

Workshop building your own GraphQL server in .NET 8 Rider Edition

To be able to build the GraphQL service, you need to have the Web-templates installed in Rider.

The workshop will be done by following a tutorial with background information and assignments. **You will have to do the lines in bold in your own environment.**

Should you run into problems, please contact me during the session. The completed code can be found at: [JohanSmarius/GraphQLWorkshop \(github.com\)](https://github.com/JohanSmarius/GraphQLWorkshop)

Step 1: Create a new WebApi project.

In Rider create a new project using the Web Project Type and the Web API template. Do make sure that .NET 8.0 is selected as the target Framework. You can choose a name for the solution and project (Figure 1). In the samples ShopSolution and ShopAPI will be used.

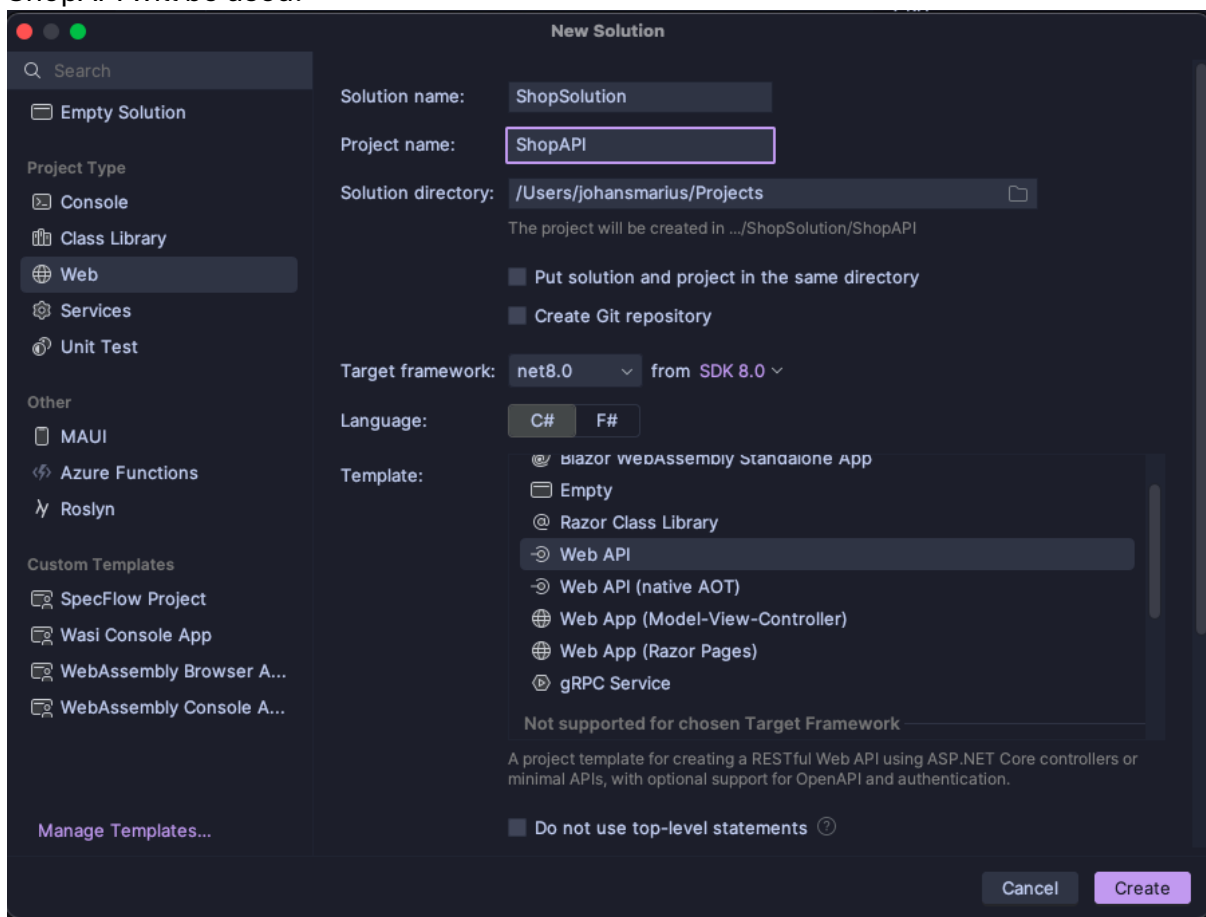


Figure 1 Create project.

You can leave the more advanced options to their default settings (Figure 2).

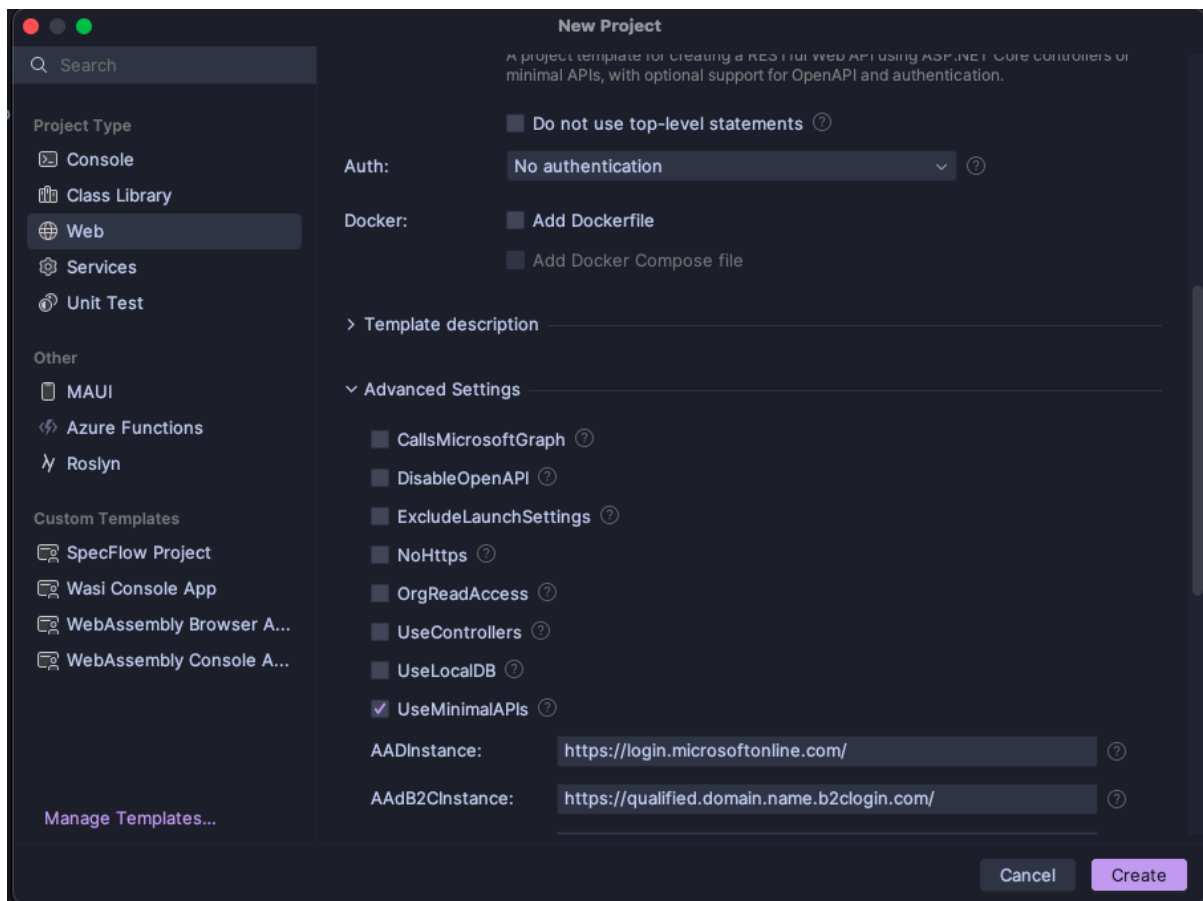
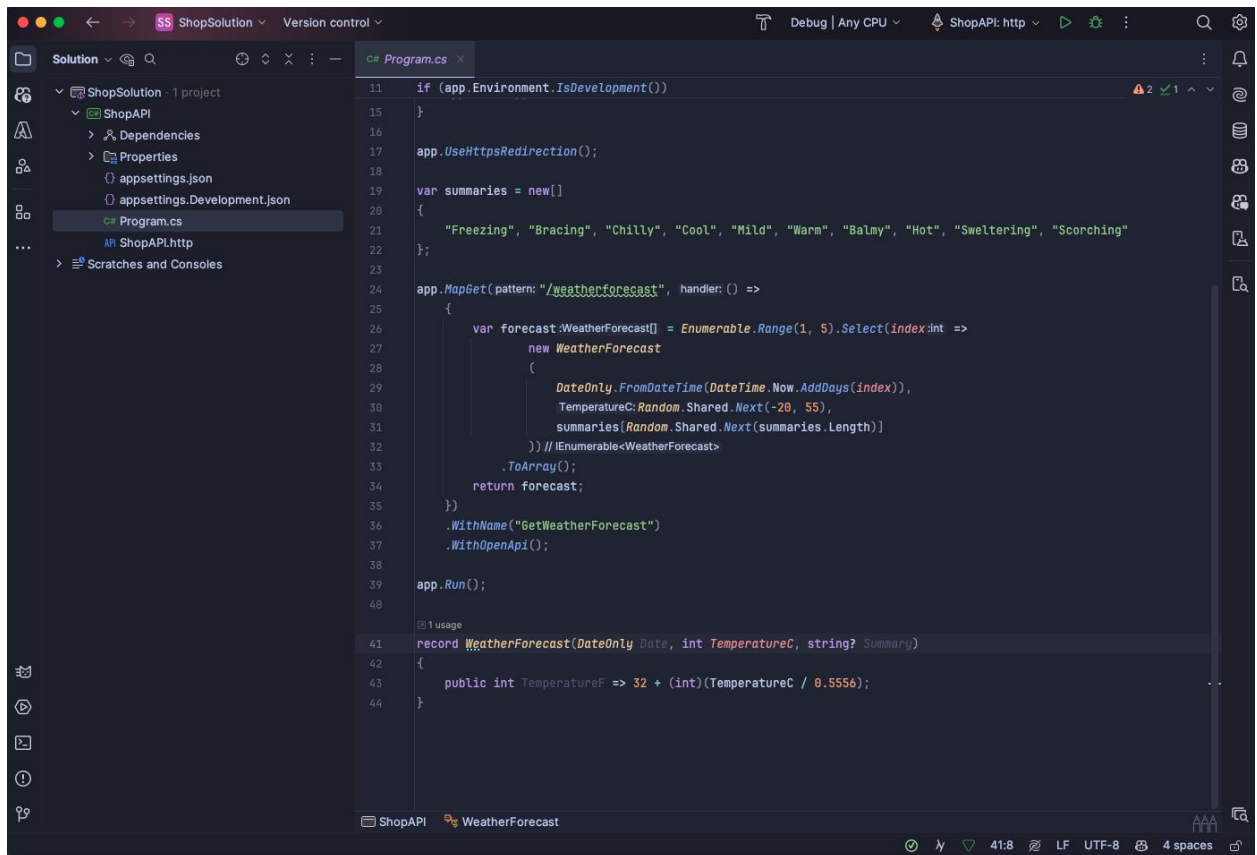


Figure 2 Advanced settings.

A new project with a minimal API will be generated (Figure 3).



```
11 if (app.Environment.IsDevelopment())
12 {
13 }
14
15 app.UseHttpsRedirection();
16
17 var summaries = new[]
18 {
19     "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
20 };
21
22 app.MapGet(pattern: "/weatherforecast", handler: () =>
23 {
24     var forecast: WeatherForecast[] = Enumerable.Range(1, 5).Select(index: int =>
25         new WeatherForecast
26         (
27             DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
28             TemperatureC: Random.Shared.Next(-20, 55),
29             summaries[Random.Shared.Next(summaries.Length)]
30         )) //IEnumerable<WeatherForecast>
31         .ToArray();
32     return forecast;
33 })
34 .WithName("GetWeatherForecast")
35 .WithOpenApi();
36
37 app.Run();
38
39
40
41 record WeatherForecast(DateOnly Date, int TemperatureC, string? Summary)
42 {
43     public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
44 }
```

Figure 3 Generated code.

Just to make sure that everything works, run the project, and verify that the API does produce some random temperatures. If you are not familiar with Minimal API's, the endpoint can be reached by adding “/weatherforecast” after the URL of the service.

The output should look something like Figure 4.

```
[{"date": "2024-05-03", "temperatureC": 47, "summary": "Chilly", "temperatureF": 116}, {"date": "2024-05-04", "temperatureC": 14, "summary": "Scorching", "temperatureF": 57}, {"date": "2024-05-05", "temperatureC": 21, "summary": "Hot", "temperatureF": 69}, {"date": "2024-05-06", "temperatureC": 27, "summary": "Chilly", "temperatureF": 80}, {"date": "2024-05-07", "temperatureC": 48, "summary": "Mild", "temperatureF": 118}]
```

Figure 4 Sample output.

Step 2: Add the required NuGet package.

As shortly explained in the presentation we will be using HotChocolate for adding GraphQL support to our .NET solution. **For this you need to add the HotChocolate.ASPNETCore package to your solution.** In NuGet there are some preview versions available for the upcoming 14 release. Since we don't want to run the risk of hitting some bugs, we will be using the **last stable version 13.9.0** for this workshop (Figure 5).

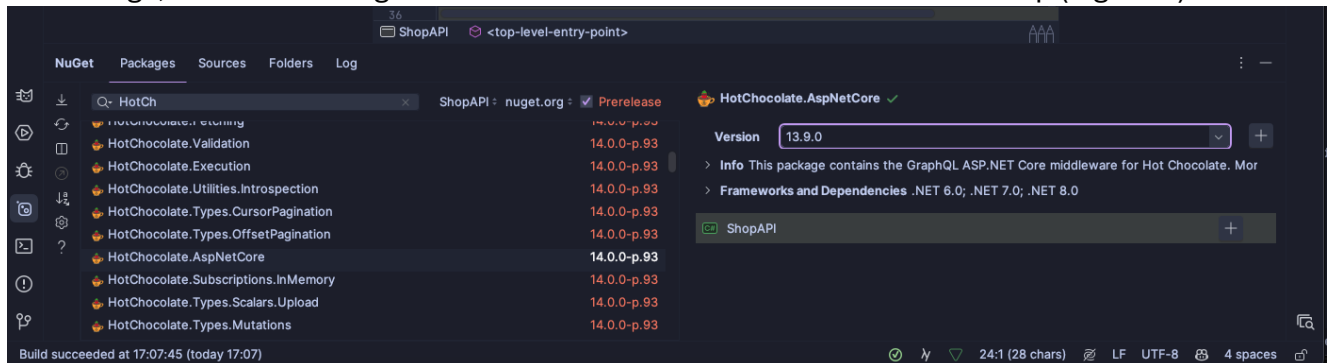


Figure 5 NuGet package.

Step 3: Add the model classes.

Just like any other solution, we do need some domain classes to work with. Just to keep our project clean, we are going to store the domain classes in their own directory in our project. In the sample code I will use Model for this directory (Figure 6).

