Contents

You can **include related entities** in a query to fetch or pre-cache a graph of entities that you know you will soon use. For example, when querying for an Order, you might also include the OrderDetails, Products, and shipping Address for the order. This saves additional trips to the server and can dramatically improve performance.

The Include method

To load related entities in the same query, use the *Include* method. For each related entity you want to include, specify the "property path" that you would use to navigate to that entity.

For example:

```
var query = anEntityManager.Customers.Where(c =>c.CompanyName.StartsWith{"A")).Include("Orders");
```

```
Dim query = anEntityManager.Customers.Where(c =>c.CompanyName.StartsWith{"A")).Include("Orders")
```

In this case, the "property path" between *Customers* and *Orders* was very simple; the *Customer* type has an *Orders* property and it is the name of this property that we are using in the *Include* method call. This query will return all of the customers with company names starting with "A" and **will also preload the EntityManager's cache with all of their related orders**.

More complex includes are also possible:

```
var query = anEntityManager.Customers
  .Where(c =>c.CompanyName.StartsWith{"A"))
  .Include("Orders.OrderDetails.Product")
  .Include("Orders.SalesRep")
```

```
Dim query = anEntityManager.Customers _
  .Where(c =>c.CompanyName.StartsWith{"A")) _
  .Include("Orders.OrderDetails.Product") _
  .Include("Orders.SalesRep")
```

In this case, the *Order* type has an "OrderDetails" property and a "SalesRep" property, and the *OrderDetail* type has a "Product" property.

Note that the property paths may not be valid navigation paths. i.e. you couldn't actually execute ***anOrder.OrderDetails.Product*** because the OrderDetails property returns a collection and it is the instances of this collection and not the collection itself that has a Product property.

In the preceding examples we have added the Include methods to the end of the query chain, but we would get exactly the same results if we were to rewrite the query as follows:

```
var query = anEntityManager.Customers
  .Include("Orders.OrderDetails.Product")
  .Include("Orders.SalesRep")
  .Where(c =>c.CompanyName.StartsWith{"A"))
```

```
Dim query = anEntityManager.Customers _
  .Include("Orders.OrderDetails.Product") _
  .Include("Orders.SalesRep") _
  .Where(c =>c.CompanyName.StartsWith{"A"))
```

Important points

- *Includes* always operate on the 'final' result type of a query, regardless of where they appear in the query chain.
- When using an *Include* like "Orders.OrderDetails.Product" with multiple "parts" to the property path, each of the intermediate results will also be added to the *EntityManager's* entity cache.
- Note that the use of the *Include* method doesn't change the list of entities returned from the query. The caller still receives the same results with or without the use of an *Include* method. The difference is that **before** returning these results, the "Include" processing fetches the related entities and merges them into the cache. This occurs behind the scenes and does not effect the query result.
- When using an *Include* such as "Orders" for a Customer, the *Include* retrieves all orders of each selected Customer. You cannot filter the "Included" orders. If you need to do that, consider a projection query.

Includes in an asynchronous world

In Silverlight and other n-tier environments where all data retrieval must be asynchronous, the benefits of preloading data ("eager querying") are even more general. In the following snippet, we preload, using *Include* method calls, a large object graph a group of Customers that meet a specified condition. Having done so, all of our subsequent queries for entities can be cache-only and synchronous:

```
public async void PreloadCustomerOrderData() {
  IEntityQuery<Customer> query = anEntityManager.Customers.Where(c => c.Country == "France")
    .Include("Orders.OrderDetails.Product.Supplier")
    .Include("Orders.SalesRep");
  await _em1.ExecuteQueryAsync(query);
 // all of the following queries can now execute synchronously and will return
 // the 'related' entities from the query
 var customers    = anEntityManager.Customers.With(QueryStrategy.CacheOnly).ToList();
  var employees    = anEntityManager.Employees.With(QueryStrategy.CacheOnly).ToList();
  var orders       = anEntityManager.Orders.With(QueryStrategy.CacheOnly).ToList();
  var orderDetails = anEntityManager.OrderDetails.With(QueryStrategy.CacheOnly).ToList();
  var products     = anEntityManager.Products.With(QueryStrategy.CacheOnly).ToList();
  var suppliers    = anEntityManager.Suppliers.With(QueryStrategy.CacheOnly).ToList();
}
```

```
Public Async Sub PreloadCustomerOrderData()
 Dim query As IEntityQuery(Of Customer) = anEntityManager.Customers.Where(Function(c) c.Country = "France") _
    .Include("Orders.OrderDetails.Product.Supplier") _
    .Include("Orders.SalesRep")
  Await _em1.ExecuteQueryAsync(query)
 ' all of the following queries can now execute synchronously and
 ' will return the 'related' entities from the query above.
 Dim customers    = anEntityManager.Customers.With(QueryStrategy.CacheOnly).ToList()
 Dim employees    = anEntityManager.Employees.With(QueryStrategy.CacheOnly).ToList()
 Dim orders       = anEntityManager.Orders.With(QueryStrategy.CacheOnly).ToList()
 Dim orderDetails = anEntityManager.OrderDetails.With(QueryStrategy.CacheOnly).ToList()
 Dim products     = anEntityManager.Products.With(QueryStrategy.CacheOnly).ToList()
 Dim suppliers    = anEntityManager.Suppliers.With(QueryStrategy.CacheOnly).ToList()
End Sub
```

