GraphQL With Hot Chocolate

本文件是以.Net Core 6.0為基礎建置，所以開始前確認已經安裝.Net Core 6.0的SDK

環境設定

建立專案

使用dotnet指令新建專案，建立專案之後，會在該目錄下建立TestGQL的專案目錄

```
dotnet new web -n TestGQL
```

安裝相關資源庫

最後安裝完成的結果如下

```
  <ItemGroup>
    <PackageReference Include="GraphQL.Server.UI.Voyager" Version="6.1.1" />
    <PackageReference Include="HotChocolate.AspNetCore" Version="11.3.8" />
    <PackageReference Include="HotChocolate.Data.Entityframework" Version="11.3.8" />
    <PackageReference Include="Microsoft.EntityframeworkCore.Design" Version="6.0.7">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityframeworkCore.SqlServer" Version="6.0.7" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0" />
  </ItemGroup>
```

安裝 HotChocolate.AspNetCore

注意：只能安裝11.x.x的版本，不然新版的程式在Resolver的處理會發生異常，這裡我們安裝的是11版的最新版本11.3.8

```
dotnet add package HotChocolate.AspNetCore  --version 11.3.8
```

安裝 HotChocolate.Data.Entityframework

注意：只能安裝11.x.x的版本，不然新版的程式在Resolver的處理會發生異常，這裡我們安裝的是11版的最新版本11.3.8

```
dotnet add package HotChocolate.Data.Entityframework --version 11.3.8
```

PS. 之前有提到Hot chocolate只能使用11.x.x版本，這部分在Resolver修改增加[Parent]的宣告之後，可以直接使用最新版，相關升級說明可以參看以下連結(2022/08/08補充)

https://chillicream.com/docs/hotchocolate/api-reference/migrate-from-11-to-12

接下來安裝的資源庫部分，沒有懸念一律安裝最新的版本

安裝 Microsoft.EntityframeworkCore.Design

```
dotnet add package Microsoft.EntityframeworkCore.Design
```

安裝 Microsoft.EntityframeworkCore.SqlServer

```
dotnet add package Microsoft.EntityframeworkCore.SqlServer
```

安裝 GraphQL.Server.UI.Voyager

```
dotnet add package GraphQL.Server.UI.Voyager
```

設定路由

開啟Program.cs增加以下兩個區塊的設定(.Net Core 6.0已經移除Startup.cs，合併到Program.cs中定義)

增加GraphQLServer的服務

```c# //增加GraphQLServer的服務 builder.Services .AddGraphQLServer();

#### 增加graphql的路由設定

```c#
//增加graphql的路由設定
app.UseRouting().UseEndpoints(endpoints =>
{
    endpoints.MapGraphQL();
});
```

完整的Program.cs設定如下：

```c# var builder = WebApplication.CreateBuilder(args);

//增加GraphQLServer的服務 builder.Services .AddGraphQLServer();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

//增加graphql的路由設定 app.UseRouting().UseEndpoints(endpoints => { endpoints.MapGraphQL(); });

app.Run();

#### 編譯並進行測試

執行以下指令進行編譯

```ba
dotnet build
```

執行以下指令執行專案

```
dotnet run
```

開啟測試網頁

網頁網址：https://localhost:7218/graphql/

應該會看到 Banana Cake Pop 類似 PostMan 的介面，如果有出現代表路由正確，GrapgQL Server有正確運作

利用Entity Framework建立資料連線操作

建立第一個Model

```c# using System.ComponentModel.DataAnnotations;

namespace TestSQL.Models { public class Customer { [Key] public int Id { get; set; }

```
    [Required]
    public string Name { get; set; }

    public string PhoneNo { get; set; }
}
```

}

### 建立DbContext

建立AppDbContext.cs，定義

```c#
using TestSQL.Models;
using Microsoft.EntityFrameworkCore;

namespace TestGQL.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions options) : base(options)
        {
        }
        public DbSet<Customer> Customers { get; set; }

    }
}
```

設定資料庫連線

開啟appsettings.Development.json檔案，增加以下設定
```

```
 "ConnectionStrings": {
    "GQLConStr":"Server=localhost,1433;Database=GQLDB;User Id=sa;Password=27559865SS"
  }
```

完整檔案如下

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "GQLConStr":"Server=localhost,1433;Database=OrderDB;User Id=sa;Password=27559865SS"
  }
}
```

開啟Program.cs增加以下設定

