連通

Docker-compose 寫法：

```

mynodered:
  image: nodered/node-red
  hostname: mynodered
  ports:
    - "1880:1880"
  environment:
    - COMPOSE_CONVERT_WINDOWS_PATHS=1
  volumes:
    - C:\Users\keywinRD0010\nodered-data:/data
  container_name: mynodered

```

4 InfluxDB

4.1 Introduction

InfluxDB 是一個開源的時間序列資料庫。由 Go 寫成，被廣泛應用於系統的監控數據，IoT 行業的即時數據等。

The diagram shows a table of data points with the following structure:

time	butterflies	honeybees	location	scientist
2015-08-18T00:00:00Z	12	23	1	langstroth
2015-08-18T00:00:00Z	1	4	1	perpetua
2015-08-18T06:00:00Z	11	2	1	perpetua
2015-08-18T00:06:00Z	3	28	1	perpetua
2015-08-18T05:54:00Z	2	11	2	langstroth
2015-08-18T06:00:00Z	1	10	2	langstroth
2015-08-18T06:06:00Z	8	23	2	perpetua
2015-08-18T06:12:00Z	7	22	2	perpetua

Callouts in the image:

- Measurement**: points to the 'name: census' header.
- Field key**: points to the column headers 'butterflies', 'honeybees', 'location', and 'scientist'.
- Field value**: points to the numerical values in the 'butterflies' and 'honeybees' columns.
- Tag key**: points to the 'scientist' column header.
- Tag value**: points to the values in the 'scientist' column.
- Timestamp**: points to the 'time' column header.

Figure 6: InfluxDB Concepts

4.2 Concepts

- **database**
資料庫
- **measurement**
表示資料的對應含意，類似 SQL 中的資料表，表示整個 time series data。
- **Point**
資料點，基本上由 time、一個或多個 field、一個或多個 tag 組成。

- **time**
即表示時間戳記，視為 primary index。
- **field**
表示一個或多個 <key,value> 的欄位，儲存該時間點所在的屬性與其數值，像是 <temparture, 28.5> 是 InfluxDB 真正儲存資料訊息的地方，可接受的資料類型包括：**string**、**int64**、**float64**、**boolean**。請特別注意，不接受 **Null**
- **tag**
表示一個或多個 <key,value> 的欄位，但型態皆是字串，與 field 不同，tag 為默認的索引值，用來 query 資料。這個欄位是 optional。

4.3 Installation

- 執行 Container

```
docker run -d --name <container-name> -p 8086:8086 -p 8083:8083
-v <Host-Volume-Path>:/var/lib/influxdb influxdb
```

- -d Run container in background and print container ID
- 8086 為 HTTP API port
- conf 文件初始化
在預設執行目錄下會有一個從 docker 中複製出來的 influxdb.conf

```
docker run --rm influxdb influxd config > influxdb.conf
```

- 指定 config

```
docker run -d --name <container-name> -p 8086:8086 -p 8083:8083 -v
<Host-Config-Volume-Path>:/etc/influxdb/influxdb.conf:ro -v
<Host-Volume-Path>:/var/lib/influxdb influxdb
```

4.4 How to use?

- 進入終端機

```
docker exec -it influxdb influx
```

- 查看資料庫列表

```
show databases
```

- 建立資料庫

```
create database <database_name>
```

- 使用資料庫進行任何操作都需先指定資料庫

```
use <database_name>
```

- 查看 Measurement 列表

```
show measurements
```

- 刪除 Measurement

```
drop measurement <measurement_name>
```

- 刪除資料庫

```
drop database <database_name>
```

4.5 In Node-Red

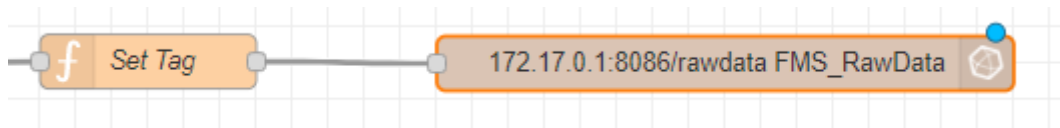


Figure 7: Insert into influxDB in Node-Red

- Set Tag Function Node 將資料轉換成輸入格式

```
//設定 tag 欄位格式
msg.payload.tag.DEVICE_ID=msg.payload.V0.DEVICE_ID
//設定 time 欄位格式 (Unix epoch time in milliseconds)
msg.payload.V0.time=dateTime*1000000;
var arr=[];

//因為 influxdb field 值不接受 NULL，所以將值為 Null 的屬性篩選掉。
Vodata=filterParams(msg.payload.V0);

arr.push(Vodata);
//設定 tag 為 deviceid
arr.push(msg.payload.tag);
msg.payload=arr;
```

- 因為 influxdb field 值不接受 NULL，所以需要先將值為 Null 的屬性篩選掉。
- msg.payload 的指定格式要是一個 Array 包含兩個物件，第一個物件會寫入 fields 中，第二個物件會寫入 tags 中。
- Influxdb out Node
 - 按鉛筆編輯 Influxdb 連線資訊
 - 設定連接字串和指定的 Measurement。

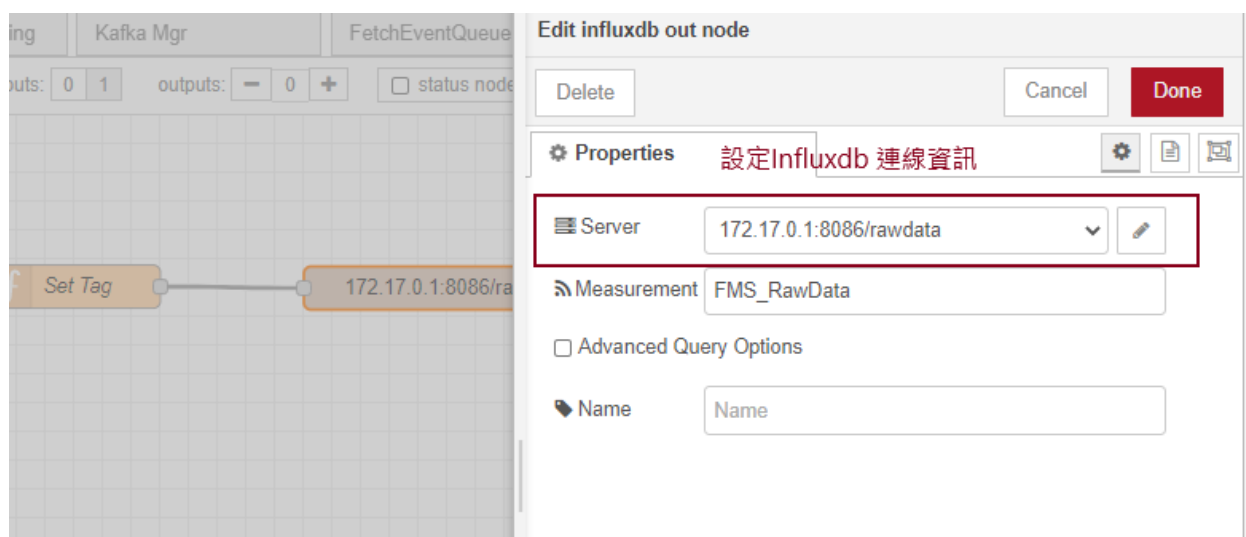


Figure 8: Influxdb out Node

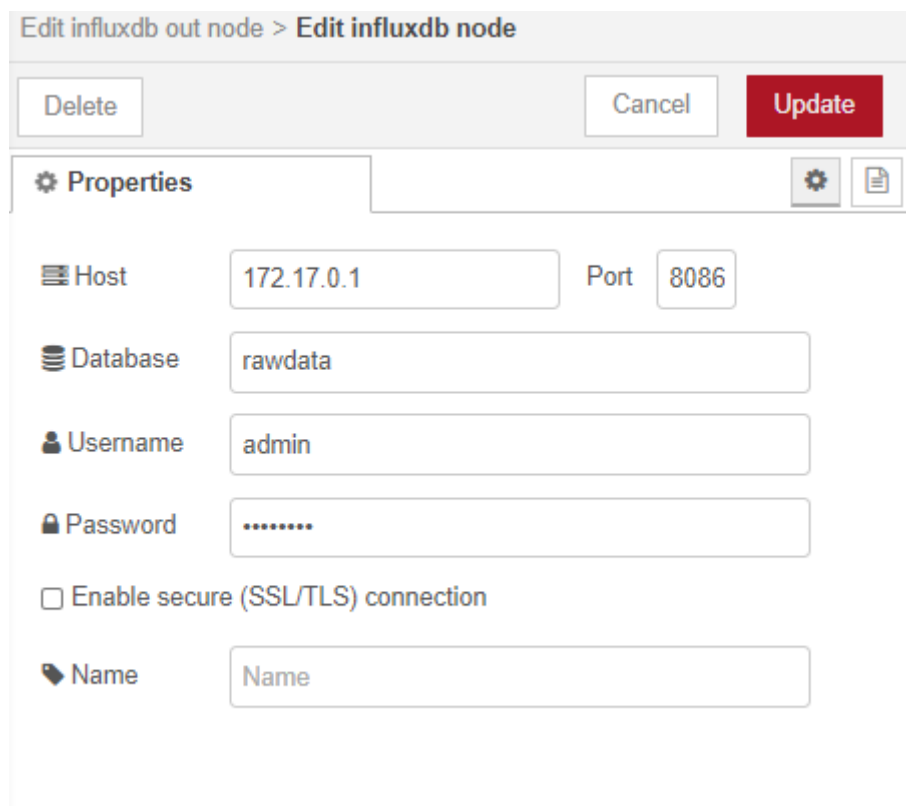


Figure 9: Influxdb 連線資訊設定

5 gRPC

5.1 Introduction

gRPC 是 Google 的一個開源遠端程序呼叫系統 (**Remote Procedure Calls**)。該系統基於 HTTP/2 協定傳輸，使用 Protocol Buffers 作為介面描述語言。適合於微服務框架，能於跨語言、跨平台的服務之間高效互動。支持多種語言：C++、Java、Go、Python、Ruby、C#、Node.js、Android Java、Objective-C、PHP 等。

