LINQ-to-SharePoint runtime library specification

- Version 0.2

Abstract

This paper describes the runtime library infrastructure used in the LINQ-to-SharePoint implementation, required to translate LINQ queries into CAML and to provide an entity model.

For comments or questions, please send e‑mail to info@bartdesmet.net.

Contents

[LINQ-to-SharePoint runtime library specification 1](#_Toc168997566)

[Introduction 3](#_Toc168997567)

[Overview of entities 3](#_Toc168997568)

[Design goals 3](#_Toc168997569)

[SharePointListEntity base class 3](#_Toc168997570)

[ListAttribute 3](#_Toc168997571)

[FieldAttribute 4](#_Toc168997572)

[Supported field types 5](#_Toc168997573)

[None 6](#_Toc168997574)

[Boolean 6](#_Toc168997575)

[Calculated 6](#_Toc168997576)

[Choice 6](#_Toc168997577)

[Counter 7](#_Toc168997578)

[Currency 7](#_Toc168997579)

[DateTime 7](#_Toc168997580)

[Integer 8](#_Toc168997581)

[Lookup 8](#_Toc168997582)

[LookupMulti 8](#_Toc168997583)

[MultiChoice 9](#_Toc168997584)

[Note 9](#_Toc168997585)

[Number 9](#_Toc168997586)

[Text 9](#_Toc168997587)

[URL 9](#_Toc168997588)

[ChoiceAttribute 10](#_Toc168997589)

© 2007 Bart De Smet. All rights reserved. This specification is provided under the license agreements for LINQ-to-SharePoint. For further details on licensing, please refer to: <http://www.codeplex.com/LINQtoSharePoint/Project/License.aspx>.

© 2007 Bart De Smet. All rights reserved.

[www.codeplex.com/LINQtoSharePoint](http://www.codeplex.com/LINQtoSharePoint)

# Introduction

This paper describes the runtime infrastructure provided by LINQ-to-SharePoint, including the entity framework and the query parser that translates LINQ queries into the corresponding CAML queries.

The reader should be familiar with core LINQ concepts, such as the LINQ query syntax for C# 3.0 and VB 9.0 and custom data providers using the IQueryable interface. The reader should also be familiar with the SharePoint technology, especially the concept of lists. Prior knowledge of CAML isn’t a must but highly recommended.

# Overview of entities

Entities are used to represent SharePoint list items in a strongly-typed fashion, ready for use in LINQ-capable languages such as C# 3.0 and VB 9.0. Entities can be created manually or can be generated automatically using a tool called SpMetal which is subject of another specification.

## Design goals

LINQ-to-SharePoint entities should:

* Provide a mapping between SharePoint types and .NET types.
* Carry metadata information about the underlying SharePoint list.
* Allow for updates that are fed back to the SharePoint list.

## SharePointListEntity base class

All entities in LINQ-to-SharePoint derive from a SharePointListEntity base class. This type is responsible for storing list item field data in a (weakly-typed) Dictionary-based manner, mapping friendly field names on objects. Two methods, GetValue and SetValue, are provided to work with the stored data.

Subclasses (*entities*) provide a strongly-typed layer on top of this infrastructure by means of properties that use the GetValue method and (optionally) the SetValue method. The LINQ-to-SharePoint library itself can use the GetValue and SetValue methods at will, e.g. to store query results. Using this level of indirection, changes to entities can be monitored in order to support updating.

The class diagram is shown in Figure 1. Beside the GetValue and SetValue methods, the class has a default constructor too. The class modifier is public; the methods are non-virtual.

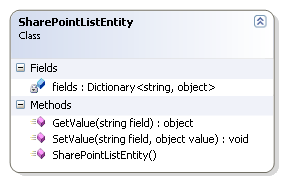


Figure 1 - SharePointListEntity class

## ListAttribute

In order to keep metadata about the SharePoint list represented by a SharePointListEntity subclass *entity*, a custom attribute called ListAttribute is provided. It keeps the following information:

* List name (e.g. Customers)
* List identifier (e.g. 34c90895-fbf3-4da7-a260-4b3ddc67146d)
* Relative path to the list on the SharePoint site (e.g. /Lists/Customers)
* The list version number (e.g. 24)

The list name can be set using the constructor only and is mandatory. Other attribute properties provide getters and setters and can be set using appropriate syntax, as shown below (C#):

[List("Customers", Id = "34c90895-fbf3-4da7-a260-4b3ddc67146d",  
 Version = 24, Path = "/Lists/Customers")]  
class Demo : SharePointListEntity  
{

This attribute is used by the LINQ-to-SharePoint runtime infrastructure to connect to the right list for query execution. More specifically, the list name (List) is used when connecting over SharePoint web services. When using the SharePoint object model, the path to the list (Path) is used to point at the list. When version enforcement is requested by the developer, the version number (Version) will be used by the runtime infrastructure to be compared to the version number of the actual list definition on the server. The list identifier (Id) isn’t used by the runtime infrastructure itself but is kept for reference purposes and for future usage by the runtime and/or supporting tools. Although this value represents a Guid it’s kept as a string value, in a hypen-separated form without trailing brackets (“D” format).

