# Entity Framework Core grouping

|  |
| --- |
| Note this should had been in part 4 of the repository branch but got messed up and went to part 3. Way too difficult to remedy so it’s here. |

Grouping is one of the most powerful capabilities of LINQ. The following examples show how to group data in various ways:

* By a single property
* By a compound key (multiple properties)
* By the first letter of a string property
* By a computed numeric range
* By Boolean predicate or other expression.

Let’s look at a simple SQL SELECT/GROUP-BY

In our north wind database, orders table, customers want to know who has the most orders. The following query can be used.

|  |
| --- |
| SELECT  EmployeeID,  COUNT(EmployeeID) AS Counter  FROM  NorthWind2020.dbo.Orders  GROUP BY  EmployeeID  ORDER BY  Counter DESC; |

Results (note the last row with a NULL)

|  |
| --- |
| 4,156  3,127  1,123  8,104  2,96  7,72  6,67  9,43  5,42  NULL,0 |

Let’s get rid of the NULL record(s).

|  |
| --- |
| SELECT  EmployeeID,  COUNT(EmployeeID) AS Counter  FROM  dbo.Orders  WHERE  EmployeeID **is not null**  GROUP BY  EmployeeID  ORDER BY  Counter DESC; |

To return the employee identifier and count of orders using a data provider.

* Karen will supply
  + Add NuGet package [System.Data.SqlClient](https://www.nuget.org/packages/System.Data.SqlClient/4.8.2?_src=template)
  + NorthWindCoreUnitTest.Classes.SqlOperations in GitHub
  + A unit test class with a test method
  + Walkthrough second and third bullet

Now let’s do the same with EF Core.

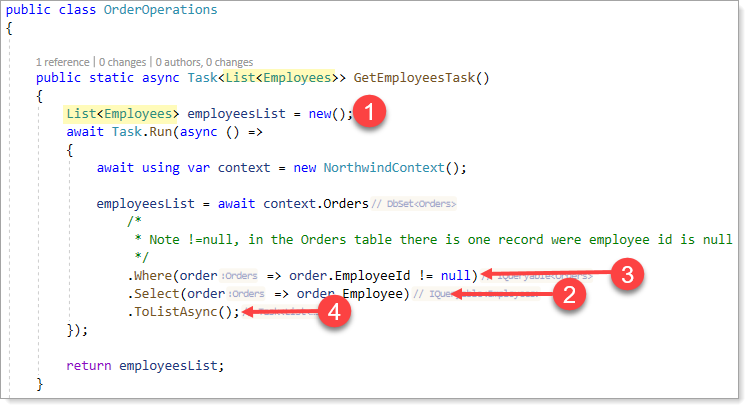
* Karen will provide OrderOperations class code

First step, get a list of employee in a method separate from performing a group-by which means this method is reusable for other operations.

In the database table, there is one record without an employee identifier which should never happen but we know there is a chance so Karen setup for failure if a developer does not assert for null.

OrderOperations class

1. Create an instance for a list of Employees
2. Query Orders
3. Where employee id is not null
4. Perform the query asynchronously



Next, the following method accepts a list of employees obtained from GetEmployeesTask method above

1. Group by employee identifier
2. Perform a descending order on count which returns the count of elements in a sequence
3. Ask for the first employee using FirstOrDefault. Note with FirstOrDefault a check can be done for null while a sister extension First if there are no elements in a sequence will throw a runtime exception.
4. Return a IGrouping<int, Employees> where int is the count of orders for the employee with the most orders

Time to write unit test

1. Create a new class in the root of the project NorthWindCoreUnitTest named
2. Change the class signature to public partial class OrderTests : TestBase hint: needs a using statement, look at the other test classes.
3. Add the following attribute [TestClass] hint: needs a using statement, look at the other test classes.
4. Add the following test method (Karen will walk through the code) and supply SqlOperations code.

[TestMethod]

[TestTraits(Trait.GroupingEntityFramework)]

public async Task GroupByEmployeeIdentifierGetHighCountInOrders()

{

    List<Employees> employeeList = await OrderOperations.GetEmployeesTask();

    IGrouping<int, Employees> employee = OrderOperations.EmployeeMostOrders(employeeList);

    Assert.IsTrue(employee != default);

    Debug.WriteLine($"Order count: {employee.Count()} employee id: {employee.Key}");

    SqlOperations.Server = ".\\SQLEXPRESS";

    SqlOperations.Database = "NorthWind2020";

    //    |   |

    //    V   V    <- Discards

    var ( \_ , \_ , dictionary ) = SqlOperations.EmployeeMostOrders();

    Assert.AreEqual(employee.Count(), dictionary.FirstOrDefault().Value);

}