```c# builder.Services.AddPooledDbContextFactory(options => {
options.UseSqlServer(builder.Configuration.GetConnectionString("GQLConStr")); });
```

設定引用

AppDbContext 引用

```c#
using TestGQL.Data;
```

options.UseSqlServer 引用

```c# using Microsoft.EntityFrameworkCore;
```

完整檔案如下：

```c#
using TestGQL.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);


builder.Services
    .AddGraphQLServer();

builder.Services.AddPooledDbContextFactory<AppDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("GQLConStr"));
});

var app = builder.Build();

app.MapGet("/", () => "Hello World!");


app.UseRouting().UseEndpoints(endpoints =>
{
    endpoints.MapGraphQL();
});

app.Run();
```

Migrate Model to Database

安裝dotnet ef tool

```
dotnet tool install --global dotnet-ef
```

如果已經安裝可以用以下語法進行更新

```
dotnet tool update --global dotnet-ef
```

建立Migration file

Migration的檔案是依照Model的定義及AppDbContext中的內容，來產生相關的資料表Sql Script，可以打開20220731161040_AddCustomerToDb.cs來檢視相關語法內容。

```
dotnet ef migrations add AddCustomerToDb
```

執行完成之後，專案目錄應該會增加一個Migrations的目錄，裡面放著以下三個檔案。

```
 20220731161040_AddCustomerToDb.Designer.cs
20220731161040_AddCustomerToDb.cs
AppDbContextModelSnapshot.cs
```

移除Migrations的內容

```
dotnet ef migrations remove
```

執行Migration將Model映射到資料庫中

執行以下指令，就會依照前面所設定的資料庫連線參數，建立資料庫並且按照Migrations的內容建立資料表

```
dotnet ef database update
```

建立完成之後可以開啟SQL Server Management Studio或是Azure Data Studio來檢視相關建立的資料庫內容。

建立Query

建立Query.cs

建立GraphQL的目錄，在此目錄中建立Query.cs

```c# using HotChocolate; using HotChocolate.Data; using TestGQL.Data; using TestSQL.Models;

namespace TestGQL.GraphQL { public class Query { [UseDbContext(typeof(AppDbContext))] public IQueryable GetCustomer([ScopedService] AppDbContext context) { return context.Customers; } } }

```
Service vs ScopedService

簡單說明，Service==>Without Pool，單工無法進行多層代理查詢邏輯，ScopedService==> With Pool，可以進行多層代理

設定依賴注入Query

開啟Program.cs加入以下設定

```c#
using TestGQL.GraphQL;

builder.Services
    .AddGraphQLServer()
    .AddQueryType<Query>();
```

完整的Program.cs修改如下所示

```c# using TestGQL.Data; using TestGQL.GraphQL; using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services .AddGraphQLServer() .AddQueryType();

builder.Services.AddPooledDbContextFactory(options => {
```

options.UseSqlServer(builder.Configuration.GetConnectionString("GQLConStr")); });

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.UseRouting().UseEndpoints(endpoints => { endpoints.MapGraphQL(); });

app.Run();

```
### 建立測試資料

開啟SSMS登入GQLDB資料庫，執行以下指令建立測試資料

```sql
USE OrderDB;
Insert Into Customers(Name,PhoneNo)values('Jerry','0928456781');
Insert Into Customers(Name,PhoneNo)values('Adam','0928456782');
Insert Into Customers(Name,PhoneNo)values('Eric','0928456783');
Insert Into Customers(Name,PhoneNo)values('Hannan','0928456784');
Insert Into Customers(Name,PhoneNo)values('Steven','0928456785');
Insert Into Customers(Name,PhoneNo)values('Johnathon','0928456786');
```

測試GraphQL

執行以下指令編譯專案

```
dotnet build
```

執行以下指令執行專案

```
dotnet run
```

兩種測試方式

GraphQL查詢語法

```
Query{
  Customer{
  id,
  name
}
}
```

1. Banana Cake pop

   直接開啟以下網址：https://localhost:7283/graphql/

   輸入GraphQL查詢語法進行查詢

2. Insomnia

    1. 建立Insomnia的Requests專案
```

2. 建立GraphQL的Request
3. 輸入GraphQL的Server位址==>https://localhost:7238/graphql
4. 輸入GraphQL查詢語法進行查詢

建立第二個Model

建立第二個Order Model

相關建立語法如下

