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](#_Toc169002083)

[Introduction 4](#_Toc169002084)

[Overview of entities 4](#_Toc169002085)

[Design goals 4](#_Toc169002086)

[SharePointListEntity base class 4](#_Toc169002087)

[ListAttribute 4](#_Toc169002088)

[FieldAttribute 5](#_Toc169002089)

[Supported field types 6](#_Toc169002090)

[None 7](#_Toc169002091)

[Boolean 7](#_Toc169002092)

[Calculated 7](#_Toc169002093)

[Choice 7](#_Toc169002094)

[Counter 8](#_Toc169002095)

[Currency 8](#_Toc169002096)

[DateTime 8](#_Toc169002097)

[Integer 9](#_Toc169002098)

[Lookup 9](#_Toc169002099)

[LookupMulti 9](#_Toc169002100)

[MultiChoice 10](#_Toc169002101)

[Note 10](#_Toc169002102)

[Number 10](#_Toc169002103)

[Text 10](#_Toc169002104)

[URL 10](#_Toc169002105)

[ChoiceAttribute 11](#_Toc169002106)

[Lazy loading 11](#_Toc169002107)

[Query parser 12](#_Toc169002108)

© 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.

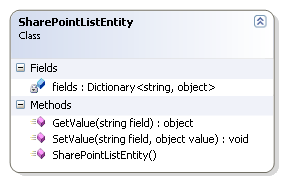


Figure 1 - SharePointListEntity class

## 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.

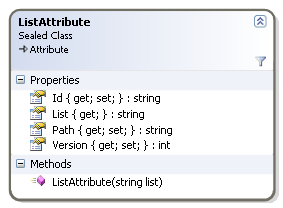


Figure 2 - ListAttribute custom attribute

## 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.

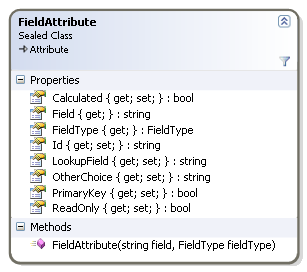


Figure 3 - FieldAttribute custom attribute

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 Figure 4. 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.

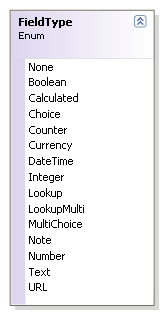


Figure 4 - Supported field types

### 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 Figure 5. 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”.

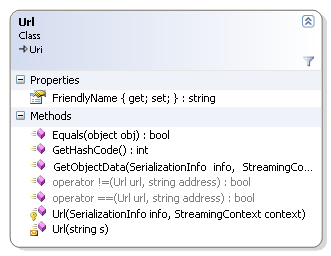


Figure 5 - Url type

## 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, [Choice("Laurel & Hardy")] LaurelHardy  
 }

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.

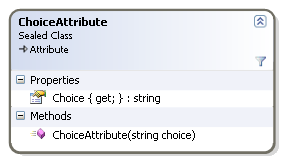


Figure 6 - ChoiceAttribute

## Lazy loading

LINQ-to-SharePoint implements *lazy loading* for Lookup and LookupMulti fields. This allows for more efficient querying. 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. This lazy loading thunk acts as a placeholder for one or more to-be-retrieved linked entities.

The ILazyLoadingThunk interface and the corresponding implementation are shown in Figure 7.

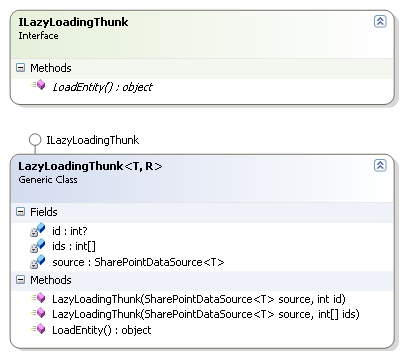


Figure 7 - Lazy loading thunk

When the SharePointListEntity’s GetValue method encounters a ILazyLoadingThunk object when looking for the value in the fields dictionary, it calls the thunk’s LoadEntity method to retrieve the referenced entity (for Lookup fields) or list of entities (for MultiLookup fields). Once retrieved, the thunk is replaced by the retrieved object or IList<T> of objects.