The class diagram is shown in Figure 2. Fields are hidden. The custom attribute class is sealed, derives from System.Attribute and has an AttributeUsage that targets only AttributeTargets.Class with AllowMultiple set to false.

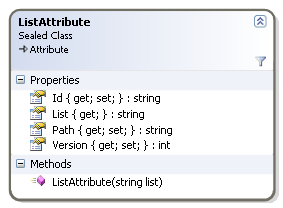


Figure 2 - ListAttribute custom attribute

## FieldAttribute

Entities contain properties that act as mappings for list item fields (list columns). To keep metadata for these fields, a custom attribute called FieldAttribute is provided. It keeps the following information:

* Field name
* Field type from the FieldType enumeration
* Field identifier
* Indicator for calculated fields
* LookupField for specification of display fields in lookup fields
* OtherChoice reference to a property containing a non-standard choice in (Multi)Choice fields
* Indicator for primary key fields
* Indicator for read-only fields

The field name and field type can be set using the constructor only and are mandatory. Other attribute properties provide getters and setters and can be set using appropriate syntax. An example of a simple field mapping is shown below (C#):

[Field("Title", FieldType.Text)]  
 public string Title  
 {  
 get { return (string)GetValue("Title"); }  
 set { SetValue("Title", value); }  
 }

Notice the use of the SharePointListEntity base class GetValue and SetValue methods in the definition for the property getter and setter.

The class diagram is shown in Figure 2. Fields are hidden. The custom attribute class is sealed, derives from System.Attribute and has an AttributeUsage that targets only AttributeTargets.Property with AllowMultiple set to false.

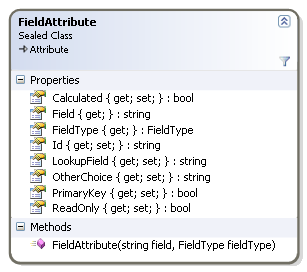


Figure 3 - FieldAttribute custom attribute

No support is provided for class fields. Entities can only provide mappings to SharePoint list item fields by means of class properties. Field attributes are required for all fields that participate in LINQ-to-SharePoint queries. No default mappings are done if field attributes are missing.

The field name (Field) is used by the runtime infrastructure to generate CAML query FieldRef elements that are used to refer to list fields. The field name gets XML-encoded before any use by the runtime. Entity definitions shouldn’t attempt any XML encoding whatsoever and can use regular Unicode field names. For example, for field names containing spaces the following is correct:

[Field("First name", FieldType.Text)]  
 public string FirstName

It’s up to the runtime to transform the field name in a format consumable by the SharePoint object model or the SharePoint web services.

For a discussion of the supported types (FieldType) and other associated properties (Calculated, LookupField, OtherChoice), refer to the next section in this specification.

The field identifier (Id) isn’t used by the runtime infrastructure itself but is kept for reference purposes and for future usage by the runtime and/or supporting tools. Although this value represents a Guid it’s kept as a string value, in a hypen-separated form without trailing brackets (“D” format).

In the current implementation, the read-only field indicator (ReadOnly) isn’t used but it’s reserved for future usage. Properties for fields marked as ReadOnly should not have a setter accessor.

## Supported field types

This section describes all of the supported field types in LINQ-to-SharePoint as well as the associated field mappings. All field types are kept in a FieldTypes enumeration as shown in . The values in this enum are not set automatically but are mapped to the corresponding values from the Microsoft.SharePoint.SPFieldType enumeration as available in the SharePoint object model v12.0.0.0.

Note: The LookupMulti field type isn’t available in the SPFieldType enumeration and has a value of 101 in the FieldType enumeration. Future field types that aren’t in SPFieldType should be numbered linearly starting from 102.

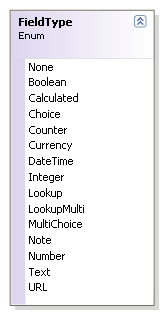


Figure 4 - Supported field types

### None

Mapped to SPFieldType.Invalid (value 0); shouldn’t be used in entities. Any use of this field type will result in a runtime exception of type InvalidOperationException.

### Boolean

Used for Yes/No fields; should be used on System.Boolean fields only. In case the target list field isn’t mandatory, a mapping should be done on Nullable<System.Boolean> (*bool?* in C#).