```c# using System.ComponentModel.DataAnnotations;

namespace TestSQL.Models { public class Order { [Key] public int Id { get; set; }

```
    [Required]
    public string? OrderNo { get; set; }

    public string? OrderDate { get; set; }
    public int? CustId { get; set; }
    public Customer? Customer { get; set; }
}
```

}

### 修改第一個Customer.cs

增加以下屬性定義

```c#
public ICollection<Order> Orders { get; set; } = new List<Order>();
```

完整程式如下：

```c# using System.ComponentModel.DataAnnotations;

namespace TestSQL.Models { public class Customer { [Key] public int Id { get; set; }

```
    [Required]
    public string? Name { get; set; }

    public string? PhoneNo { get; set; }

    public ICollection<Order> Orders { get; set; } = new List<Order>();

}
```

}

修改AppDbContext

相關修改語法如下：

增加訂單檔的資料

```c#
public DbSet<Order> Order { get; set; }
```

設定訂單與客戶關聯

```c# protected override void OnModelCreating(ModelBuilder modelBuilder) { modelBuilder .Entity() .HasMany(p => p.Commands) .WithOne(p => p.Platform) .HasForeignKey(p => p.PlatformId);
```

```
        modelBuilder
            .Entity<Command>()
            .HasOne(p => p.Platform)
            .WithMany(p => p.Commands)
            .HasForeignKey(p => p.PlatformId);
    }
```

建立Migration及建立相關資料庫物件

執行以下指令產生Script並且更新資料庫物件定義

```bash
dotnet ef migrations add AddOrderToDb
```

```
dotnet ef database update
```

建立Order的測試資料

```
USE OrderDB;
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456781','20220801',1);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456782','20220801',2);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456712','20220801',2);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456783','20220801',3);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456713','20220801',3);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456784','20220801',4);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456713','20220801',4);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456726','20220801',4);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456715','20220801',5);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456725','20220801',5);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456786','20220801',6);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456777','20220801',6);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456788','20220801',6);
Insert Into Orders(OrderNo,OrderDate,CustId)values('O123456789','20220801',6);
```

修改Query.cs 加入訂單查詢

加入GetOrder的方法

```c# [UseDbContext(typeof(AppDbContext))] public IQueryable GetOrder([ScopedService] AppDbContext context) { return context.Orders; }```

完整檔案修改如下：

```c#
using HotChocolate;
using HotChocolate.Data;
using TestGQL.Data;
using TestSQL.Models;

namespace TestGQL.GraphQL
{
    public class Query
    {
        [UseDbContext(typeof(AppDbContext))]
        public IQueryable<Customer> GetCustomer([ScopedService] AppDbContext context)
        {
            return context.Customers;
        }

        [UseDbContext(typeof(AppDbContext))]
        public IQueryable<Order> GetOrder([ScopedService] AppDbContext context)
        {
            return context.Orders;
        }
    }
}
```
```

測試訂單查詢

執行dotnet build & dotnet run

測試語法如下：

```
query {
    order {
        id
        orderNo
        orderDate
        custId
    }
}
```

加入GraphQL Voyager

修改Program.cs增加Voyager中間件並配置URL

```c# // 增加Voyager中間件並配置URL app.UseGraphQLVoyager(new VoyagerOptions { GraphQLEndPoint = "/graphql", }, "/graphql-voyager");
```

開啟以下網址檢視，可以看到視覺化的Schema

Https://localhost:7283/graphql-voyager

增加Model說明內容

開啟Model增加說明內容，引用部分須增加 using HotChocolate;

Customer Model

```c#
using System.ComponentModel.DataAnnotations;
using HotChocolate;

namespace TestSQL.Models
{
    [GraphQLDescription("會員基本資料檔")]
    public class Customer
    {
        [Key]
        [GraphQLDescription("會員序號")]
        public int Id { get; set; }

        [Required]
        [GraphQLDescription("會員姓名")]
        public string? Name { get; set; }

        [GraphQLDescription("會員電話")]
        public string? PhoneNo { get; set; }

        [GraphQLDescription("會員訂單資料")]
        public ICollection<Order> Orders { get; set; } = new List<Order>();

    }
}
```

Order Model

```c# using System.ComponentModel.DataAnnotations; using HotChocolate;

namespace TestSQL.Models { [GraphQLDescription("訂單資料檔")] public class Order { [Key] [GraphQLDescription("訂單序號")] public int Id { get; set; }
```

