Contents

- The EntityKeyQuery
  - Creating an EntityKeyQuery
  - Disadvantages of the EntityKeyQuery
- Query by Id

The *EntityKey* of an entity defines the unique identity of an entity. You can query by *EntityKey* using the ***EntityKeyQuery***.

The EntityKeyQuery

Because an *EntityKey* by definition uniquely identifies a single entity, the *EntityKeyQuery* is optimized in a way that other query types cannot be. The EntityManager, given an *EntityKey*, can determine by looking in its entity cache whether or not an entity with this key has already been fetched from the database. For other query types, DevForce uses its query cache to determine whether a query has already been executed.

The determination of whether a given query has already been executed against a database is a DevForce performance enhancement. Even with an *EntityKeyQuery*, this optimization can be suppressed by using the *DataSourceOnly* *QueryStrategy*.

Creating an EntityKeyQuery

There are two basic ways to create an *EntityKeyQuery*: with an *EntityKey*, or with a list of *EntityKeys* called an *EntityKeyList*.

You might wonder how to obtain an *EntityKey* since it's a property of an *Entity* (through its *EntityAspect*). You can construct an *EntityKey* if you know the entity type and the key values.

For example, if you want to build an *EntityKey* for Employee 1, you could do the following:

```
var key = new EntityKey(typeof(Employee), 1);
```

```
Dim key = New EntityKey(GetType(Employee), 1)
```

You can then build the *EntityKeyQuery* for this key. One easy way is to use the helper method *ToKeyQuery*:

```
var query = key.ToKeyQuery();
```

```
Dim query = key.ToKeyQuery()
```

Or using the *EntityKeyQuery* constructor:

```
var query = new EntityKeyQuery(key);
```

```
Dim query = New EntityKeyQuery(key)
```

Building an *EntityKeyQuery* from a list of *EntityKeys* is similar. An *EntityKeyList* is simply a strongly-typed collection of *EntityKeys*. All keys in the list must be for the same type or abstract type.

```
var key1 = new EntityKey(typeof(Employee), 1);
var key2 = new EntityKey(typeof(Employee), 2);
var keyList = new EntityKeyList(typeof(Employee), new[] { key1, key2 });
```

```
Dim key1 = New EntityKey(GetType(Employee), 1)
Dim key2 = New EntityKey(GetType(Employee), 2)
Dim keyList = New EntityKeyList(GetType(Employee), { key1, key2 })
```

You can create the *EntityKeyQuery* from the list in familiar ways:

```
var query = keyList.ToKeyQuery();
... or
var query = new EntityKeyQuery(keyList);
```

```
Dim query = keyList.ToKeyQuery()
'...or
Dim query = New EntityKeyQuery(keyList)
```

With the query in hand then you can then do many of the usual things you might do with a query. The *EntityKeyQuery* is not a LINQ query so not all features will be available to it, but you can set its *QueryStrategy* and *EntityManager*, and execute it via the *ExecuteAsync* or *ExecuteQueryAsync* methods.

Because the *EntityKeyQuery* is not a generically typed class, its result is a simple *IEnumerable*. You can cast the result an *IEnumerable<T>*.

```
var employees = entityManager.ExecuteQuery(query).Cast<Employee>();
```

```
Dim employees = entityManager.ExecuteQuery(query).Cast(Of Employee)()
```

```
entityManager.ExecuteQueryAsync(query), op => {
  var items = op.Results.Cast<Employee>();
});
```

```
Dim employees = entityManager.ExecuteQuery(query).Cast(Of Employee)()
```

Disadvantages of the EntityKeyQuery

While the *EntityKeyQuery* has its uses, it has some pretty substantial shortcomings when compared with a standard LINQ query. The biggest of these is that the *EntityKeyQuery* is not composable. This means that we cannot apply any additional restrictions, projections, ordering etc. on these queries. We can of course perform all of the operations on the results of an *EntityKeyQuery* after the query query returns but the server will have still needed to perform the entire query.

The second disadvantage of the *EntityKeyQuery* is that it is really only intended for small numbers of *EntityKeys*. The reason for this is that these methods are implemented so that they in effect create a large "IN" or "OR" query for all of the desired entities by key. The query expression itself can therefore become very large for large numbers of entities. Expressions that are this large will have performance impacts in both serialization as well as query compilation. For those cases where very large numbers of entities need to be refreshed, it is usually a better idea to write a "covering" query that is much smaller textually but returns approximately the same results. You may find that even though you return more entities than are needed with this covering query, the resulting overall performance is still better.

Query by Id

You might wonder how an *EntityKeyQuery* differs from a simple LINQ query by Id. For example, instead of:

```
var key = new EntityKey(typeof(Employee), 1);
var query = key.ToKeyQuery();
```

```
Dim key = New EntityKey(GetType(Employee), 1)
Dim query = key.ToKeyQuery()
```

... we could instead have built an *EntityQuery*:

```
var query =  entityManager.Employees.Where(e => e.EmployeeID == 1);
//.. more useful, use an EntityQuery with FirstOrNullEntity()
var emp =  entityManager.Employees.FirstOrNullEntity(e => e.EmployeeID == 1);
```

```
Dim query = entityManager.Employees.Where(Function(e) e.EmployeeID = 1)
'.. more useful, use an EntityQuery with FirstOrNullEntity()
Dim emp = entityManager.Employees.FirstOrNullEntity(Function(e) e.EmployeeID = 1)
```

The primary difference to DevForce is that it doesn't know that the *EntityQuery* is querying only by the *EntityKey* value, and will thus treat the query as any other query in terms of optimization: it will look for the query in the *QueryCache* and if not present send the query to the datastore, even if the queried entity was already in the entity cache. With the *EntityKeyQuery*, DevForce will first search the entity cache for the requested entity, and only if not present send the query to the datastore.

A second difference is that the LINQ query allows you to return a null entity if the requested entity was not found, while the *EntityKeyQuery* will return an empty enumeration.

Another difference is that the *EntityQuery* is composable, so you can use all standard LINQ operators supported by the *EntityQuery*