What is a QueryStrategy?

QueryStrategy is a class in the *IdeaBlade.EntityModel* namespace. Every query has a *QueryStrategy* property that returns an instance of the *QueryStrategy* type. This property has a default value of null, but can be set on any query as follows:

```
EntityQuery<Order> query01 = myEntityManager.Orders;
query01.QueryStrategy = QueryStrategy.DataSourceThenCache;
```

```
Dim query01 As EntityQuery(Of Order) = myEntityManager.Orders
query01.QueryStrategy = QueryStrategy.DataSourceThenCache
```

In addition, every EntityManager has a *DefaultQueryStrategy* property that is used whenever you do not explicitly specify the query strategy you want to use with a particular query. The *DefaultQueryStrategy* is also used whenever you explicitly set a query's *QueryStrategy* property to null (Nothing in VB). By default the *DefaultQueryStrategy* has a value of *QueryStrategy.Normal* but you can also set it as follows:

```
myEntityManager.DefaultQueryStrategy = QueryStrategy.DataSourceOnly;
```

```
myEntityManager.DefaultQueryStrategy = QueryStrategy.DataSourceOnly
```

Entity navigation (e.g., myEmployee.Orders) is implemented with relation queries governed by the *DefaultQueryStrategy*. In addition, any query whose *QueryStrategy* property has a value of null will be executed with the *DefaultQueryStrategy* for the EntityManager under which it is run.

The *QueryStrategy* class is immutable and has several properties that uniquely define it:

QueryStrategy.FetchStrategy
The *FetchStrategy* controls where DevForce looks for the requested data: in the cache, in the datasource, or in some combination of the two.

QueryStrategy.MergeStrategy
The *MergeStrategy* controls how DevForce resolves conflicts between the states of objects which, although already in the cache, are also retrieved from an external source.

QueryStrategy.InversionMode
The *InversionMode* controls whether DevForce attempts to retrieve objects that are referenced in the query but are not the target type (e.g., the query "give me all Customers with Orders in the current year" will return references to Customer objects, but must process Order objects along the way).

QueryStrategy.TransactionSettings
The *TransactionSettings* permits you to control the TimeOut and IsolationLevel associated with a query, and also whether and how to use the Microsoft Distributed Transaction Coordinator. Defaults to *TransactionSettings.Default*.

QueryStrategy.CacheQueryOptions
The *CacheQueryOptions* control how the query will be run against cache to share the same semantics as the corresponding query against the Entity Framework and the backend database. Defaults to *CacheQueryOptions.Default*.

QueryStrategy.CommunicationRetryPolicy
The retry policy to use for possibly transient communication errors to a remote EntityServer. If not specified, the *CommunicationSettings.Default.DefaultRetryPolicy* is used.

There are five static (*Shared* in VB) properties in the *QueryStrategy* class that return the five most common combinations of a *FetchStrategy*, a *MergeStrategy*, and an *InversionMode*. These will be named and discussed momentarily, but are much easier to understand after examining the available *FetchStrategy*, *MergeStrategy*, and *InversionMode* options.

Pre-Defined QueryStrategies

Every *QueryStrategy* combines a *FetchStrategy*, a *MergeStrategy*, and a *InversionMode*. Since there are five FetchStrategies, five MergeStrategies, and four InversionModes, there are potentially 100 versions of *QueryStrategy*, even keeping the *TransactionSettings* constant. However, in practice, a much smaller set of QueryStrategies suffices for the great majority of purposes. DevForce has identified five of them as being of particular significance, enshrining them as static (Shared in VB) properties of the *QueryStrategy* class. These pre-defined QueryStrategies combine *FetchStrategy*, *MergeStrategy*, and *InversionMode* strategies as shown in the table below:

Fetch and merge strategies of the common query strategies

QueryStrategy	Fetch Strategy	Merge Strategy	InversionMode
Normal	Optimized	PreserveChanges	Try
CacheOnly	CacheOnly	(Not Applicable)	(Not Applicable)