```
    [Required]
    [GraphQLDescription("訂單編號")]
    public string? OrderNo { get; set; }

    [GraphQLDescription("下訂日期")]
    public string? OrderDate { get; set; }

    [GraphQLDescription("訂單會員編號")]
    public int? CustId { get; set; }

    [GraphQLDescription("訂單會員資料")]
    public Customer? Customer { get; set; }
}
```

}

修改完成之後，可以重新開啟以下網址，確認是否有呈現相關的規格說明

https://localhost:7283/graphql-voyager

另外，同樣功能也可以套用在Query的定義上，試試看！

增加CustomerType With Resolver

GraphQL的Type有點類似Model的代理，可以增加很多自定義操作在Model的各項屬性上，例如以下的幾種操作：

1. 增加自定義說明
2. 隱藏Model屬性
3. 針對特定欄位增加Resolver的自定義處理

增加自訂義說明及隱藏Model屬性

首先，先嘗試進行定義說明及隱藏屬性的處理

在GraphQL目錄下，新增一個目錄為Customers，並新增CustomerType.cs

```c#
using HotChocolate.Types;
using TestSQL.Models;

namespace TestGQL.old.GraphQL.Customers
{
    public class CustomerType : ObjectType<Customer>
    {
        protected override void Configure(IObjectTypeDescriptor<Customer> descriptor)
        {
            descriptor.Description("客戶基本資料Type");
            descriptor.Field(p=>p.PhoneNo).Ignore();
        }
    }
}
```

在Program.cs中，增加Type的引用，因爲引入了CustomerType，所以需要增加using TestGQL.GraphQL.Customers;

```c# using TestGQL.GraphQL.Customers; builder.Services.AddType();
```

完整修改的程式碼如下所示：

```c#
using TestGQL.Data;
using TestGQL.GraphQL;
using Microsoft.EntityFrameworkCore;
using GraphQL.Server.Ui.Voyager;
using TestGQL.old.GraphQL.Customers;

var builder = WebApplication.CreateBuilder(args);


builder.Services
    .AddGraphQLServer()
    .AddQueryType<Query>()
    .AddType<CustomerType>();

builder.Services.AddPooledDbContextFactory<AppDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("GQLConStr"));
});

var app = builder.Build();

// app.MapGet("/", () => "Hello World!");


app.UseRouting().UseEndpoints(endpoints =>
{
    endpoints.MapGraphQL();
});
// 增加Voyager中間件並配置URL
app.UseGraphQLVoyager(new VoyagerOptions
{
    GraphQLEndPoint = "/graphql",
}, "/graphql-voyager");

app.Run();
```

增加Resolver

在CustomerType.cs中，增加Resolvers的Class及GetOrder的功能

```c# private class Resolvers { public IQueryable GetOrders([Parent] Customer customer, [ScopedService] AppDbContext context) { return context.Orders.Where(p => p.CustId == customer.Id); } }
```

PS. 之前有提到Hot chocolate只能使用11.x.x版本，這部分在Resolver修改增加[Parent]的宣告之後，可以直接使用最新版

https://chillicream.com/docs/hotchocolate/api-reference/migrate-from-11-to-12

在Configure的功能中增加以下的引用處理

```c#
            descriptor
                .Field(p => p.Orders)
                .ResolveWith<Resolvers>(p => p.GetOrders(default!, default!))
                .UseDbContext<AppDbContext>()
                .Description("查詢該客戶的訂單明細");
```

重新編譯執行之後，可以利用GraphQL的查詢工具執行以下查詢

```
query {
    customer {
        id
        name
        orders{
            orderNo,
            orderDate
        }
    }
}
```

試試用同樣方式進行OrderType.cs的建置

OrderType建置

如果試不出來，可以參考以下的最終結果進行實作

Orders\OrderType.cs

```c# using HotChocolate; using HotChocolate.Types; using TestGQL.Data; using TestSQL.Models;

namespace TestGQL.GraphQL.Orders { public class OrderType:ObjectType { protected override void Configure(IObjectTypeDescriptor descriptor) { descriptor.Description("訂單資料Type");
```

```
        descriptor
            .Field(p => p.Customer)
            .ResolveWith<Resolvers>(p => p.GetCustomers(default!, default!))
            .UseDbContext<AppDbContext>()
            .Description("查詢該客戶的訂單明細");
    }

    private class Resolvers
    {
        public IQueryable<Customer> GetCustomers([Parent] Order order, [ScopedService] AppDbContext co
        {
            return context.Customers.Where(p => p.Id == order.CustId);
        }
    }
}
```

}

修改Program.cs