Create this directory in your own solution.

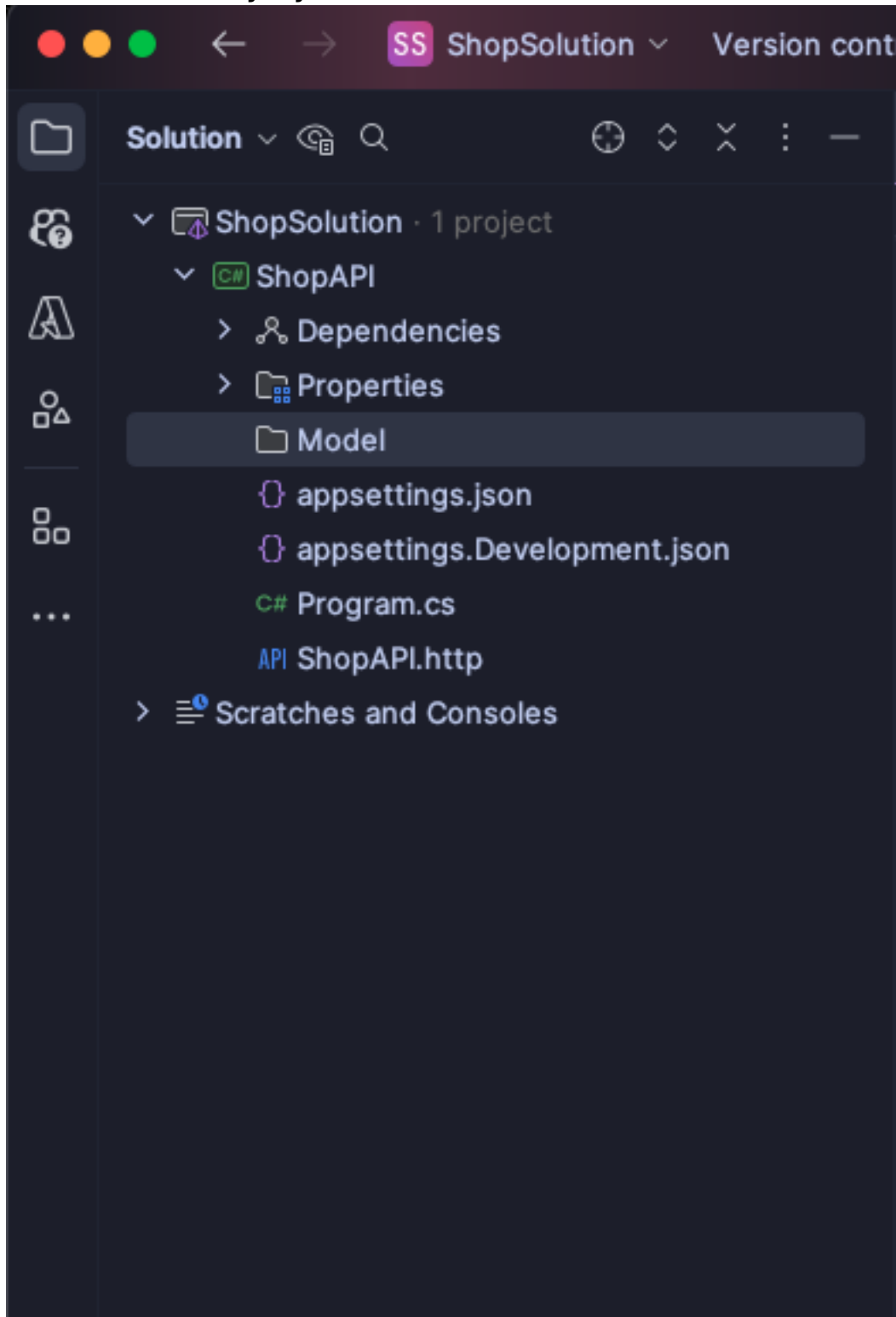


Figure 6 Directory structure.

Within this directory we need to create some classes. Let's start with creating a Customer class.

Copy this code to or type this code in a new class file named Customer.cs.

```
public class Customer
{
    public int Id { get; set; }

    public required string Name { get; set; }

    public string? EMailAddress { get; set; }
}
```

The Name property for the Customer must be set during construction, so I added the required keyword to this property.

The next class that we will create is the Product class.

Copy this code to or type this code in a new class file named Product.cs.

```
public class Product
{
    public int Id { get; set; }

    public required string Name { get; set; }

    public string? EanCode { get; set; }

    public decimal Price { get; set; }

    public decimal Weight { get; set; }
}
```

Just like the Customer class, the Name property has been set to required as well.

Now we have the supporting classes in place, we can focus on the Order specific class. First we add the OrderLine class to the project. We will be integrating Entity Framework later in the workshop, so we already take this into account when creating the code for this OrderLine. So, we add both the Id and the reference to the Product class and the yet to be built Order class. In this code we suppress the nullability check.

Copy this code to or type this code in a new class file named OrderLine.cs.

```
public class OrderLine
{
    public int Id { get; set; }

    public int OrderId { get; set; }

    public Order Order { get; set; } = null!;

    public Product Product { get; set; } = null!;

    public int ProductId { get; set; }

    public int Quantity { get; set; } = 1;

    public decimal DiscountPercentage { get; set; } = 0;
}
```

The Order class can be added next. The reference to Customer will be handled later on by Entity Framework, so we suppress this nullability check for now.

Copy this code to or type this code in a new class file named Order.cs.

```
public class Order
{
    public int Id { get; set; }

    public ICollection<OrderLine> OrderLines { get; set; } = new
List<OrderLine>();

    public Customer Customer { get; set; } = null!;

    public int CustomerId { get; set; }

    public DateTime OrderTime { get; set; }

    public OrderStatus OrderStatus { get; set; }
}
```

The only thing that is missing from our domain classes is the enumeration for OrderStatus.

Copy this code to or type this code in a new class file named OrderStatus.cs.

```
public enum OrderStatus
{
    NEW,
    PAID,
    OPENTOPICK,
    SHIPPED,
    DELIVERED,
    RETURNED
}
```

You can now build your solution and if everything is correct, the solution will build without errors.

Step 4: Add the Query support for GraphQL

Fetching data can be done in GraphQL through a Query-class. We will store all the GraphQL code in a separate directory under the project named GraphQL. This is just a convention that I use. It is not enforced in any way. In this directory I will create a class name OrderQuery. I choose this name, because I want the queries to be done from the order level as root. This class is just a plain old C# class with no special classes to inherit from.

Create the directory GraphQL in your solution and add a new class named OrderQuery to it.

```
public class OrderQuery
{

}
```

We do want to provide a method to allow the user of the API to fetch all orders. Since we don't have a database yet, we do need to get a way to get a list of orders. We will implement a helper method within the OrderQuery class to generate a list of Orders.

Add this method to your OrderQuery class.

```
private List<Order> GenerateTestOrders()
{
    var order1 = new Order()
    {
        Id = 1,
        Customer = new Customer()
        {
            Id = 1, Name = "Someone", EMailAddress =
"noreply@nowhere.com"
        },
        OrderStatus = OrderStatus.PAID,
        OrderTime = DateTime.Now
    };

    order1.OrderLines.Add(new OrderLine()
    {
        Id = 1,
        Order = order1,
        Product =
            new Product() {Id = 1, Name = "Samsung S20", Price =
900, Weight = 100, EanCode = "4985791347598"},
        Quantity = 2,
        DiscountPercentage = 20
    });
}
```

```

order1.OrderLines.Add(new OrderLine()
{
    Id = 2,
    Order = order1,
    Product = new Product()
        {Id = 1, Name = "iPhone 13", Price = 1200, Weight =
100, EanCode = "498430958307598"},
    Quantity = 1,
    DiscountPercentage = 10
});

var order2 = new Order()
{
    Id = 2,
    Customer = new Customer()
    {
        Id = 1, Name = "Iris", EMailAddress =
"whoami@nowhere.com"
    },
    OrderStatus = OrderStatus.SHIPPED,
    OrderTime = DateTime.Now.AddDays(-2)
};

order2.OrderLines.Add(new OrderLine()
{
    Id = 1,
    Order = order1,
    Product = new Product() {Id = 1, Name = "Clean code",
Price = 20, Weight = 200, EanCode = "94359324590380"},
    Quantity = 1,
    DiscountPercentage = 0
});
order2.OrderLines.Add(new OrderLine()
{
    Id = 2,
    Order = order1,
    Product = new Product()
    {
        Id = 1, Name = "The unlikely success of a copy-paste
developer", Price = 30, Weight = 150,
        EanCode = "87439587975927"
    },
    Quantity = 2,
    DiscountPercentage = 5
});

return [order1, order2];
}

```

With this helper method in place, we can write the actual query method.

Add this code to your OrderQuery class.

```
public IEnumerable<Order> GetAllOrders()
{
    return GenerateTestOrders();
}
```

This method is also just C#, so no special attributes are needed yet.

Step 5: Add GraphQL to ASP.NET Core

So far, we have not done anything to make ASP.NET Core aware of the GraphQL classes. We do need to add some code in program.cs to change this.

Before the line with builder.Build() we have to add the graphql service to the pipeline using the following code.

Add this code at the correct location in your own program.cs file.

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

And before app.Run() we need to add GraphQL to the middleware using the following code.

Add this code to your program.cs as well.

```
app.MapGraphQL();
```

Step 6: Run our first query

All the code we have added is enough to get GraphQL to work in a very basic setting. If we run the code, an empty page is shown. This is not an error, but simply the default page for the site we created. We can start interacting with our service by adding “/graphql” after the url. This will open up the BanakaCakePop test site which we can use to interact with our service (Figure 7).

Start your service and navigate to the /graphql endpoint.

During the workshop we will restart the service at several times. Normally a new tab is opened in the browser per debug/run session. You can however simply refresh this first tab and work from there.

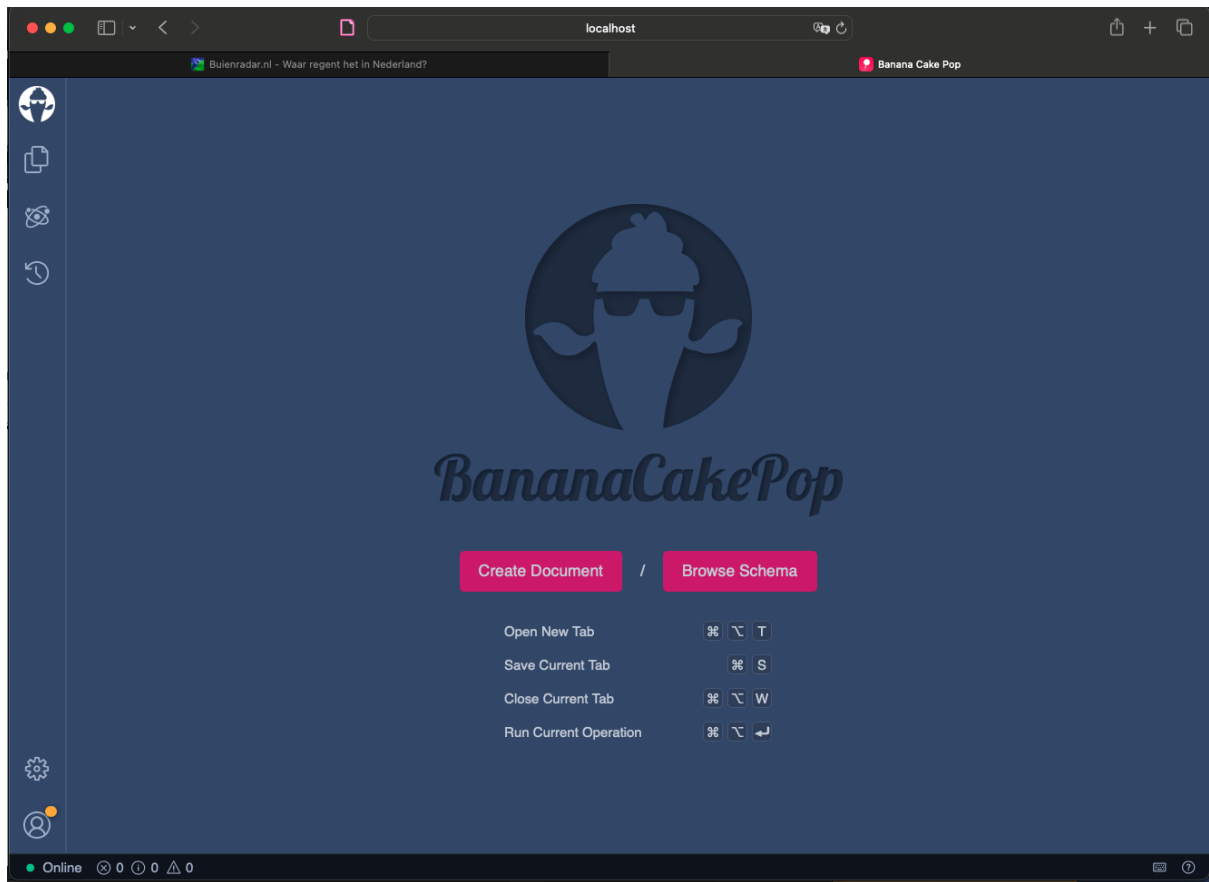


Figure 7 BananaCakePop home screen.

GraphQL works through a schema. In this workshop we are not going to use the schema directly to write a service, but it is good to know a bit about the schema being used. When you click the Browse Schema button, the generated schema for OrderQuery is shown (Figure 8).

