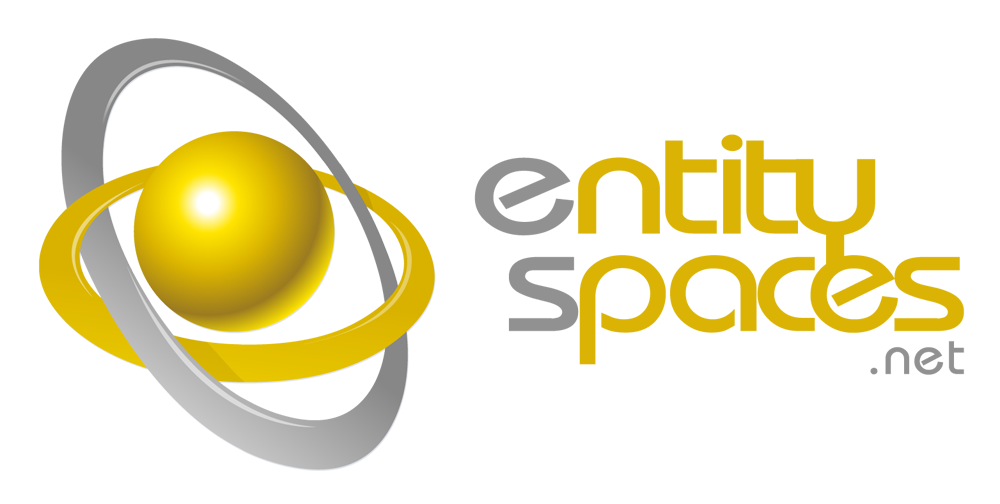
Dynamic Query

Using the dynamic query API

February 11, 2012  
EntitySpaces, LLC  
The EntitySpaces Team

**[](http://www.entityspaces.net/)**

Persistence Layer and Business Objects for Microsoft .NET

Contents

[Queries 4](#_Toc317317332)

[Jumping Right in 4](#_Toc317317333)

[Select Clause 5](#_Toc317317334)

[Select 5](#_Toc317317335)

[Select TOP 5](#_Toc317317336)

[Select DISTINCT 5](#_Toc317317337)

[SelectAll 6](#_Toc317317338)

[SelectAllExcept() 7](#_Toc317317339)

[Selecting and Aliasing Columns 7](#_Toc317317340)

[Selecting Columns with Arithmetic Operators to Create Derived Columns 8](#_Toc317317341)

[Where Clause 9](#_Toc317317342)

[A Simple Where Clause 9](#_Toc317317343)

[Where using AND / OR operators 9](#_Toc317317344)

[Where Sub-operators 10](#_Toc317317345)

[OrderBy Clause 10](#_Toc317317346)

[GroupBy Clause 10](#_Toc317317347)

[Joins 11](#_Toc317317348)

[Any, All and Some 12](#_Toc317317349)

[Paging 13](#_Toc317317350)

[Using PageNumber and PageSize for Paging 13](#_Toc317317351)

[Using Skip and Take for Paging 13](#_Toc317317352)

[Prefetching 14](#_Toc317317353)

[Advanced Samples 15](#_Toc317317354)

[Selecting the Count of a Table 15](#_Toc317317355)

[Selecting the Count of a Table with a Where Clause 15](#_Toc317317356)

[LastQuery and Parse 15](#_Toc317317357)

[Select Case When Then (Sample 1) 16](#_Toc317317358)

[Select Case When Then (Sample 2) 17](#_Toc317317359)

[Correlated Sub-Query 17](#_Toc317317360)

[Where In with Nested Sub-Query 18](#_Toc317317361)

[From Sub-Query 19](#_Toc317317362)

[Join On Sub-Query 20](#_Toc317317363)

[Raw SQL Injection 21](#_Toc317317364)

[Select against a SQL View rather than a SQL Table 22](#_Toc317317365)

[Operators 23](#_Toc317317366)

[Comparison Operators 23](#_Toc317317367)

[Arithmetic Operators 23](#_Toc317317368)

[Logical-Operators 23](#_Toc317317369)

[Sub-Operators 23](#_Toc317317370)

# Queries

The EntitySpaces Dynamic Query API is modeled after the SQL language syntax. The Dynamic Query API allows you to easily create queries using Intellisense that are clear and concise. Before we jump into specific examples let’s take a look at a full blown query to give you a feel for the syntax.