```c#
using TestGQL.Data;
using TestGQL.GraphQL;
using Microsoft.EntityFrameworkCore;
using GraphQL.Server.Ui.Voyager;
using TestGQL.GraphQL.Customers;
using TestGQL.GraphQL.Orders;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddGraphQLServer()
    .AddQueryType<Query>()
    .AddType<CustomerType>()
    .AddType<OrderType>();

builder.Services.AddPooledDbContextFactory<AppDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("GQLConStr"));
});

var app = builder.Build();

app.UseRouting().UseEndpoints(endpoints =>
{
    endpoints.MapGraphQL();
});
// 增加Voyager中間件並配置URL
app.UseGraphQLVoyager(new VoyagerOptions
{
    GraphQLEndPoint = "/graphql",
}, "/graphql-voyager");

app.Run();
```

重新編譯之後執行以下的查詢

```
query {
    order {
        id
        orderNo
        orderDate
        customer{
            name
        }
    }
}
```

篩選及排序

增加屬性的設定之後，就可以提供前端進行排序及篩選的功能

排序

開啟查詢Query.cs，增加排序的定義

```c# [UseSorting]
```

完整程式如下

```c#
using HotChocolate;
using HotChocolate.Data;
using TestGQL.Data;
using TestSQL.Models;

namespace TestGQL.GraphQL
{
    [GraphQLDescription("查詢API功能")]
    public class Query
    {
        [GraphQLDescription("會員資料查詢")]
        [UseDbContext(typeof(AppDbContext))]
        [UseSorting]
        public IQueryable<Customer> GetCustomer([ScopedService] AppDbContext context)
        {
            return context.Customers;
        }

        [GraphQLDescription("訂單資料查詢")]
        [UseDbContext(typeof(AppDbContext))]
        [UseSorting]
        public IQueryable<Order> GetOrder([ScopedService] AppDbContext context)
        {
            return context.Orders;
        }
    }
}
```

在Program.cs中，增加排序的宣告

```c# builder.Services.AddSorting();
```

完整程式如下

```c#
using TestGQL.Data;
using TestGQL.GraphQL;
using Microsoft.EntityFrameworkCore;
using GraphQL.Server.Ui.Voyager;
using TestGQL.GraphQL.Customers;
using TestGQL.GraphQL.Orders;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddGraphQLServer()
    .AddQueryType<Query>()
    .AddType<CustomerType>()
    .AddType<OrderType>()
    .AddSorting();

builder.Services.AddPooledDbContextFactory<AppDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("GQLConStr"));
});

var app = builder.Build();

app.UseRouting().UseEndpoints(endpoints =>
{
    endpoints.MapGraphQL();
});
// 增加Voyager中間件並配置URL
app.UseGraphQLVoyager(new VoyagerOptions
{
    GraphQLEndPoint = "/graphql",
}, "/graphql-voyager");

app.Run();
```

以下面的語法進行測試

```
query {
    customer (order:{name:DESC}){
        id
        name
    }
}
```

篩選

開啟查詢Query.cs，增加篩選的定義

```c# [UseFiltering]
```

完整程式如下

```c#
using HotChocolate;
using HotChocolate.Data;
using TestGQL.Data;
using TestSQL.Models;

namespace TestGQL.GraphQL
{
    [GraphQLDescription("查詢API功能")]
    public class Query
    {
        [GraphQLDescription("會員資料查詢")]
        [UseDbContext(typeof(AppDbContext))]
        [UseSorting]
        [UseFiltering]
        public IQueryable<Customer> GetCustomer([ScopedService] AppDbContext context)
        {
            return context.Customers;
        }

        [GraphQLDescription("訂單資料查詢")]
        [UseDbContext(typeof(AppDbContext))]
        [UseSorting]
        [UseFiltering]
        public IQueryable<Order> GetOrder([ScopedService] AppDbContext context)
        {
            return context.Orders;
        }
    }
}
```

在Program.cs中，增加篩選的宣告