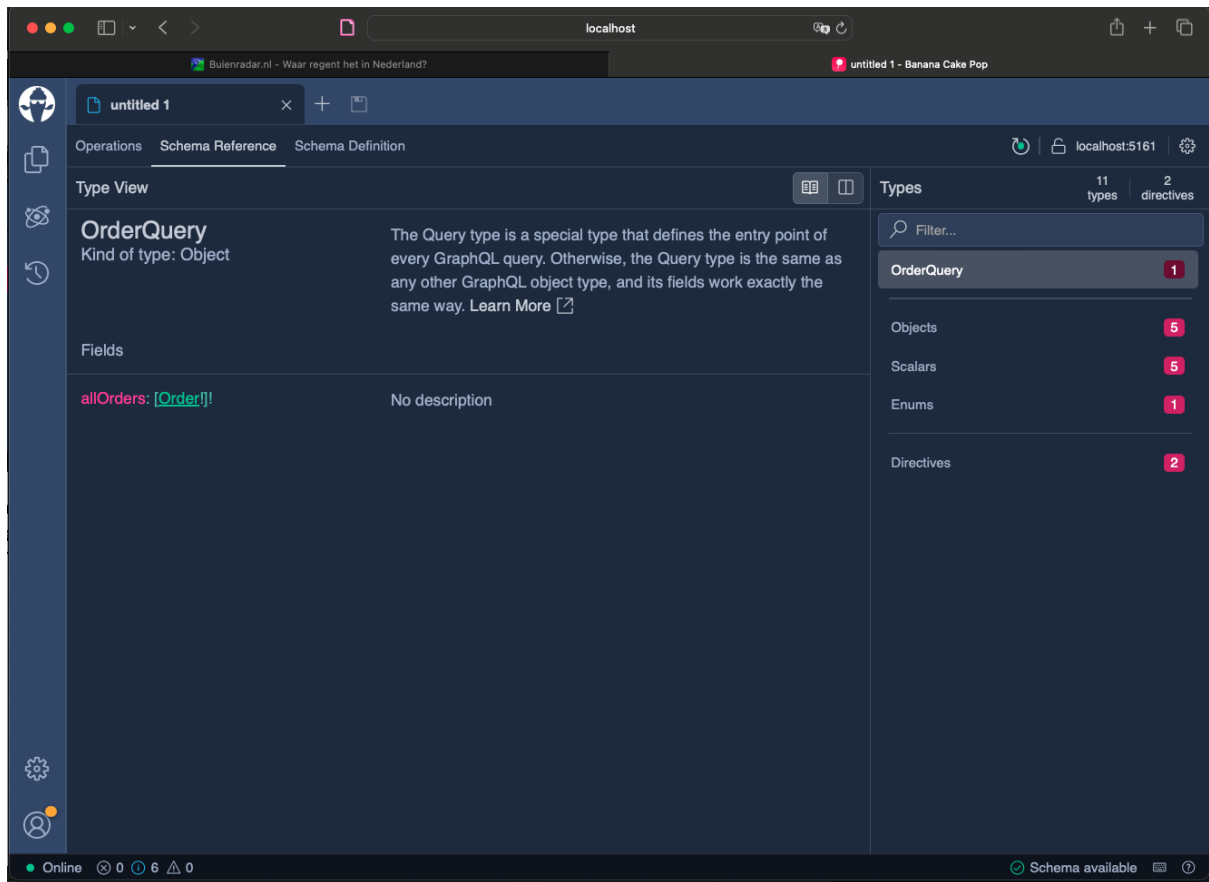


Figure 8 Schema for OrderQuery

Our method GetAllOrders got generated into a field allOrders. The Get is removed automatically from the code. The [] indicate that a list of Orders is returned, and the exclamation mark indicates that the value cannot be null. By default, values in GraphQL can be null.

When we click on the Order type, the details of the schema for Order are shown (Figure 9).

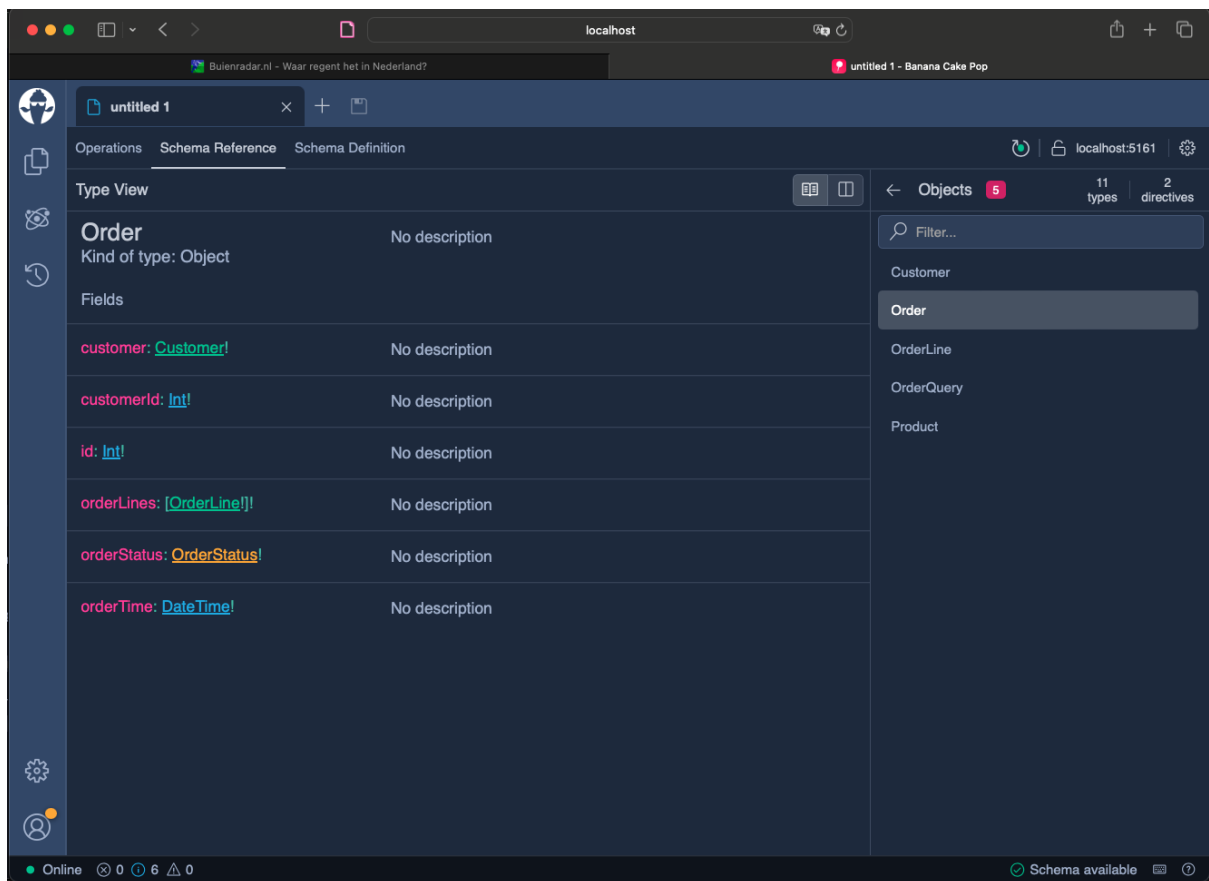


Figure 9 Schema for OrderType

Every field has its own type specified, and you could drill further into the types by clicking on the links. If you want to know more about the GraphQL schema language, you can visit <https://graphql.org/learn/schema/>.

Since this workshop is focused around building a service, let's use the schema to construct a query. **So go back to the Operations tab (Figure 10).**

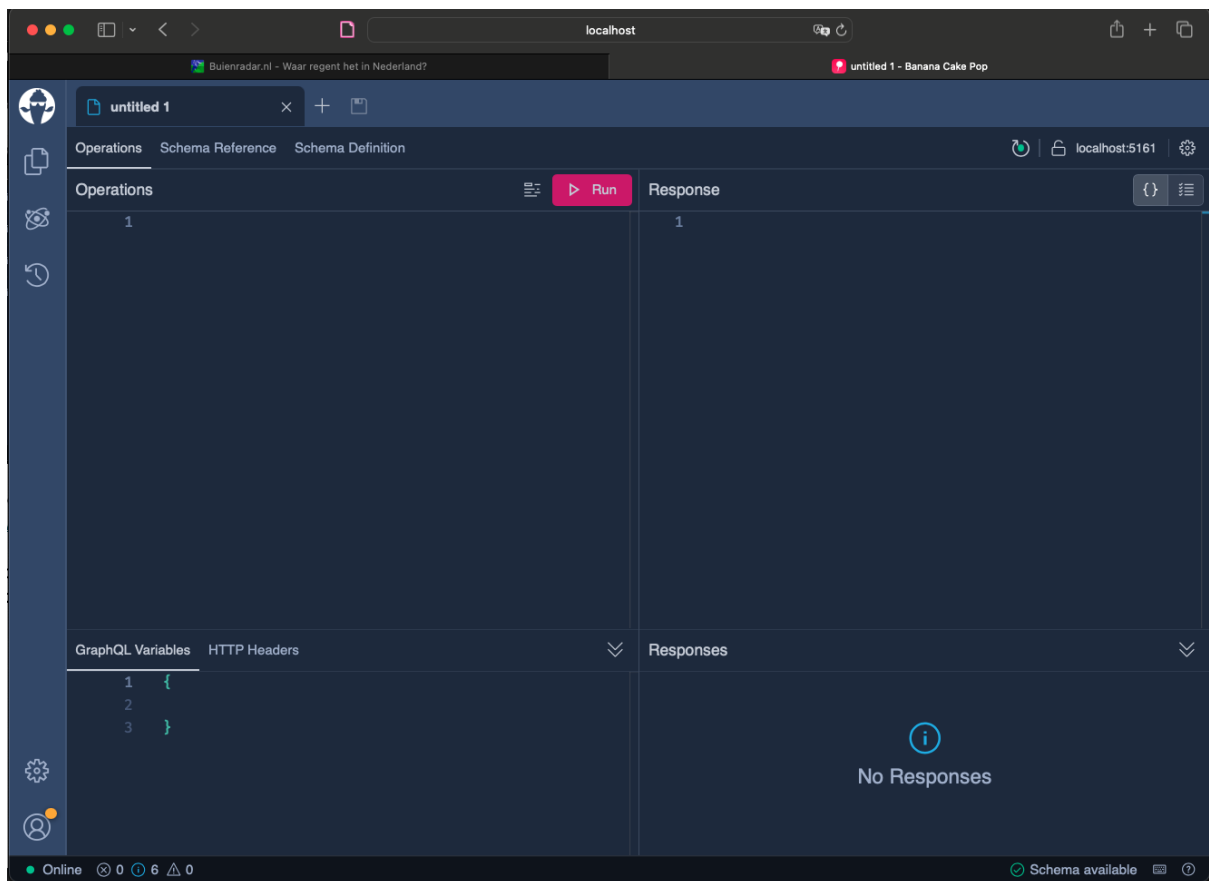


Figure 10 Operations tab.

In the operations tab we can add a query.

All queries start with the keyword query. If you only have one query, you don't have to specify a name.

We start by simply adding a query that uses our allOrder schema (Figure 11).

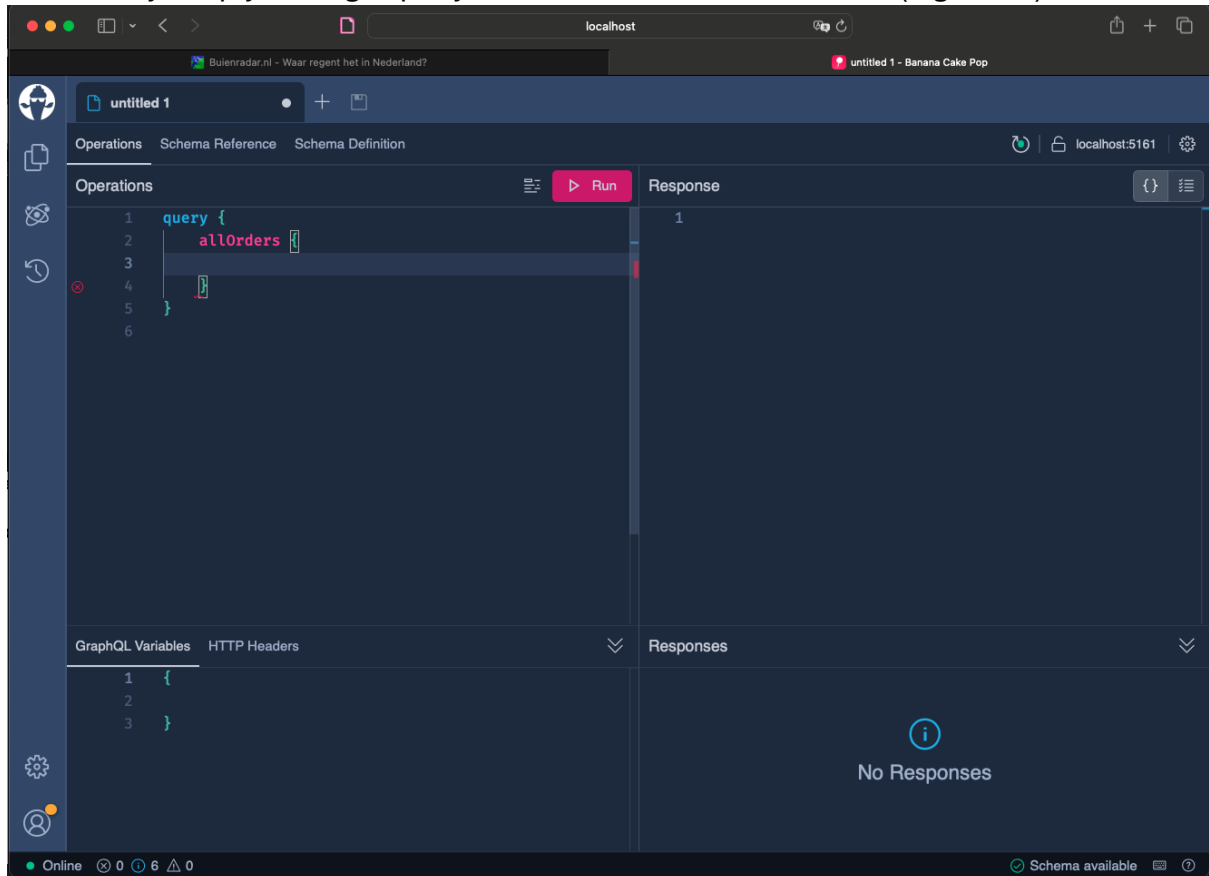


Figure 11 Start of Query

Enter the query as listed above in the operations tab.

And within this query we can ask for all the fields that are available in the type (Figure 12). This allows us to control the data that is returned from the service, so we can prevent overfetching field that are not of interest to us.

Add these fields to the query.

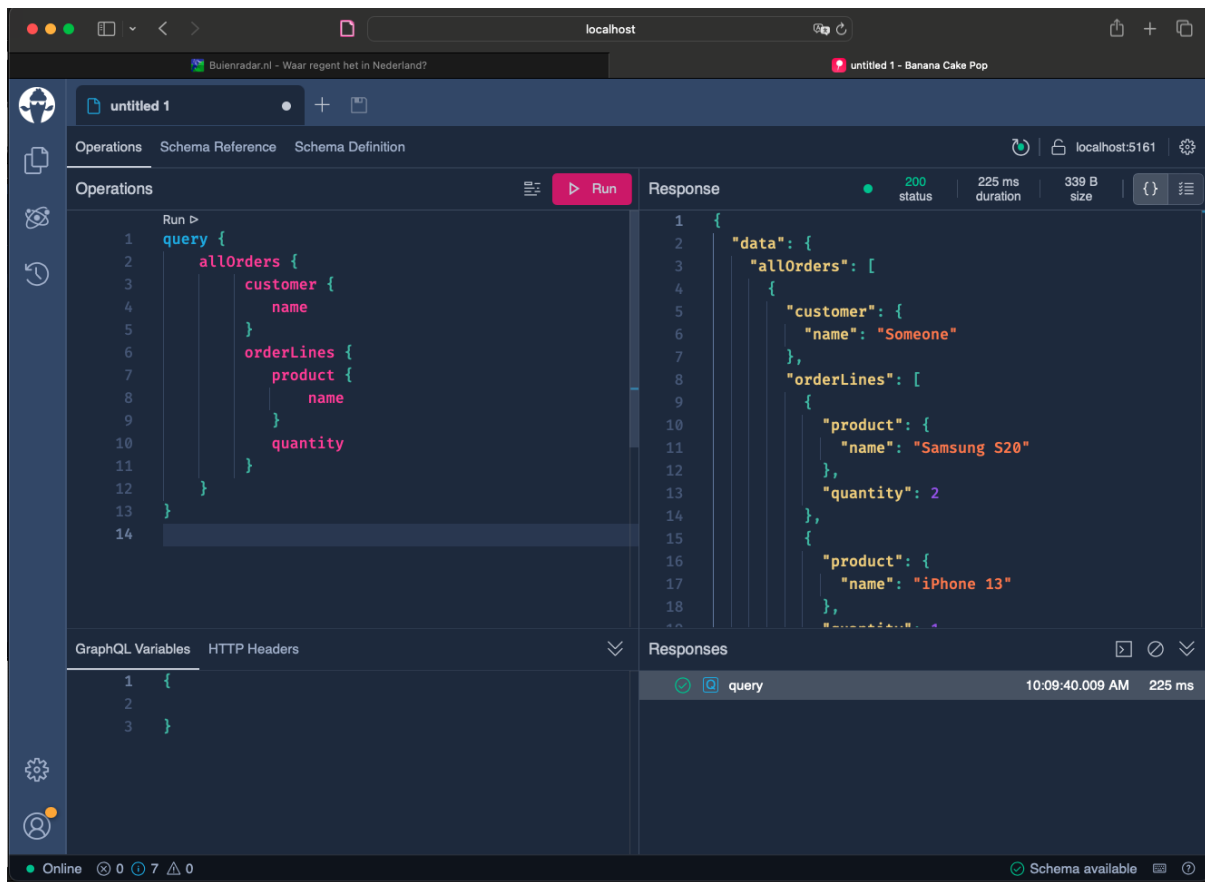


Figure 12 Added requested fields to the query.

