Contents

You can construct *Select*, *SelectMany* and *GroupBy* clauses dynamically with the *IdeaBlade.Linq.ProjectionSelector* when you can't specify them at compile time.

The LINQ *Select*, *SelectMany* and *GroupBy* clauses change the shape of a query result. Queries that uses these clauses are called "projections" and we refer to them as "projection clauses" in the following discussion.

When you need to construct a projection query but you don't know the types involved at compile time, you can create an instance of the *ProjectionSelector* class and pass in the type information at runtime.

Select examples

When creating a *ProjectionSelector* class you specify the properties to select (AKA, "project"). You can also specify the type of object to which the properties belong ... if you know the type ... or delay type identification until you use the selector.

Consider a simple example in which we query for *Customers* and return ("project") their *CompanyNames*. If we knew we were going to do this at compile time, we'd write:

```
var companyNames = anEntityManager.Customers.Select(c => c.CompanyName).ToList();
```

```
Dim companyNames = anEntityManager.Customers.Select(Function(c) c.CompanyName).ToList()
```

But we don't know. So we write a general purpose function, *DoSelect*, that is capable of projecting an arbitrary property belonging to an arbitrary entity type. To see it in action, we call it with the *Customer* type and specify the *CompanyName* type as before.

```
public IEnumerable DoSelect(EntityManager manager, Type entityType, string propertyName) {
   var selector = new ProjectionSelector(propertyName);
   var rootQuery = EntityQuery.Create(entityType, manager);
   var query = rootQuery.Select(selector); // selector learns its type from rootQuery
  var results = query.Execute();          // synchronous query execution
  return results;
}
// The company names of every Customer
var companyNames = DoSelect(anEntityManager, typeof(Customer), "CompanyName");
```

```
Public Function DoSelect(ByVal manager As EntityManager, ByVal entityType As Type,
                         ByVal propertyName As String) As IEnumerable
  Dim selector = New ProjectionSelector(propertyName)
  Dim rootQuery = EntityQuery.Create(entityType, manager)
  Dim query = rootQuery.Select(selector)  ' selector learns its type from rootQuery
  Dim results = query.Execute()           ' synchronous query execution
  Return results
End Function
' The company names of every Customer
Dim companyNames = DoSelect(anEntityManager, GetType(Customer), "CompanyName")
```

Any property of a class can be projected by passing in its property name, or as in the next example, passing in a nested property name.

```
// The company names of every Order's Customer
var companyNames = DoSelect(anEntityManager, typeof(Order), "Customer.CompanyName");
```

```
' The company names of every Order's Customer
Dim companyNames = DoSelect(anEntityManager, GetType(Order), "Customer.CompanyName")
```

Select into a dynamic type

We can use the *Select* clause to project multiple values into a single instance of a DevForce dynamic type (we explain dynamic types below). In our next example, we get a few *Product*s, find their related *Category* entities, and project two of the *Category* properties into a dynamic type. For ease of exposition, we specify *Product*, *Category* and its properties in the code. If you actually knew the types and properties at design time you'd use strongly typed LINQ statements and wouldn't bother with the *ProjectionSelector*. We trust you appreciate our true intention.

```
IProjectionSelector selector = new ProjectionSelector("Category.Name", "CatName");
selector = selector.Combine("Category.Description", "CatDesc");
var rootQuery = EntityQuery.Create(typeof(Product), anEntityManager);
var query = rootQuery.Select(selector);
var results = query.Execute();
```

```
Dim selector = New ProjectionSelector("Category.Name", "CatName") as IProjectionSelector
selector = selector.Combine("Category.Description", "CatDesc"))
Dim rootQuery = EntityQuery.Create(GetType(Product), anEntityManager)
Dim query = rootQuery.Select(selector)
Dim results = query.Execute()
```