### Calculated

(Obsolete) Shouldn’t be used directly in entity field definitions; instead, the underlying type (of the calculated value) should be used for the FieldType property, combined with a Calculated property set to true. Since all calculated columns are read-only, the Calculated property should be combined with a true-valued ReadOnly property. Because of this, no setter is allowed. For example (C#):

[Field("DoubleAge", FieldType.Number, ReadOnly = true,  
 Calculated = true)]  
 public double? DoubleAge  
 {  
 get { return (double?)GetValue("DoubleAge"); }  
 }

The underlying type corresponds to the ResultType value in the SharePoint list definition schema.

### Choice

Used for SharePoint Choice fields commonly known as radio buttons or drop-down menus. For checkboxes, see MultiChoice. Choice fields are mapped using enums. An example (C#):

[Field("Membership Type", FieldType.Choice)]  
 public MembershipType? MembershipType  
 {  
 get { return (MembershipType?)GetValue("MembershipType"); }  
 set { SetValue("MembershipType", value); }  
 }

enum MembershipType : uint { Platinum, Gold, Silver, Bronze }

The used enums are of type uint and contain the available choices in no particular order with compiler-aided auto-numbering. In case the field is not required, the mapping should use the Nullable<T> type for the property (*T?* in C#), where T stands for the enum type.

If fill-in choices are enabled on the target field, an OtherChoice field attribute property should be specified, pointing to a string-typed entity property that will hold the fill-in choice value (if present). The name for this property is arbitrary (SpMetal will suffix the field name with “Other” to generate the fill-in choice string property name). The referenced entity property should have the FieldType.Text field type and should have the same field name and field identifier as the Choice field. An example (C#):

[Field("Favorite food", FieldType.Choice,  
 Id = "bdf129e3-b899-4aa0-badb-6529a630a01e",  
 OtherChoice = "FavoriteFoodOther")]  
 public FavoriteFood? FavoriteFood  
 {  
 get { return (FavoriteFood?)GetValue("FavoriteFood"); }  
 set { SetValue("FavoriteFood", value); }  
 }  
  
 [Field("Favorite food", FieldType.Text,  
 Id = "bdf129e3-b899-4aa0-badb-6529a630a01e")]  
 public string FavoriteFoodOther  
 {  
 get { return (string)GetValue("FavoriteFoodOther"); }  
 set { SetValue("FavoriteFoodOther", value); }  
 }

Note: Because of this mapping, users can’t rely on a simple null-check on the entity Choice property to determine whether or not a Choice field has been set. An additional null-check on the referenced fill-in choice field will have to be performed. Versioning might be another problem if additional choices are added to the list definition. In such a case, pre-defined values will be stored in fill-in choice fields if the enum isn’t up to date. List schema version enforcement (see further) will help to solve this.

See the section about ChoiceAttribute for more information about choice value mappings.

### Counter

Used for auto-numbered fields, typically for primary key list item identification fields; shouldn’t be used for anything other than primary key fields with the PrimaryKey indicator set. Counter fields should be configured ReadOnly too, without a setter accessor. Example (C#):

[Field("ID", FieldType.Counter, PrimaryKey = true, ReadOnly = true)]  
 public int ID  
 {  
 get { return (int)GetValue("ID"); }  
 }

### Currency

Used for Currency fields; mapped to double values, possibly Nullable<double> (*double?* in C#).

Note: This mapping doesn’t provide any information about the currency unit or currency format, which should be known by the end-user when appropriate.

### DateTime

Used for Date and Time fields; mapped on DateTime values, possibly Nullable<DateTime> (*DateTime?* in C#). In case the Date and Time format has been set to ‘Date Only’, a midnight hour will be considered during the mapping.

Note: No calendar, time zone and/or date/time format considerations are made for/during this mapping. The end-user should be aware of the SharePoint server’s date/time settings when relevant.

### Integer

Used for numerical integer-valued fields (0 decimals); mapped on Int32 values, possibly Nullable<Int32> (*int?* in C#).

### Lookup

Used for single-valued Lookup fields; mapped on an entity type. Lookup fields should have the LookupField attribute property set, referring to the name of the property on the referenced entity type that’s being used as the display property for the Lookup field in SharePoint. An example (C#):

[List("Products", ...)]  
class Product  
{  
 ...  
  
 [Field("Supplier", FieldType.Lookup, LookupField = "Title")]  
 public Supplier Supplier  
 {  
 get { return (Supplier)GetValue("Supplier"); }  
 set { SetValue("Supplier", value); }  
 }  
}  
  
[List("Suppliers", ...)]  
class Supplier  
{  
 ...  
  
 [Field("Title", FieldType.Text)]  
 public string Title  
 {  
 get { return (string)GetValue("Title"); }  
 set { SetValue("Title", value); }  
 }  
}

Note: When loading entities that contain Lookup fields, lazy loading will be applied. Only the target entity’s primary key value will be kept in a *lazy loading thunk* till the property is retrieved via the GetValue method of SharePointListEntity. For more information, see the section about lazy loading further in this specification.