## Jumping Right in

Take a look at this query that does an inner join.

// Query for the Join

OrderDetailsQuery odq = new OrderDetailsQuery("odq");

// Orders with discounted items

OrdersQuery oq = new OrdersQuery("oq");

oq.Select(oq.OrderID, odq.Discount);

oq.InnerJoin(odq).On(oq.OrderID == odq.OrderID);

oq.Where(odq.Discount > 0);

OrdersCollection collection = new OrdersCollection();

collection.Load(oq);

**The Resulting SQL**

SELECT oq.[OrderID],odq.[Discount]  
FROM [Orders] oq  
INNER JOIN [Order Details] odq ON oq.[OrderID] = odq.[OrderID]  
WHERE odq.[Discount] > @Discount1

Notice that we return the OrderDetails.Discount column which is not a column in our Orders table. If you were to bind the OrdersCollection to a grid the “Discount” extra column would appear to the grid as if it were a strongly typed property in the OrdersCollection. EntitySpaces takes care of this for you.

Also, the Collection.Load() method returns a Boolean value as do all methods that load data in EntitySpaces. Typically, you will write code like this:

OrdersCollection collection = new OrdersCollection();

if (collection.Load(oq))

{

// Then at least one record was loaded

}

For the sake of brevity our sample code will not use the “if” statement syntax in each sample.

## Select Clause

It is important to note that If you do not use one of the following select methods the default is to select all columns and it is done with “SELECT \* FROM [MyTable]”. If you want all columns you don’t have to call any Select() method.

### Select

Select will return only the columns you explicitly list.

EmployeesQuery q = new EmployeesQuery("e");

q.**Select**(q.EmployeeID);

**Resulting SQL**

SELECT e.[EmployeeID] FROM [Employees] e

### Select TOP

Select the TOP n records in any query is simple.

EmployeesQuery q = new EmployeesQuery("e");

q.es.**Top** = 5;

q.Select(q.EmployeeID);

**Resulting SQL**

SELECT TOP 5 e.[EmployeeID] FROM [Employees] e

### Select DISTINCT

The DISTINCT syntax is as follows.

EmployeesQuery q = new EmployeesQuery("e");

q.es.**Distinct** = true;

q.Select(q.EmployeeID);

**Resulting SQL**

SELECT DISTINCT e.[EmployeeID] FROM [Employees] e

### SelectAll

SelectAll columns explicitly by name.

EmployeesQuery q = new EmployeesQuery("e");

q.**SelectAll**();

**Resulting SQL**

SELECT e.[EmployeeID],  
 e.[LastName],  
 e.[FirstName],  
 e.[Title],  
 e.[TitleOfCourtesy],  
 e.[BirthDate],  
 e.[HireDate],  
 e.[Address],  
 e.[City],  
 e.[Region],  
 e.[PostalCode],  
 e.[Country],  
 e.[HomePhone],  
 e.[Extension],  
 e.[Photo],  
 e.[Notes],  
 e.[ReportsTo],  
 e.[PhotoPath]  
FROM [Employees] e

### SelectAllExcept()

SelectAllExcept will select all columns except for those you pass in as parameters. The example below is the perfect case for just this approach. We want to select our Employee data but not bring back the very large “Photo” image column.

EmployeesQuery q = new EmployeesQuery("e");

q.**SelectAllExcept**(q.Photo);

**Resulting SQL**

SELECT e.[EmployeeID],  
 e.[LastName],  
 e.[FirstName],  
 e.[Title],  
 e.[TitleOfCourtesy],  
 e.[BirthDate],  
 e.[HireDate],  
 e.[Address],  
 e.[City],  
 e.[Region],  
 e.[PostalCode],  
 e.[Country],  
 e.[HomePhone],  
 e.[Extension],  
 e.[Notes],  
 e.[ReportsTo],  
 e.[PhotoPath]  
FROM [Employees] e

### Selecting and Aliasing Columns

You can use the As() method to alias a column.