A perhaps more graceful syntax, especially when there are many properties, might be:

```
var selector = new AnonymousProjectionSelector()
                .Combine("Category.Name",        "CatName")
                .Combine("Category.Description", "CatDesc");
var rootQuery = EntityQuery.Create(typeof(Product), anEntityManager);
var query = rootQuery.Select(selector);
var results = query.Execute();
```

```
Dim selector = New AnonymousProjectionSelector()
                .Combine("Category.Name",        "CatName")
                .Combine("Category.Description", "CatDesc")
Dim rootQuery = EntityQuery.Create(typeof(Product), anEntityManager)
var query = rootQuery.Select(selector)
Dim results = query.Execute()
```

This form draws proper attention to the *AnonymousProjectionSelector* and the use of the *Combine* extension method to build up the properties of the projection.

The "CatName" and "CatDesc" arguments are the aliases for the projected nested properties; they become the names of the two properties of the DevForce dynamic type objects returned in the *results*.

Finally, a quite general projection function which suggests the potential for this approach:

```
public IEnumerable DoSelect(
    EntityManager manager, Type entityType, params string[] propertyNames)
{
    var selector = new AnonymousProjectionSelector();
   foreach (var name in propertyNames)
    {
       var alias = name.Replace(".", "_");
       selector = selector.Combine(name, alias);
    }
    var rootQuery = EntityQuery.Create(entityType, manager);
    var query = rootQuery.Select(selector);  // selector learns its type from rootQuery
   var results = query.Execute();           // synchronous query execution
   return results;
}
```

```
Public Function DoSelect(ByVal manager As EntityManager, _
   ByVal entityType As Type, ByVal ParamArray propertyNames As String()) As IEnumerable
   Dim selector = new AnonymousProjectionSelector()
   For Each name As String In propertyNames
      Dim alias = name.Replace(".", "_")
      selector = selector.Combine(name, alias)
   Next
   Dim rootQuery = EntityQuery.Create(entityType, manager)
   Dim query = rootQuery.Select(selector)  ' selector learns its type from rootQuery
   Dim results = query.Execute()           ' synchronous query execution
   Return results
End Function
```

The DevForce dynamic type

The DevForce dynamic type returned in query results may be treated exactly like a standard .NET anonymous type. DevForce cannot return an actual .NET anonymous type because the compiler insists that anonymous types be known at compile time. The dynamic types DevForce creates are not defined until runtime. Therefore, DevForce dynamically creates a type that has the same semantics as an anonymous type by emitting IL at runtime.

Unknown macro: IBNote

```
The "IBNote" macro is not in the list of registered macros. Verify the spelling or contact your administrator.
```

A DevForce dynamic type has one additional benefit: it is a *public* class which means that Silverlight controls can bind to it. Silverlight controls can't bind to .NET anonymous classes becauses they are declared *internal*.

You access properties of a dynamic type much as you would properties of a .NET anonymous type as we see in the following two examples. The first makes use of the *dynamic* keyword introduced in .NET 4.

```
foreach (dynamic item in results) {
  String categoryName        = (String) item.CatName;
  String categoryDescription = (String)item.CatDesc;
}
```

```
For Each item As dynamic In results
 Dim categoryName As String        = CType(item.CatName, String)
 Dim categoryDescription As String = CType(item.CatDesc, String)
Next item
```

The second uses the DevForce *IdeaBlade.Core.AnonymousFns* helper class to deconstruct a dynamic type into an object array.

```
var elementType = results.Cast<Object>().FirstOrDefault().GetType();
var items = AnonymousFns.DeconstructMany(results, false, elementType);
foreach (var item in items) {
  String categoryName        = (String) item[0];
  String categoryDescription = (String)item[1];
}
```

```
Dim elementType = results.Cast(Of Object)().FirstOrDefault().GetType()
Dim items = AnonymousFns.DeconstructMany(results, False, elementType)
For Each item In items
 Dim categoryName As String        = CType(item(0), String)
 Dim categoryDescription As String = CType(item(1), String)
Next item
```

The memory footprint of dynamic types

Dynamic projections are convenient but should be used with care. Every .NET type definition consumes memory that cannot be reclaimed by the .NET garbage collector. This is as true of the dynamic types that DevForce creates as it is of .NET anonymous types. Queries that return the same dynamic type "shape" - the same properties, of the same types, in the same order, with the same names - are not a problem; dynamic type definitions are cached and a type with matching shape is reused. But it's possible for client applications to create an endless variety of dynamic type shapes.

