C1 CMS – IData Interface

2017-02-14

Orckestra, Europe
Nygårdsvej 16
DK-2100 Copenhagen
Phone +45 3915 7600

www.orckestra.com

Contents

1 Introduction

A big part of C1 CMS is the data services - C1 CMS uses LINQ as the primary interface to query data, and all data is 100% based on CLR types. Thus .NET types (interfaces) play a big role when querying and working with data in C1 CMS.

The data layer in C1 CMS totally abstracts the underlying stores like SQL Server or XML files away from the developer without sacrificing developer control. Instead of writing SQL statements or XML queries to get data, you write LINQ statements.

All data items have one common interface they all inherit from: Composite.Data.IData. In C1 CMS's terminology an IData is what database developers would call a "table" or "schema" – an interface that inherits from IData is a definition of a data object which holds properties (name, type) and meta data like primary keys, foreign keys etc.

You can write a LINQ query without caring about the underlying store. If the underlying store is a SQL Server, then the LINQ query will be translated into a perfectly optimized SQL statement that gets the job done. And the same query, without any rewriting or even recompiling, can be used if the underlying store is XML – or any other kind of store for that matter.

Below is an example of a query that works for any data store (SQL, XML, …)

```
using (DataConnection connection = new DataConnection())
{
var q =
    from page in connection.Get<IPage>()
    where page.Title == "My Title"
    orderby page.UrlTitle
    select page;
}
```

Here is another example that will result in an inner join statement in the SQL database and in-memory object join when XML is used.

```
using (DataConnection connection = new DataConnection())
{
var q =
    from page in connection.Get<IPage>()
    from pagetype in connection.Get<IPageType>()
    where page.PageTypeId == pagetype.Id
    orderby page.Title
    select new { page.Title, pagetype.Name };
}
```

The last example shows a query that will result in a left outer join in the SQL database and in-memory object join when XML is used.

```
using (DataConnection connection = new DataConnection())
{

var q =
    from pagetype in connection.Get<IPageType>()
    join page in connection.Get<IPage>() on pagetype.Id equals
page.PageTypeId into sub
    from s in sub.DefaultIfEmpty()
    select s;
}
```

In all these examples IPage and IPageType are data interfaces that inherit from the base data interface IData. Below is a minimal example of a data type in C1 CMS:

```
[KeyPropertyName("Id")]
[DataScope(DataScopeIdentifier.PublicName)]
[ImmutableTypeId("{87122C34-E622-4e97-BD36-CBC398B862F9}")]
public interface IPerson : IData
{
    [StoreFieldType(PhysicalStoreFieldType.Guid)]
    [ImmutableFieldId("{172DD44C-426B-4812-834B-6B45366E78CB}")]
    Guid Id { get; set; }

    [StoreFieldType(PhysicalStoreFieldType.String, 249)]
    [ImmutableFieldId("{ADB24D3D-FA2A-496a-BBE9-91CFEB88336F}")]
    string Name { get; set; }
}
```

As seen in this example, the class and field attributes are heavily used. C1 CMS uses these attributes as meta-information about the data type – much like you have more information about SQL tables than just the table name and column name / type. This information is used throughout the system, from the low-level data store to the UI and is the basis of data types in C1 CMS.

Please note that these custom data interfaces must be defined in a Class Library project, built as an assembly (DLL) and placed in /Bin. You cannot create them in /App_Code.

If you were to create a new Data Type using the Data type wizards in the C1 CMS console, restart the C1 CMS site (Tools | Restart Server) and then examine "/bin/Composite.Generated.dll" using a program like .NET Reflector you would see that your data definition is actually a CLR type now, with properties and attributes like, the one shown above.

So – in short – all data schemas are defined as CLR types (interfaces) whether they are defined by C1 CMS (like pages, users, templates etc.), you as a C# developer or users that use the visual data creation tools. And they are all accessible via LINQ.

When you are working with custom IData you typically only define an interface – concrete classes are generated by C1 CMS automatically.

In this document we will go into the technical detail about the C1 CMS IData concept. The intended audience is C# developers who need to create custom data types for use in C1 CMS. The document will describe how to prepare your custom IData for features like publishing workflow, page meta data, content localization and more.

2 Super interfaces

By making your data type inherit from one or more of the data interfaces in C1 CMS you can “subscribe” to functionality and behavior to your type. The currently super data interfaces supplied by C1 CMS are described below.

2.1 IData

All data types in C1 CMS have to derive from the IData interface.

