LINQ to SQL Beta 2 to RTM Breaking Changes

November 2007

For the latest information, please see [www.microsoft.com/vstudio](http://www.microsoft.com/vstudio)

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2007 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, MS, Windows, Windows NT, MSDN, Active Directory, BizTalk, SQL Server, SharePoint, Outlook, PowerPoint, FrontPage, Visual Basic, Visual C++, Visual J++, Visual InterDev, Visual SourceSafe, Visual C#, Visual J#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

LINQ to SQL Beta 2 to RTM Breaking Changes

There have been changes to LINQ to SQL since Beta 2 that may affect applications you choose to upgrade. This document covers such changes but does not list new features that do not affect conversion of Beta 2 applications.

# Table<T>.Add and .Remove renamed to .InsertOnSubmit and .DeleteOnSubmit

The methods on **Table<T>** for inserting and deleting rows from tables have been renamed to better convey that the updates are deferred until the call to **DataContext.SubmitChanges**. Note that these changes apply only to the methods on **Table<T>** and not **EntitySet<T>**.

|  |  |
| --- | --- |
| Beta 2 Name | RTM Name |
| Add | InsertOnSubmit |
| AddAll | InsertAllOnSubmit |
| Remove | DeleteOnSubmit |
| RemoveAll | DeleteAllOnSubmit |

We have also changed the members of **ChangeSet** to match these new names:

|  |  |
| --- | --- |
| Beta 2 Name | RTM Name |
| AddedEntities | Inserts |
| RemovedEntities | Deletes |
| ModifiedEntities | Updates |

### Action Item

Rename references to these members to match the new names.

# Table<T>.Attach changed to throw an exception if attached entity has not been detached

The purpose of **Table<T>.Attach** method is to let you make a **DataContext** aware of entities that are transferred from other tiers, usually through serialization/deserialization. These entities are typically retrieved using a different instance of the **DataContext** than the one used for making updates. Once the entities are attached, the **DataContext** can track their original state for update purposes as if they had been fetched initially by this **DataContext**. However, it is important to make sure that **Attach** is called only for detached instances obtained through deserialization. **Attach** is not intended for transferring entities from one live **DataContext** instance to another within the same process. If the **DataContext** instance used to retrieve an entity is live, there may be deferred loaders that effectively keep the entity attached to the **DataContext**. Unlike in Beta 2, an exception will now be thrown if **Attach** is called on a second **DataContext** instance for such an entity.

In RTM, we also more intelligently handle related entities linked to the entity being attached. They can be attached later for appropriate action.

### Action Item

Be sure to call **Attach** only on entities that are not in scope of another live **DataContext** instance on this tier.

# Partial initialization of entities within queries no longer allowed

The typical way to return data from a query is to select out the range variable you referenced in the **from** clause:

 var q = from c in db.Customers

 where Country == "USA"

 select c;

The query above constructs actual **Customer** entity objects and passes them on to you.

In Beta 2, the following type of query containing an explicit **new Customer** clause was also possible:

 var q = from c in db.Customers

 where Country == "USA"

 select new Customer { Name = c.Name, City = c.City };

This code seems fairly reasonable, projecting out the **Name** and **City** of a certain customer into a **Customer** object, but there are multiple problems you can encounter if you then try to use this object like a normal **Customer** object. First, the **Customer** object is only partially filled in, leaving many fields blank including the primary key field, **CustomerID**. Without the **CustomerID** field, the **DataContext** cannot persist changes you make to this **Customer** back to the database on **SubmitChanges**(). Even if the **CustomerID** field is loaded, any other missing fields will cause the optimistic concurrency checks to fail on update as the full set of original field values is required to use optimistic concurrency. In addition, after fetching this partially constructed **Customer** entity, the object will be stored in the entity cache and any future queries that need to materialize this entity will retrieve this partially constructed version instead of the full version.

To protect against all of these problems, we now throw an exception during query translation if the query contains a constructor call to a known entity class (non-entity classes may still be freely constructed). If you wish to project out a restricted set of result fields, the correct way to do so is to project into an anonymous type. This is done by saying **select new {…}** instead of **select new Customer {...}**. The object you get back is not of type **Customer**, so there are no changes to track, and the partial object will not end up in the entity cache:

 var q = from c in db.Customers

 where Country == "USA"

 select new { Name = c.Name, City = c.City };

### Workaround

Replace entity constructors within query expressions with either non-entity type initializers (anonymous or nominal) or use the range variable from the **from** clause to get a complete entity.

# OnValidate breaking changes

## ChangeAction parameter added to OnValidate

**OnValidate** now has a parameter of type **ChangeAction** to indicate whether the change being validated was an Insert, Update, or Delete:

partial void OnValidate(System.Data.Linq.ChangeAction action);

namespace System.Data.Linq

{

 public enum ChangeAction

 {

 None = 0,

 Delete,

 Insert,

 Update

 }

}

### Action Item

Add a parameter named **action** of type **ChangeAction** to any **OnValidate** implementations you have.