DataSourceOnly	DataSourceOnly	OverwriteChanges	Off
DataSourceOnlyWithInversion	DataSourceOnly	OverwriteChanges	On
DataSourceThenCache	DataSourceThenCache	OverwriteChanges	Try

Here's how you assign a pre-defined *QueryStrategy*:

```
query.QueryStrategy = QueryStrategy.DataSourceThenCache;
```

```
query.QueryStrategy = QueryStrategy.DataSourceThenCache
```

Custom QueryStrategies

As just noted, only five of the possible combinations of a *FetchStrategy* and a *MergeStrategy* are covered by the named QueryStrategies. What if you want one of the other combinations?

You can create your own *QueryStrategy* by supplying the fetch and merge strategy enumerations to its constructor. The result is a new immutable *QueryStrategy* instance. Immutable meaning that we can get the component fetch and merge strategies but we cannot reset them.

Here's an example of the creation and assignment of a custom *QueryStrategy*:

```
QueryStrategy aQueryStrategy =
 new QueryStrategy(FetchStrategy.DataSourceThenCache,
  MergeStrategy.PreserveChanges,
  QueryInversionMode.On);
```

```
Dim aQueryStrategy As New QueryStrategy( _
  FetchStrategy.DataSourceThenCache, _
  MergeStrategy.PreserveChanges, _
  QueryInversionMode.On)
```

There is another, often more useful, method of creating a custom *QueryStrategy*, via the use of one of the *With* overloads provided by the *QueryStrategy* class. These *With* methods allow you to create a new *QueryStrategy* based on an existing *QueryStrategy* with one of the properties changed. For example:

```
QueryStrategy queryStrategy1 = QueryStrategy.Normal.With(FetchStrategy.DataSourceOnly);
QueryStrategy queryStrategy2 = QueryStrategy.DataSourceOnly.With(MergeStrategy.PreserveChangesUpdateOriginal);
QueryStrategy queryStrategy3 = queryStrategy1.With(QueryInversionMode.Off);
```

```
Dim queryStrategy1 As QueryStrategy = _
  QueryStrategy.Normal.With(FetchStrategy.DataSourceOnly)
Dim queryStrategy2 As QueryStrategy = _
  QueryStrategy.DataSourceOnly.With(MergeStrategy.PreserveChangesUpdateOriginal)
Dim queryStrategy3 As QueryStrategy = _
  queryStrategy1.With(QueryInversionMode.Off)
```

DefaultQueryStrategy

We mentioned earlier that the DevForce EntityManager has a *DefaultQueryStrategy* property that can be used to shape the fetch and merge behavior of queries where the *QueryStrategy* is not explicitly specified. The default setting for the EntityManager's *DefaultQueryStrategy* is *QueryStrategy.Normal*. If you leave this setting at its default value, and in an individual query do nothing to countermand the default settings, then the *FetchStrategy* of *Optimized* will be used in combination with the *MergeStrategy* of *PreserveChanges*.

If for some reason you wanted a EntityManager where the default QueryStrategy would always involve a trip to the data source, you could assign a different QueryStrategy, such as *DataSourceOnly,* to the PM's DefaultQueryStrategy property. For a given query, you could still use any desired QueryStrategy by explicitly specifying a different one.

When to use the different QueryStrategies

For most users, most of the time, the DevForce defaults are perfect:

- Satisfy a query from the entity cache whenever possible;
- When a trip to the data source is found necessary, resolve any conflicts that occur between incoming data and data already cache by giving the local version priority; and
- Perform query inversion as needed; if needed and undoable, revert to a *DataSourceOnly FetchStrategy*.

Your choice of a non-default strategy can be driven by a variety of things. For example, suppose your application supports online concert ticket sales. Your sales clerks need absolutely up-to-date information about what seats are available at the

time they make a sale. In that use case, it will be essential to direct your query for available seats against the data source, so a *FetchStrategy* of *DataSourceOnly* might be in order.