2.2 IPublishControlled

If your type should support publishing workflows, your type has to derive from the IPublishControlled interface. Types that support publishing can exist in two scopes:

- administrated
- public

Published data is used when pages are rendered for the public site. See the Creating a publishable data type section for more information on types that support publishing workflows. Actually, IPublishControlled is an IData itself that has some special handling in C1 CMS. Here is how it’s defined:

```
[DataScope(DataScopeIdentifier.PublicName)]
[DataScope(DataScopeIdentifier.AdministratedName)]
public interface IPublishControlled : IProcessControlled
{
    [StoreFieldType(PhysicalStoreFieldType.String, 64, IsNullable = false)]
    [ImmutableFieldId("{FAB1CF0C-66B0-11DC-A47E-CF6356D89593}")]
    [DefaultFieldStringValue("")]
    [BeforeSet(typeof(PublishControlledSetPropertyHandler))]
    [FieldPosition(50)]
    string PublicationStatus { get; set; }
}
```

So, when your interface inherits the IPublishControlled interface, a property is added to your type. Also, two data scopes are added.

2.3 ILocalizedControlled

If your type should support localization workflows, your type has to derive from the ILocalizedControlled interface. Types that support localization can have instances with the same ID in each active locale in a running system. In other words, if da-DK, en-US and it-IT locales have been added to a running C1 CMS solution, then an instance of your type with the ID of ‘X’ can exist in those three locales, but do not have to. If at one point a request to rendering a page in the en-US locale is made and data of your type is requested, only instances that have been added to the en-US scope will be returned. Like IPublishControlled, ILocalizedControlled is also just a specialized IData interface with some special handling in C1 CMS:

```
public interface ILocalizedControlled : IProcessControlled
{
    [StoreFieldType(PhysicalStoreFieldType.String, 16)]
    [ImmutableFieldId("{E271D3EB-A8EB-49ea-9BB5-E5A54F88298F}")]
    [NotNullValidator()]
    [DefaultFieldStringValue("")] // Invariant
    string CultureName { get; set; }

    [StoreFieldType(PhysicalStoreFieldType.String, 16)]
    [ImmutableFieldId("{0456EBB0-7FB1-46cd-9A23-4AE9AA3337FA}")]
    [NotNullValidator()]
    [DefaultFieldStringValue("")] // Invariant
    string SourceCultureName { get; set; }
}
```

3 Interface Attributes

The following are interface attributes grouped as:

- Required attributes
- Optional attributes
- Expert attributes

3.1 Required Attributes

These are required attributes:

- ImmutableTypeId Attribute
- KeyPropertyName Attribute
- DataScope Attribute

Please consider using the LabelPropertyName attribute, too, to provide user-friendly labels for data items, for example, in function parameters of the DataReference<T> type. Otherwise, GUID-like labels will be used.

3.1.1 ImmutableTypeId Attribute

This attribute specifies a unique ID for the type. The ID should be unique for the running C1 CMS solution. The value of the argument should be a string representation of a GUID. This unique ID is used by C1 CMS to identify the type. This means that it is possible to rename the type and the type would still work. Though, when coding your own type and using the type name in code (your own or code that might use yours), the same compile rules apply as those with normal C# interfaces. Example:

```
[ImmutableTypeId("{D261B424-3629-4e00-9D24-BDA763DE8DD8}")]
```

3.1.2 KeyPropertyName Attribute

Use this attribute to specify one or more key property names. A key property is the property that will be used as a key for data type. No more than one instance of your data type may have the same key value (or keys values) in a given scope. The value of the argument should be a string with the name of one of the properties of your interface. Example:

```
[KeyPropertyName("Id")]
```

3.1.3 DataScope Attribute

This attribute specifies which data scopes items of the data types should exist in. Currently two scopes are supported: DataScopeIdentifier.Public and DataScopeIdentifier.Administrated. If your type should not support publishing workflows, then add this attribute once with the value DataScopeIdentifier.PublicName. If your type should support publishing workflows, you should add this attribute twice with the argument values: DataScopeIdentifier.PublicName and DataScopeIdentifier.AdministratedName. See the Creating a publishable data type section for more information on types that support publishing workflows.