#RPC Remote Procedure Call(RPC) 便是將本地程式的 function，外顯出來，利用 TCP/UDP/HTTP 各種網路通訊方法，讓別台機器可以透過網路呼叫你的這個 function。讓 Client 像是呼叫本地 function 一樣得到 return 值。

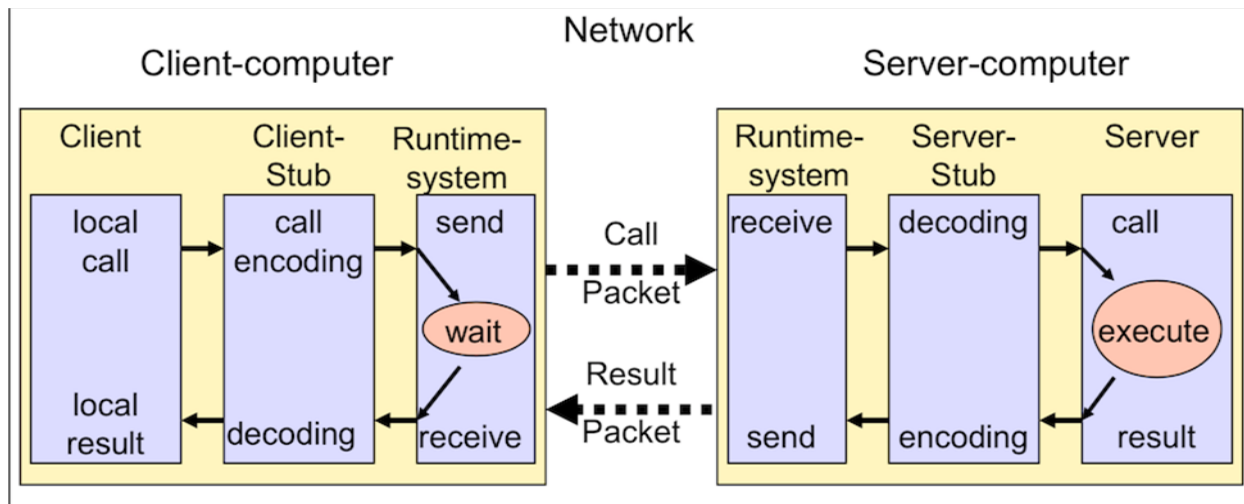


Figure 10: RPC

5.2 Protocol Buffers

5.3 How to use?

Protocol Buffers 是由 Google 所定義的一種資料格式，檔案的副檔名是 .proto，可以看成是 API 的定義檔，gRPC 中的 Client Stub 及 Server Stub 必須依賴 proto 檔來產生。

```
service Eventer {
  //Get Event Queue
  rpc GetEvent (GetEventRequest) returns (GetEventReply);
}

// The request message containing How many eventqueue shoud be Fetch.
message GetEventRequest {
  int32 message_count=1;
}

// The response message containing the events.
message GetEventReply {
  repeated EventVO V0=1;
}
```

```
//The message containing the event.
message EventVO{

    string ONE_WIRE_DATA=1;
    string ONE_WIRE_DATA_2=2;
    string RFID_DATA=3;
    int32 GPIO12=4;
    int32 GPIO13=5;
    int32 GPIO14=6;
    int32 GPIO15=7;
    int32 GPIO16=8;
    int32 AD1=9;
    int32 AD2=10;
    int32 GPS_STATUS=11;
    double GPS_LATITUDE=12;
    double GPS_LONGITUDE=13;
    double GPS_SPEED=14;
    double GPS_HEADING=15;
    int32 GPS_ALTITUDE=16;
    int32 GPS_SAT_COUNT=17;
    int32 GPIO11=18;
    int32 GPS_ODOMETER=19;
    int32 GPIO10=20;
    int32 GPIO8=21;
    string EVENT_ID=22;
    string DEVICE_ID=23;
    string RAW_DATA_ID=24;
    int32 USER_NUMBER_PARAM1=25;
    int32 FORMAT_CODE_PARAM2=26;
    int32 GPIO1=27;
    int32 GPIO2=28;
    int32 GPIO3=29;
    int32 GPIO4=30;
    int32 GPIO5=31;
    int32 GPIO6=32;
    int32 GPIO7=33;
    int32 GPIO9=34;
}
```

- syntax：描述 Protocol Buffers 格式是哪一個版本
- package：類似 Namespace 的效果，是 Optional 的欄位，但是如果遇到 service 或 message 有衝突的時候，package 有助於明確化。
- option csharp_namespace：C# 在產生程式碼時所使用的 Namespace 名稱
- service：定義一個服務
- rpc ... returns ...：定義一個 API
- message：定義一個訊息結構
- int32、int64、string、...：Protocol Buffers 的資料類型
- 欄位後面的數字：這個是指定欄位在結構中的順序，從 1 開始，也是 gRPC 實際上在對應 message 欄位時的順序。

5.4 Protobuf 的資料類型

- 純量資料類型：

原型	C# 類別
double	double
float	float
int32	int
int64	long
bool	bool
string	string
uint32	uint
uint64	ulong
bytes	ByteString

- 其他

原型	C# 類別
DateTime	google.protobuf.Timestamp
TimeSpan	google.protobuf.Duration
int?	google.protobuf.Int32Value
double?	google.protobuf.DoubleValue

- DateTime & google.protobuf.Timestamp 轉換實例 (C#)

```
// Create Timestamp and Duration from .NET DateTimeOffset and TimeSpan
var meeting = new Meeting
{
    Time = Timestamp.FromDateTimeOffset(meetingTime), // also FromDateTime()
    Duration = Duration.FromTimeSpan(meetingLength)
};

// Convert Timestamp and Duration to .NET DateTimeOffset and TimeSpan
DateTimeOffset time = meeting.Time.ToDateTimeOffset();
TimeSpan? duration = meeting.Duration?.ToTimeSpan();
```

5.5 Service (以.Net Core 為例)

完成 proto 檔之後，我們必須建置專案以產生 Server Stub，那麼 proto 檔到這邊可以告一個段落了，接下來我們要撰寫 Server Service。Service 會依據 Proto 檔產生一個輸入參數為 GetEventRequest，輸出參數為 Task 的 GetEvent 函數。

```
“{C# eval=FALSE} public class EventerService : Eventer.EventerBase { private readonly ILogger __logger;
public EventerService(ILogger logger) { __logger = logger; }
```

```
public override async Task<GetEventReply> GetEvent(GetEventRequest request, ServerCallContext context)
{
    ...
    var result = new GetEventReply()
```

```

    {
        Events = { new VO() {
        }
        }
    };

allData.ForEach(data =>
{
    try
    {
        var vo = new EventVO()
        {

            AD1 = data.AD1,
            AD2 = data.AD2,
            GPIO1 = data.GPIO1,
            GPIO10 = data.GPIO10,
            GPIO11 = data.GPIO11,
            GPIO12 = data.GPIO12,
            GPIO13 = data.GPIO13,
            GPIO14 = data.GPIO14,
            GPIO15 = data.GPIO15,
            GPIO16 = data.GPIO16,
            GPIO2 = data.GPIO2,
            GPIO3 = data.GPIO3,
            GPIO4 = data.GPIO4,
            GPIO5 = data.GPIO5,
            GPIO6 = data.GPIO6,
            GPIO7 = data.GPIO7,
            GPIO8 = data.GPIO8,
            GPIO9 = data.GPIO9,
            GPSALTITUDE = data.GPS_ALTITUDE,
            GPSHEADING = data.GPS_HEADING,
            GPSLATITUDE = data.GPS_LATITUDE,
            GPSLONGITUDE = data.GPS_LONGITUDE,
            GPSPosition = data.GPS_Position != null ? data.GPS_Position : "",
            GPSODOMETER = data.GPS_ODOMETER,
            GPSSATCOUNT = data.GPS_SAT_COUNT,
            GPSSPEED = data.GPS_SPEED,
            GPSSTATUS = data.GPS_STATUS,
            GPSTRIPODOMETER = data.GPS_TRIP_ODOMETER,
            GPSUTCDATE = Timestamp.FromDateTime(data.GPS_UTC_DATE.ToUniversalTime()),
            GPSUTCTIME = Timestamp.FromDateTime(data.GPS_UTC_TIME.ToUniversalTime()),
            ONEWIREDATA = data.ONE_WIRE_DATA,
            ONEWIREDATA2 = data.ONE_WIRE_DATA_2,
            ONEWIREDATA3 = data.ONE_WIRE_DATA_3 != null ? data.ONE_WIRE_DATA_3 : "",
            RAWDATAID = data.RAW_DATA_ID != null ? data.RAW_DATA_ID : "",
            RFIDDATA = data.RFID_DATA != null ? data.RFID_DATA.ToString() : "",
            RTC = Timestamp.FromDateTime(data.RTC.ToUniversalTime()),

        };

        result.Events[0].VO.Add(vo);
    }
    catch (Exception ex)

```

```

        {
            Log(ex, DateTime.UtcNow)
        }
    });
});
return await Task.FromResult(result);
}

```

In Node-Red

![gRPC in Node-Red](gRPC_0.png)