EmployeesQuery q = new EmployeesQuery("e");

q.Select(q.FirstName.**As**("MyAlias"));

**Resulting SQL**

SELECT e.[FirstName] AS 'MyAlias' FROM [Employees] e

### Selecting Columns with Arithmetic Operators to Create Derived Columns

You can use your native language operators to create derived columns on the fly. For instance, let’s concatenate the FirstName and LastName columns into a new column aliased as “FullName” using the + operator.

EmployeesQuery q = new EmployeesQuery("e");

q.Select((q.LastName **+** ", " **+** q.FirstName).As("FullName"));

**Resulting SQL**

SELECT ((e.[LastName] + ', ') + e.[FirstName]) AS 'FullName'   
FROM [Employees] e

You can also use operators to perform mathematical operations in a select. This example brings back a column aliased as “Total Price” which is the result of multiplying the Quanty and UnitPrice column using the \* operator.

OrderDetailsQuery od = new OrderDetailsQuery("od");

od.Select((od.Quantity **\*** od.UnitPrice).As("Total Price"));

**Resulting SQL**

SELECT (od.[Quantity] \* od.[UnitPrice]) AS 'Total Price'  
FROM [Order Details] od

## Where Clause

The Where clause can be most any expression you dream up and serves to limit the number of records returned in your query.

### A Simple Where Clause

This is a very simple where clause. Notice that the value 32 is passed in as parameter. This prevents SQL injection techniques. Also notice that we can use the natural language features such as “>” for greater than.

EmployeesQuery q = new EmployeesQuery("e");

q.**Where**(q.EmployeeID > 32);

**Resulting SQL**

SELECT \* FROM [Employees] e WHERE e.[EmployeeID] **>** @EmployeeID1

### Where using AND / OR operators

You can use your natural languages and / or operators. Notice our use of the && operator below. In VB.NET you would simply use the ‘And’ keyword.

EmployeesQuery q = new EmployeesQuery("e");

q.Where(q.EmployeeID > 4 **&&** q.EmployeeID < 10);

**Resulting SQL**

SELECT \* FROM [Employees] e  
WHERE (e.[EmployeeID] > @EmployeeID1 **AND** e.[EmployeeID] < @EmployeeID2)

If you need to get more complicated you can use parenthesis as shown below. In this example we use both the && and the || or operators.

EmployeesQuery q = new EmployeesQuery("e");

q.Where((q.EmployeeID > 4 **&&** q.EmployeeID < 10) **||** q.EmployeeID == 100);

**Resulting SQL**

SELECT \* FROM [Employees] e  
WHERE ((e.[EmployeeID] > @EmployeeID1 **AND** e.[EmployeeID] < @EmployeeID2)  
 **OR** e.[EmployeeID] = @EmployeeID3)

### Where Sub-operators

There are many sub-operators specific to the Where clause. In the example below we use the IsNotNull() operator is used to return Employees whose LastName is not null.

EmployeesQuery q = new EmployeesQuery("e");

q.Where(q.LastName.**IsNotNull**());

**Resulting SQL**

SELECT \* FROM [Employees] e WHERE e.[LastName] **IS NOT NULL**

See the list of Sub-Operators in the "Operators" section at the end of this section.

## OrderBy Clause

Use the OrderBy clause to sort the data coming back from the database.

EmployeesQuery q = new EmployeesQuery("e");

q.Where(q.LastName.Like("%a%"));

q.**OrderBy**(q.LastName.Descending);

**Resulting SQL**

SELECT \* FROM [Employees] e WHERE e.[LastName] LIKE @LastName1  
**ORDER BY** e.[LastName] DESC

Ordering by multiple columns.

EmployeesQuery q = new EmployeesQuery("e");

q.Where(q.LastName.Like("%a%"));

q.**OrderBy**(q.LastName.Descending, q.FirstName.Ascending);

**Resulting SQL**

SELECT \* FROM [Employees] e WHERE e.[LastName] LIKE @LastName1  
**ORDER BY** e.[LastName] DESC, e.[FirstName] ASC

## GroupBy Clause

The group by clause is used to group by columns and rollup sub totals and alike.

