Contents

The need for the ability to create a dynamic *where* clause occurs fairly frequently in applications that need to filter data based on a users input. In these cases you don't really know ahead of time what **properties** a user is going to want to query or what the **conditions** of the query are likely to be. In fact, you may not even know the **type** that you will be querying for until runtime. So you will need to be able to compose the query based on the inputs determined at runtime.

Flavors of dynamic where clauses

Ideally, when building a query we want to specify as little dynamically as we have to. The less dynamic a query is the better the compile time checking and intellisense will be for the query. Everything depends upon what we know and when we know it. Do we know the type of the query at compile time but need to wait until runtime to determine the properties to be queried and the conditions regarding these properties? Or do we not even know the type of the query until runtime?

Whatever the case, the *IdeaBlade.Linq.PredicateBuilder* and the *IdeaBlade.Linq.PredicateDescription* classes are the tools used to deal with compile time uncertainty. As we review the *PredicateBuilder* API, you will see both typed and untyped overloads for most methods in order to support both compile and runtime scenarios.

What's a predicate?

A LINQ where clause is intimately tied to the concept of a "predicate". So what is a predicate and how does it relate to a LINQ *where* clause.

A predicate is a function that evaluates an expression and returns true or false. The code fragment...

```
p.ProductName.Contains("Sir")
```

```
p.ProductName.Contains("Sir")
```

...is a predicate that examines a product and returns true if the product's *ProductName* contains the **Sir** string.

The CLR type of the predicate in our example is:

```
Func<Product, bool>
```

```
Func(Of Product, Boolean)
```

Which we can generalize to:

```
Func<T, bool>
```

```
Func(Of T, Boolean)
```

This is almost what we need in order to build a LINQ where clause. The LINQ *Where* extension method is defined as follows:

```
public static IQueryable<T> Where<TSource>(
 this IQueryable<T> source1, Expression<Func<T,bool>> predicate)
```

```
public static IQueryable(Of T) Where(Of TSource) _
  (Me IQueryable(Of T) source1, Expression(Of Func(Of T,Boolean)) predicate)
```

Note that the "predicate" parameter above is

```
Expression<Func<T, bool>>
```

```
Expression(Of Func(Of T, Boolean))
```

When we refer to the idea that a "predicate" is needed in order to build a LINQ *where* clause, what we really mean is that we need to build a "predicate expression", or a *Expression* that is resolvable to a simple predicate.

Create a *Where* clause with two predicates

The snippet below comprises two statements, each of which uses *PredicateBuilder.Make* to create a *PredicateDescription* representing a single predicate (filter criteria).

```
PredicateDescription p1 = PredicateBuilder.Make(typeof(Product), "UnitPrice",
  FilterOperator.IsGreaterThanOrEqualTo, 24);
```

```
PredicateDescription p2 = PredicateBuilder.Make(typeof(Product), "Discontinued",
FilterOperator.IsEqualTo, true);
```

```
Dim p1 As PredicateDescription = PredicateBuilder.Make(GetType(Product), "UnitPrice", _
  FilterOperator.IsGreaterThanOrEqualTo, 24)
Dim p2 As PredicateDescription = PredicateBuilder.Make(GetType(Product), "Discontinued", _
  FilterOperator.IsEqualTo, True)
```

A new query can then be composed and executed by an EntityManager as usual:

```
var query = anEntityManager.Products.Where(p1.And(p2))
// The above query is the same as:
//var queryb = anEntityManager.Products.Where(p => p.UnitPrice > 24 && p.Discontinued);
```

```
Dim query = anEntityManager.Products.Where(p1.And(p2))
' The above query is the same as:
'var queryb = anEntityManager.Products.Where(p => p.UnitPrice > 24 && p.Discontinued);
```

We could have combined the individual *PredicateDescriptions* into another *PredicateDescription* variable:

```
CompositePredicateDescription p3 = p1.And(p2);
var query = anEntityManager.Products.Where(p3);
```

```
Dim p3 As CompositePredicateDescription = p1.And(p2)
Dim query = anEntityManager.Products.Where(p5)
```

Learn more about combining predicates with *PredicateBuilder* and *PredicateDescription* classes.

Completely dynamic query

The *Where* clause in the previous examples were still applied to a strongly typed (non-dynamic) *IQueryable<Product>* implementation. To be more precise, the query root was of type *IEntityQuery<Product>*.

What if we didn't know we were querying for *Product* at compile type? We'd know the type at runtime but we didn't know it as we wrote the code. We could use the *EntityQuery.CreateQuery* factory method and pass it the runtime type. *CreateQuery* returns a nominally-untyped *EntityQuery* object.

```
var queryType = typeof(Product); // imagine this was passed in at runtime.
var baseQuery = EntityQuery.CreateQuery(queryType , anEntityManager);
var query = baseQuery.Where(p1.And(p2));
```

```
Dim queryType = GetType(Product) ' imagine this was passed in at runtime.
Dim baseQuery = EntityQuery.CreateQuery(queryType , anEntityManager)
Dim query = baseQuery.Where(p1.And(p2))
```

The *EntityQuery* result of *CreateQuery* implements *ITypedEntityQuery*. The *Where* clause in this example made use of the *ITypedEntityQuery* extension method instead of the *IQueryable<T>* extension method:

```
public static ITypedEntityQuery Where(this ITypedEntityQuery source, IPredicateDescription predicateDescription);
```

```
Public Shared ITypedEntityQuery Where(Me ITypedEntityQuery source, _
  IPredicateDescription predicateDescription)
```

The query that will be executed is **exactly** the same regardless of which extension method is used. Both queries are *Product* queries. The critical difference is that the compiler can't determine the type of the query in the second case. That's why it resolved to a query typed as *ITypedEntityQuery*, which implements IQueryable, instead of *IEntityQuery<T>* which implements *IQueryable<T>*.

There are two ramifications:

- Completely dynamic queries, those typed as *ITypedEntityQuery*, must be executed with one of the *EntityManager.Execute* methods instead of ToList().
- The compiler declares that the type of the dynamic query execution result is *IEnumerable* instead of *IEnumerable<T>*.

The following example illustrates:

```
// IQueryable<T> implementation
var query1 = anEntityManager.Products.Where(p3);
IEnumerable<Product> products1 = query1.ToList();
// IQueryable implementation
var baseQuery = EntityQuery.CreateQuery(typeof(Product), anEntityManager);
var query2 = baseQuery.Where(p1.And(p2));
```

```
// can't call query2.ToList() because query2 is not an IQueryable<T>
IEnumerable results = anEntityManager.ExecuteQuery(query2);
// next line is required in order to
IEnumerable<Product> products2 = results.Cast<Product>();
```

```
' IQueryable<T> implementation
Dim query1 = anEntityManager.Products.Where(p3)
Dim products1 As IEnumerable(Of Product) = query1.ToList()
' IQueryable implementation
Dim baseQuery = EntityQuery.CreateQuery(GetType(Product), anEntityManager)
Dim query2 = baseQuery.Where(p1.And(p2))
' can't call query2.ToList() because query2 is not an IQueryable<T>
Dim results As IEnumerable = anEntityManager.ExecuteQuery(query2)
' next line is required in order to
Dim products2 As IEnumerable(Of Product) = results.Cast(Of Product)()
```