Contents

One common issue when working with entities is how to **return part of an entity** when only some of the properties of an entity or collection of entities are needed.

Returning part of an entity, or a "slice", can be useful if an entity type has many properties, some of which are large (e.g, BLOBS), and a query could return too much data or take too long to load. Perhaps you are presenting a list of such entities. You won't edit them. If you do edit them, you will use the list as a reference for identifying entities to edit (e.g., by user selection); then you will launch a separate editor and load it with the full entity version retrieved from the database expressly for this purpose.

Slice projections are appropriate in such high volume, "read-only" scenarios where the savings are measurable and significant. In most cases, though, entities should still be the primary tool for building any application that requires the update of a domain model's data.

There are two ways to select only part of an entity - using *anonymous projections*, and projections into a custom type. We'll describe both below.

Anonymous projections are not supported in Windows Store, Windows Phone and Windows Universal applications. Instead, use the project into custom type capability.

Anonymous projections

DevForce supports all of the standard LINQ mechanisms for projection of an anonymous type from any query. For example, in the snippet below we build a LINQ query that projects out three properties from the *Employee* entity type:

```
var query = manager.Employees
  .Select(emp => new { emp.EmployeeID, emp.FirstName, emp.LastName });
```

```
Dim query = manager.Employees _
  .Select(Function(emp) New With { emp.EmployeeID, emp.FirstName, emp.LastName })
```

Let's look at query syntax too, since it's easier to use in VB:

```
var query = from emp in manager.Employees
            select new {emp.EmployeeID, emp.FirstName, emp.LastName};
```

```
Dim query = From emp In manager.Employees
            Select New With {emp.EmployeeID, emp.FirstName, emp.LastName}
```

When the query is executed, an *IEnumerable* of anonymous types is returned. Here's where implicit type names come in handy, since the anonymous type is not statically defined:

```
var anonItems = query.ToList();
foreach (var anonItem in anonItems) {
 int    id    = anonItem.EmployeeID;
  String fName = anonItem.FirstName;
  String lName = anonItem.LastName;
}
```

```
Dim anonItems = query.ToList()
For Each anonItem In anonItems
 Dim id As Integer = anonItem.EmployeeID
 Dim fName As String = anonItem.FirstName
 Dim lName As String = anonItem.LastName
Next anonItem
```

It's worth pointing out that anonymous projection queries can be sent to the EntityServer, like any other query type.

We can issue synchronous or asynchronous queries with anonymous projections too.

```
var results = await query.ExecuteAsync();
foreach (var anonItem in results) {
   int id       = anonItem.EmployeeID;
    String fName = anonItem.FirstName;
    String lName = anonItem.LastName;
}
```

```
});
```

```
Dim results = Await query.ExecuteAsync()
For Each anonItem In results
 Dim id As Integer = anonItem.EmployeeID
 Dim fName As String = anonItem.FirstName
 Dim lName As String = anonItem.LastName
Next anonItem
```

We can also do aggregates and calculations within the projections. For example:

```
var query = manager.Customers
  .Select(c => new { c.CustomerID, TotalFreightWithDiscount = c.Orders.Sum(os => os.Freight * .85m) });
```

```
Dim query = manager.Customers _
  .Select(Function(c) New With {Key c.CustomerID, Key .TotalFreightWithDiscount = _
   c.Orders.Sum(Function(os) os.Freight *.85D)})
```

Grouping operations are also supported, as shown in the asynchronous example below.

```
var query = manager.Customers
  .Where(c => c.CompanyName.StartsWith("C"))
  .GroupBy(c => c.Country)
  .Select(g => new { Country = g.Key, Count = g.Count() });
var results = await query.ExecuteAsync();
foreach (var item in results) {
 string country = item.Country;
 int count = item.Count;
}
```

```
Dim query = manager.Customers.Where(Function(c) c.CompanyName.StartsWith("C")) _
  .GroupBy(Function(c) c.Country).Select(Function(g) New With _
  {Key .Country = g.Key, Key .Count = g.Count()})
Dim results = Await query.ExecuteAsync()
For Each item In results
 Dim country As String = item.Country
 Dim count As Integer = item.Count
Next item
```

You can also select related entities and lists. This is useful too if a filter needs to be applied to the related entities.

```
var anonItems = manager.Customers.Select(c => new { c.CompanyName, c.OrderSummaries }).ToList();
```

```
Dim anonItems = manager.Customers.Select(Function(c) New With { c.CompanyName, c.OrderSummaries }).ToList()
```

Anonymous types are not public types

Notice that the anonymous type returned by these queries is not public. Anonymous types have *internal* (*Friend* in VB) visibility.

Because of this visibility, you can't directly bind to an anonymous type in Silverlight. Fortunately, DevForce offers the *DynamicTypeConverter* helper class that can convert a non-public anonymous query result into a public type suitable for binding.

```
var anonItems = manager.Customers.Select(c => new { c.CompanyName, c.OrderSummaries }).ToList();
var bindableResult = IdeaBlade.Core.DynamicTypeConverter.Convert(anonItems);
```

```
Dim anonItems = manager.Customers.Select(Function(c) New With _
  {Key c.CompanyName, Key c.OrderSummaries}).ToList()
Dim bindableResult = IdeaBlade.Core.DynamicTypeConverter.Convert(anonItems)
```

DevForce also offers a helper method which allows you to test whether a query returned an anonymous type, *AnonymousFns.IsAnonymousType* .