OrderDetailsQuery od = new OrderDetailsQuery("od");

od.Select(od.OrderID, (od.UnitPrice \* od.Quantity).Sum().As("OrderTotal"));

od.**GroupBy**(od.OrderID);

**Resulting SQL**

SELECT od.[OrderID], *Sum*((od.[UnitPrice] \* od.[Quantity])) AS 'OrderTotal'  
FROM [Order Details] od  
**GROUP BY** od.[OrderID]

## Joins

The Dynamic Query API supports InnerJoin, LeftJoin, RightJoin, and FullJoin.

// Query for the Join

OrderDetailsQuery oq = new OrderDetailsQuery("oq");

OrdersQuery o = new OrdersQuery("o");

o.Select(o, oq.Discount); // notice we pass in 'o' which means o.\*

o.InnerJoin(oq).On(o.OrderID == oq.OrderID);

OrdersCollection coll = new OrdersCollection();

if (coll.Load(o))

{

foreach (Orders order in coll)

{

// We only need to access ‘Discount’ this way because it doesn't belong to our Order class

string discount = (string)order.GetColumn(OrderDetailsMetadata.ColumnNames.Discount);

}

}

**Resulting SQL**

SELECT o.\*, oq.[Discount]  
FROM [Orders] o  
**INNER JOIN** [Order Details] oq ON o.[OrderID] = oq.[OrderID]

Notice that we bring back the OrderDetails.Discount column in our OrdersQuery. If we want to access the Discount column we have no strongly typed property in our Orders entity to do so. Thus, we simply use the GetColumn() method to access the Discount column.

Finally, we encourage you to never use hard coded string names. Notice how we use the OrderDetailsMeta.ColumnNames.Discount property. The reason we encourage this is if you were to drop the Discount column and regenerate you will get a compile error as opposed to a runtime crash if you were to use a hard coded string.

## Any, All and Some

The Dynamic Query API supports using Any, All, and Some. They are accessible on he .es property.

// SubQuery using the ALL operator.

// Get a list of customers whose zip code

// is greater than all the customers in Oregon.

CustomersQuery cq2 = new CustomersQuery("c2");

cq2.es.**All** = true;

cq2.Select(cq2.PostalCode);

cq2.Where(cq2.Region == "OR");

CustomersQuery cq1 = new CustomersQuery("c1");

cq1.Select(cq1.CustomerID, cq1.CompanyName, cq1.PostalCode);

cq1.Where(cq1.PostalCode > cq2);

CustomersCollection coll = new CustomersCollection();

coll.Load(cq1);

**Resulting SQL**

SELECT c1.[CustomerID],  
 c1.[CompanyName],  
 c1.[PostalCode]  
FROM [Customers] c1  
WHERE c1.[PostalCode] > **ALL** (SELECT c2.[PostalCode]  
 FROM [Customers] c2  
 WHERE c2.[Region] = @Region1)

## Paging

### Using PageNumber and PageSize for Paging

EmployeesQuery q = new EmployeesQuery("e");

q.es.**PageSize** = 20;

q.es.**PageNumber** = 5;

q.OrderBy(q.HireDate.Descending);

**Resulting SQL**

Note that is this the SQL 2005 syntax, which is pretty ugly for a simple paging request.

WITH [withStatement]  
 AS (SELECT \*, **Row\_number**() OVER( ORDER BY e.[HireDate] DESC) AS ESRN  
 FROM [Employees] e)  
SELECT \*  
FROM [withStatement]  
WHERE ESRN BETWEEN 81 AND 100  
ORDER BY ESRN ASC

### Using Skip and Take for Paging

EmployeesQuery q = new EmployeesQuery("e");

q.**Take**(20);

q.**Skip**(5);

q.OrderBy(q.HireDate.Descending);

**Resulting SQL**

SELECT \* FROM [Employees] e ORDER BY e.[HireDate] DESC   
OFFSET 5 ROWS FETCH NEXT 20 ROWS ONLY

## Prefetching

Prefetching enables you to, in a single trip to the database, fetch an entire hierarchical set of data from the database. Prefetching can be much faster than using the lazy load hierarchical model. However, often times the lazy loading approach is the better approach as you only pay for what you access.

This example fetches all of the Employees, each employees Orders, and each orders OrderDetails in a single query.