And when you run the query, the results are returned as shown in Figure 12. **Verify that you do see similar results.**

Since the test environment allows code completion, simply try to change the query to include the OrderStatus and OrderTime.

Step 7: Working with parameters

GraphQL also supports passing parameters to a query. To demonstrate this, we are going to build a new query method to return an order based on its Id. This is just for demo purposes. In a real-life service, you would solve this in a different way. We will get to this later in the workshop.

Add the following code to the OrderQuery class.

```
public Order GetOrderById(int id)
{
    return GenerateTestOrders().Single(order => order.Id == id);
}
```

The Single ensures that only 1 record can ever be retrieved from the collection. Since the id field can be considered to be the primary key this will hold true for this workshop.

We do not need to change anything in program.cs for this to work. **All we have to do is run the application again.**

If we look at the schema that is now generated, we can see that a new field has been added (Figure 13).

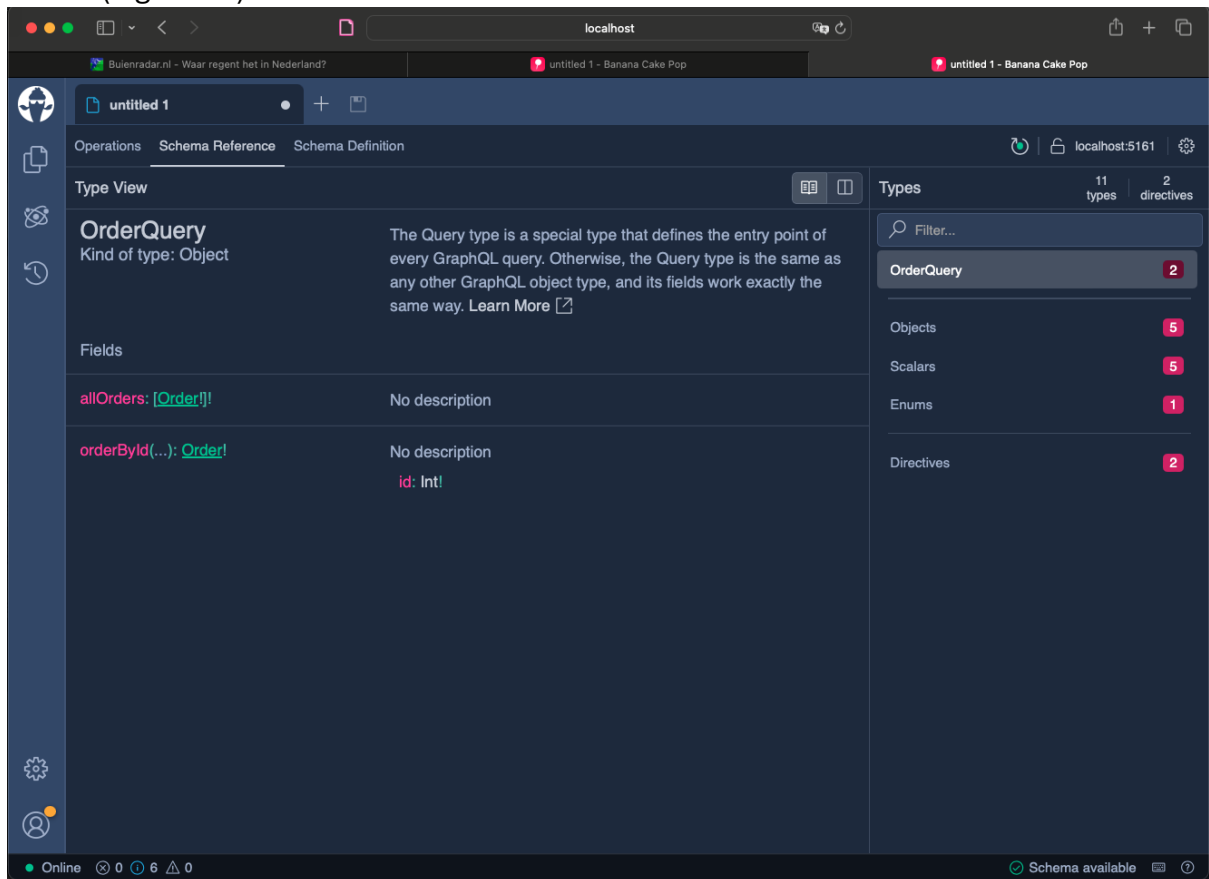


Figure 13 Updated schema.

We can use this field in the query as well.

First give a name (allOrders) to the first query we wrote (Figure 14).

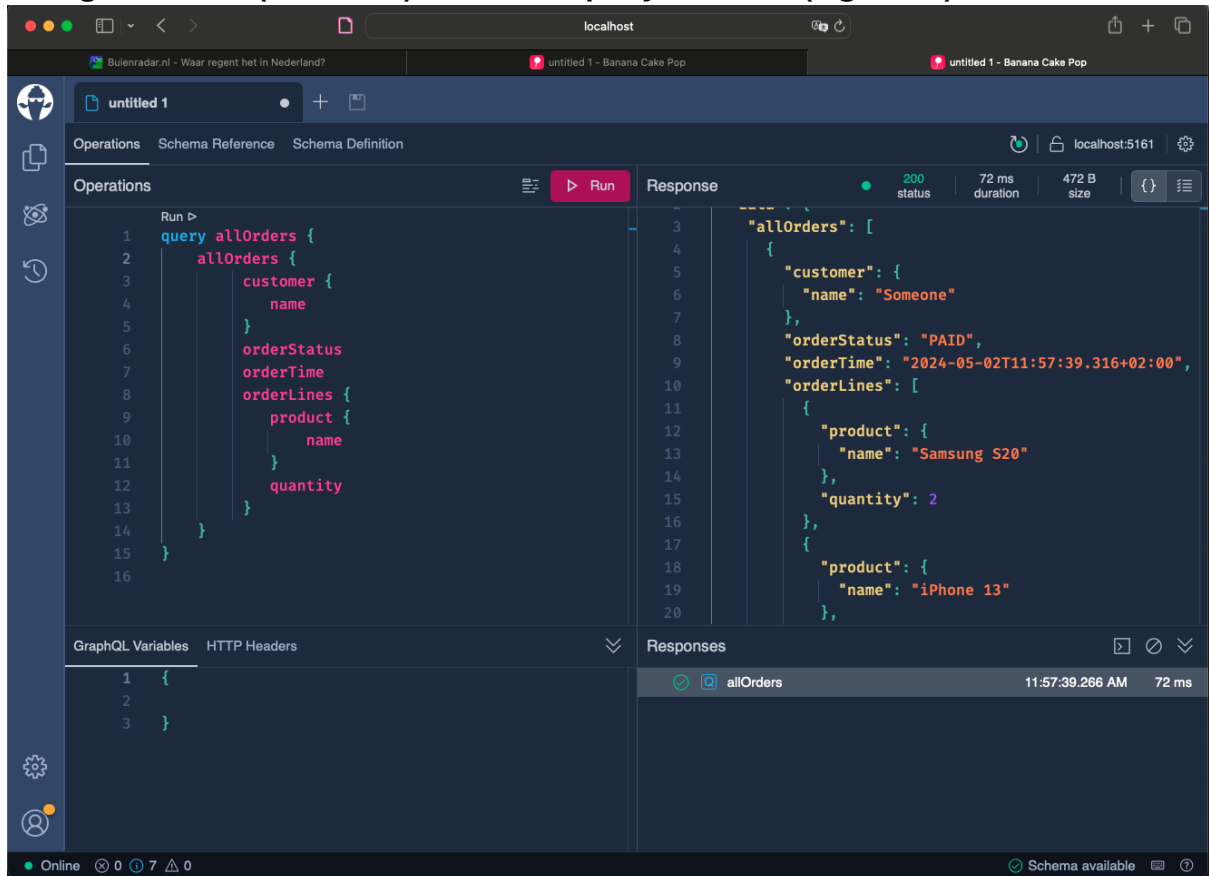


Figure 14 Name first query.

We can now create a new query for the operation we just created (Figure 15).

Add this query to your own test site.

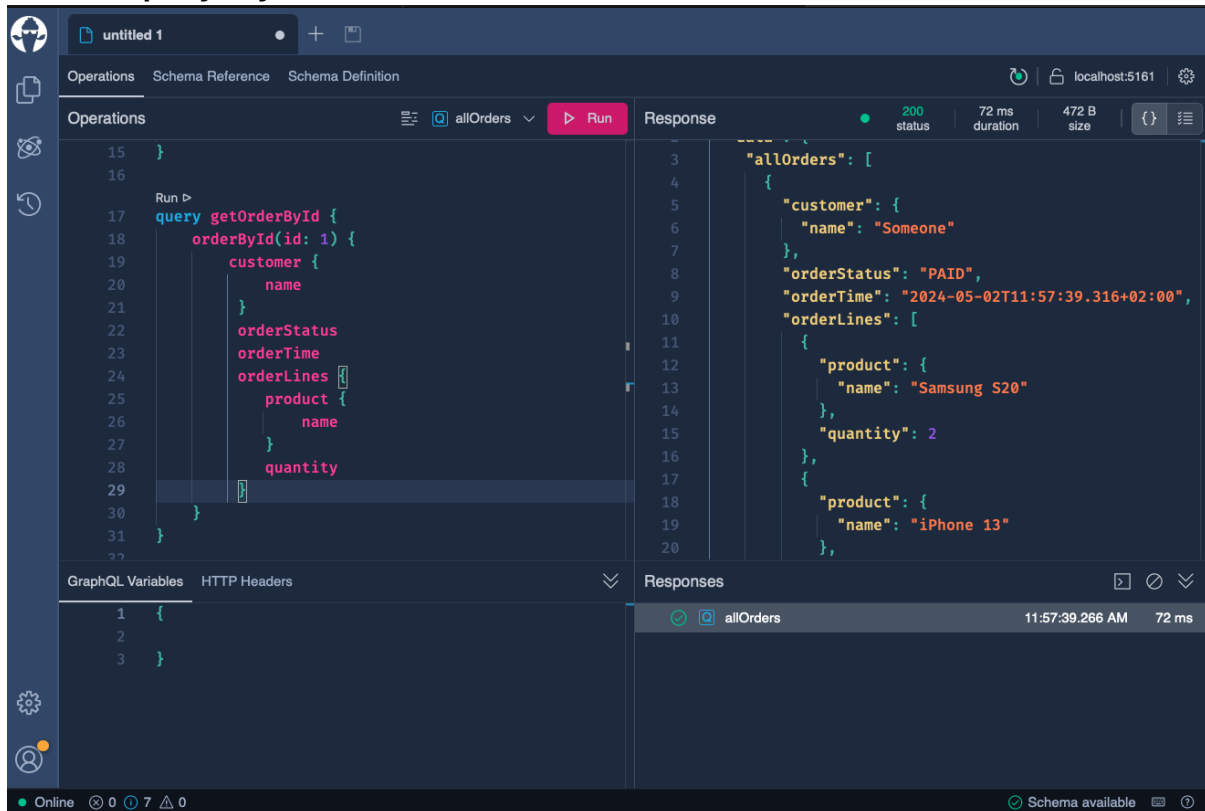


Figure 15 Call second operation.

The parameter can be passed between the rounded brackets. This parameter is now fixed for this query, but we also have the option to add the parameter to the query itself (Figure 16).

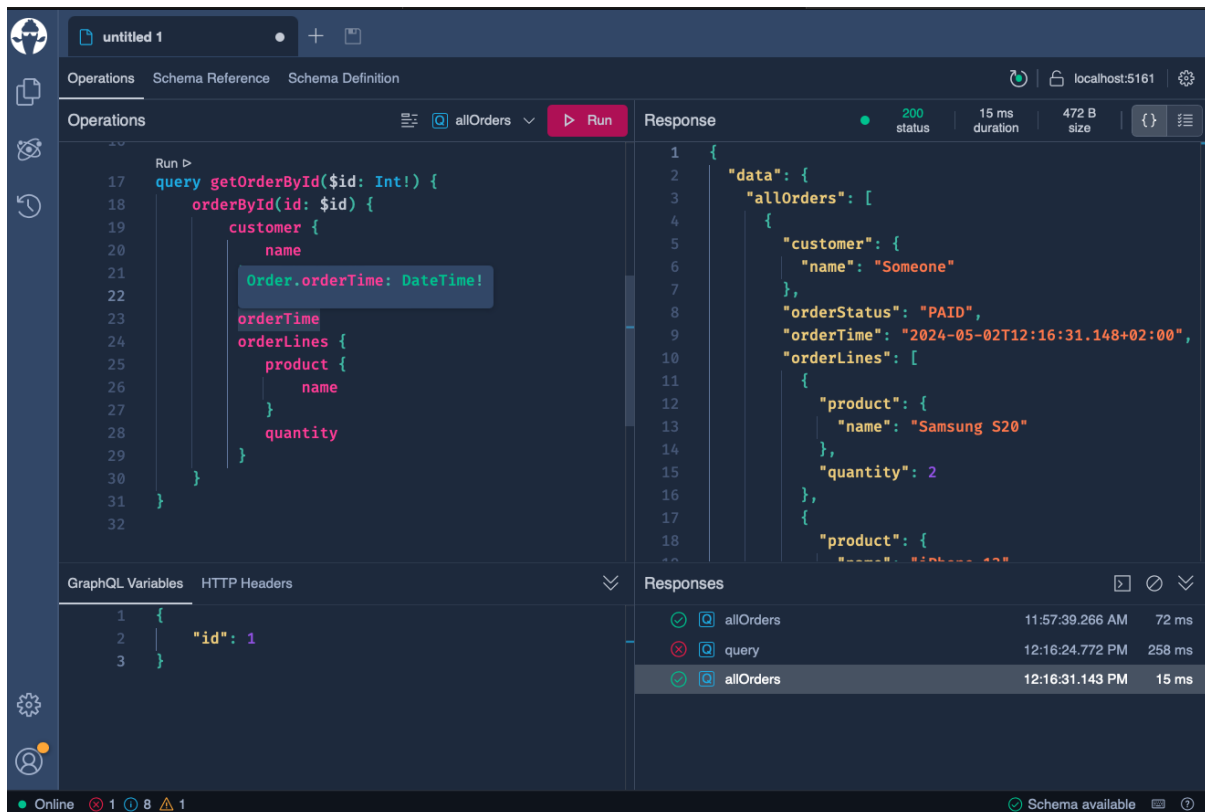


Figure 16 Add parameter to query.