Limitations

The "slice" projection described here returns an anonymous type, not an entity. The returned object will not enter the entity cache. The EntityManager does not "remember" the query and will send the query to the server every time (assuming that a server visit is consistent with the query strategy and the present state of the *EntityManager* itself).

Remember that the projected result is not an entity. Even if you could make changes, you couldn't save those changes ... not in the way you would save entities.

You will also be responsible for coordinating projected values with changes to the source entities. When you display a list of Employees and change one of them, the displayed employee list updates to reflect those changes. Change "Nancy" to "Sally" and it changes in the list too. Not so if the list contains projected employee data - which is disconcerting to users. You can coordinate with the list yourself at the cost of added complexity, maintenance, and testing.

Finally, the anonymous types returned by slice projections are difficult to work with.

- You can't edit the anonymous result.
- You can't bind directly to the anonymous type (without the assistance of the *DyanmicTypeConverter*).
- In Silverlight you can't access type members outside the assembly.

You may want to transfer the result into a locally defined type to make it easier to work with. If you expect the projection to have long term, widespread use, you should consider "table splitting" - an Entity Framework modeling technique which defines two (or more entities) mapped to the same table. Or you may prefer to define a serializable type that matches the expected shape and define your projection directly into that, which we'll describe next.

One final note about projecting into an anonymous type: depending on how many properties are in your anonymous type and the number of items queried, you may find that a query projection into a custom type will execute faster.

Project into a custom type

Instead of projecting part of an entity into an anonymous type you have the option of projecting into a custom type which you've defined. This option can be useful since it alleviates the difficulties in working with anonymous types. It does, however, require additional setup.

You must first define the custom type. It must be serializable, a known type, and available on both client and server. Here's a simple example, using the *Employee* slice we used above.

```
[DataContract]
public class EmployeeSlice : IKnownType {
 // Employee properties
 [DataMember]
 public int EmployeeID { get; set; }
  [DataMember]
 public string FirstName { get; set; }
  [DataMember]
 public string LastName { get; set; }
  [DataMember]
 public int OrderCount { get; set; }
 // Calculated property
 public string FullName { get { return FirstName + " " + LastName; } }
 }
```

```
<DataContract>
Public Class EmployeeSlice
Implements IKnownType
' Employee properties
<DataMember>
Public Property EmployeeID() As Integer
<DataMember>
Public Property FirstName() As String
<DataMember>
Public Property LastName() As String
<DataMember>
Public Property OrderCount() As Integer
' Calculated property
Public ReadOnly Property FullName() As String
Get
Return FirstName & " " & LastName
End Get
End Property
End Class
```

Note the markup with the DataContract and DataMember attributes - these allow instances of this class to be moved between client and server tiers. Next note the implementation of the DevForce marker interface *IKnownType*. DevForce probes for this interface during its discovery of known types, and this ensures that DevForce is "aware" that the type may be used in queries. Also note that the data member properties are *public*. This isn't strictly required, but avoids serialization issues. In Silverlight applications you can define *internal* setters with a bit of extra work.

You'll want to define this class on both client and server. Why? Because your query will be created on the client and executed on the server. Both sides must know about it. How do you define the class on both tiers? The easiest approach, and the one DevForce uses with the generated entity model, is to create the class file in a "server-side" project and link to the file (or files) from the "client-side" project. If the project assembly will be deployed to both tiers, then linking the files is of course not necessary, but you will need to ensure the assembly is deployed to both tiers.

You'll also want to ensure the class can be discovered by DevForce. If you've modified the discovery options, be sure that the assembly containing this type will be included.

The query is only slightly different than the anonymous projection query. The "Select" clause features a "new" keyword followed by our receiving type, "EmployeeSlice". The projection is no longer anonymous.

```
var query = manager.Employees
  .Select(emp => new EmployeeSlice {
    EmployeeID = emp.EmployeeID,
    FirstName = emp.FirstName,
    LastName = emp.LastName,
    OrderCount = emp.Orders.Count(),
  });
```

```
Dim query = manager.Employees _
.Select(Function(emp) New EmployeeSlice With {
  .EmployeeID = emp.EmployeeID,
  .FirstName = emp.FirstName,
  .LastName = emp.LastName,
  .OrderCount = emp.Orders.Count()
})
```

Use with POCO types

Although we are projecting into a known type, that type is not an entity. The query is not delivering entities. The result objects do not enter the *entity cache*. The *EntityManager* does not remember the query and will re-send it to the server with every execution unless otherwise barred from doing so. And you cannot save instances of this type to the database.

If you *must* store this data in the *entity cache* you can work around these limitations by assigning a Key attribute to one or more properties of the type. If instances of the type have a unique identity then they will be stored in the *entity cache* when queried, and can be manipulated and saved as POCO types. The query itself won't be placed in the *query cache*.