* Set input Function Node
依據proto檔設定輸入資料

```

```bash
msg.payload={message_batch:12}

```

- gRPC Node
  - 按鉛筆編輯 gRPC 連線資訊及 proto 檔
  - 設定 Service 和 Method。

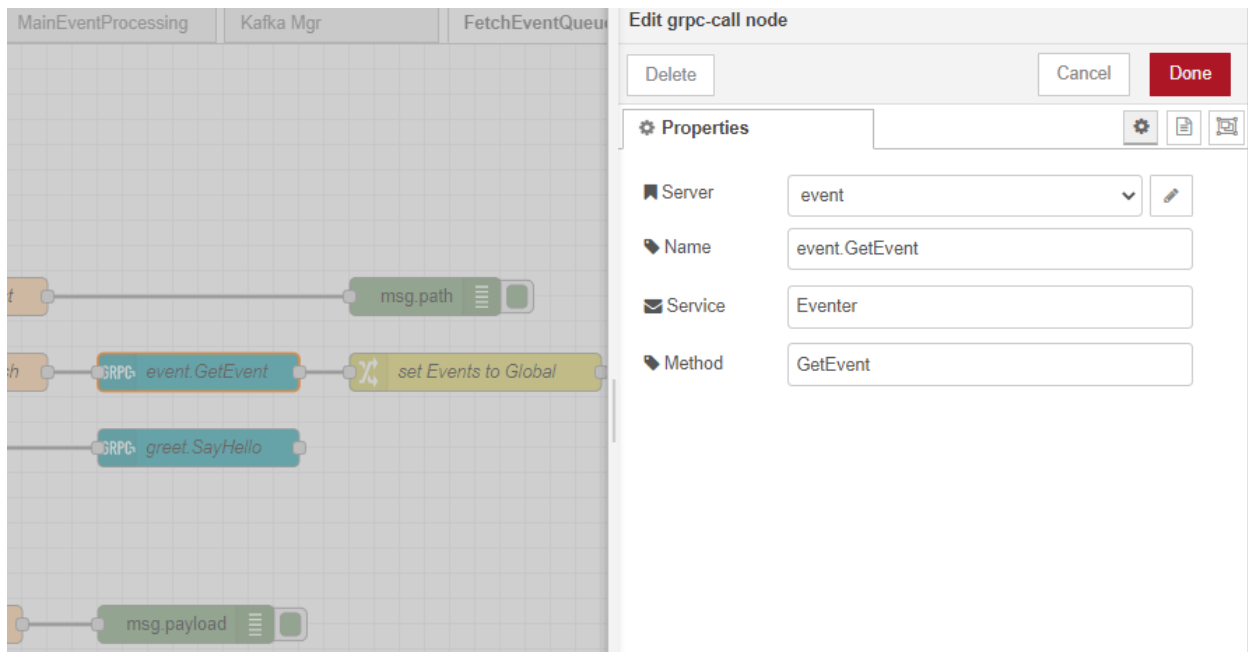


Figure 11: RPC Node

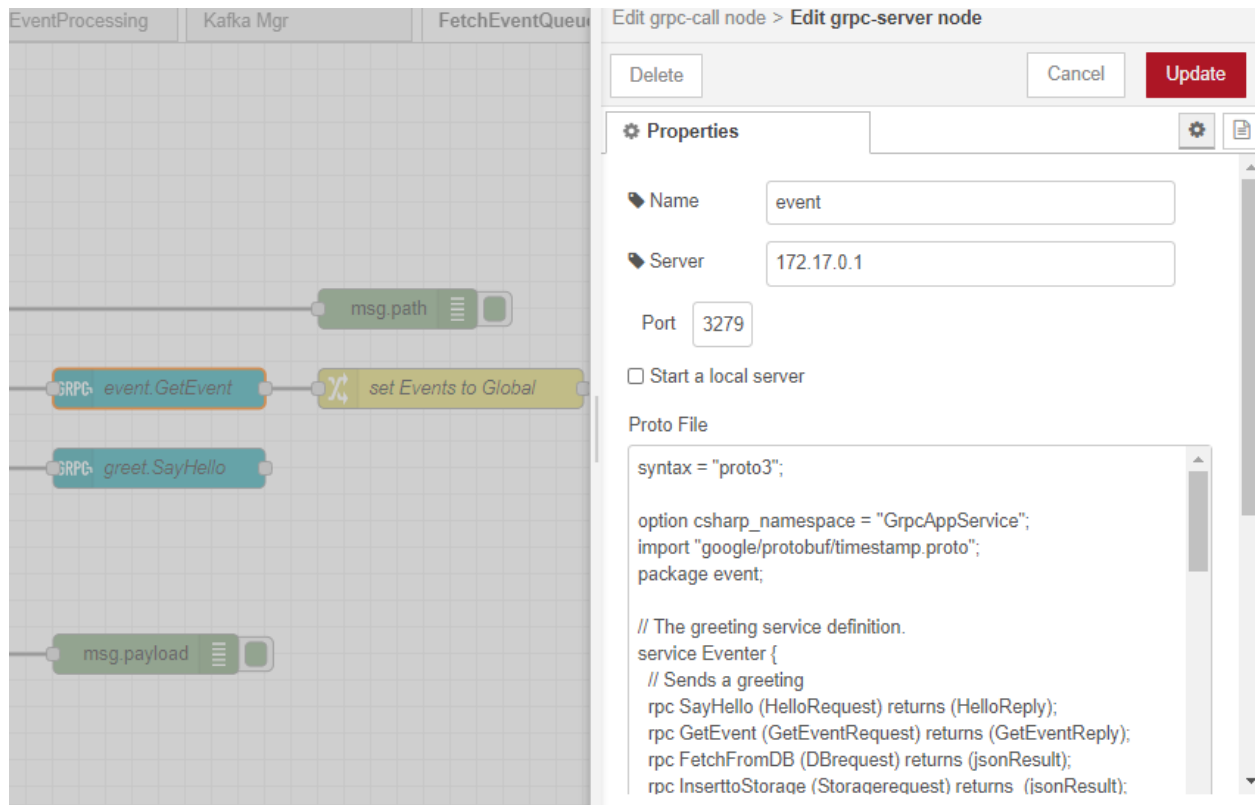


Figure 12: gRPC 連線資訊設定

## 6 Grafana

### 6.1 Introduction

Grafana 是一個開源的分析和互動式可視化平台。連接到支持的 Data Source 後，它會提供 Web 的圖表，圖形和警報。可以通過套件進行擴展。使用戶能夠簡單的建構可視化儀表板。

### 6.2 Installation