If we specify a parameter in the query, then we need to supply the variable in the section of GraphQL variables as in the picture.

Update your query to include the parameter and add the parameter value in the variables pane.

Step 8: Working with fragments in GraphQL.

The information about the fields we want to retrieve from the query is now duplicated. Just as with code, code duplication should be avoided if possible. Fortunately, this is possible by using fragment. In the fragment we must specify for which type we are specifying a fragment for (Figure 17).

Add this code to your own test site.

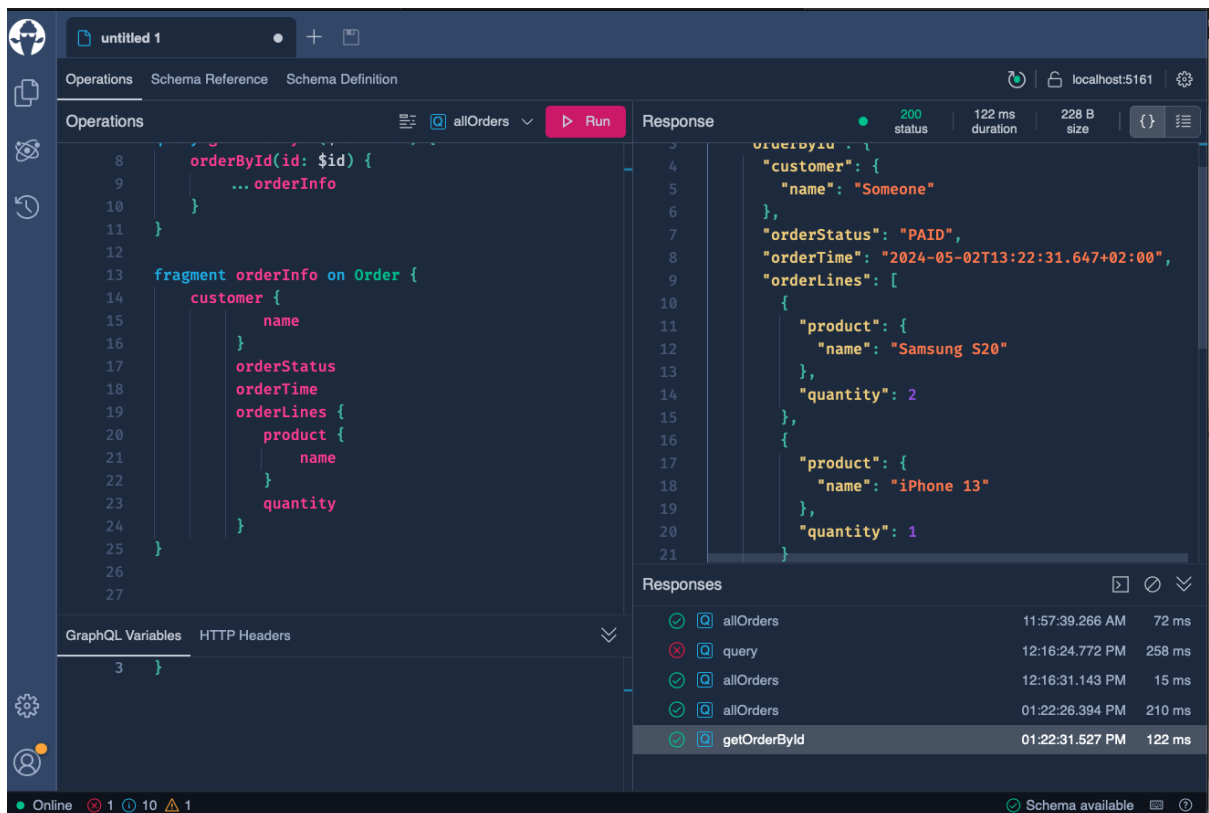


Figure 17 Add fragment.

The fragment can be used inside the query by using the ... construction (Figure 18).

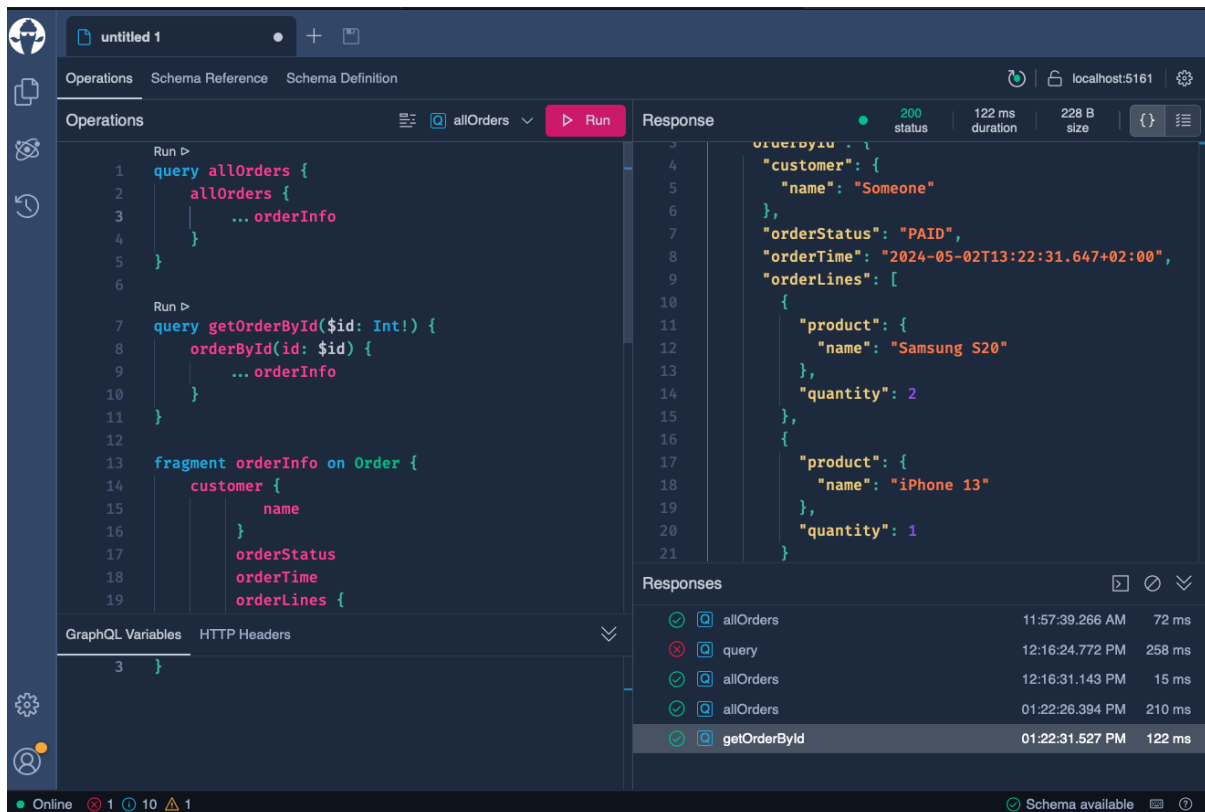


Figure 18 Use of fragment in query.

Update your own query to use the fragment.

Step 9: Connect GraphQL to Entity Framework Core

Most services do need some kind of data storage to work. In this workshop we will be using Entity Framework Core in combination with SQLite for this. SQLite works cross platform and is therefore better suited to support Windows, Mac and Linux users. In this workshop we don't enforce a certain operating system.

To support Entity Framework Core, we first need to add the NuGet packages to the solution. For this workshop we will be using the following packages:

- Microsoft.EntityFrameworkCore version 8.0.4
- Microsoft.EntityFrameworkCore.Tools version 8.0.4
- Microsoft.EntityFrameworkCore.Sqlite version 8.0.4
- HotChocolate.Data.EntityFrameworkCore version 13.9.0

Add these NuGet packages to your own solution.

After installing the packages, the list of packages should look like Figure 19.

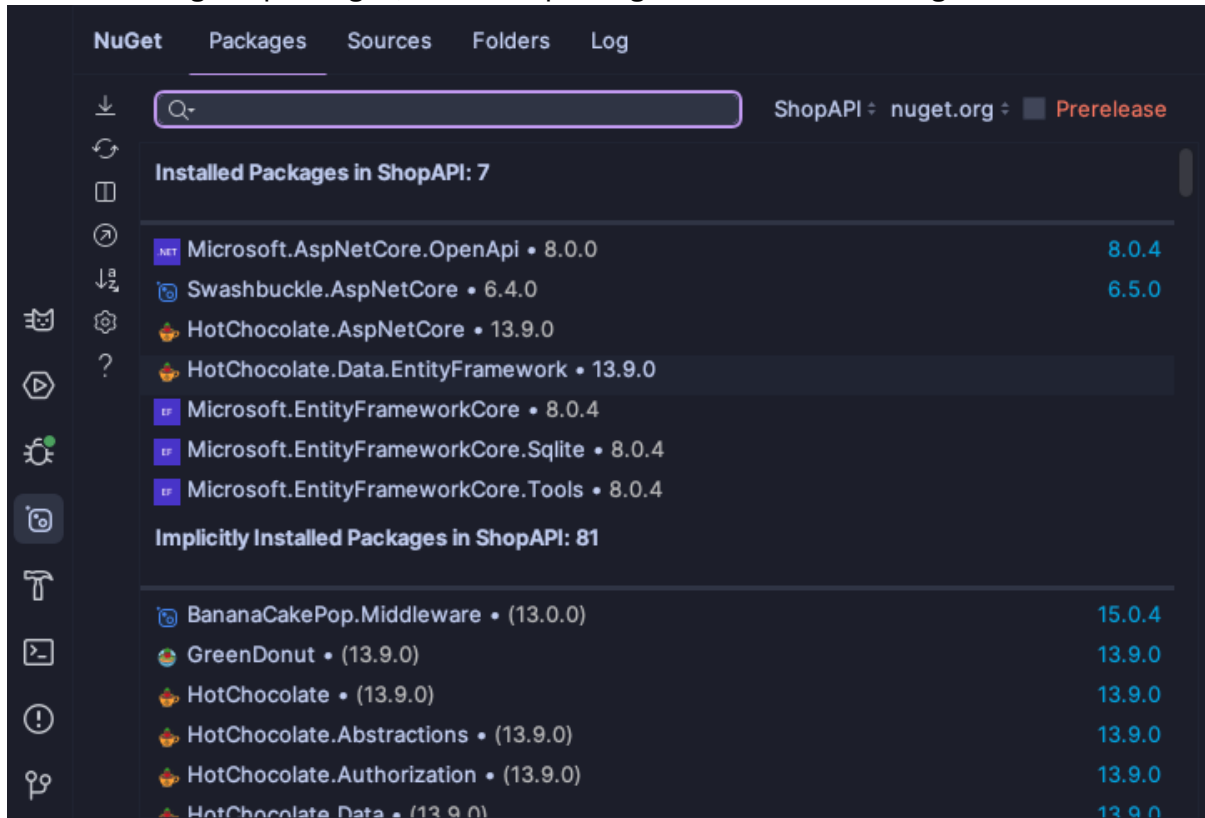


Figure 19 Packages after installing EF Core

Before we can connect GraphQL to Entity Framework Core we need to write a DataContext for this. We will put this class in its own directory named Database. Since we want to retrieve data from the database, we will add seeding to the context.

Create the directory in your solution and add this code to a new code file named OrderContext.cs.

```
public class OrderContext : DbContext
{
    public DbSet<Order>? Orders { get; set; }

    public DbSet<OrderLine>? Orderlines { get; set; }
    public DbSet<Customer>? Customers { get; set; }

    public OrderContext(DbContextOptions<OrderContext> options)
: base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
    }
}
```



```

        modelBuilder.Entity<Customer>().HasData(
            new Customer() { Id = 1, Name = "Customer 1"},
            new Customer() { Id = 2, Name = "Customer 2"}
        );
        modelBuilder.Entity<Product>().HasData(
            new Product() { Id = 1, Name = "Product 1", EanCode
= "123439900", Price = 100, Weight = 300 },
            new Product() { Id = 2, Name = "Product 2", EanCode
= "37034034039", Price = 200, Weight = 700}
        );
        modelBuilder.Entity<Order>().HasData(
            new List<Order>()
            {
                new()
                {
                    Id = 1, CustomerId = 1, OrderTime =
DateTime.Now
                },
                new()
                {
                    Id = 2, CustomerId = 2, OrderTime =
DateTime.Now
                }
            }
        );
        modelBuilder.Entity<OrderLine>().HasData(
            new OrderLine() {Id = 2, OrderId = 1, ProductId = 1,
Quantity = 2, DiscountPercentage = 0},
            new OrderLine() {Id = 1, OrderId = 1, ProductId = 2,
Quantity = 5, DiscountPercentage = 5},
            new OrderLine() {Id = 3, OrderId = 2, ProductId = 1,
Quantity = 7, DiscountPercentage = 0},
            new OrderLine() {Id = 4, OrderId = 2, ProductId = 2,
Quantity = 10, DiscountPercentage = 20.0m}
        );
    }
}

```

With this DbContext in place, we can add the support for EntityFramework to our program.cs class.

Just above the code that adds the GraphQLService, add the following code:

```

builder.Services.AddPooledDbContextFactory<OrderContext>(
    options => options.UseSqlite("Data Source=Orders.db")

.EnableSensitiveDataLogging()).AddLogging(Console.WriteLine);

```

This will add the DbContext to the Services collection. We do need to add a pooled context factory due to the inner workings of GraphQL. This framework can run parallel queries and if you just register the DbContext in the normal way, this would not work. Please do not add sensitive data logging in your own applications, but we now need it now to test some query features later on.

GraphQL should also be made aware of the Entity Framework link.

So, add the following code for RegisterDbContext after our Query registration.

```
builder.Services.AddGraphQLServer()  
    .AddQueryType<OrderQuery>()  
    .RegisterDbContext<OrderContext>(DbContextKind.Pooled);
```

Because we are using a pooled connection, we also specify this in the RegisterDbContext.

To start working with our database, we need to generate an initial migration. To support Entity Framework Migrations, I use the Rider Plugin “Entity Framework Core UI” (Figure 20). Of course, you can use the command line if you rather use that.