```c# builder.Services.AddFiltering();
```

完整程式如下

```c#
using TestGQL.Data;
using TestGQL.GraphQL;
using Microsoft.EntityFrameworkCore;
using GraphQL.Server.Ui.Voyager;
using TestGQL.GraphQL.Customers;
using TestGQL.GraphQL.Orders;

var builder = WebApplication.CreateBuilder(args);



builder.Services
    .AddGraphQLServer()
    .AddQueryType<Query>()
    .AddType<CustomerType>()
    .AddType<OrderType>()
    .AddSorting()
    .AddFiltering();

builder.Services.AddPooledDbContextFactory<AppDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("GQLConStr"));
});

var app = builder.Build();

// app.MapGet("/", () => "Hello World!");



app.UseRouting().UseEndpoints(endpoints =>
{
    endpoints.MapGraphQL();
});
// 增加Voyager中間件並配置URL
app.UseGraphQLVoyager(new VoyagerOptions
{
    GraphQLEndPoint = "/graphql",
}, "/graphql-voyager");

app.Run();
```

以下面的語法進行測試

```
query {
    order(where: { custId: { eq: 1 } }) {
        id
        orderNo
        orderDate
        customer
        {
            name
        }
    }
}
```

資料維護 Mutation

替GraphQL增加資料維護的功能

增加AddCustomerInput.cs

在GraphQL\Customer目錄下，增加AddCustomerInput.cs，接收前端傳入的參數

```c# namespace TestGQL.GraphQL.Customers { public record AddCustomerInput(string name) {

```
}
```

}

```
在GraphQL\Customer目錄下，增加AddCustomerPayload.cs，更新完成後回傳結果

```c#
using TestSQL.Models;

namespace TestGQL.GraphQL.Customers
{
    public record AddCustomerPayload(Customer customer);
}
```

在GraphQL目錄下，增加Mutation.cs檔案，並且增加AddCustomerAsync的方法進行資料更新

```c# using HotChocolate; using HotChocolate.Data; using TestGQL.Data; using TestGQL.GraphQL.Customers; using TestSQL.Models;

namespace TestGQL.GraphQL { public class Mutation { [UseDbContext(typeof(AppDbContext))] public async Task AddCustomerAsync( AddCustomerInput input, [ScopedService] AppDbContext context ) { var customer = new Customer { Name = input.name }; context.Customers.Add(customer); await context.SaveChangesAsync(); return new AddCustomerPayload(customer); } } }

```
修改Program.cs內容，增加Mutation的宣告

```c#
builder.Services.AddMutationType<Mutation>();
```

完整程式如下

```c# using TestGQL.Data; using TestGQL.GraphQL; using Microsoft.EntityFrameworkCore; using GraphQL.Server.Ui.Voyager; using TestGQL.GraphQL.Customers; using TestGQL.GraphQL.Orders;

var builder = WebApplication.CreateBuilder(args);

builder.Services .AddGraphQLServer() .AddQueryType() .AddType() .AddMutationType() .AddType() .AddSorting() .AddFiltering();

builder.Services.AddPooledDbContextFactory(options => {
options.UseSqlServer(builder.Configuration.GetConnectionString("GQLConStr")); });

var app = builder.Build();

app.UseRouting().UseEndpoints(endpoints => { endpoints.MapGraphQL(); }); // 增加Voyager中間件並配置URL
app.UseGraphQLVoyager(new VoyagerOptions { GraphQLEndPoint = "/graphql", }, "/graphql-voyager");

app.Run();
```

可以用以下的語法進行測試，新增完成資料後，就會回傳資料

```json
mutation {
    addCustomer(input: { name: "Anderson" }) {
        customer {
            id
            name
        }
    }
}
```

可以用相同邏輯進行Order的Mutation功能開發

增加AddOrderInput.cs

在GraphQL\Customer目錄下，增加AddOrderInput.cs，接收前端傳入的參數

```c# namespace TestGQL.GraphQL.Orders { public record AddOrderInput(string orderno, string orderdate, int custid) {

}
```

}

在GraphQL\Customer目錄下，增加AddOrderPayload.cs，更新完成後回傳結果

```c#
using TestSQL.Models;

namespace TestGQL.GraphQL.Orders
{
    public record AddOrderPayload(Order order);
}
```

修改GraphQL\Mutation.cs檔案，並且增加AddOrderAsync的方法進行資料更新

```c# [UseDbContext(typeof(AppDbContext))] public async Task AddOrderAsync( AddOrderInput input, [ScopedService] AppDbContext context ) { var order = new Order { OrderNo = input.orderno, OrderDate = input.orderdate, CustId = input.custid }; context.Orders.Add(order); await context.SaveChangesAsync();

```
        return new AddOrderPayload(order);
    }
```