EmployeesQuery q = new EmployeesQuery("e");

q.Prefetch<OrdersQuery>(Employees.Prefetch\_OrdersCollectionByEmployeeID);

q.Prefetch<OrderDetailsQuery>(Employees.Prefetch\_OrdersCollectionByEmployeeID, Orders.Prefetch\_OrderDetailsCollectionByOrderID);

// Load It

EmployeesCollection coll = new EmployeesCollection();

coll.Load(q);

// This code will not lazy load any data, it's already been loaded

foreach (Employees emp in coll)

{

foreach (Orders order in emp.OrdersCollectionByEmployeeID)

{

foreach (OrderDetails detail in order.OrderDetailsCollectionByOrderID)

{

}

}

}

Prefetching allows you some customization. The following is an example that uses a Where clause.

// The Main Query

EmployeesQuery q = new EmployeesQuery("e");

q.Where(q.EmployeeID < 5);

// The OrdersCollection

OrdersQuery o1 = q.Prefetch<OrdersQuery>(Employees.Prefetch\_OrdersCollectionByEmployeeID);

EmployeesQuery emp1 = o1.GetQuery<EmployeesQuery>();

o1.Where(emp1.EmployeeID < 5 && o1.ShipCity == "Indianapolis");

// The OrdersDetailsCollection

OrderDetailsQuery od = q.Prefetch<OrderDetailsQuery>(Employees.Prefetch\_OrdersCollectionByEmployeeID, Orders.Prefetch\_OrderDetailsCollectionByOrderID);

EmployeesQuery emp2 = od.GetQuery<EmployeesQuery>();

OrdersQuery o2 = od.GetQuery<OrdersQuery>();

od.Where(emp2.EmployeeID < 5 && o2.ShipCity == "Indianapolis");

// Load It

EmployeesCollection coll = new EmployeesCollection();

coll.Load(q);

Granted, the above is pretty complicated. Improving the prefetch syntax is going to be a major part of our second release of EntitySpaces 2012.

## Advanced Samples

These samples are more advanced and show some or richness of the Dynamic Query API.

### Selecting the Count of a Table

EmployeesQuery q = new EmployeesQuery("e");

q.es.CountAll = true;

int count = q.ExecuteScalar<int>();

**Resulting SQL**

**SELECT** **COUNT(\*)** **AS** 'Count' **FROM** **[**Employees**]** e

### Selecting the Count of a Table with a Where Clause

EmployeesQuery q = new EmployeesQuery("e");

q.es.CountAll = true;

q.Where(q.FirstName.Like("%a%"));

int count = q.ExecuteScalar<int>();

**Resulting SQL**

**SELECT** **COUNT(\*)** **AS** 'Count' **FROM** **[**Employees**]** e **WHERE** e**.[**FirstName**]** **LIKE** @FirstName1

### LastQuery and Parse

#### LastQuery

The text of each query is available after the query has been executed (even if a SQL exception occurs) via the query.es.LastQuery property. The code below shows how to access the LastQuery.

EmployeesQuery q = new EmployeesQuery("e");

q.Where(q.FirstName.Like("%a%"));

EmployeesCollection coll = new EmployeesCollection();

coll.Load(q);

string lastQuery = q.es.**LastQuery**;

#### Parse

Parse can be used to create the syntax that a query will execute without hitting the database. The should be used for debugging purposes only.

EmployeesQuery q = new EmployeesQuery("e");

q.Where(q.FirstName.Like("%a%"));

string sql = q.**Parse**();

### Select Case When Then (Sample 1)

OrderDetailsQuery oq = new OrderDetailsQuery();

oq.Select

(

oq.Quantity,

oq.UnitPrice,

oq.UnitPrice

.**Case**()

.**When**(oq.Quantity < 50).Then(oq.UnitPrice)

.**When**(oq.Quantity >= 50 && oq.Quantity < 70).Then(oq.UnitPrice \* .90)

.**When**(oq.Quantity >= 70 && oq.Quantity < 99).Then(oq.UnitPrice \* .80)

.**Else**(oq.UnitPrice \* .70)

.**End**().As("Adjusted Unit Price")

).OrderBy(oq.Quantity.Descending);

OrderDetailsCollection coll = new OrderDetailsCollection();

coll.Load(oq);