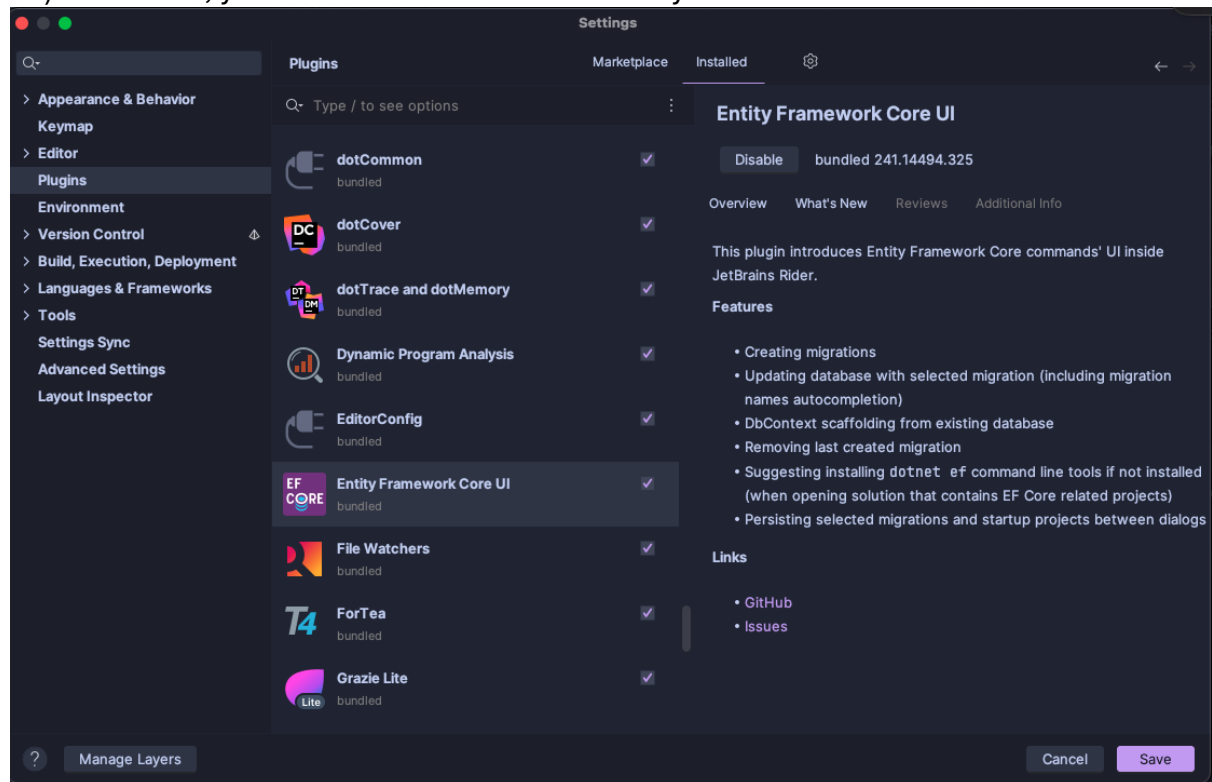


Figure 20 EF Core Plugin

With the tool in place, you can simply choose Add Migration from the Tools | Entity Framework Core menu (Figure 21).

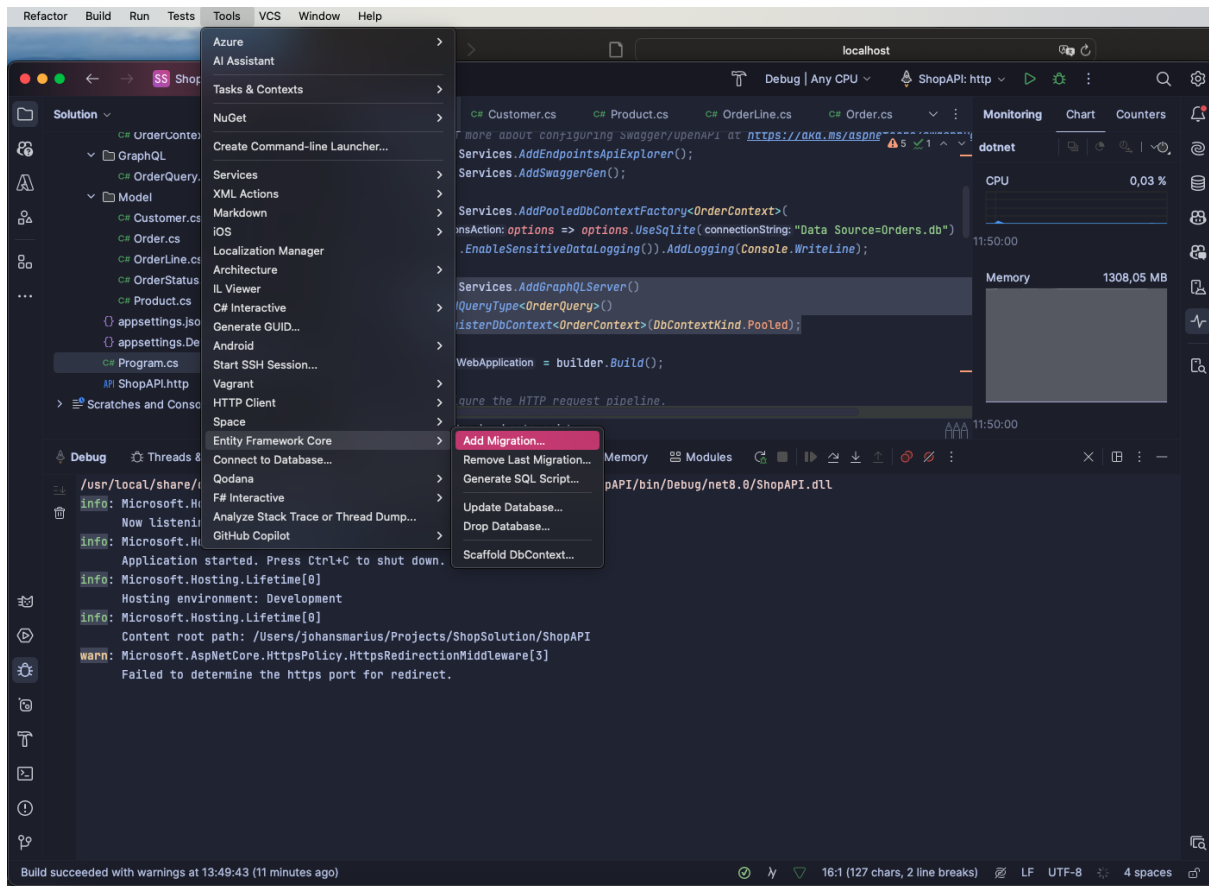


Figure 21 Add Migration

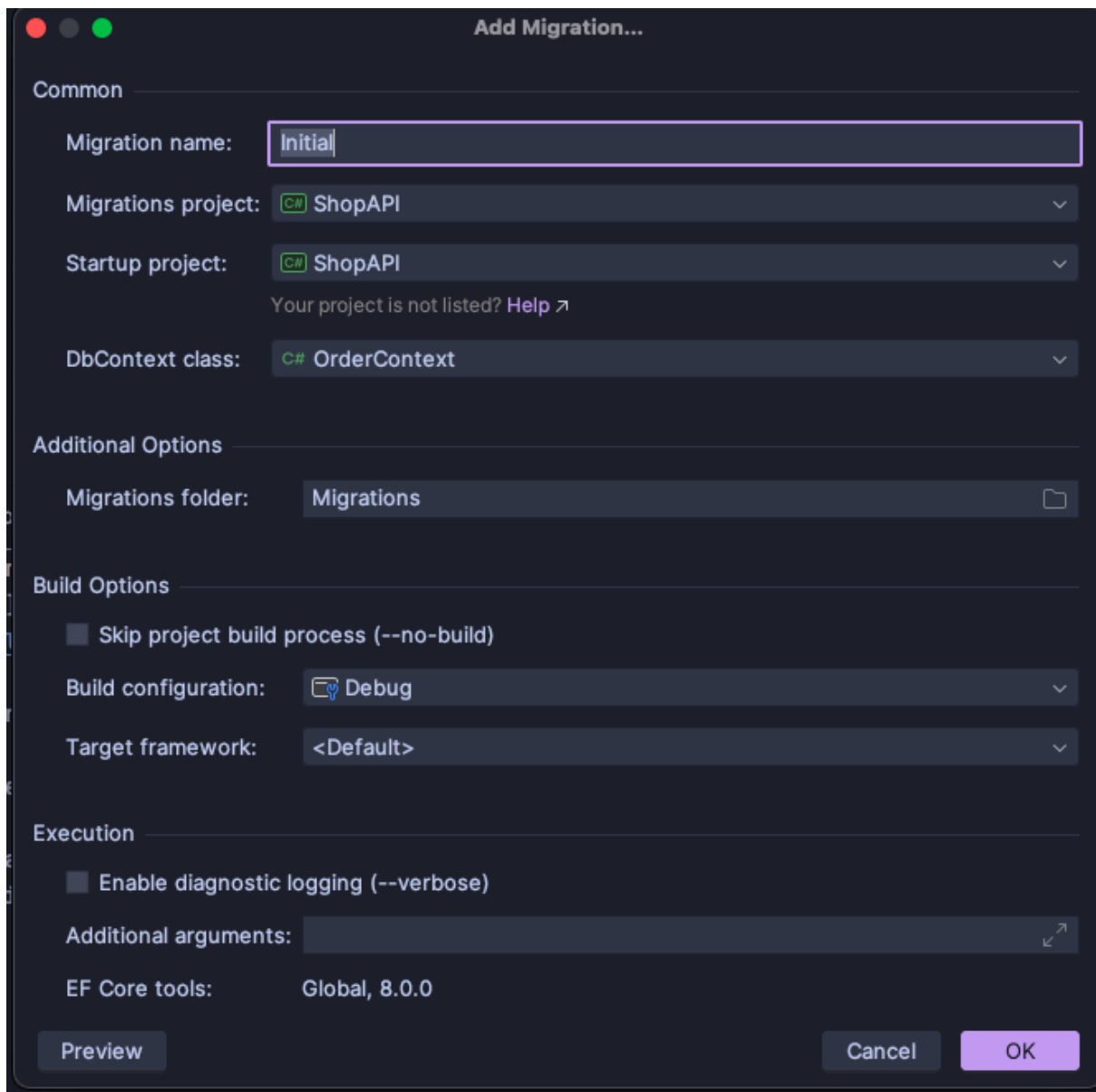


Figure 22 Options for migration.

You can just leave the default options and click “OK” (Figure 22). The migration is now created for you.

This migration must be pushed to a database, so we are going to do this by running the Update Database command (Figure 23).

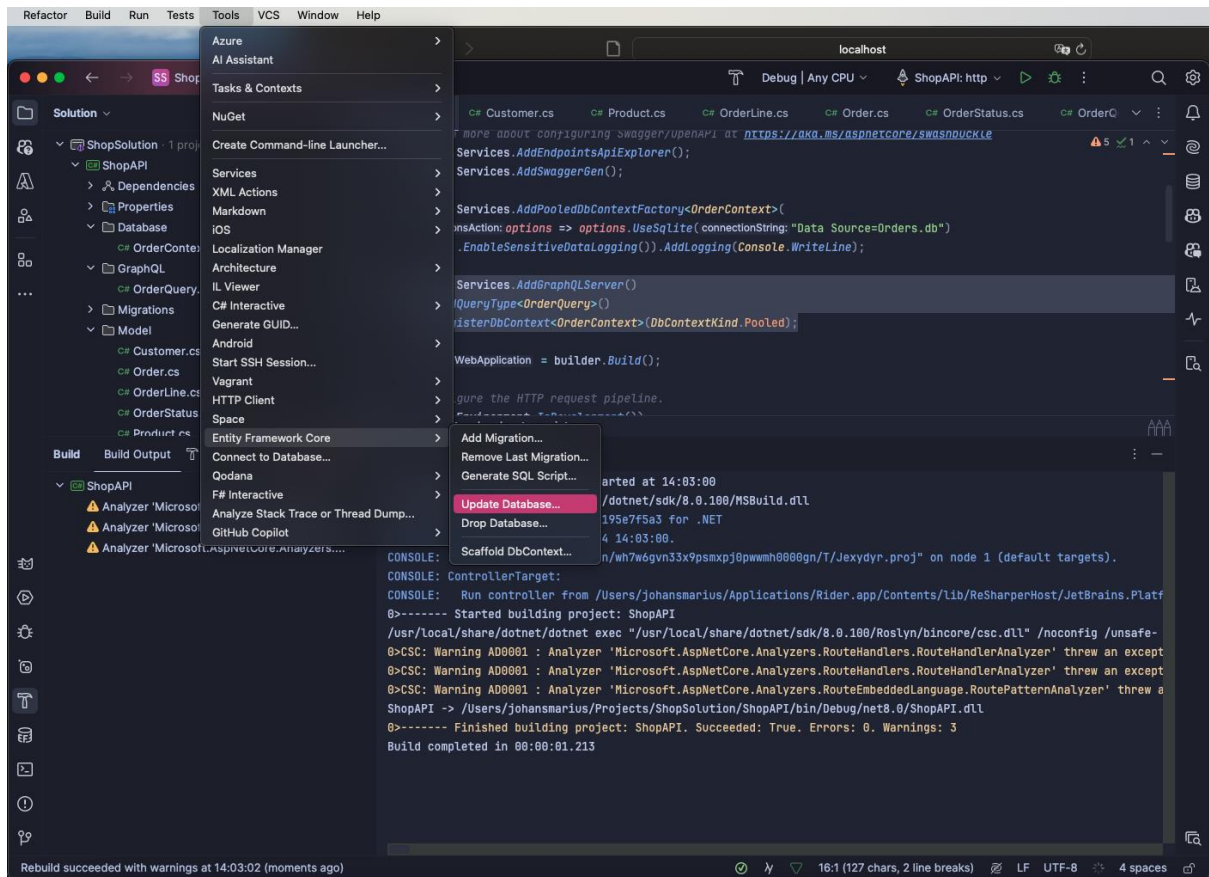


Figure 23 Update Database command.

In the solution the newly created database is now visible (Figure 24).