Examples:

```
[DataScope(DataScopeIdentifier.PublicName)]
```

and

```
[DataScope(DataScopeIdentifier.AdministratedName)]
```

3.2 Optional Attributes

These are optional attributes:

- Title Attribute
- AutoUpdateble Attribute
- LabelPropertyName Attribute
- RelevantToUserType Attribute
- Caching Attribute
- PublishProcessControllerType Attribute

3.2.1 Title Attribute

Use this attribute to assign a more user-friendly name for your type. The value of the argument should be a string. This will be used by C1 CMS when the type's name needs to be displayed in the UI. Example:

```
[Title("Employee")]
```

3.2.2 AutoUpdateble Attribute

This attribute will make C1 CMS auto add and update your type. The type will be added to the default data provider when data is added for the first time. If reading data is done before the type has been added, zero items are returned. If you change your type, adding a new property, then C1 CMS will update the underlying data store automatically. If you omit this attribute, you have to manually add it to the data layer and future changes to the type also have to be made manually. It is recommended that your type has this attribute. See the Manually adding or removing a data type to or from C1 CMS chapter for manually adding and removing a data type. Example:

```
[AutoUpdateble]
```

3.2.3 LabelPropertyName Attribute

Use this attribute to specify which property value should be used as label for items of the data type. The label is used when listing items in trees or other kinds of lists.

If the property used as a label is nullable (optional) or is a string and the value of the property is null, then the title will be of the form: "(undefined [PROPERTY_NAME])", where PROPERTY_NAME is the name of the property specified with this attribute.

If the property is a reference property (See ForeignKey attribute) then the label of the referenced data will be displayed. The value of the argument should be a string with exactly the same name and casing as one of the properties of your interface. Example:

```
[LabelPropertyName("Name")]
```

Although this attribute is optional, we recommend using it to provide user-friendly labels for data items, for example, in function parameters of the DataReference<T> type.

3.2.4 RelevantToUserType Attribute

If you add this attribute to your type then your type will be selectable in the UI. Examples: Data references, adding a Visual function for your type and being a part of the functions calls in XSLT functions. At the moment the only supported argument value for this attribute is: UserType.Developer. Example:

```
[RelevantToUserType(UserType.Developer)]
```

3.2.5 Caching Attribute

By specifying this attribute on your type, instances of your type will be cached by C1 CMS, thus making data access to your type faster. Caching is done in memory, so avoid caching a type where large data sets are expected. Example:

```
[Caching(CachingType.Full)]
```

3.2.6 PublishProcessControllerType Attribute

When your type supports publishing workflows this attribute should be specified. The value of the argument should be a type that implements the IPublishProcessController interface. For a default behavior you can use the type GenericPublishProcessController as an argument value for this attribute. See the Creating a publishable data type chapter for more information on types that supports publishing workflows. Example:

```
[PublishProcessControllerType(typeof(GenericPublishProcessController))]
```

3.3 Expert Attributes

These are expert attributes:

- PublishControlledAuxiliary Attribute
- BuildNewHandler Attribute

3.3.1 PublishControlledAuxiliary Attribute

Use this attribute to get some custom code to run after the IPublishProcessController has done its work. The argument value of this attribute should be a type that implements the interface IPublishControlledAuxiliary. See the Creating a publishable data type chapter for more information on types that supports publish workflows. Example:

```
public class MyPublishControlledAuxiliary : IPublishControlledAuxiliary
{
    // IPublishControlledAuxiliary Members
}

[PublishControlledAuxiliary(typeof(MyPublishControlledAuxiliary))]
public interface IMyData : IData
{
    // Properties
}
```

3.3.2 BuildNewHandler Attribute

When the DataConnection.New<T>() is made, an object of a type that implements T is created. If you want to control which type is used to create a new object, then you should add this attribute to your type. The value of the argument should be a type value to a type that implements the interface IBuildNewHandler. Example:

```
public class MyBuildNewHandler : IBuildNewHandler
{
    // IBuildNewHandler Members
}

[BuildNewHandler(typeof(MyBuildNewHandler))]
public interface IMyData : IData
{
    // Properties
}
```

4 Property Attributes

The following are property attributes grouped as:

- Required attributes
- Optional attributes
- Validation attributes

4.1 Required Attributes

These are required attributes:

- ImmutableFieldId Attribute
- StoreFieldType Attribute

4.1.1 ImmutableFieldId Attribute

This attribute specifies a unique ID for the property. The ID should be unique for the running C1 CMS solution. The value of the argument should be a string representation of a GUID. This unique ID is used by C1 CMS to identify the property. This means that it is possible to rename the property and the type would still work. Though, when coding your own type and using the property name in code (your own or code that might use yours), the same compile rules apply as with normal C# interfaces. Example:

```
[ImmutableFieldId("{D261B424-3629-4e00-9D24-BDA763DE8DD8}")]
```

4.1.2 StoreFieldType Attribute

This attribute specifies the underlying type of a given property. This information is passed to the data provider (i.e. XML or SQL) and used by the data provider when a type needs to be created on the underlying data layer. The first argument is required and should be the value of PhysicalStoreFieldType. There are two optional arguments, the first one is the max length and can only be used with the first argument value of PhysicalStoreFieldType.String. The max length value specifies that the strings stored in the given property will never be longer than the specified max length. The second one is the numeric precision and numeric scale and these can only be used with the first argument value of PhysicalStoreFieldType.Decimal and specify the precision and scale of the decimal values that the property will hold. All properties can be made nullable by using the optional argument IsNullable.