In order to retrieve the referenced entity or entities, the query execution engine will keep the entity identifier(s) (primary key field) as a field in the LazyLoadingThunk<T,R> object. The T type parameter refers to the parent list; R refers to the referenced entity type. The LoadEntity implementation relies on the SharePointDataSource<R>’s GetEntityById or GetEntityByIds method to retrieve an object of type R or an IList<R> object based on the kept id or ids array (see further).

# Query parser

At the core of the LINQ-to-SharePoint runtime infrastructure is the query parser, responsible for the translation of LINQ query expression trees into CAML. This section of the specification describes the query parser in detail.

## SharePointDataSource<T>

In LINQ-to-SharePoint each data source connecting to a SharePoint list is represented as an instance of type SharePointDataSource<T> with T the entity type for the list. This allows for strongly-typed queries to be written using comfortable LINQ syntax.

The SharePointDataSource<T> class is responsible to translate the LINQ query into CAML and to send the query to the SharePoint web site either using the SharePoint web services or using the SharePoint object model.

### Constructors

Two public constructors have to be provided:

**SharePointDataSource(Uri siteUrl)**

Used toconnect to the SharePoint list using web services. The Uri has to point to the root of the SharePoint website to connect to. The system will connect to the Lists web service by appending */vti\_bin/lists.asmx*.

A wsdl.exe v2.0.0.50727 generated web service proxy class called Lists should be created to establish web service communication with the Lists service. Only the methods GetList and GetListItems should be retained.

The web service proxy instance should be kept in a field called *\_ws*. If this field is null during execution, a connection using the SharePoint object model will be considered. The web service proxy object should set the System.Net.CredentialCache.DefaultNetworkCredentials on its Credentials property. This default can be overriden using the Credentials property of SharePointDataSource<T> (see further).

Also, the used Uri should be kept in a *\_uri* field for further reference. The name of the list as specified on T’s ListAttribute’s List property should be kept in a *\_wsList* field for further reference. Nevertheless, the system shouldn’t depend on a *\_wsList* null-check to determine between web services or the SharePoint object mode; the *\_ws* field should be used as a decision basis instead.

If the passed Uri object is null, an ArgumentNullException should be thrown.

**SharePointDataSource(SPSite site)**

Used to connect the the SharePoint list using the SharePoint object model. The SPSite object has to point to the SharePoint site with the RootWeb that contains the list to connect to, and will be kept in a *\_site* field. This field may be used in a null-check to determine between web services or the SharePoint object mode.

The list to connect to will be inferred from T’s ListAttribute’s Path property that can be passed to the GetList method of the SPSite RootWeb property (of type SPWeb). The retrieved SPList object should be kept in a *\_list* field. This field may be used in a null-check to determine between web services or the SharePoint object mode.

If the passed SPSite object is null, an ArgumentNullException should be thrown.

Note: This connection methodology can only be used on the same machine as the one the SharePoint site is running on, due to technical limiations of the SharePoint object model. It’s up to the end-users to decide on the connection technology and that choice shouldn’t affect further operation.

Note: This constructor isn’t CLS compliant because SPSite isn’t. Apart from this member, the rest of the SharePointDataSource<T> implementation should be CLS compliant.

In addition, the following private constructors are required:

**SharePointDataSource(Lists ws, string list)**

Used internally to create a new SharePoint datasource object using the SharePoint web services. The proxy object (ws) and the list name (list) should be copied to the *\_ws* and *\_wsList* fields respectively.

If any of both parameters is null, an ArgumentNullException should be thrown.

**SharePointDataSource(SPList list, bool performCheck)**

Used to connect the the SharePoint list using the SharePoint object model. The SPSite object (list) should be copied to the *\_list* field. If the performCheck parameter is true, the system should check for the presence of the ListAttribute on the referenced T entity type. This can be done using a GetListAttribute method (see further).