Verify that you do see the file Orders.db

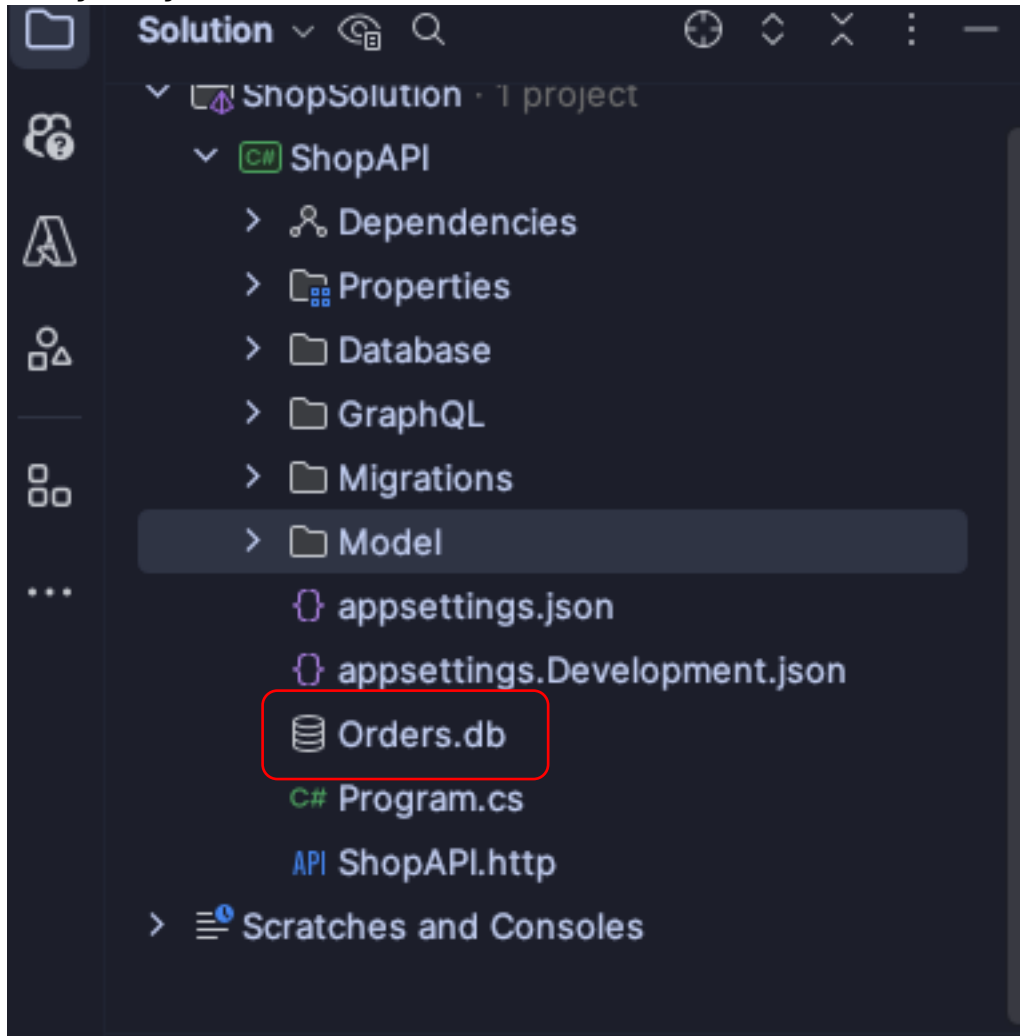


Figure 24 Created database.

We can now update our Query class to use the DbContext.

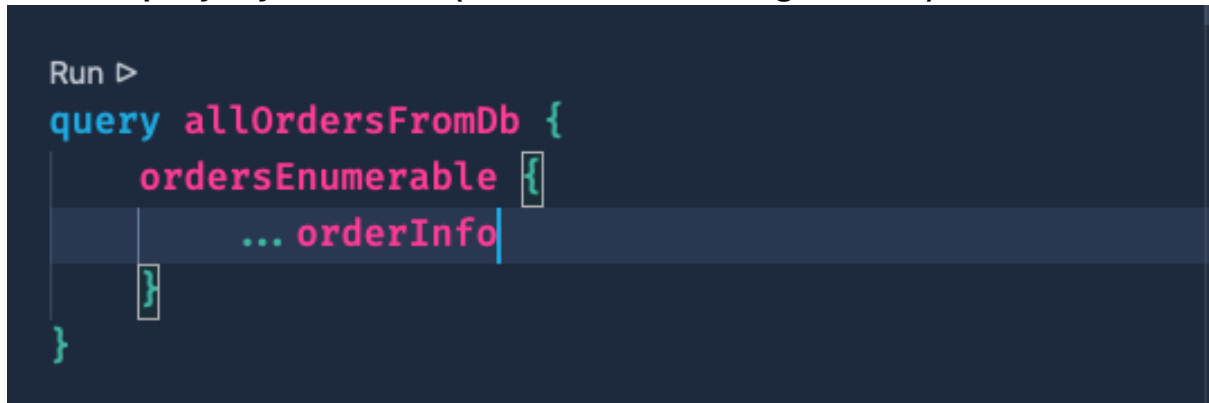
Add the following code to this class:

```
public IEnumerable<Order> GetOrdersEnumerable(OrderContext
orderContext) =>
    orderContext.Orders
        .Include(order =>
order.OrderLines).ThenInclude(orderline => orderline.Product)
        .Include(order => order.Customer);
```

With this code we can fetch all the information about our orders using our OrderContext. To be able to fetch all the dependent objects I include OrderLines, Product and Customer in this query.

If we run this code, we can use this new method to query our database (Figure 25).

Add this query to your test site (of course after running the code).

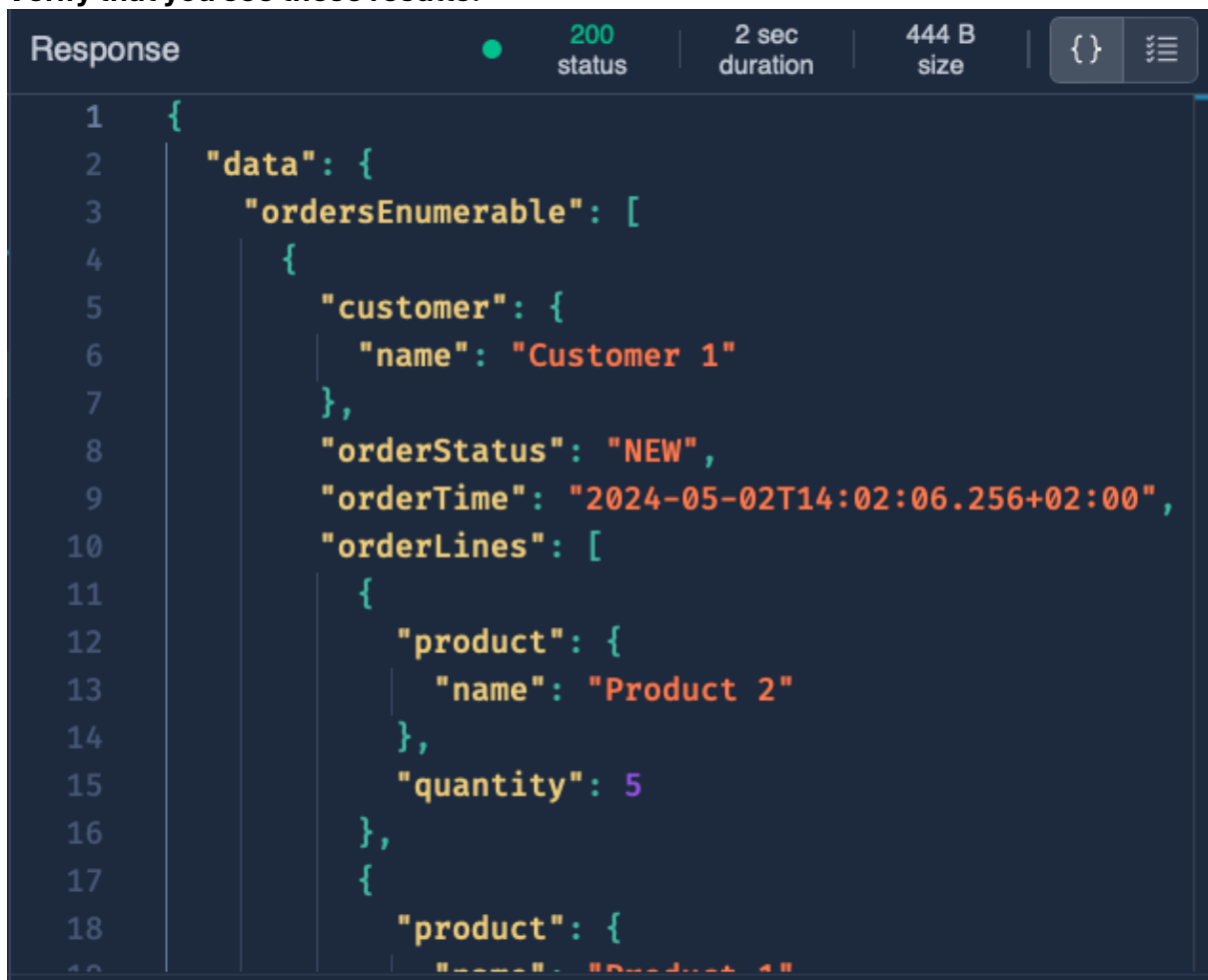


```
Run ▶
query allOrdersFromDb {
  ordersEnumerable {
    ... orderInfo
  }
}
```

Figure 25 Call query from db.

And if we run this query, we will see that the orders we seeded in the database are returned (Figure 26).

Verify that you see these results.

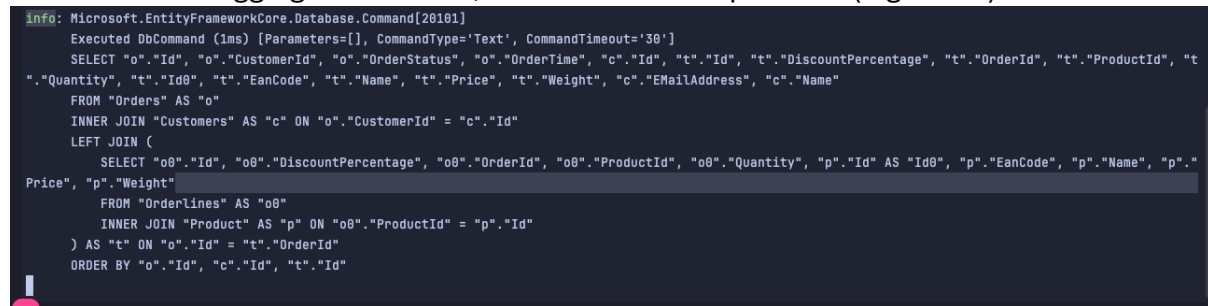


Response 200 status 2 sec duration 444 B size

```
1 {
2   "data": {
3     "ordersEnumerable": [
4       {
5         "customer": {
6           "name": "Customer 1"
7         },
8         "orderStatus": "NEW",
9         "orderTime": "2024-05-02T14:02:06.256+02:00",
10        "orderLines": [
11          {
12            "product": {
13              "name": "Product 2"
14            },
15            "quantity": 5
16          },
17          {
18            "product": {
19              "name": "Product 1"
20            }
21          }
22        ]
23      }
24    ]
25  }
26 }
```

Figure 26 Results from database.

So, the integration between HotChocolate and Entity Framework works quite well, but if we look at the logging information, we can still see a problem (Figure 27).



```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT "o"."Id", "o"."CustomerId", "o"."OrderStatus", "o"."OrderTime", "c"."Id", "t"."Id", "t"."DiscountPercentage", "t"."OrderId", "t"."ProductId", "t"."Quantity", "t"."Id8", "t"."EanCode", "t"."Name", "t"."Price", "t"."Weight", "c"."EmailAddress", "c"."Name"
      FROM "Orders" AS "o"
      INNER JOIN "Customers" AS "c" ON "o"."CustomerId" = "c"."Id"
      LEFT JOIN (
        SELECT "o8"."Id", "o8"."DiscountPercentage", "o8"."OrderId", "o8"."ProductId", "o8"."Quantity", "p"."Id" AS "Id8", "p"."EanCode", "p"."Name", "p"."Price", "p"."Weight"
        FROM "OrderLines" AS "o8"
        INNER JOIN "Product" AS "p" ON "o8"."ProductId" = "p"."Id"
      ) AS "t" ON "o"."Id" = "t"."OrderId"
      ORDER BY "o"."Id", "c"."Id", "t"."Id"
```

Figure 27 Logging for query.

All the fields from the the database are still retrieved from the database, so there is still overfetching taking place. Let us fix this.

Step 10: Fix overfetching from DB

Fortunately, HotChocolate has a good option to prevent overfetching. We do need to add a new method that returns an IQueryable for this.

Add the following code to your OrderQuery class.

```
public IQueryable<Order> GetOrders(OrderContext orderContext) =>
    orderContext.Orders;
```

This code simply returns the Orders in the Context. Because of the IQueryable behaviour the actual query that will be sent to the database can be changed by changing this result after it has been returned.

Add the attribute UseProjection above this method.

```
[UseProjection]
public IQueryable<Order> GetOrders(OrderContext orderContext) =>
    orderContext.Orders;
```

Adding projection support does require an additional step when registering the Query type in program.cs.

Add the AddProjections option after the registration of the DbContext.

```
builder.Services.AddGraphQLServer()
    .AddQueryType<OrderQuery>()
    .RegisterDbContext<OrderContext>(DbContextKind.Pooled)
    .AddProjections();
```

If we now run the code, we can add a new query to our console (Figure 28).

Add this query to your own site.


```

12
13 Run ▾
14 query allOrdersProjected {
15     orders {
16         ... orderInfo
17     }
18 }

```

Figure 28 Query for projection.

If you run this query the new method will be called.

And if we now look at the logging, we can see that the database is no longer overfetched (Figure 29).

```

info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (12ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT 1, "c"."Name", "o"."OrderStatus", "o"."OrderTime", "o"."Id", "c"."Id", "t"."c", "t"."Name", "t"."Quantity", "t"."Id", "t"."Id0"
      FROM "Orders" AS "o"
      INNER JOIN "Customers" AS "c" ON "o"."CustomerId" = "c"."Id"
      LEFT JOIN (
        SELECT 1 AS "c", "p"."Name", "o0"."Quantity", "o0"."Id", "p"."Id" AS "Id0", "o0"."OrderId"
        FROM "OrderLines" AS "o0"
        INNER JOIN "Product" AS "p" ON "o0"."ProductId" = "p"."Id"
      ) AS "t" ON "o"."Id" = "t"."OrderId"
      ORDER BY "o"."Id", "c"."Id", "t"."Id"

```

Figure 29 Logging after projection.

So now we not only do not transmit unnecessary information over the wire, but we also do not fetch this information from the database.

Step 11: Adding support for filtering (the right way)

In step 7 we added a parameter to a method just to be able to fetch a specific record. This functionality is supported in GraphQL out of the box by using filters.

All you have to do is add the `UseFiltering` attribute above our method.

```
[UseProjection]
[UseFiltering]
public IQueryable<Order> GetOrders(OrderContext orderContext) =>
    orderContext.Orders;
```

And add filtering support when registering the service.

```
builder.Services.AddGraphQLServer()
    .AddQueryType<OrderQuery>()
    .RegisterDbContext<OrderContext>(DbContextKind.Pooled)
    .AddProjections()
    .AddFiltering();
```

If you now run your code, we can make use of special filter option is the GraphQL language (Figure 30).