**Resulting SQL**

SELECT [Quantity], [UnitPrice],[Adjusted Unit Price] =   
 CASE WHEN [Quantity] < @Quantity1   
 THEN [UnitPrice]  
 WHEN ( [Quantity] >= @Quantity2 AND [Quantity] < @Quantity3 )   
 THEN ( [UnitPrice] \* 0.9 )  
 WHEN ( [Quantity] >= @Quantity4 AND [Quantity] < @Quantity5 )   
 THEN ( [UnitPrice] \* 0.8 )  
 ELSE ( [UnitPrice] \* 0.7 )  
 END  
FROM [Order Details]  
ORDER BY [Quantity] DESC

### Select Case When Then (Sample 2)

EmployeesQuery q = new EmployeesQuery();

q.Select

(

q.LastName,

q.LastName

.**Case**()

.**When**(q.LastName.Like("%a%")).Then("Last Name Contains an A")

.**Else**("Last Name Doesnt Contain an A")

.**End**().As("SpecialLastName")

);

EmployeesCollection coll = new EmployeesCollection();

coll.Load(q);

**Resulting SQL**

SELECT [LastName],[SpecialLastName] =   
 CASE WHEN [LastName] LIKE @LastName1 THEN 'Last Name Contains an A'  
 ELSE 'Last Name Doesnt Contain an A'  
 END  
FROM [Employees]

### Correlated Sub-Query

OrderDetailsQuery oiq = new OrderDetailsQuery("oi");

ProductsQuery pq = new ProductsQuery("p");

oiq.Select(

oiq.OrderID,

(oiq.Quantity \* oiq.UnitPrice).Sum().As("Total")

);

oiq.Where(oiq.ProductID

.In(

pq.Select(pq.ProductID)

.Where(oiq.ProductID == pq.ProductID)

)

);

oiq.GroupBy(oiq.OrderID);

OrderDetailsCollection collection = new OrderDetailsCollection();

collection.Load(oiq);

**Resulting SQL**

SELECT oi.[OrderID],  
 *Sum*(( oi.[Quantity] \* oi.[UnitPrice] )) AS 'Total'  
FROM [Order Details] oi  
WHERE oi.[ProductID] IN (SELECT p.[ProductID]  
 FROM [Products] p  
 WHERE oi.[ProductID] = p.[ProductID])  
GROUP BY oi.[OrderID]

### Where In with Nested Sub-Query

// OrderID and ProductID for Order Details

// with Products whose name begins with "G"

// from Suppliers whose city begins with "S".

OrderDetailsQuery oq = new OrderDetailsQuery("o");

ProductsQuery pq = new ProductsQuery("p");

SuppliersQuery sq = new SuppliersQuery("s");

oq.Select(

oq.OrderID,

oq.ProductID

).Where(oq.ProductID.In(

pq.Select(pq.ProductID)

.Where(pq.ProductName.Like("G%") && pq.SupplierID.In(

sq.Select(sq.SupplierID)

.Where(sq.City.Like("S%"))

)

)

)

);

OrderDetailsCollection collection = new OrderDetailsCollection();

collection.Load(oq);

**Resulting SQL**

SELECT o.[OrderID],  
 o.[ProductID]  
FROM [Order Details] o  
WHERE o.[ProductID] IN (SELECT p.[ProductID]  
 FROM [Products] p  
 WHERE ( p.[ProductName] LIKE @ProductName1  
 AND p.[SupplierID] IN (SELECT s.[SupplierID]  
 FROM [Suppliers] s  
 WHERE s.[City] LIKE @City2) ))

### From Sub-Query

OrdersQuery oq = new OrdersQuery("o");

OrderDetailsQuery oiq = new OrderDetailsQuery("oi");

oq.Select(oq.CustomerID, oq.OrderDate, "<sub.OrderTotal>");

oq.**From**

(

oiq.Select(oiq.OrderID,

(oiq.UnitPrice \* oiq.Quantity).Sum().As("OrderTotal"))

.GroupBy(oiq.OrderID)

).As("sub");

oq.InnerJoin(oq).On(oq.OrderID == oiq.OrderID);

OrdersCollection collection = new OrdersCollection();

collection.Load(oq);

**Resulting SQL**

SELECT o.[CustomerID],  
 o.[OrderDate],  
 sub.OrderTotal  
FROM (   
 SELECT oi.[OrderID],  
 *Sum*(( oi.[UnitPrice] \* oi.[Quantity] )) AS 'OrderTotal'  
 FROM [Order Details] oi  
 GROUP BY oi.[OrderID]  
 ) AS sub  
INNER JOIN [Orders] o ON o.[OrderID] = sub.[OrderID]