```
docker run --name <container-name> -p 3000:3000
-v <Host-etc-Volume-path>:/etc/grafana
-v <Host-data-Volume-path>:/var/lib/grafana grafana/grafana
```

### 6.3 How to use?

- 新增 Data Source
- 進入localhost:3000
- 點選Configuration>DataSource>Add data source
- 選擇DataBase(這裡選擇InfluxDB)
- 填寫連接字串並測試連線

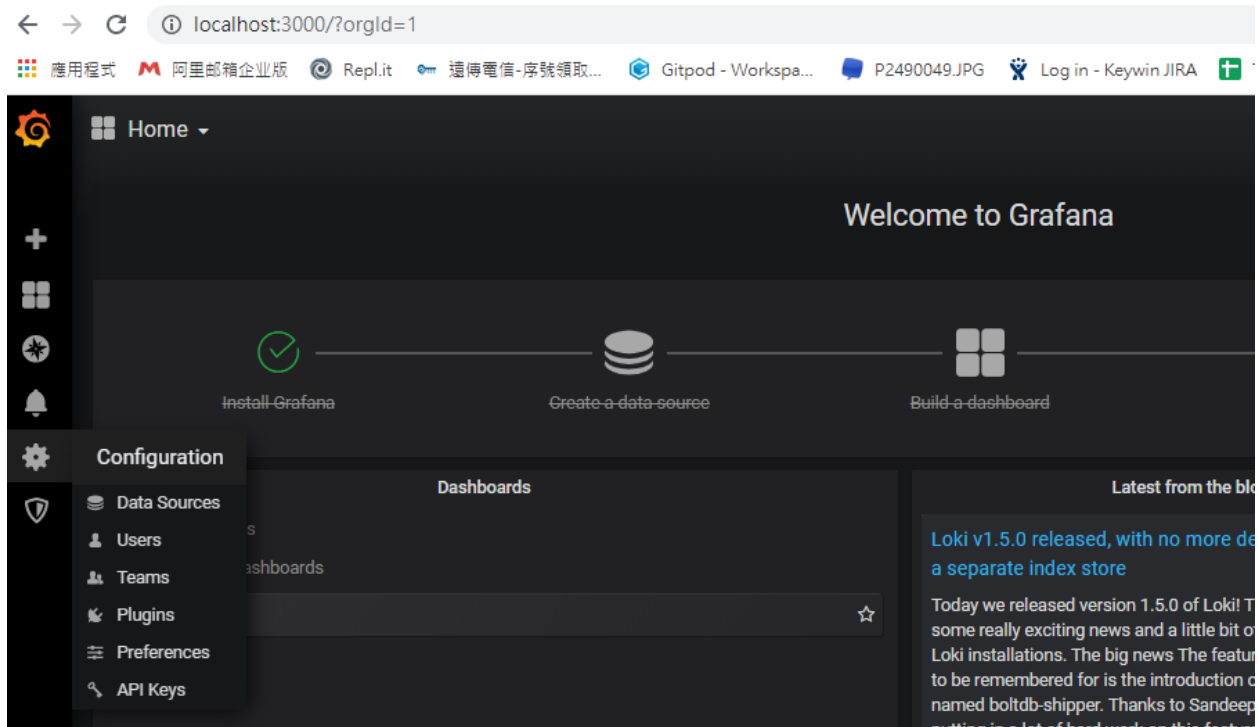


Figure 13: Grafana Create Data Source

- 新增 Dashboard
  - 點選 Create>Dashboard>Add Query

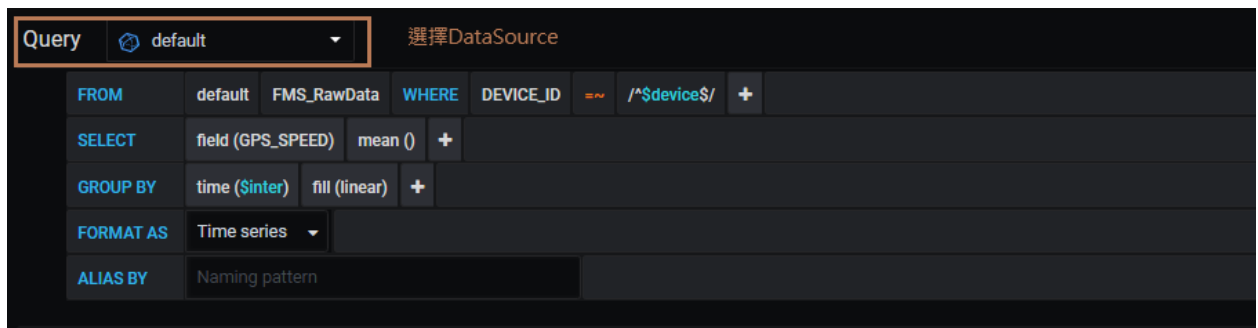


Figure 14: Grafana Add Query

- \* SELECT mean("GPS\_SPEED") FROM "FMS\_RawData" WHERE ("DEVICE\_ID" =~ /~\$device/) AND timeFilterGROUPBYtime(inter) fill(linear)
- \* 此為車速曲線圖的 Query
- \* 在 FMS\_RawData 這個 measurement 取 DEVICE\_ID 為/~\$device\$/ 資料依 \$inter 時間間隔的平均車速。
- \* /~\$device\$/ 為 Dashboard 設定 DeviceID 變數。
- \* \$inter 為 Dashboard 設定時間間隔變數。
- \* fill(linear) 表示當中間時間沒有資料時，會自動線性填充值。
- \* fill(none) 表示中間沒有資料時，不填充任何值。
- \* fill(previous) 表示中間沒有資料時，填充上一筆值。



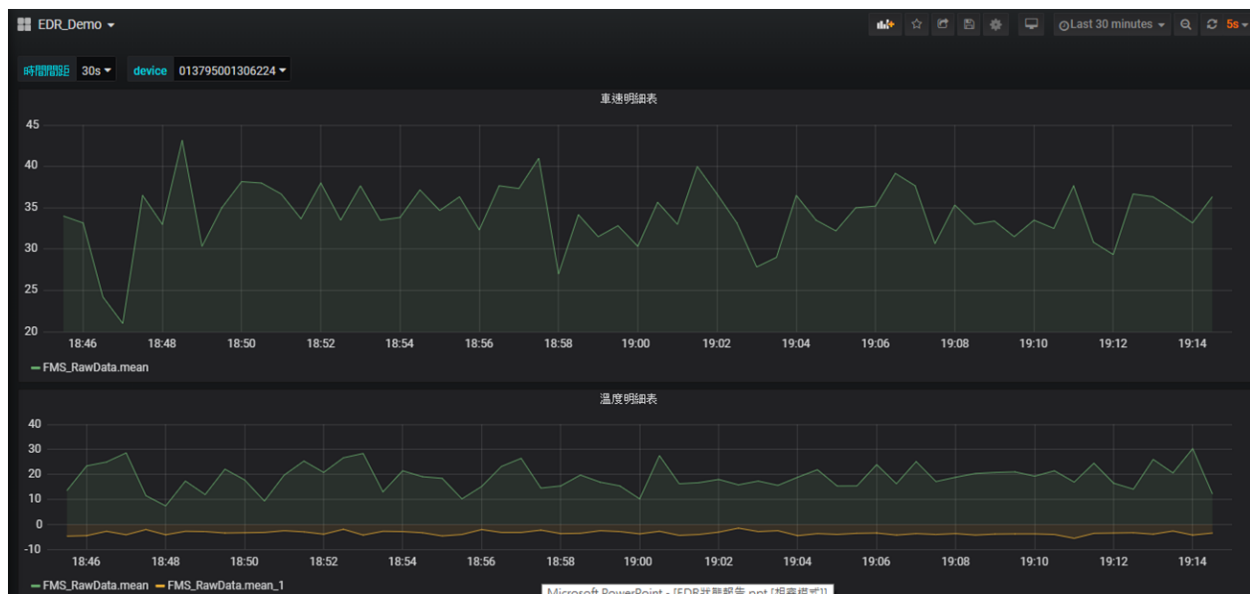


Figure 15: Grafana -車速明細表及溫度明細表

## 7 Kafka

### 7.1 Introduction

Kafka 是由 Apache 軟體基金會開發的一個開源分散式訊息處理平台，由 Scala 和 Java 編寫。該專案的目標是為處理即時資料提供一個統一、高吞吐、低延遲的平台。

### 7.2 Concepts

- Broker：Kafka 採用叢集的架構，由一至多台的機器所組成，一台 Kafka 伺服器就是一個 Broker，一個集群由多個 broker 組成，一個 broker 可以有多個 topic。broker 承擔著中間緩存和分發的作用，broker 將 producer 發送的數據分發到註冊 consumer 中。
- Topic：放訊息類別的通道 (Queue)。
- Partition：每一個 Topic 內可以切分成多個 Partition，每個 Partition 在物理上都對應一個文件夾，該文件夾存儲這個 Partition 的所有消息和索引文件。Partition 中的消息都會被分配為一個有序的 ID (offset)
- Producer：負責發布消息到 Kafka Topic 的角色。
- Consumer：負責接收/訂閱 Kafka Topic 內的訊息做處理。
- Zookeeper：每個 Broker 中的訊息同步及叢集資源配置是透過 Zookeeper 來達成。APACHE KAFKA 是一個分散式的 messaging system，裡面主要有三個角色，broker，producer，consumer。producer 是生產者，負責生產 message 並傳送至 broker(中間人、代理者)，consumer 則是消費者負責拉走 message。

### 7.3 Installation

docker-compose

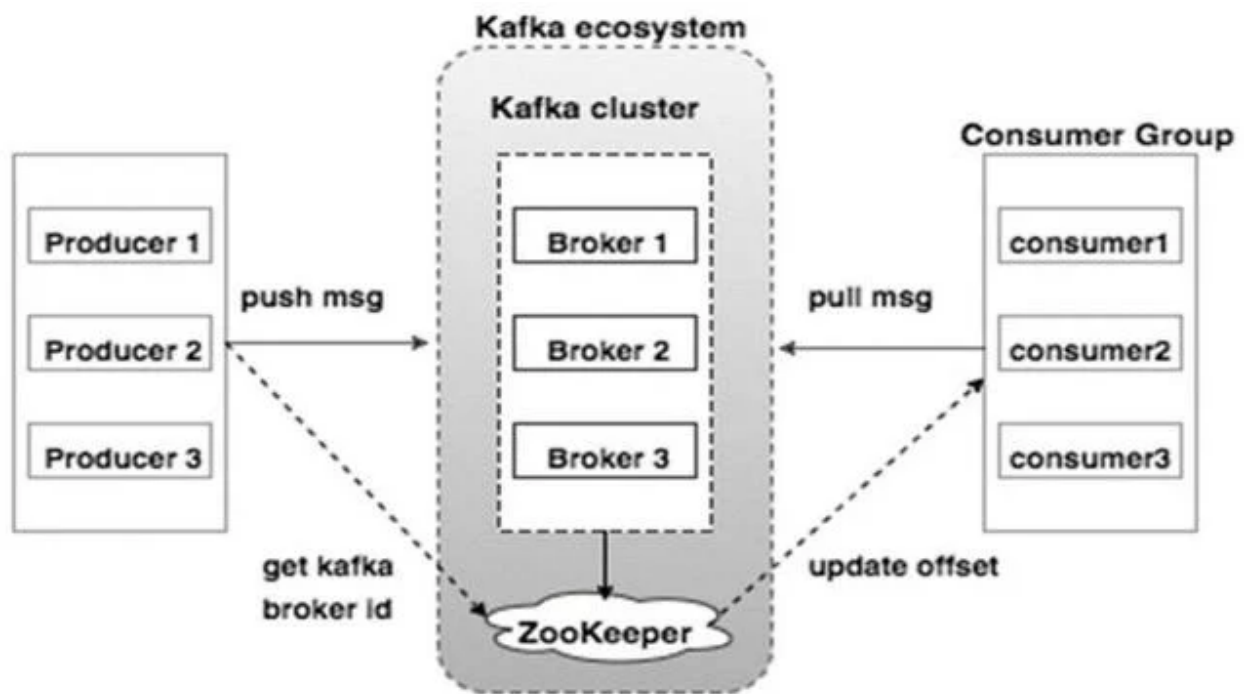


Figure 16: Kafka Architecture

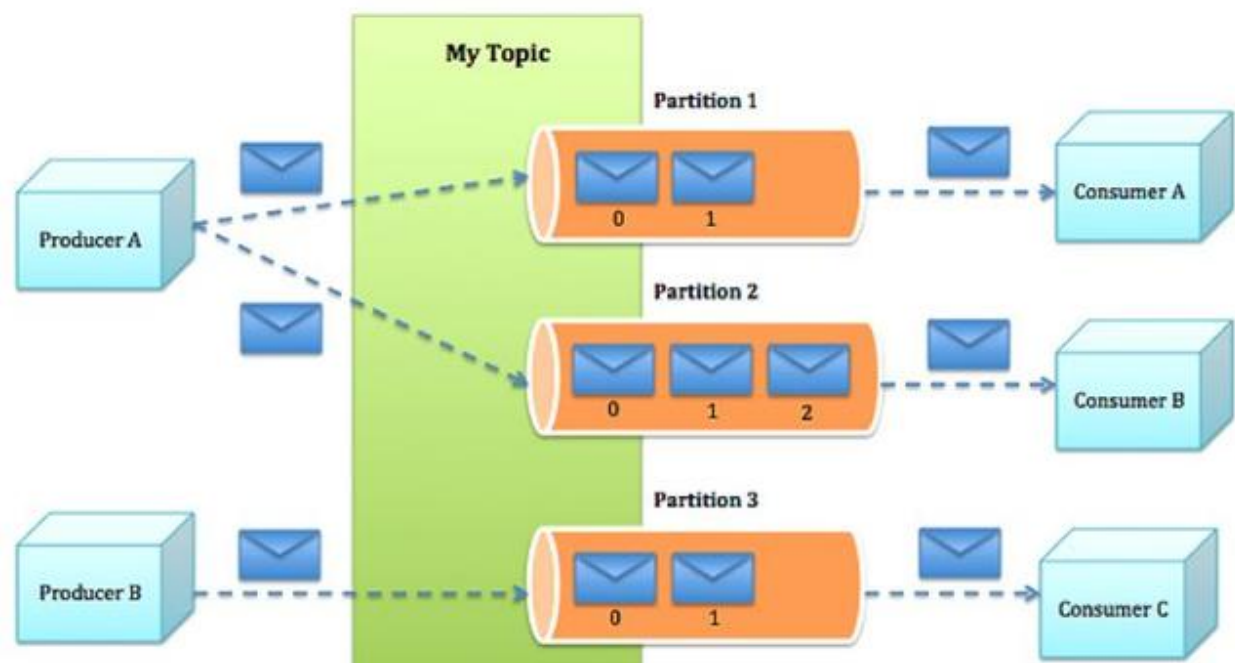


Figure 17: Kafka