Below is a total list of possible PhysicalStoreFieldTypes:

```
[StoreFieldType(PhysicalStoreFieldType.Boolean)]
bool IsChild { get; set; }

[StoreFieldType(PhysicalStoreFieldType.DateTime)]
DateTime Created { get; set; }

[StoreFieldType(PhysicalStoreFieldType.Decimal, 10, 2)]
decimal Price { get; set; }

[StoreFieldType(PhysicalStoreFieldType.Guid)]
Guid Id { get; set; }

[StoreFieldType(PhysicalStoreFieldType.Integer)]
int Level { get; set; }

[StoreFieldType(PhysicalStoreFieldType.LargeString)]
string Description { get; set; }

[StoreFieldType(PhysicalStoreFieldType.Long)]
long Distance { get; set; }

[StoreFieldType(PhysicalStoreFieldType.String, 128)]
string Name { get; set; }
```

Example of the nullable:

```
[StoreFieldType(PhysicalStoreFieldType.Guid, IsNullable=true)]
Guid Id { get; set; }
```

4.2 Optional Attributes

These are optional attributes:

- DefaultFieldValue Attribute
- ForeignKey Attribute
- FunctionBasedNewInstanceDefaultFieldValue Attribute

4.2.1 DefaultFieldValue Attribute

This attribute is passed to the data provider and thereby the underlying data layer (i.e. SQL or XML). This is not used for giving your properties a value when an instance of your type is first created (DataConnection.New<T>()). For assigned values to properties of newly created (DataConnection.New<T>()) instances of your type, see the FunctionBasedNewInstanceDefaultFieldValue Attribute section for more information. But this is used if, for example, a new property is added to your type and the SQL data provider needs to extend the given table with a new column. Then all existing records will get the value of this attribute's value. You can use one of the following attributes: DefaultFieldStringValueAttribute, DefaultFieldIntValueAttribute, DefaultFieldDecimalValueAttribute, DefaultFieldBoolValueAttribute, DefaultFieldGuidValueAttribute, DefaultFieldNewGuidValueAttribute or DefaultFieldNowDateTimeValueAttribute.

4.2.2 ForeignKey Attribute

This attribute can be used to specify a relation from your type to another existing data type. In C1 CMS, when given a data item where the type is "pointing" to one or more other types, it is easy to get the data items that are "pointed" to. Also, cascade deletes and reference integrity are performed when AddNew, Update or Delete is performed for types with references to other types.

The two primary and required arguments for this attribute are the type that your type is "pointing" to and the name of the key property of that type.

There are also four optional arguments. AllowCascadeDeletes can be specified to enable/disable cascade deletes. NullReferenceValue, NullReferenceValueType, NullableString are used to control the behavior of null references. NullReferenceValue specifies the value for a null reference. NullReferenceValueType specifies the type of the null reference value.

Ex: “{00000000-0000-0000-0000-000000000000}” as the value and typeof(Guid) as the value type

NullableString can be set to true to allow null to be used as a non-reference value.

For sample code, please see “Static IData Types”, Example #2.

4.2.3 FunctionBasedNewInstanceDefaultFieldValue Attribute

This attribute can be used to have C1 CMS assign a value to newly build instances of your data type. This is not the same as using the DefaultFieldValue attribute, see DefaultFieldValue Attribute for more information. The value of the argument to this attribute is a function markup call. A call to this function will be made when DataConnection.New<T>() is done. And the value of that function call will be assigned to the property where this attribute is specified.

4.3 Validation Attributes

Common for all validation attributes is that when adding or updating a data item, validation is done by C1 CMS and an exception is thrown if validation fails. When editing data on the client, these exceptions are shown as balloons if validation fails. Some of the validation attributes will result in client side validation.

4.3.1 NotNullValidator

Use this attribute to specify that the property (if nullable, including strings) shall have a value. This attribute will also be used by the client (client-side validation) so that the user will be shown a warning (“Required” in red) if no value is entered.

4.3.2 RegexValidator

Use this attribute to specify that property should conform to the given regular expression. This attribute will also be used by the client (client-side validation) so that the user will be shown a warning if the value does not conform to the given regular expression.

4.3.3 DecimalPrecisionValidatorAttribute

Use this attribute to specify that the property should conform to the given decimal precision.

4.3.4 GuidNotEmptyAttribute

Use this attribute to specify that empty GUIDs (0-guids) are not allowed as a value for the given property.