### Join On Sub-Query

// Query for the Join

OrderDetailsQuery odq = new OrderDetailsQuery("odq");

// SubQuery of OrderItems with a discount

OrderDetailsQuery odsq = new OrderDetailsQuery("odsq");

odsq.es.Distinct = true;

odsq.Select(odsq.Discount);

odsq.Where(odsq.Discount > 0);

// Orders with discounted items

OrdersQuery oq = new OrdersQuery("o");

oq.Select(oq.OrderID, odq.Discount);

oq.InnerJoin(odq).On(oq.OrderID == odq.OrderID & odq.Discount.**In**(**odsq**));

OrdersCollection collection = new OrdersCollection();

collection.Load(oq);

**Resulting SQL**

SELECT o.[OrderID],  
 oi.[Discount]  
FROM [Orders] o  
 INNER JOIN [Order Details] oi  
 ON (o.[OrderID] = oi.[OrderID]  
 AND oi.[Discount] IN (SELECT DISTINCT ods.[Discount]  
 FROM [Order Details] ods  
 WHERE ods.[Discount] > @Discount1))

## Raw SQL Injection

There may be times when you need to access some SQL feature that is not supported by the DynamicQuery API. Probably the last thing you want to do is stop and go write a stored procedure or create a view. The Dynamic Query API supports a technique referred to as raw injection. The way it works is raw SQL is passed as string surrounded by < > angle brackets. This indicates to the EntitySpaces DataProviders the raw SQL should be passed directly to the database engine “as is”.

Here is an example query. You would never write a query like this in reality. EntitySpaces supports the simple query shown below without having to use < > angle brackets. This is just to show all of the places that can accept the raw SQL injection technique:

EmployeesQuery q = new EmployeesQuery();

q.Select("<FirstName>", q.HireDate);

q.Where("<EmployeeID = 1>");

q.GroupBy("<FirstName>", q.HireDate);

q.OrderBy("<FirstName ASC>");

EmployeesCollection coll = new EmployeesCollection();

if (coll.Load(q))

{

// Then we found at least one record

}

**Resulting SQL**

SELECT FirstName,[HireDate]  
FROM [Employees]  
WHERE ( EmployeeID = 1 )  
GROUP BY FirstName, [HireDate]  
ORDER BY FirstName ASC

## Select against a SQL View rather than a SQL Table

You can trick your DynamicQuery to load from a SQL view rather than a SQL Table. In the case below, we are going to use the horribly named “Customer and Supplier by City” view from the Northwind database.

The contents of the view is as follows:

SELECT City, CompanyName, ContactName, 'Customers' AS Relationship

FROM Customers

UNION SELECT City, CompanyName, ContactName, 'Suppliers'

FROM Suppliers

We use the QuerySource property to instruct our query to load from our view rather than the Customer table.

CustomersQuery q = new CustomersQuery();

q.es.**QuerySource** = "Customer and Suppliers by City";

q.Where(q.City == "Anchorage");

CustomersCollection coll = new CustomersCollection();

coll.Load(q);

return coll;

**Resulting SQL**

SELECT \*  
FROM [Customer and Suppliers by City]  
WHERE [City] = @City1

## Operators

### Comparison Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | C# | VB.NET |
| Greater Than | > | > |
| Less Than | < | < |
| Greater Than or Equal | >= | >= |
| Less Than or Equal | <= | <= |
| Equal | == | = |
| Not Equal | != | <> |

### Arithmetic Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | C# | VB.NET |
| Plus | + | + |
| Minus | - | - |
| Multiply | \* | \* |
| Divide | / | / |
| Modulus | % | Mod |

### Logical-Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | C# | VB.NET |
| AND | && | And |
| OR | || | Or |

### Sub-Operators

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Avg | Between | Case | Cast | Coalesce | Contains | Count |
| Date | DatePart | Distinct | In | IsNotNull | IsNull | Length |
| Like | LTrim | Max | Min | Round | RTrim | StdDev |
| Substring | Sum | ToLower | ToUpper | Trim |  |  |