In code to handle concurrency conflicts, one might need a *QueryStrategy* with a *MergeStrategy* of *PreserveChangesUpdateOriginal* to make an entity in conflict savable. (The data source version of the conflicted entity would only be retrieved and used to partially overwrite the cache version after the concurrency conflict had been resolved by some predetermined strategy.)

You can and will think of your own reasons to use different combinations of *FetchStrategy*, *MergeStrategy*, and *InversionMode*. Just ask yourself, for a given data retrieval operation, whether the data in the cache is good enough, or you need absolutely current data from the data source. Then ask yourself how you want to resolve conflicts between data already cached and duplicate incoming data. Then consider the process DevForce will use to satisfy the query and make sure it will have the data it needs to give you a correct result. DevForce gives you the flexibility to set the behavior exactly as need it.

Changing a QueryStrategy for a single query execution

You may find yourself with an existing *IEntityQuery* object that you don't want to disturb in any way, but which you would like to run with a different *QueryStrategy* for a specific, one-time purpose. DevForce provides an extension method on *IEntityQuery*, called *With()*, that permits you to this. (Our topic here is *QueryStrategy*, but in fact some overloads of the With() method also (or alternatively) permit you to make a one-time change to the EntityManager against which the query will be run.) Note that this is a different method from the *QueryStrategy.With()* method mentioned earlier.

When a call to *With()* is chained to a query, the result may be either a new query or a reference to the original query. Normally it will be a new query, but if the content of the *With()* call is such that the resultant query would be the same as the original one, a reference to the original query is returned instead of a new query.

If you ever want to be sure that you get a new query, use the *Clone()* extension method instead of *With()*. *With()* avoids the overhead of a *Clone()* when a copy is unnecessary.

```csharp
IEntityQuery<Customer> query00 = _em1.Customers
  .Where(c => c.CompanyName.ToLower().StartsWith("a"));
query00.QueryStrategy = QueryStrategy.DataSourceOnly;
// The With() call in the right-hand side of the following statement
// specifies a query that is materially different from query0, in
// that it has a different QueryStrategy associated with it.
// Accordingly, the right-hand side of the statement will return
// a new query:
IEntityQuery<Customer> query01 =
  query00.With(QueryStrategy.CacheOnly);
// Because the content of the With() call in the right-hand side
// of the following statement doesn't result in a modification
// of query0, the right-hand side will return a reference to
// query0 rather than a new query.
IEntityQuery<Customer> query02 =
  query00.With(QueryStrategy.DataSourceOnly);
// If you want to be certain you get a new query, use Clone()
// rather than With():
EntityQuery<Customer> query03 =
  (EntityQuery<Customer>)query00.Clone();
query03.QueryStrategy = QueryStrategy.DataSourceOnly;
```

```vbnet
Dim query00 As IEntityQuery(Of Customer) = _
  _em1.Customers.Where(Function(c) c.CompanyName.ToLower().StartsWith("a"))
    query00.QueryStrategy = QueryStrategy.DataSourceOnly
    ' The With() call in the right-hand side of the following statement
    ' specifies a query that is materially different from query0, in
    ' that it has a different QueryStrategy associated with it.
    ' Accordingly, the right-hand side of the statement will return
    ' a new query:
Dim query01 As IEntityQuery(Of Customer) = _
  query00.With(QueryStrategy.CacheOnly)
    ' Because the content of the With() call in the right-hand side
    ' of the following statement doesn't result in a modification
    ' of query0, the right-hand side will return a reference to
    ' query0 rather than a new query.
Dim query02 As IEntityQuery(Of Customer) = _
  query00.With(QueryStrategy.DataSourceOnly)
    ' If you want to be certain you get a new query, use Clone()
    ' rather than With():
Dim query03 As EntityQuery(Of Customer) = _
```

```
CType(query00.Clone(), EntityQuery(Of Customer))
    query03.QueryStrategy = QueryStrategy.DataSourceOnly
```