```

zookeeper:
 image: wurstmeister/zookeeper
 restart: unless-stopped
 hostname: zookeeper
 ports:
 - "2181:2181"
 container_name: zookeeper
kafka1:
 image: wurstmeister/kafka
 ports:
 - "9092:9092"
 - '29094:29094'
 # 配置容器環境變數
 environment:
 # match your docker host IP (Note: Do not use localhost or 127.0.0.1 as the host ip if you want t
 KAFKA_ADVERTISED_HOST_NAME: 162.18.0.1
 KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
 #kafka broker 綁定接口
 KAFKA_LISTENERS: LISTENER_INSIDE://kafka1:29094,LISTENER_OUTSIDE://kafka1:9092
 # 當 ADVERTISED_LISTENERS 和 LISTENERS 不同時，會將 ADVERTISED_LISTENERS 註冊至 zookeeper
 # 供 client 端使用
 KAFKA_ADVERTISED_LISTENERS: LISTENER_INSIDE://kafka1:29094,LISTENER_OUTSIDE://localhost:9092
 KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: LISTENER_INSIDE:PLAINTEXT,LISTENER_OUTSIDE:PLAINTEXT
 KAFKA_INTER_BROKER_LISTENER_NAME: LISTENER_INSIDE
 KAFKA_BROKER_ID: 1
 # 決定 Topic 的副本數，實質上的意義為" 含自身 broker" 共有幾份相同的資料被分別儲存在 broker 中。
 KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 2
 #automatically create topics in Kafka during creation
 KAFKA_CREATE_TOPICS: "stream-in:1:1,stream-out:1:1"
 depends_on:
 - zookeeper
 container_name: kafka1
kafka2:
 image: wurstmeister/kafka
 ports:
 - "9091:9092"
 - '29093:29094'

 environment:
 KAFKA_ADVERTISED_HOST_NAME: localhost
 KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
 KAFKA_LISTENERS: LISTENER_INSIDE://kafka2:29094,LISTENER_OUTSIDE://kafka2:9092
 KAFKA_ADVERTISED_LISTENERS: LISTENER_INSIDE://kafka2:29094,LISTENER_OUTSIDE://localhost:9091
 KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: LISTENER_INSIDE:PLAINTEXT,LISTENER_OUTSIDE:PLAINTEXT
 KAFKA_INTER_BROKER_LISTENER_NAME: LISTENER_INSIDE
 KAFKA_BROKER_ID: 2
 KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 2
 KAFKA_CREATE_TOPICS: "stream-in:1:1,stream-out:1:1"
 depends_on:
 - zookeeper
 container_name: kafka2

```

- 當 Docker Network 中的 Container 互相通訊時，使用 listener INSIDE:kafka1:29094 連接

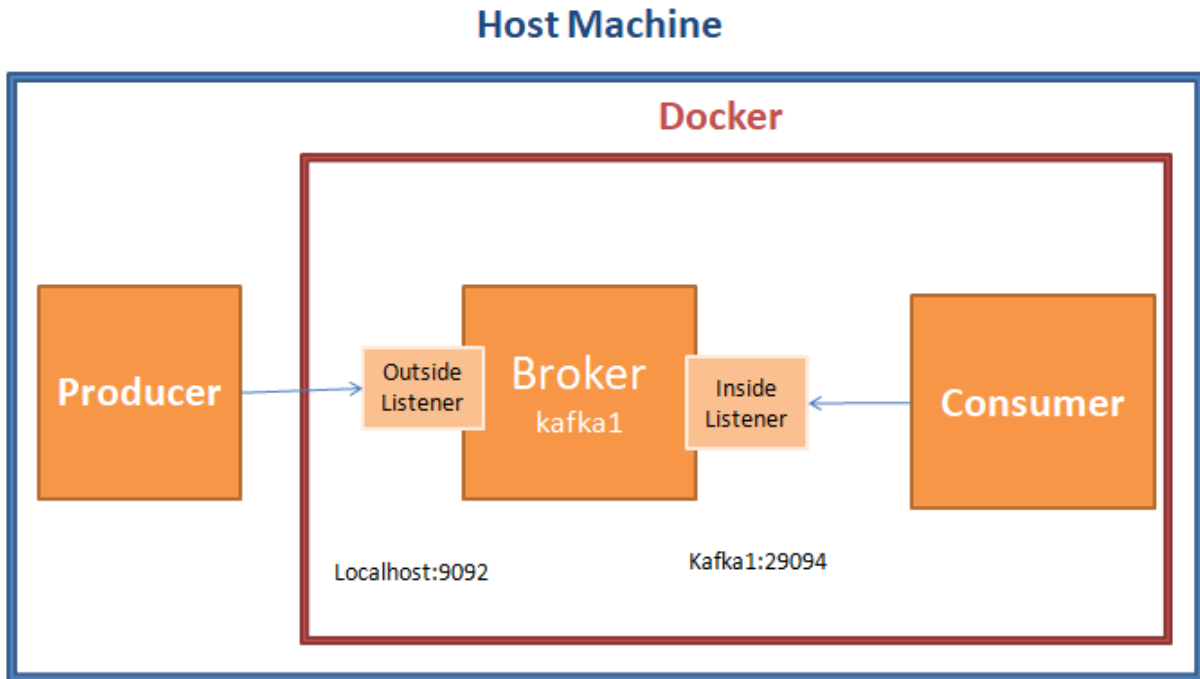


Figure 18: Kafka Listener Config

- 當 Docker Network 外部的 Client 端要與 broker 通訊時，使用 listener INSIDE:kafka1:9092 連接
- 設定 volumes/networks 和開啟所有的 containers

```
docker-compose up
```

- 停止所有的 containers 和移除 cont/vol/net

```
docker-compose up
```

## 7.4 How to use?

- 查看 topic 列表

```
/opt/kafka_2.12-2.4.1/bin/kafka-topics.sh --list
--zookeeper <zookeeper-host&port>
```

- 使用 cmd 當 producer

```
/opt/kafka_2.12-2.4.1/bin/kafka-console-producer.sh
--broker-list <broker-host&port>
--topic <topic-name>
```

- 使用 cmd 當 consumer