If the list parameter is null, an ArgumentNullException should be thrown.

All constructors should set a field called ­*\_originalType* based on the runtime type of the T type parameter. This field will be used by the SharePointDataSource to create entity instances as part of or during the projection operation.

### Properties

The following properties are intended for direct use by end-users of SharePointDataSource:

**ICredentials Credentials**

Used to get/set the network credentials for web services communication. Calling this property will result in an InvalidOperationException when the SharePointDataSource instance is used with the SharePoint object model (see constructors). By default the System.Net.CredentialCache.DefaultNetworkCredentials will be used, as set by the **SharePointDataSource(Uri siteUrl)** constructor.

**TextWriter Log**

Gets/sets a System.IO.TextWriter object used to log diagnostic information about generated CAML queries. More information about logging will be provided further in this specification.

**bool CheckListVersion**

If set, the runtime will check the online list version against the version number kept in the ListAttribute for the entity type T. See further for more information. This validation mechanism can be used to detect list version mismatches.

### Interface implementations

The signature of the SharePointDataSource<T> class is displayed below:

public sealed class SharePointDataSource<T> : IOrderedQueryable<T>, IDisposable

The class should be declared as sealed and should implement System.Linq.IOrderedQuerable<T> (see further) and IDisposable. The Dispose method should make sure all resources used for connections etc. are disposed properly. This makes it possible to use the SharePointDataSource in a *using* block in C# and VB.

## Query parsing logic

The core of the SharePointDataSource<T> implementation takes care of parsing LINQ queries. To enable this, the class implements IOrderedQueryable<T>. This section will outline the rules applied when parsing queries and will specify the CAML constructs used in the translation.

### Supported LINQ query keywords

This section outlines the query keywords and constructs supported by LINQ-to-SharePoint (C# 3.0 and higher):

* **from *r* in *src***  
  One single from clause per LINQ query is currently supported. The *src* object is of type SharePointDataSource<T> with T the entity type for the list to be queried.
* **where *pred***One single where clause per LINQ query is currently supported. The *pred* predicate is subject to restrictions as outlined further in this specification.
* **select *proj***  
  The select projection operation is fully supported with all flexibility.
* **orderby *fld* [ascending | descending]**  
  One or more ordering clauses are supported, including decending ones.

Non-supported constructs include groupby and join. Currently, LINQ-to-SharePoint queries cannot be combined with other data sources as part of one cross-domain LINQ query.

Note: Similar LINQ query keywords are supported in VB 9.0 and higher. Where appropriate, this specification will detail differences between both languages if it affects the construction of the LINQ-to-SharePoint query parser.

### CAML query operators

*This is an informative section based on the Windows SharePoint Services SDK[[1]](#endnote-2) information.*

CAML (Collaborative Application Markup Language) is an XML-based language used for various purposes – including quyering – in SharePoint. It supports the following query elements (**Query Schema**):

Query   
 Where  
 Logical Joins  
 And  
 [Logical Joins]  
 [Comparison Operators]  
 Or  
 [Logical Joins]  
 [Comparison Operators]

Comparison Operators  
 BeginsWith  
 FieldRef  
 Value  
 Contains  
 FieldRef  
 Value  
 Eq  
 FieldRef  
 Value  
 Geq  
 FieldRef  
 Value  
 Gt  
 FieldRef  
 Value  
 Leq  
 FieldRef  
 Value  
 Lt  
 FieldRef  
 Value  
 Neq  
 FieldRef  
 Value  
 DateRangesOverlap (\*)  
 IsNotNull  
 FieldRef  
 IsNull  
 FieldRef

Order/Group Operators  
 OrderBy  
 FieldRef  
 GroupBy (\*)

\* currently not supported by LINQ-to-SharePoint

Note: Irrevelant elements such as <XML> elements in CAML were omitted from the diagram above.