See LookupMulti for multi-valued Lookup fields.

### LookupMulti

Used for multi-valued Lookup fields; mapped on an IList<T> of the referenced entity type. Lookup fields should have the LookupField attribute property set, referring to the name of the property on the referenced entity type that’s being used as the display property for the Lookup field in SharePoint. An example (C#):

[List("Products", ...)]  
class Product  
{  
 ...  
  
 [Field("Supplier", FieldType.LookupMulti, LookupField = "Title")]  
 public IList<Supplier> Suppliers  
 {  
 get { return (IList<Supplier>)GetValue("Supplier"); }  
 }  
}

Only a getter accessor is provided for the entity property. This avoids the list to be replaced by another one. Instead, the list itself can be manipulated. A future IList<T> implementation will provide support for LookupMulti field manipulation with update support.

A IList<T> collection type is used to preserve the ordering of the referenced entities as kept in the SharePoint list. This allows end-users to rely on this ordering when appropriate, for example if the LookupMulti field has some kind of priority ordering semantics.

Note: LookupMulti has value 101 in the FieldType enumerator since no equivalent enum value is present in the SPFieldType enumeration of the SharePoint object model.

### MultiChoice

Used for SharePoint Choice fields commonly known as checkboxes. MultiChoice fields are mapped using flagged enums. An example (C#):

[Field("Activities", FieldType.MultiChoice)]  
 public Activities? Activities  
 {  
 get { return (Activities?)GetValue("Activities"); }  
 set { SetValue("Activities", value); }  
 }  
  
 [Flags] enum Activities : uint { Quiz = 1, Adventure = 2, Culture = 4 }

The used enums are of type uint and contain the available choices with powered two values starting with 1. In case the field is not required, the mapping should use the Nullable<T> type for the property (*T?* in C#), where T stands for the enum type.

If fill-in choices are enabled on the target field, an OtherChoice field attribute property should be specified, pointing to a string-typed entity property that will hold the fill-in choice value (if present). See the section about Choice fields for more information about this, together with an example.

See the section about ChoiceAttribute for more information about choice value mappings.

### Note

Used for Note fields; mapped on string entity properties. The entity model does not distinguish between Note and Text fields for what the mapping is concerned.

### Number

Used for numerical fields; mapped on Double entity properties, possibly Nullable<Double> (*double?* in C#).

### Text

Used for Text fields; mapped on string entity properties. The entity model does not distinguish between Note and Text fields for what the mapping is concerned.

### URL

Used for Hyperlink or Picture fields; mapped on a special Url type as shown in . This type combines System.Uri functionality with a friendly name (FriendlyName property), representing the Url field’s caption and underlying URL.

The Url type derives from System.Uri and adds a FriendlyName property with getter and setter. It’s marked as Serializable and overrides the Equals and GetHashCode methods for equality checks. It also provides operator overloads for equality (==) and inequality (!=) to make field usage in LINQ queries easier. An additional constructor with SerializationInfo and StreamingContext parameters has to be provided to allow deserialization, as well as a new GetObjectData method with similar parameters for serialization to XML.

Note: This enum value should be named in an all-uppercase format in order to reflect the corresponding value from SPFieldType. This conflicts with the framework design guideline CA1705 concerning “Long acronyms should be Pascal-cased”.

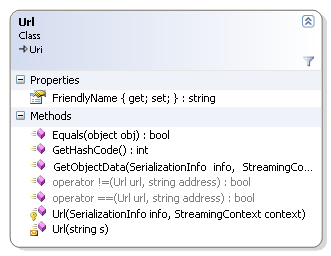


Figure 5 - Url type

## ChoiceAttribute

When mapping Choice and MultiChoice fields to enums it’s possible to be faced with pre-set choice values that cannot be mapped directly to the target language. Using the ChoiceAttribute, an enum field can be mapped to an underlying choice name. If no ChoiceAttribute is present, the name of the enum field will be considered to match the target choice name as defined on the SharePoint list. An example of a choice mapping is shown below (C#):

enum Actors : uint  
 {  
 Chaplin = 1,  
 [Choice("Laurel & Hardy")] LaurelHardy = 2  
 }

The Choice property is the only property defined on ChoiceAttribute and is read-only. It can be set using the constructor’s parameter. No XML encoding should be performed on the property value, this will be done by the runtime when required.

The class diagram is shown in Figure 6. Fields are hidden. The custom attribute class is sealed, derives from System.Attribute and has an AttributeUsage that targets only AttributeTargets.Field with AllowMultiple set to false.

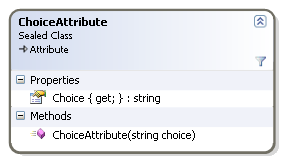


Figure 6 - ChoiceAttribute

## Lazy loading

Dfd

# LINQ query provider

Entities are used to represent S