完整的Mutation.cs程式如下

```c#
using HotChocolate;
using HotChocolate.Data;
using TestGQL.Data;
using TestGQL.GraphQL.Customers;
using TestGQL.GraphQL.Orders;
using TestSQL.Models;

namespace TestGQL.GraphQL
{
    public class Mutation
    {

        [UseDbContext(typeof(AppDbContext))]
        public async Task<AddCustomerPayload> AddCustomerAsync(
            AddCustomerInput input,
            [ScopedService] AppDbContext context
            )
        {
            var customer = new Customer
            {
                Name = input.name
            };
            context.Customers.Add(customer);
            await context.SaveChangesAsync();

            return new AddCustomerPayload(customer);
        }

        [UseDbContext(typeof(AppDbContext))]
        public async Task<AddOrderPayload> AddOrderAsync(
            AddOrderInput input,
            [ScopedService] AppDbContext context
            )
        {
            var order = new Order
            {
                OrderNo = input.orderno,
                OrderDate = input.orderdate,
                CustId = input.custid
            };
```
```

```
            context.Orders.Add(order);
            await context.SaveChangesAsync();

            return new AddOrderPayload(order);
        }
    }
}
```

可以用以下的語法進行測試，新增完成資料後，就會回傳資料

```
mutation {
    addOrder(input: {
        orderno:"TEST00001",
        orderdate:"20220808",
        custid:1
    }) {
        order {
            id
            orderNo,
            orderDate,
            custId
        }
    }
}
```

增加Subscription功能

增加Subscription.cs

主要定義回傳的資料內容有哪些？

```c# using TestSQL.Models; using HotChocolate; using HotChocolate.Types;

namespace TestSQL.GraphQL { public class Subscription { [Subscribe] [Topic] public Customer OnCustomerAdded([EventMessage] Customer customer) { return customer; } } }
```

修改Mutation.cs內容

傳入參數增加以下定義

```c#
[Service] ITopicEventSender eventSender,
  CancellationToken cancellationToken
```

完整程式如下

```c# [UseDbContext(typeof(AppDbContext))] public async Task AddCustomerAsync( AddCustomerInput input, [ScopedService] AppDbContext context, //增加訂閱相關參數 [Service] ITopicEventSender eventSender, CancellationToken cancellationToken ) { var customer = new Customer { Name = input.name }; context.Customers.Add(customer); //修改增加訂閱相關參數 await context.SaveChangesAsync(cancellationToken); //訂閱觸發的設定 await eventSender.SendAsync(nameof(Subscription.OnCustomerAdded), customer, cancellationToken);
```

```
        return new AddCustomerPayload(customer);
    }
```

修改Program.cs

開啟Program.cs增加Subscription的宣告

```c#
builder.Services.AddSubscriptionType<Subscription>();
builder.Services.AddInMemorySubscriptions();
```

增加Websocket的宣告，這部分需要優先宣告

```c# app.UseWebSockets();
```

完整程式如下：

```c#
using TestGQL.Data;
using TestGQL.GraphQL;
using Microsoft.EntityFrameworkCore;
using GraphQL.Server.Ui.Voyager;
using TestGQL.GraphQL.Customers;
using TestGQL.GraphQL.Orders;
using TestSQL.GraphQL;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddGraphQLServer()
    .AddQueryType<Query>()
    .AddType<CustomerType>()
    .AddMutationType<Mutation>()
    .AddSubscriptionType<Subscription>()
    .AddType<OrderType>()
    .AddSorting()
    .AddFiltering()
    .AddInMemorySubscriptions();

builder.Services.AddPooledDbContextFactory<AppDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("GQLConStr"));
});

var app = builder.Build();

app.UseWebSockets();

app.UseRouting().UseEndpoints(endpoints =>
{
    endpoints.MapGraphQL();
});
// 增加Voyager中間件並配置URL
app.UseGraphQLVoyager(new VoyagerOptions
{
    GraphQLEndPoint = "/graphql",
}, "/graphql-voyager");

app.Run();
```

接下來開啟https://localhost:7283/graphql為訂閱端，進行查詢後，會處於等待的狀況

```
subscription{
  onCustomerAdded{
    id
    name
  }
}
```

開啟Insomnia作為資料更新端，進行顧客新增的作業

```
mutation {
    addCustomer(input: { name: "Hank" }) {
        customer {
            id
            name
        }
    }
}
```