*End of informative section.*

### Implementation outline

The implementation of the query parser is outlined in this section.

When writing a LINQ query against a SharePointDataSource<T> object, an expression tree is generated by the front-end language compiler. This expression tree needs to be parsed at runtime, resulting in a translation into an appropriate domain-specific query language (in our case CAML). This mechanism is empowered by the IQueryable<T> interface, as outlined in Figure 8 (some inheritance links have been omitted for clarity). In order to support ordering, the IOrderedQueryable<T> interface is implemented by LINQ-to-SharePoint.

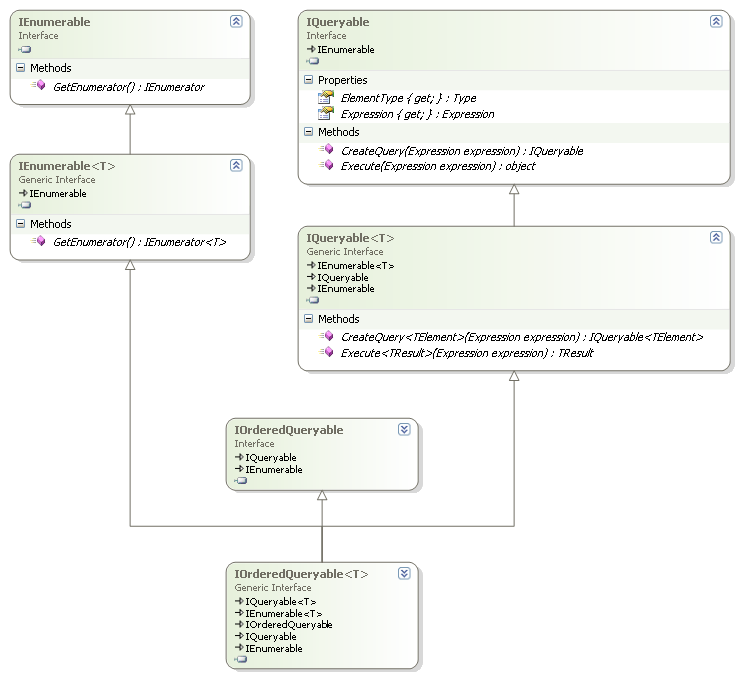


Figure 8 - IOrderedQueryable<T> interface hierarchy

At the core of the IQueryable<T> interface is the **CreateQuery<TElement>** method. Each LINQ query expression results in a chain of these calls. In here, T is the original type and TElement is the result type. For example, when applying projections (*select* clause), T and TElement might be different. We don’t provide an implementation for the Execute method right now.

As an example, consider the following simple LINQ-to-SharePoint query:

var res = from p in new SharePointDataSource<Products>(uri)  
 orderby p.ProductName  
 where p.UnitPrice > 123.45  
 select p.UnitsInStock;

In a first compilation stage, this will get transformed into the following piece of code:

var res = new SharePointDataSource<Products>(uri)  
 .OrderBy(p => p.ProductName)  
 .Where(p => p.UnitPrice > 123.45)  
 .Select(p => p.UnitsInStock);

All of these method calls are extension methods defined on the System.Linq.Queryable type, resulting in the following:

var res = Queryable.Select(  
 Queryable.Where(  
 Queryable.OrderBy(  
 new SharePointDataSource<Products>(uri),  
 p => p.ProductName),  
 p => p.UnitPrice > 123.45,  
 p => p.UnitsInStock);

Each of these methods take an IQuerable<T> object as their first parameter and an Expression as their second parameter (the lambda expressions). This causes the compiler to translate the lambda expressions into expression trees. Internally, these methods call the IQuerable<T> object’s CreateQuery<TElement> method, passing in the corresponding Expression wrapped in a *method call expression* representing the corresponding method. By returning an IQueryable<TElement> object, these method calls can be chained.