## OnValidate no longer called on “one” table when “many” membership changes

In a one-to-many relationship such as **Customers** to **Orders**, the information about which **Customer** a given **Order** is associated with is stored in the **Orders** table, not the **Customers** table. Therefore, when an **Order** is added or removed from an **EntitySet**, or when **Order.Customer** is set directly, **OnValidate** is no longer called on the **Customers** table, only on the **Orders** table.

### Action Item

Consider moving logic relevant to validating orders from the **Customer**’s **OnValidate** implementation into the **Orders** table’s **OnValidate** implementation.

# Deleting an entity now removes it from EntitySets / 1:1 relationships

In Beta 2, when you deleted an entity from its table, you also had to manually remove the entity from any **EntitySet**s and deferred 1:1 relationships in which the entity was a member to maintain the consistency of your object graph. This cleanup is now done automatically during **DeleteOnSubmit**.

### Action Item

Remove code that manually fixes up orphaned **EntitySet** or deferred 1:1 references in your object graph caused by **DeleteOnSubmit**.

# Visual Basic String = Nothing and String <> Nothing fixes

Beta 2 has incorrect SQL translations for some Visual Basic string comparisons.

In Visual Basic, if you have a **String** variable s, testing **s = Nothing** is equivalent to testing **s = “”**. In LINQ to SQL, this should be translated as **WHERE [t0].[s] = “”** to maintain equivalent semantics. This was instead getting translated to **[t0].[s] IS NULL**. An equivalent mistranslation was occurring for **s <> Nothing**.

### Action Item

Check your Visual Basic code that tests database string values to be sure that you use **s = Nothing** or **s = “”** when you want to test for empty string and **s Is Nothing** when you want to test for NULL.

# Connection strings no longer persist plaintext passwords

When the connection string provided to a **DataContext** contains a plaintext password, it is saved until either a query is executed, causing the connection to open, or **CreateDatabase** is called to generate a database from the mapping schema. The password is then removed from the cached connection string to help prevent it from becoming exposed to a malicious user. Once the password has been removed, you may no longer call **DatabaseExists**, **DeleteDatabase** or **CreateDatabase**, as these require connecting again to the master catalog, but you may continue to query the database you’ve opened.

Because you cannot call **DeleteDatabase** after a query has been executed, it is now invalid to do this:

 DataContext ctxt = new DataContext("Server=Srv;User=A;Password=B");

 Customer c = ctxt.Customers.First();

 ctxt.DeleteDatabase();

 ctxt.CreateDatabase();

However, the following pattern still works because the password is not stripped out until the call to **CreateDatabase** that sets up the main connection:

 DataContext ctxt = new DataContext("Server=Srv;User=A;Password=B");

 if (ctxt.DatabaseExists())

 ctxt.DeleteDatabase();

 ctxt.CreateDatabase();

 Customer c = ctxt.Customers.First();

### Workaround

Either cache the connection string securely yourself and create a new **DataContext** when you need to call **DeleteDatabase**/**DatabaseExists**, or refactor your code to ensure that **DeleteDatabase**/**DatabaseExists** operations are performed before you execute a query or run **CreateDatabase**.

# Code-gen breaking changes for SQLMetal and O/R Designer

Besides the LINQ to SQL runtime breaking changes above, there have also been some changes made to the code generation that are worth calling out as they will affect the meaning of your generated classes after you regenerate them for RTM.

*Please do not try to use the list below to manually convert the .CS files generated by SQLMetal or the O/R Designer from Beta 2 to RTM. There have been many minor bug fixes to the code generation that you will not obtain if you do not regenerate these .CS files after updating to RTM.*

## XML columns now map to XElement by default

The default mapping for XML database columns had previously been **XDocument**, but this default has been changed to **XElement**. This was done due to a change to **XDocument** since Beta 2 to not implement **IXmlSerializable**, thus preventing XML serialization of **XDocument**s. Mapping XML columns to **XDocument** is otherwise still supported and you may modify the DBML or the O/R Designer surface to continue to map these columns manually to **XDocument** if you wish.

### Workaround

Manually map XML columns to **XDocument** in DBML or O/R Designer or modify code referencing field to use **XElement** instead.

## Visual Basic code-gen for String columns modified to catch property changes between Nothing and empty string

The code generated to determine whether a property setter is actually making a field change was incorrect for Visual Basic when the column was of type **String**. The Visual Basic code had used the **=** operator to compare the new value coming into the property setter with the current value to determine whether to mark the field changed and call the **On*Property*Changed** partial method. As the **=** operator in Visual Basic considers an empty string and the value **Nothing** to be equivalent, it was not possible to change the value of the database field from **NULL** to **“”** or vice versa in Visual Basic. The code-gen now uses **System.Equals** to do the comparison, allowing the same changes in Visual Basic as in C#.

### Action Item

Be sure that none of your code had relied on the fact that **Nothing** and **“”** were considered equivalent in entity property setters.

## OnCreated now called after EntitySets are initialized

As part of the **Initialize** method generated for each entity, the **OnCreated** partial method is now invoked after any **EntitySet**s are initialized, instead of before. This lets you add logic to your **OnCreated** implementation that can operate on the newly initialized **EntitySet**s.

### Action Item

Remove any logic from your **OnCreated** implementation that relied on the **EntitySet**s not being initialized yet.