4.3.5 IntegerRangeValidatorAttribute

Use this attribute to specify the range that the value of the given property must lie within.

4.3.6 StringLengthValidatorAttribute

This attribute allows the length of the string-property value to lie within the range indicated by two numbers.

4.3.7 NullStringLengthValidatorAttribute

This attribute allows the string-property value to be a null string or a string the length of which lie within the range indicated by two numbers.

5 Reserved attributes

The following attributes are reserved and should never be used as the support for them might be changed or removed in the future.

- CodeGenerated Attribute
- FieldPosition Attribute
- GroupByPriority Attribute
- AssociationVisabilityType Attribute
- DataAssociation Attribute

6 Obsolete Attributes

The following attributes should never be used as the support for them might be removed in the future.

- BeforeSetAttribute
- VersionProcessControllerTypeAttribute

7 Creating a publishable data type

To create a publishable data type like IPage, your type has to derive from the IPublishControlled interface.

Two interface attributes are also required, namely DataScope and PublishProcessControllerType. There is one optional attribute to consider when creating a publishable data type and that is PublishControlledAuxiliary.

8 Creating a Page folder data type

The sample code and its key pointers in the following sections demonstrate how to create a Page folder data type.

8.1 Minimal example

```
[AutoUpdateble]
[DataScope(DataScopeIdentifier.PublicName)]
[ImmutableTypeId("{500AF1F0-998D-47c8-A411-C42DB7711D43}")]
[DataAncestorProvider(typeof(NoAncestorDataAncestorProvider))]
public interface IMyDataFolder : IPageFolderData
{
    [StoreFieldType(PhysicalStoreFieldType.String, 255)]
    [ImmutableFieldId("{47459211-C41F-4065-93BE-731E87A991DA}")]
    string Title { get; set; }
}
```

9 Creating a Page meta data type

The sample code and its key pointers in the following sections demonstrate how to create a meta data type.

9.1 Minimal example

```
[AutoUpdateble]
[ImmutableTypeId("{49F23108-2EA8-4f1b-945F-30676777F8AB}")]
[DataAncestorProvider(typeof(NoAncestorDataAncestorProvider))]
public interface IMyMetaData : IPageMetaData
{
    [StoreFieldType(PhysicalStoreFieldType.String, 255)]
    [ImmutableFieldId("{1F3F5C32-4075-4899-89B6-9C01A041B712}")]
    string Title { get; set; }
}
```

10 Manually adding a meta type to a page (branch)

The sample code and its key pointers in the following sections demonstrate how to manually add a meta type to a page.

10.1 Minimal Example

```
using (DataConnection connection = new
DataConnection(PublicationScope.Unpublished))
{
  IPage page = connection.Get<IPage>().Last(); // The page in question
  Guid containerId =
PageMetaDataFacade.GetAllMetaDataContainers().First().Key;
  Guid myTypeId = typeof(IMyDataType).GetImmutableTypeId();
  Page.AddMetaDataDefinition("MyName", "MyLabel", containerId, myTypeId);
  // Adding default values
  IMyDataType newDataTemplate = DataConnection.New<IMyDataType>();
  newDataTemplate.Name = "Some name";
  // Fill in the rest of the fields
  page.AddNewMetaDataToExistingPages("MyName", newDataTemplate);
}
```

11 Manually adding a new meta data category (tab)

By default, a “Meta Data” category (tab) exists in C1 CMS. If no new categories are added, then all meta data added through the UI will be added to this category. It is possible to create and add a new category to C1 CMS. Below is an example of how to add a new meta data category. You can use your own categories when you are adding meta data to a page manually.

11.1 Minimal Example

```
using (DataConnection connection = new DataConnection())
{
    ICompositionContainer compositionContainer = 
DataConnection.New<ICompositionContainer>();
    compositionContainer.Id = Guid.NewGuid();
    compositionContainer.Label = "My Container";
    compositionContainer = 
connection.Add<ICompositionContainer>(compositionContainer);
}
```

12 Manually adding or removing a data type

The sample code in the following sections demonstrates how to manually add or remove a data type in C1 CMS.

12.1 Adding

If you want to add your data type to the system, you should at one point make this call:

```
DynamicTypeManager.EnsureCreateStore(typeof(IMyDataType));
```

It is safe to call this repeatedly, but it is recommended only doing it one time per system startup.

12.2 Removing

If you want to remove your type from the system, you do the following:

```
DataTypeDescriptor dtd =
DynamicTypeManager.GetDataTypeDescriptor(typeof(IMyDataFolder));
DynamicTypeManager.DropStore(dtd);
```

This can only be call one time.