For the fragment above, the following expression trees are generated for the corresponding lambdas:

**p => p.ProductName**

ParameterExpression p = Expression.Parameter(typeof(Products), "p");  
 MemberExpression body = Expression.Property(p, "ProductName");  
 LambdaExpression orderby = Expression.Lambda(body, p);

**p => p.UnitPrice > 123.45**

ParameterExpression p = Expression.Parameter(typeof(Products), "p");  
 MemberExpression field = Expression.Property(p, "UnitPrice");  
 ConstantExpression maxPrice = Expression.Constant(123.45);  
 BinaryExpression body = Expression.GreaterThan(field, maxPrice);  
 LambdaExpression predicate = Expression.Lambda(body, p);

**p => p.UnitsInStock**

ParameterExpression p = Expression.Parameter(typeof(Products), "p");  
 MemberExpression body = Expression.Property(p, "UnitsInStock");  
 LambdaExpression projection = Expression.Lambda(body, p);

The Queryable methods used in the codefragment above have an implementation that’s conceptually equivalent to:

1. Take the IQueryable source’s current expression tree representation by calling its Expression property getter;
2. Make a new MethodCallExpression with the current method’s method name and the expression from step 1;
3. Call the IQueryable source’s CreateQuery method, passing in the expression created in the previous step.
4. Return the result of the CreateQuery method call.

This mechanism ultimately creates an expression tree representing the entire query structure. For maximum flexibility, a custom query provider could just participate in the tree building logic, keeping the query expression tree in some field/property. When execution is requested, the tree can be parsed and translated into the appropriate target query language. Our implementation takes a shortcut however.

**Warning!** This specification puts serious restrictions on the query expressiveness that can be reached using LINQ-to-SharePoint. By not implementing the Queryable.Execute method (as outlined further on), several methods supported by Queryable won’t be supported since these rely on a working Execute method. This limitation will be reconsidered in subsequent releases.

The IQueryable.Expression property will be implemented trivally, returning the current node as a *constant expression*. The other IQueryable property, ElementType, will have a trivial implementation too. Finally, the IQueryable.Execute method won’t be implemented:

public Type ElementType  
 {  
 get { return typeof(T); }  
 }  
  
 public Expression Expression  
 {  
 get { return Expression.Constant(this); }  
 }  
  
 public object Execute(Expression expression)  
 {  
 throw new NotImplementedException();  
 }

Instead, the LINQ-to-SharePoint query parser will perform *online parsing* during the **CreateQuery** method calls. For each call to CreateQuery the portion of the query being fed in will be translated into some intermediary structure right away, putting the pieces of the CAML query together. The core part of the CreateQuery method will therefore consist of a big switch that switches on the method name of the (MethodCall)Expression passed in:

* Where: call the *predicate parser*
* OrderBy, OrderByDescending, ThenBy, ThenByDescending: call the *ordering parser*
* Select: call the *projection parser*
* Take: *set result restriction*

Each of these subparsers (italic) will be specified in the next sections of this specification.

Note: Take is the only supported Queryable method in LINQ-to-SharePoint that does not correspond to a built-in language query operator. Take calls will be mapped to row count restrictions on the result set of the query. Details are outlined further in this specification.

Each call to CreateQuery results in a new IQueryable<TElement> object to be created. This object is of type SharePointDataSource<TElement> and is created using the appropriate constructor so that the connection methodology is kept the same. The private constructors as specified before are used for this purpose. Furthermore, all fields of the current SharePointDataSource<T> (the ones that hold information about the parsed query amongst the others for logging, connections, etc) are copied to the newly created instance in order to keep state. Once the whole query has been composed by a chain of CreateQuery calls, the final SharePointDataSource object will have collected all parsed information about the query, ready for execution.

1. Windows SharePoint Services 3.0: Software Development Kit (SDK) – version 2 – 01/22/2007 – <http://www.microsoft.com/downloads/details.aspx?familyid=05E0DD12-8394-402B-8936-A07FE8AFAFFD&displaylang=en> [↑](#endnote-ref-2)