```
/opt/kafka_2.12-2.4.1/bin/kafka-console-consumer.sh
--broker-list <broker-host:port>
--topic <topic-name>
```

## 7.5 In Node-Red

- Kafka Consumer Node 設定 kafka Broker 連接資訊及 Topic

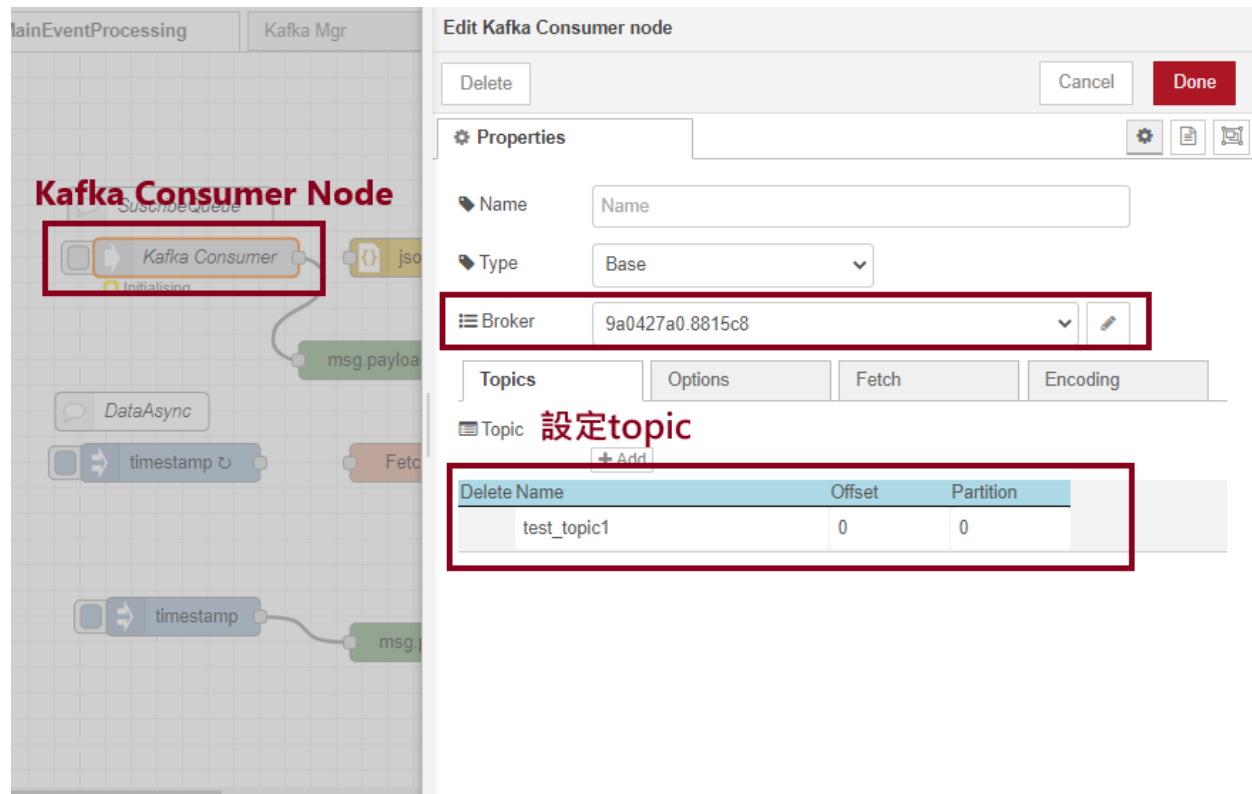


Figure 19: Kafka Consumer Node

## 8 VoltDB

### 8.1 Introduction

VoltDB 是一個符合 ACID（原子性、一致性、隔離和持久性）的分散式記憶體資料庫系統，將資料儲存在記憶體 (RAM) 中，減少進入磁盤存取數據，提升運算性能。在使用 SQL 的條件下，達到可伸縮性，可靠性和高可用性。在本是 RuleEngine 中，主要負責貯存 SQL 類型的基本資料，減少呼叫 SQL 資料庫的次數。

- \* Atomicity (原子性)：一個 transaction 中的所有操作，或者全部完成，或者全部不完成，不會結束在中間某個環節。事務在執行過程中發生錯誤，會被 Rollback 到事務開始前的狀態。
- \* Consistency (一致性)：在事務開始之前和事務結束以後，資料庫的完整性沒有被破壞。這表示寫入的資料必須完全符合所有的預設約束、觸發器、級聯回滾等。
- \* Isolation (隔離性)：資料庫允許多個並發事務同時對其數據進行讀寫和修改的能力，隔離性可以防止多個事務並發執行時由於交叉執行而導致數據的不一致。
- \* Durability (持久性)：事務處理結束後，對數據的修改就是永久的，即便系統故障也不會丟失。

## 8.2 Concepts

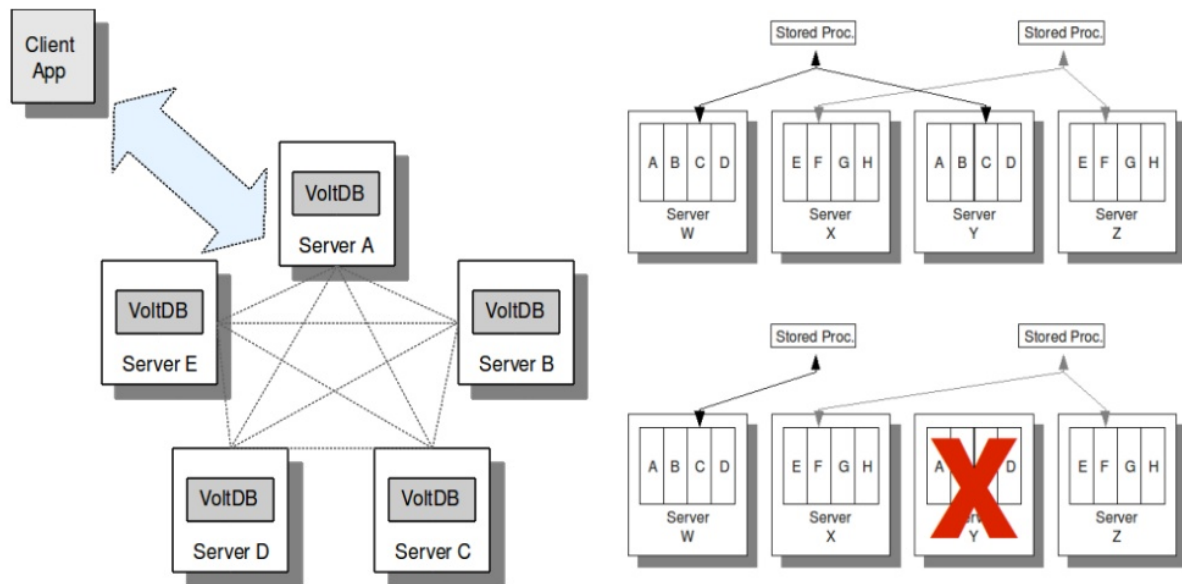


Figure 20: VoltDB Architecture-分散式儲存

K-safety : VoltDB 採分散式架構，VoltDB 會自動地複製資料庫分區，K 表示副本的個數。例如  $K=0$  時表示沒有副本，所以任何一個節點的故障都會導致整個資料庫 Cluster 的停止服務。當  $K=1$  時表示有 1 個副本，即一共 2 份拷貝，如上圖所示。 $k=?$  即為成 K-safety 的值。要注意的是：VoltDB 中的副本是可以讀寫的。

## 8.3 Installation

- Create User defined network for the VoltDBCluster  

```
{bash eval=FALSE}docker network create -d bridge voltLocalCluster
```
- Start N nodes

```
docker run -d -P -e HOST_COUNT=<host_count>
-e HOSTS=<Comma separated list of node's Hostnames/IP>
--name=<node's-Hostnames>
-v <host-volume-path>:/tmp
--network=<user-defined-voltdb-cluster-network>
voltdb/voltdb-community:6.6
```

Example :

```
docker run -d -P -e HOST_COUNT=3
-e HOSTS=node1,node2,node3 --name=node1
-v C:\Users\keywinRD0010\VoltDB-Data\data:/tmp
--network=voltLocalCluster voltdb/voltdb-community:6.6
```

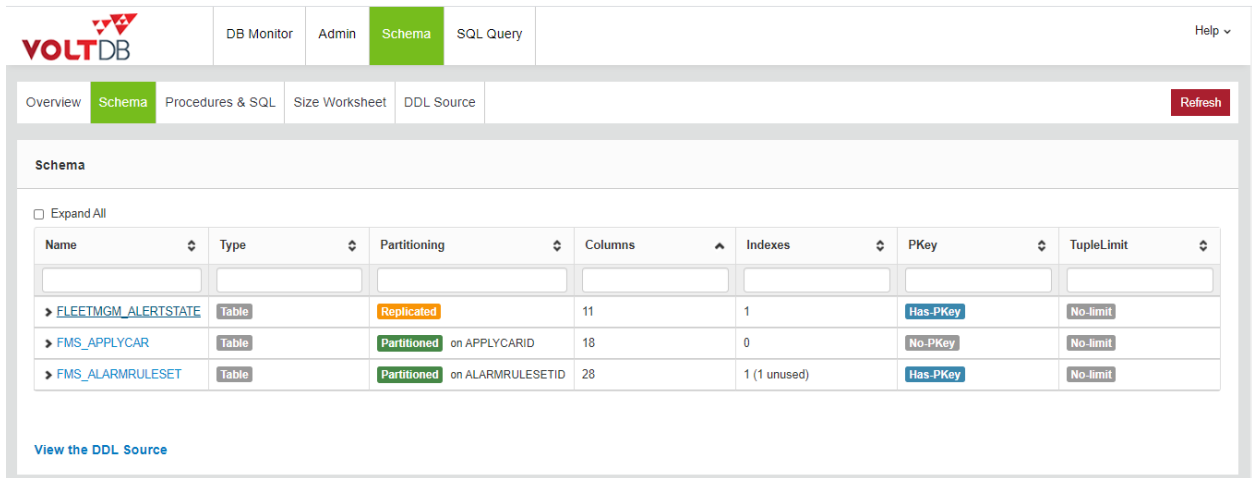
- Execute Data Definition Language