While the amount of memory consumed by a single type is tiny and almost never an issue on a single client machine, each of these types is also created on the *EntityServer* where the query is executed. A long-running *EntityServer* might accumulate large numbers of distinct dynamic type definitions. The server could run out of memory if it saw millions of different anonymous types in which case you'll have to recycle the server periodically if this becomes a problem. It's not a great risk but we felt we should mention it.

SelectMany example

A dynamic implementation of the LINQ *SelectMany* operation also makes use of the *ProjectionSelector*. The following contrived example shows how to use *SelectMany* to get the *OrderDetails* associated with the first five *Product*s in the database. It relies upon a peculiar function, *DoSelectMany*, that gets an arbitrary set of related entities from the first five instance of some kind of entity.

```
public IEnumerable DoSelectMany(
      EntityManager manager, Type entityType, string propertyName)
{
   var selector = new ProjectionSelector(propertyName);
   var rootQuery = EntityQuery.Create(entityType, manager);
   var query = rootQuery
      .Take(5) // reduce the result size for exposition.
      .SelectMany(selector);
   var results = query.Execute();
  return results;
}
var orderDetails =
       DoSelectMany(anEntityManager, typeof(Product), "OrderDetails")
          .Cast<OrderDetail>() // convert IEnumerable to IEnumerable<OrderDetails>
          .ToList();
```

```
Public Function DoSelectMany(ByVal manager As EntityManager,  _
    ByVal entityType As Type, ByVal propertyName As string) As IEnumerable

   var selector = New ProjectionSelector(propertyName)
  Dim rootQuery = EntityQuery.Create(entityType, mManager)
  Dim query = rootQuery
      .Take(2) ' reduce the result size for exposition.
     .SelectMany(selector)
  Dim results = query.Execute()
  Return results
End Function
Dim orderDetails =
      DoSelectMany(anEntityManager, GetType(Product), "OrderDetails")
         .Cast<OrderDetail>() ' convert IEnumerable to IEnumerable<OrderDetails>
         .ToList()
```

GroupBy example

A dynamic implementation of the LINQ *GroupBy* operation also makes use of the *ProjectionSelector*. In the following example, the *SimpleGroupBy* function queries for every instance of a given type and groups the results by one of the properties of that type.

```
public IEnumerable SimpleGroupBy(EntityManager manager,
   Type entityType, string groupByProperty)
{
   var groupBy = new ProjectionSelector(groupByProperty, alias);
   var query = EntityQuery.Create(entityType, manager)
                    .GroupBy(groupBy);
   var result = manager.ExecuteQuery(query);
  return result;
}
// Static equivalent: anEntityManager.Products.GroupBy(_ => _.Category.CategoryName);
var result = SimpleGroupBy(anEntityManager,
            typeof(Product), "Category.CategoryName");
var productGrps = result.Cast<IGrouping<String, Product>>().ToList();
```

```
Public Function SimpleGroupBy(ByVal manager As EntityManager,
  ByVal entityType As Type,ByVal groupByProperty As String) _
  As IEnumerable
  Dim groupBy = new ProjectionSelector(groupByProperty, alias)
  Dim query = EntityQuery.Create(entityType, manager)
                    .GroupBy(groupBy)
  Dim result = manager.ExecuteQuery(query)
  Return result
End Function
' Static equivalent: anEntityManager.Products.GroupBy(Function(_) _.Category.CategoryName)
Dim result = SimpleGroupBy(anEntityManager,
            GetType(Product), "Category.CategoryName")
Dim productGrps = result.Cast(Of IGrouping(Of String, Product))().ToList()
```