Performance details

While use of the *Include* syntax greatly reduces the number of queries submitted to *EntityServer*, it is also likely to change the number of queries performed by the backend database.

This is because a single query with an *Include* will typically perform one or two database queries depending on the complexity of the query, but the same query without an include would result in an initial single query followed by many additional queries spaced out over time as each individual client side property navigation from the resulting entities caused a further "lazy" query evaluation. Each of these "lazy evaluations" necessitates a separate trip to both the EntityServer and then to the database.

In an n-tier deployment using the *EntityServer*, these "lazy evaluations" can end up being very expensive because of the latency of the network.

But be careful! You could be trying to get too much data at one time. You could be joining data from too many tables at one time. Consequently, the query might perform more poorly than if you had made several trips. Or it might just collapse of its own weight. Make sure you test your queries.

"Eager queries" are great, but we have also seen them being badly misused. The preceding paragraphs have enumerated the advantages, but there is sometimes a tendency to think that because you might need some related entities that you should always "preload" them. This can be a bad idea if most of the time this data is not needed, and you are still paying the cost of retrieving and transmitting them.

Performance matters ... but not all time and effort spent optimizing performance returns equal results. We strongly advise instrumenting your queries during development and testing to identify performance hotspots. Then optimize where it really matters.

Strongly-typed Includes

Using the *Include* method with a string to represent the property path means that an incorrect path won't be discovered until runtime. For simple property paths you can instead use a type-safe alternative.

A "simple" property path is one that can be expressed by a simple lambda expression, so "Order.OrderDetails" is a simple property path, but "Order.OrderDetails.Product" is not.

There are two type-safe options available:

- Use the ***PathFor*** method that is generated into every DevForce entity type
- Use the *Include* extension method which accepts a lambda expression

The advantage of these syntaxes is that if any of the included property names ever change the compiler will catch the error instead of allowing the code to compile with a "bad" include.

Below we show several queries with the strongly-typed options.

```
// 1 - using a string property path
var query1a = anEntityManager.Customers
  .Where(c =>c.CompanyName.StartsWith("A"))
  .Include("Orders");
// 2 - the same query using PathFor
var query1b = anEntityManager.Customers
  .Where(c =>c.CompanyName.StartsWith{"A"))
  .Include(Customer.PathFor( c => c.Orders));
// 3 - the same query using the Include lambda
var query1c = anEntityManager.Customers
  .Where(c =>c.CompanyName.StartsWith{"A"))
  .Include(c => c.Orders);
```

```
' 1 - using a string property path
Dim query1a = anEntityManager.Customers
    .Where(Function(c) c.CompanyName.StartsWith("A"))
    .Include("Orders")
' 2 - the same query using PathFor
Dim query1b = anEntityManager.Customers
    .Where(Function(c) c.CompanyName.StartsWith{"A"))
    .Include(Customer.PathFor(Function(c) c.Orders))
' 3 - the same query using the Include lambda
Dim query1c = anEntityManager.Customers
    .Where(Function(c) c.CompanyName.StartsWith{"A"))
    .Include(Function(c) c.Orders)
```

Here's a more complex path, but still "simple":

```
// 1 - using a string property path
var query2a = anEntityManager.OrderDetails
  .Include("Product.Suppliers");
// 2 - the same query using PathFor
var query2b = anEntityManager.OrderDetails
  .Include(OrderDetail.PathFor( od => od.Product.Suppliers));
// 3 - the same query using the Include lambda
var query2c = anEntityManager.OrderDetails
  .Include(od => od.Product.Suppliers);
```

```
' 1 - using a string property path
Dim query2a = anEntityManager.OrderDetails
    .Include("Product.Suppliers")
' 2 - the same query using PathFor
Dim query2b = anEntityManager.OrderDetails
    .Include(OrderDetail.PathFor(Function(od) od.Product.Suppliers))
' 3 - the same query using the Include lambda
Dim query2c = anEntityManager.OrderDetails
    .Include(Function(od) od.Product.Suppliers)
```