```
docker exec -it node1 sh -c "echo 'file /tmp/ddl.sql;' | sqlcmd --port=21212"
```

- custom deployment  
在 Host-volume-path 中可以加入 customeDeployment.xml 來設定自定義部屬，也可在這放入 ddl.sql 方便執行資料建構。

## 8.4 How to use?

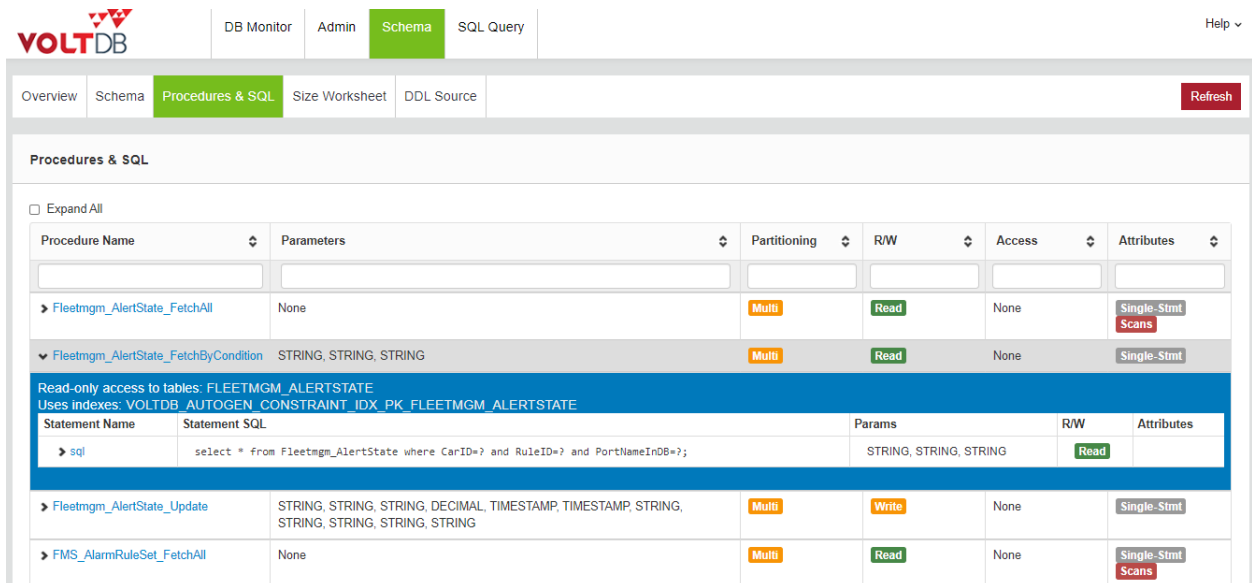
- Admin GUI 操作
  - 進入 localhost:</8080 tcp in container>



The screenshot shows the VOLTDB Admin GUI with the 'Schema' tab selected. The 'Overview' sub-tab is active, displaying a table of database schemas. The table has columns for Name, Type, Partitioning, Columns, Indexes, PKey, and TupleLimit. Three schemas are listed: FLEETMGM\_ALERTSTATE (Replicated), FMS\_APPLYCAR (Partitioned), and FMS\_ALARMRULESET (Partitioned). A 'View the DDL Source' link is at the bottom left.

Name	Type	Partitioning	Columns	Indexes	PKey	TupleLimit
FLEETMGM_ALERTSTATE	Table	Replicated	11	1	Has-PKey	No-limit
FMS_APPLYCAR	Table	Partitioned on APPLYCARID	18	0	No-PKey	No-limit
FMS_ALARMRULESET	Table	Partitioned on ALARMRULESETID	28	1 (1 unused)	Has-PKey	No-limit

Figure 21: 檢視 Data Schema



The screenshot shows the VOLTDB Admin GUI with the 'Procedures & SQL' tab selected. The 'Overview' sub-tab is active, displaying a table of database procedures. The table has columns for Procedure Name, Parameters, Partitioning, R/W, Access, and Attributes. Three procedures are listed: Fleetmgm\_AlertState\_FetchAll, Fleetmgm\_AlertState\_FetchByCondition, and Fleetmgm\_AlertState\_Update. A detailed view for Fleetmgm\_AlertState\_FetchByCondition is expanded, showing its SQL statement and parameters.

Procedure Name	Parameters	Partitioning	R/W	Access	Attributes
Fleetmgm_AlertState_FetchAll	None	Multi	Read	None	Single-Stmt Scans
Fleetmgm_AlertState_FetchByCondition	STRING, STRING, STRING	Multi	Read	None	Single-Stmt
Fleetmgm_AlertState_Update	STRING, STRING, STRING, DECIMAL, TIMESTAMP, TIMESTAMP, STRING, STRING, STRING, STRING, STRING	Multi	Write	None	Single-Stmt
FMS_AlarmRuleSet_FetchAll	None	Multi	Read	None	Single-Stmt Scans

Read-only access to tables: FLEETMGM\_ALERTSTATE  
Uses indexes: VOLTDB\_AUTOGEN\_CONSTRAINT\_IDX\_PK\_FLEETMGM\_ALERTSTATE

Statement Name	Statement SQL	Params	R/W	Attributes
sql	select * from Fleetmgm_AlertState where CarID=? and RuleID=? and PortNameInDB=?;	STRING, STRING, STRING	Read	

Figure 22: 檢視 stored Procedure

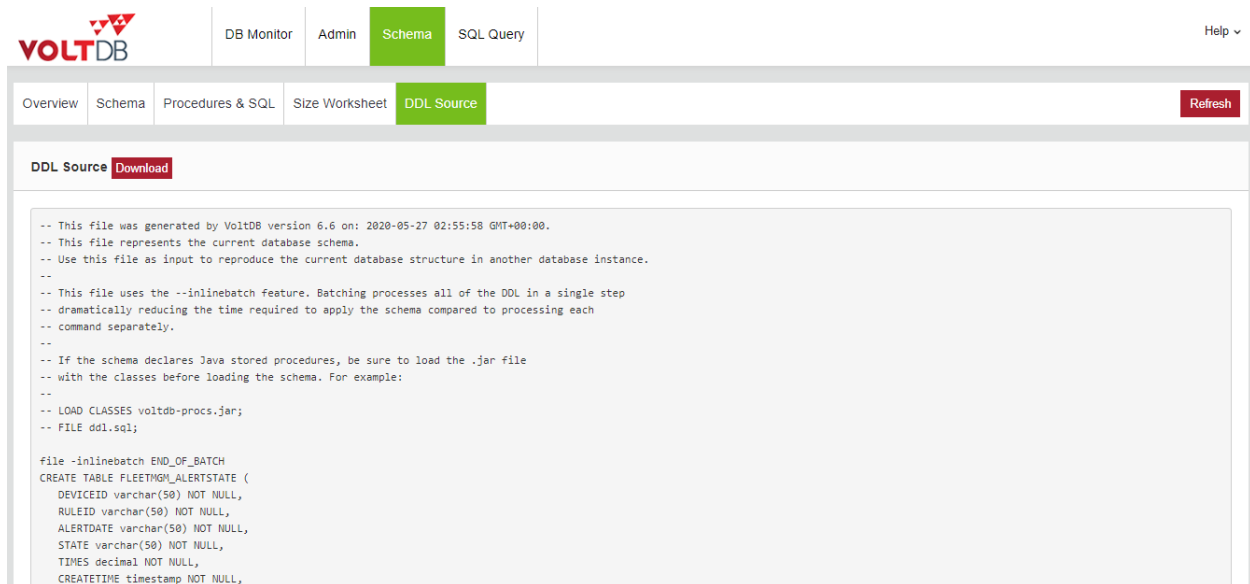


Figure 23: 檢視 ddl 檔

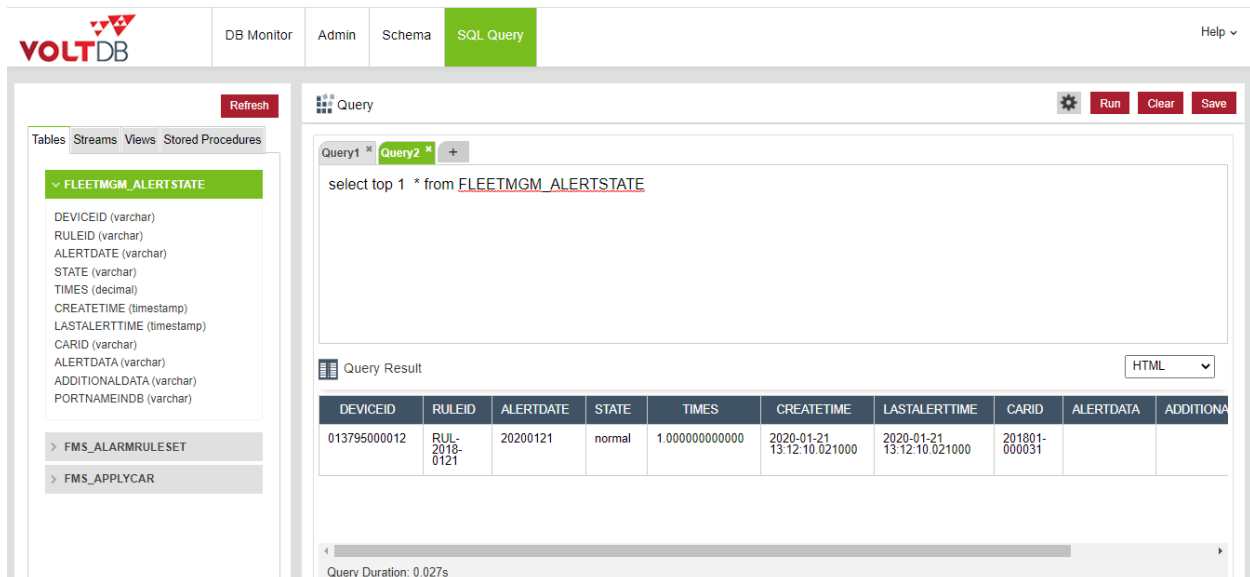


Figure 24: 進行 SQL Query



## 9 ContainerMgr

### 9.1 Docker Engine REST API

為了讓我們平台能夠設定並自動產生相對應的 Container，我們使用 Docker-Engine REST API，讓網頁能操作 Docker Engine。

Docker Engine REST API Document

#### 9.1.1 Create Container

- URL  
`{bash eval=FALSE} POST /containers/create`
- PARAMETERS

參數	類型	定義
name	string	Assign the specified name to the container. Must match <code>/?[a-zA-Z0-9_]+</code> .

- REQUEST BODY

參數	類型	定義
Hostname	string	The hostname to use for the container.
Domainname	string	The domain name to use for the container.
User	string	The user that commands are run as inside the container.
<b>ExposedPorts</b>	ExposedPorts	An object mapping ports to an empty object in the form: <code>{"&lt;port&gt;/&lt;tcp udp&gt;": {}}</code>
<i>-Additional Properties</i>	object	Default: <code>{}</code>
Image	string	The name of the image to use when creating the container
<b>Volumes</b>	volumes	An object mapping mount point paths inside the container to empty objects.
<i>-Additional Properties</i>	object	Default: <code>{}</code>
WorkingDir	string	The working directory for commands to run in.
<b>HostConfig</b>	HostConfig	Container configuration that depends on the host we are running on
<i>-NetworkMode</i>	string	Network mode to use for this container. Supported standard values are: bridge, host, none, and container:<name>
<b>NetworkingConfig</b>	NetworkingConfig	This container's networking configuration.
<i>-EndpointsConfig</i>	EndpointsConfig	A mapping of network name to endpoint configuration for that network.
<i>-Additional Properties</i>	EndpointSettings	Configuration for a network endpoint.
<i>—IPAMConfig</i>	IPAMConfig	IPAM configurations for the endpoint
<i>—Links</i>	Array of string	
<i>—Aliases</i>	Array of string	
<i>—NetworkID</i>	string	
<i>—EndpointID</i>	string	
<i>—Gateway</i>	string	

參數	類型	定義
— <i>IPAddress</i>	string	
— <i>IPPrefixLen</i>	string	
— <i>IPv6Gateway</i>	string	
— <i>GlobalIPv6Address</i>	string	
— <i>GlobalIPv6PrefixLen</i>	int64	
— <i>MacAddress</i>	string	
<b>Mounts</b>	Array of Mount	
<i>Array</i>	Array	
- <i>Target</i>	string	Container path
- <i>Source</i>	string	Mount source (e.g. a volume name, a host path).
- <i>Type</i>	string	“bind”“volume”“tmpfs”
- <i>ReadOnly</i>	boolean	Whether the mount should be read-only.
- <i>Consistency</i>	string	The consistency requirement for the mount: default, consistent, cached, or delegated.
- <b>BindOptions</b>	BindOptions	Optional configuration for the bind type
- <i>Propagation</i>		“private”“rprivate”“shared”“rshared”“slave”“rslave”
- <b>VolumeOptions</b>	VolumeOptions	Optional configuration for the volume type.
- <i>NoCopy</i>	boolean	Default:false.Populate volume with data from the target.
- <b>Labels</b>	Labels	User-defined key/value metadata.
— <i>Additional Properties</i>	string	
- <b>DriverConfig</b>	Driverconfig	Map of driver specific options
— <i>Name</i>	string	Name of the driver to use to create the volume.
— <i>Options</i>	Options	key/value map of driver specific options.
- <b>TmpfsOptions</b>	TmpfsOptions	Optional configuration for the tmpfs type.
- <i>SizeBytes</i>	int64	The size for the tmpfs mount in bytes.
- <i>Mode</i>	int	The permission mode for the tmpfs mount in an integer.

- RESPONSES

### 9.1.2 List Container

- URL