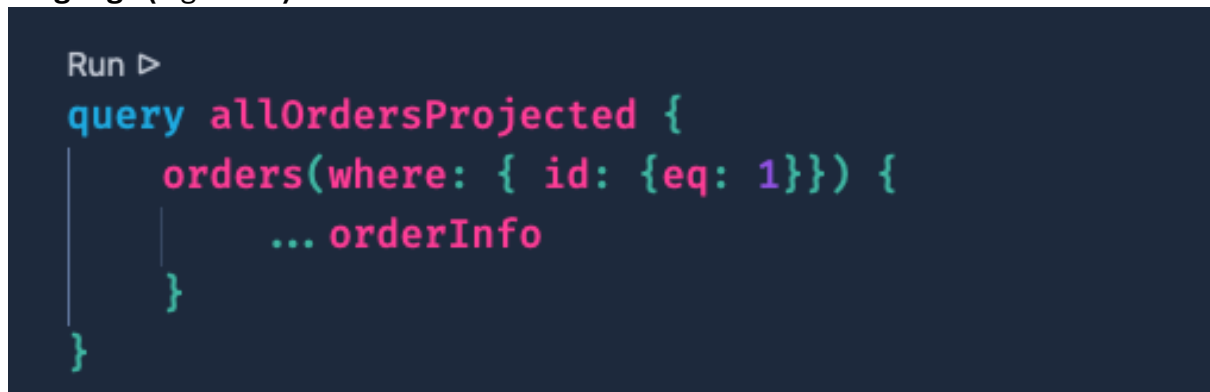
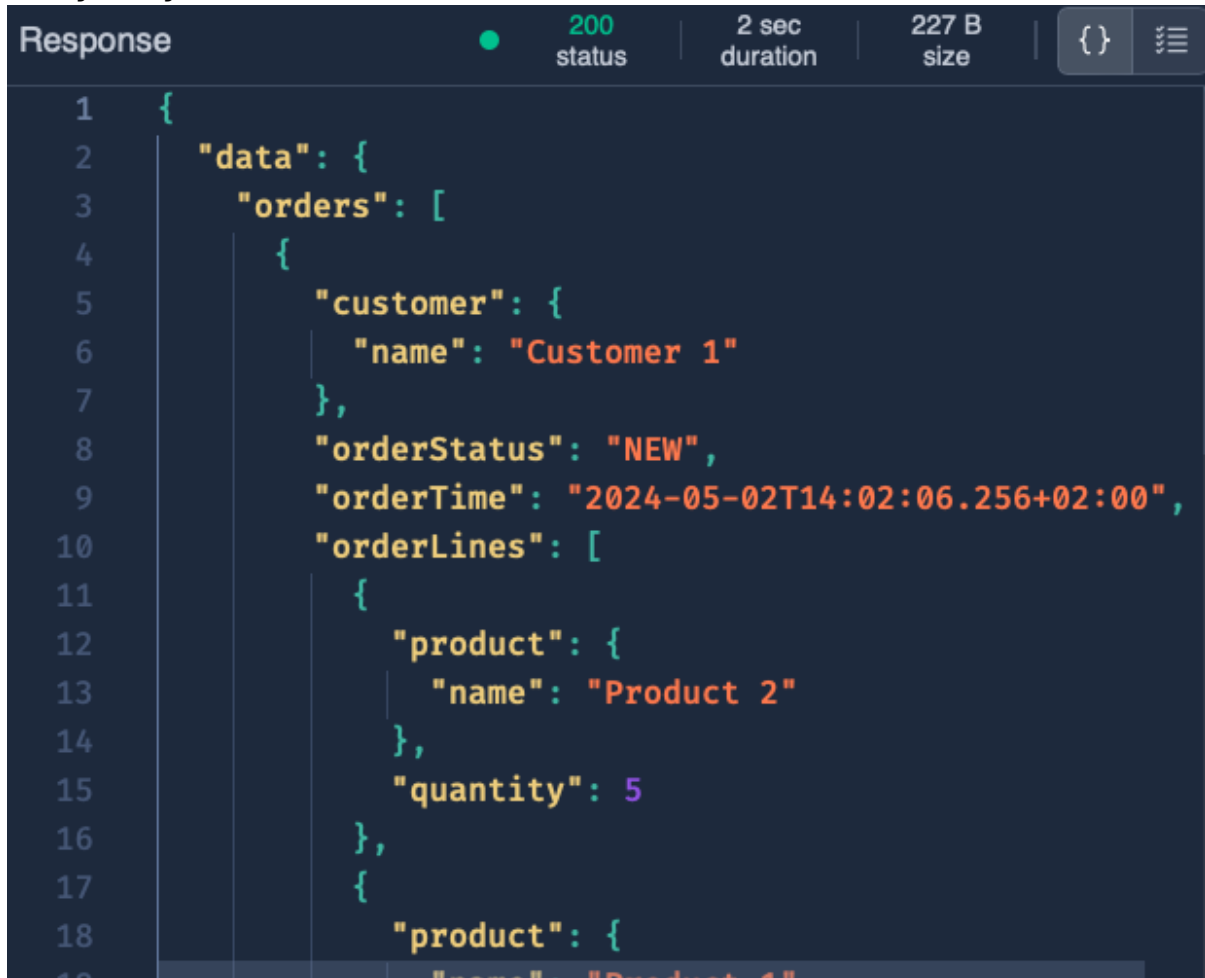


Figure 30 Query for projections.

Filtering can be specified by using the `where` keyword. If you want to know more about this feature and the options it supports you can visit <https://chillicream.com/docs/hotchocolate/v13/fetching-data/filtering>.

When we run this query, you can see that the filtering actually works (Figure 31).

Verify that you see the same results.



```
Response 200 status 2 sec duration 227 B size {}

1 {
2   "data": {
3     "orders": [
4       {
5         "customer": {
6           "name": "Customer 1"
7         },
8         "orderStatus": "NEW",
9         "orderTime": "2024-05-02T14:02:06.256+02:00",
10        "orderLines": [
11          {
12            "product": {
13              "name": "Product 2"
14            },
15            "quantity": 5
16          },
17          {
18            "product": {
19              "name": "Product 1"
20            }
21          }
22        ]
23      }
24    ]
25  }
26 }
```

Figure 31 Results after filtering.

Verify in your own solution that the filtering works.

And the logging shows that the filter is used in the generation of the SQL query as well (Figure 32).

```
Info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (15ms) [Parameters=[@__p_0='1'], CommandType='Text', CommandTimeout='30']
      SELECT 1, "c"."Name", "o"."OrderStatus", "o"."OrderTime", "o"."Id", "c"."Id", "t"."c", "t"."Name", "t"."Quantity", "t"."Id", "t"."Id0"
      FROM "Orders" AS "o"
      INNER JOIN "Customers" AS "c" ON "o"."CustomerId" = "c"."Id"
      LEFT JOIN (
        SELECT 1 AS "c", "p"."Name", "o0"."Quantity", "o0"."Id", "p"."Id" AS "Id0", "o0"."OrderId"
        FROM "Orderlines" AS "o0"
        INNER JOIN "Product" AS "p" ON "o0"."ProductId" = "p"."Id"
      ) AS "t" ON "o"."Id" = "t"."OrderId"
      WHERE "o"."Id" = @__p_0
      ORDER BY "o"."Id", "c"."Id", "t"."Id"
```

Figure 32 SQL command after projection.

Step 11: Adding support for sorting.

Many times, it can be easier to have the server sort the information for the client. HotChocolate supports this as well by using sorting.

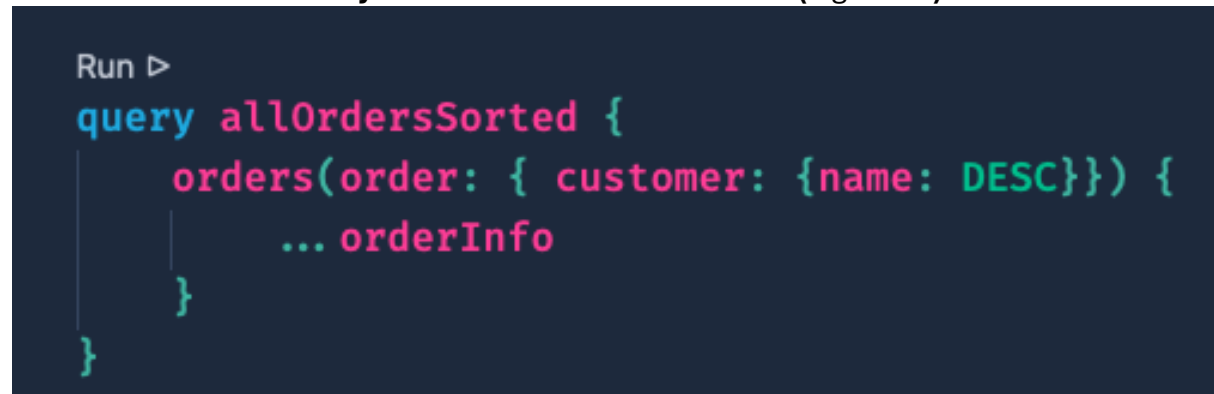
Add [UseSorting] above the GetOrders method.

```
[UseProjection]
[UseFiltering]
[UseSorting]
public IQueryable<Order> GetOrders(OrderContext orderContext) =>
    orderContext.Orders;
```

And add AddSorting to the registration.

```
builder.Services.AddGraphQLServer()
    .AddQueryType<OrderQuery>()
    .RegisterDbContext<OrderContext>(DbContextKind.Pooled)
    .AddProjections()
    .AddFiltering()
    .AddSorting();
```

Now run the code and try to sort the order information (Figure 33).



```
Run >
query allOrdersSorted {
  orders(order: { customer: { name: DESC}}) {
    ... orderInfo
  }
}
```

Figure 33 Sort query.

Verify that the second order is now listed first (Figure 34).

```
1  {
2    "data": {
3      "orders": [
4        {
5          "customer": {
6            "name": "Customer 2"
7          },
8          "orderStatus": "NEW",
9          "orderTime": "2024-05-02T14:02:06.256+02:00",
10         "orderLines": [
11           {
12             "product": {
13               "name": "Product 1"
14             },
15             "quantity": 7
16           },
17           {
18             "product": {
19               "name": "Product 2"
```

Figure 34 Result after sorting.

Step 12: Adding paging.

Many client applications do support paging in the user interface. So, it is good to provide support for paging in the service as well.

Add the [UseOffsetPaging] attribute above the GetOrders method. Please pay attention to the order of the attributes. **The paging attribute must be the first one in the list.**

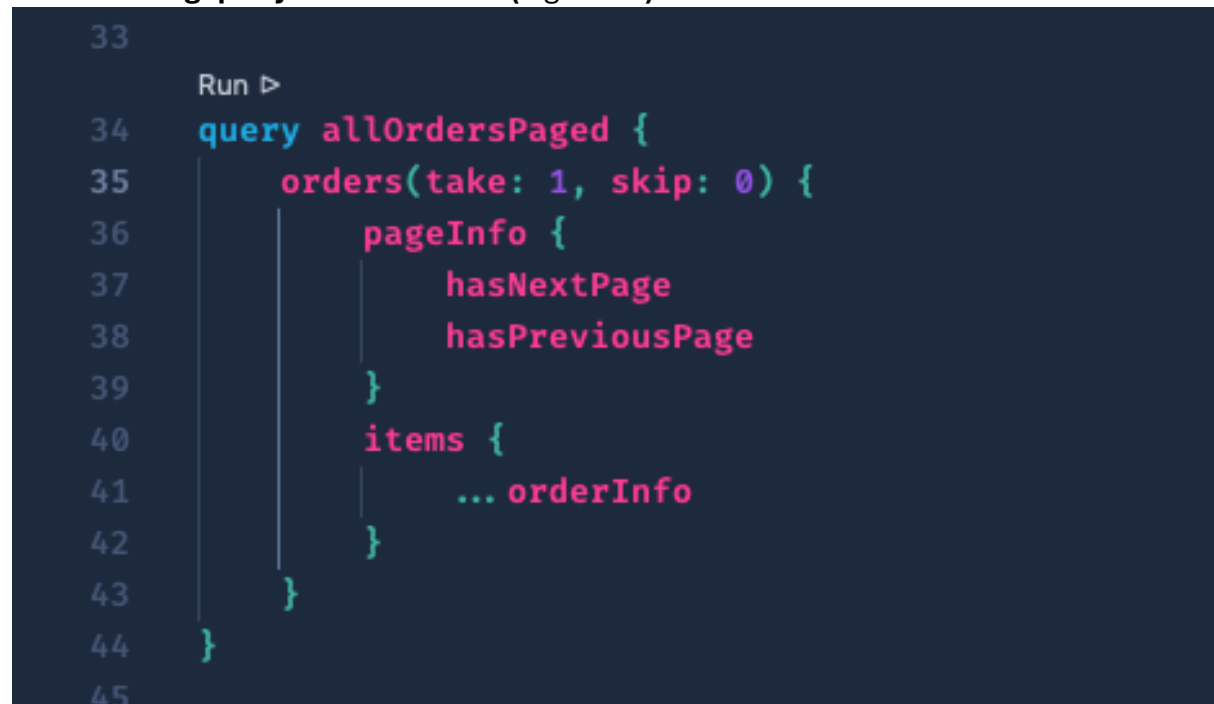
```
[UseOffsetPaging]
[UseProjection]
[UseFiltering]
[UseSorting]
public IQueryable<Order> GetOrders(OrderContext orderContext) =>
    orderContext.Orders;
```

In the service registration we do need a somewhat different setting.

Add the SetPagingOptions code to your registration.

```
builder.Services.AddGraphQLServer()  
    .AddQueryType<OrderQuery>()  
    .RegisterDbContext<OrderContext>(DbContextKind.Pooled)  
    .AddProjections()  
    .AddFiltering()  
    .AddSorting().AddFiltering()  
    .SetPagingOptions(new PagingOptions() { DefaultPageSize = 1,  
MaxPageSize = 1 });
```

This query can of course be used in the test environment, so start the app and add the following query to the console (Figure 35).



```
33  
34  Run ▶  
34  query allOrdersPaged {  
35      orders(take: 1, skip: 0) {  
36          pageInfo {  
37              hasNextPage  
38              hasPreviousPage  
39          }  
40          items {  
41              ... orderInfo  
42          }  
43      }  
44  }  
45
```

Figure 35 Query for paging.

The result type has now been changed. The paging info is part of every result because the paging attribute is on top of the other ones. The items that are returned can now be found in the items collection. If we run this query the following result will be given. Because we only take 1 record from the set, has NextPage shows true (Figure 36).

Verify that you get the same result.

```
1  {
2    "data": {
3      "orders": {
4        "pageInfo": {
5          "hasNextPage": true,
6          "hasPreviousPage": false
7        },
8        "items": [
9          {
10           "customer": {
11             "name": "Customer 1"
12           },
13           "orderStatus": "NEW",
14           "orderTime": "2024-05-02T14:02:06.256+02:00",
15           "orderLines": [
16             {
17               "product": {
18                 "name": "Product 2"
19               },
20               "quantity": 5
21             },

```

Figure 36 Results after paging.

Because of the change in attributes all the queries in the console have to be changed to include at least the items section. This is an easy step to do, so I leave this to you as attendee.

The End

Congratulations! You have now created your own GraphQL service using HotChocolate and .NET 8. Of course there is still a lot more to learn, but I hope you now feel confident enough to start further experiments.

Happy coding my friends!