```
{bash eval=FALSE} GET /containers/json
```

- PARAMETERS

參數	類型	定義
all	boolean	Default : false. Return all containers. By default, only running containers are shown.
limit	int	Return this number of most recently created containers, including non-running ones.
size	boolean	Default : false.Return the size of container as fields SizeRw and SizeRootFs.

- RESPONSES

- 200 no error

Array		
參數	類型	定義
Id	string	The ID of this container
Names	Array of string	The names that this container has been given
Image	string	The name of the image used when creating this container
ImageID	string	The ID of the image that this container was created from
Command	string	Command to run when starting the container
Created	int64	When the container was created
<b>Ports</b>	Array of Port	The ports exposed by this container
<i>Array of Port</i>		
-IP	string	
-PrivatePort	uint16	Port on th container
-PublicPort	uint16	Port exposed on the host
-Type	string	“tcp”“udp”
SizeRw	int64	The size of files that have been created or changed by this container
SizeRootFs	int64	The total size of all the files in this container
<b>Labels</b>	Labels	User-defined key/value metadata.
-Additional Properties	string	
State	string	The state of this container (e.g. Exited)
Status	string	Additional human-readable status of this container (e.g. Exit 0)
<b>HostConfig</b>	HostConfig	
-NetworkMode	string	
<b>NetworkSettings</b>	NetworkSettings	A summary of the container’s network settings
-Networks	Network	
-Additional Properties	EndpointSettings	Configuration for a network endpoint.
—IPAMConfig	IPAMConfig	IPAM configurations for the endpoint
—Links	Array of string	
—Aliases	Array of string	
—NetworkID	string	
—EndpointID	string	
—Gateway	string	
—IPAddress	string	
—IPPrefixLen	string	
—IPv6Gateway	string	
—GlobalIPv6Address	string	
—GlobalIPv6PrefixLen	int64	
—MacAddress	string	
<b>Mounts</b>	Array of Mount	
<i>Array</i>	Array	
-Target	string	Container path
-Source	string	Mount source (e.g. a volume name, a host path).
-Type	string	“bind”“volume”“tmpfs”
-ReadOnly	boolean	Whether the mount should be read-only.
-Consistency	string	The consistency requirement for the mount: default, consistent, cached, or delegated.
-BindOptions	BindOptions	Optional configuration for the bind type
-Propagation		“private”“rprivate”“shared”“rshared”“slave”“rslave”
-VolumeOptions	VolumeOptions	Optional configuration for the volume type.

Array		
<i>-NoCopy</i>	boolean	Default:false.Populate volume with data from the target.
<i>-Labels</i>	Labels	User-defined key/value metadata.
<i>-Additional Properties</i>	string	
<i>-DriverConfig</i>	Driverconfig	Map of driver specific options
<i>-Name</i>	string	Name of the driver to use to create the volume.
<i>-Options</i>	Options	key/value map of driver specific options.
<i>-TmpfsOptions</i>	TmpfsOptions	Optional configuration for the tmpfs type.
<i>-SizeBytes</i>	int64	The size for the tmpfs mount in bytes.
<i>-Mode</i>	int	The permission mode for the tmpfs mount in an integer.

- 400 bad parameter

參數	類型	定義
message	string	The error message.

- 500 server error

參數	類型	定義
message	string	The error message.

### 9.1.3 Start Container

{bash eval=FALSE} POST /containers/[id]/start

- PATH PARAMETERS

參數	類型	定義
id	string	ID or name of the container.

- QUERY PARAMETERS

參數	類型	定義
detachKeys	string	Override the key sequence for detaching a container. Format is a single character [a-Z] or <b>ctrl-<i>&lt;value&gt;</i></b> where <i>&lt;value&gt;</i> is one of: a-z, @, ^, [, , or _.

### 9.1.4 Stop Container

{bash eval=FALSE} POST /containers/[id]/start

- PATH PARAMETERS

參數	類型	定義
id	string	ID or name of the container.

- QUERY PARAMETERS

參數	類型	定義
t	int	Number of seconds to wait before killing the container , or __..

## 9.2 NodeRed Admin API

### 9.2.1 Get the active flow configuration

- URL

GET /flows

- RESPONSE

```
{
 //an optional revision identifier for the flows
 "rev": "abc-123",
 "flows": [
 {
 "type": "tab",
 "id": "396c2376.c693dc",
 "label": "Sheet 1"
 }
]
}
```

- 參數定義

欄位	定義
id	The unique id of the flow.
label	A label for the the flow

### 9.2.2 Set the active flow configuration

- URL

POST /flows

- REQUEST BODY

```
{
 "rev": "abc-123",
 "flows": [
```

```
{
 {
 "type": "tab",
 "id": "396c2376.c693dc",
 "label": "Sheet 1"
 }
}
```

- 參數定義

欄位	定義
id	The unique id of the flow.
label	A label for the the flow

### 9.2.3 Get a list of the installed nodes

- URL

GET /nodes

- HEADER

Accept:application/json

- RESPONSE

```
[
 {
 "id": "node-red/sentiment",
 "name": "sentiment",
 "types": [
 "sentiment"
],
 "enabled": true,
 "module": "node-red"
 "version": "0.10.6"
 }
]
```

- 參數定義

欄位	定義
id	The ID of the set - module/name
name	The name of the set - as defined in the module's package.json
types	A string array of the node types provided by this set
enabled	Whether this set is currently enabled
module	The name of the module providing the set. A value of node-red indicates the node was loaded from copied in files, rather than an npm module.

### 9.2.4 Install a new node module

- URL

POST /nodes

- REQUEST BODY

```
{
 "module": "node-red-node-suncalc"
}
```

- RESPONSE

```
{
 "name": "node-red-node-suncalc",
 "version": "0.0.6",
 "nodes": [
 {
 "id": "node-red-node-suncalc/suncalc",
 "name": "suncalc",
 "types": [
 "sunrise"
],
 "enabled": true,
 "loaded": true,
 "module": "node-red-node-suncalc"
 }
]
}
```

- 參數定義

欄位	定義
id	The ID of the set - module/name
name	The name of the set - as defined in the module's package.json
types	A string array of the node types provided by this set
enabled	Whether this set is currently enabled
module	The name of the module providing the set. A value of node-red indicates the node was loaded from copied in files, rather than an npm module.