A Developer Guide to Accessing Data

2017-02-14

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

www.orckestra.com

Contents

C1 CMS

1 Introduction

C1 CMS is based on .NET Framework and its data system is based on the CLR types. This implies that you can use LINQ to query, and C# to work with, data in C1 CMS. All C1 CMS data types implement a common interface - **Composite.Data.IData** and their concrete classes are generated by C1 CMS automatically.

1.1 Who Should Read This Guide?

This document is a quick guide intended for C# developers who want to learn how to perform CRUD operations on C1 CMS's data system using C# and LINQ.

Working with data is the primary focus of this guide. For information about creating your own data types by using C#, please read Data Types Using C#.

1.2 Getting Started

The following data operations are covered here:

- Querying Data
- Adding Data
- Updating Data
- Deleting Data
- Joining Data

In the following few chapters, you will learn more about these operations.

We recommend that you first read the following two sections to get familiar with the DataConnection class - the central class in accessing data in C1 CMS - and the way the data types are represented in the C# code.

1.3 Data Connection

Before you access data in C1 CMS, you need to establish a connection to the C1 CMS data system by creating an instance of the **DataConnection** class.

```
DataConnection connection = new DataConnection();
```

Listing 1: Connecting to the data store

The **DataConnection** class is available in the **Composite.Data** namespace, which you should add to your code with the **using** directive.

```
using Composite.Data;
```

Listing 2: Using Composite.Data namespace

This constructor creates an instance that inherits the current **PublicationScope** and locale (set on the call stack). When outside an existing scope, they are set to the default **Published** scope and the default language on the website.

By using the **DataConnection** instance and static member methods, you can query, add, update and delete data in C1 CMS.

To retrieve data from a specific scope or locale, you should use one of the constructor overloads.

```
DataConnection connection = new DataConnection(new CultureInfo("da-DK"));
```

Listing 3: Data connection with a specific locale

This constructor creates a **DataConnection** instance with a specific locale (and the current or default scope). It uses an instance of **CultureInfo**, initialized to a specific locale, e.g. “da-DK”.

```
DataConnection connection = new DataConnection(PublicationScope.Published);
```

Listing 4: Data connection with a specific scope

This constructor creates a **DataConnection** instance with a specific scope (and the current or default locale).

The scope of data is defined in relation to its publication status:

- Data which support publication should always be maintained in the “Unpublihed” scope
- Reading data on the public website should always be done in the “Published” scope

Normally, C1 CMS itself handles the **PublicationScope**. You may however choose to explicitly set the **PublicationScope** (for example, on new service end-points or if you need specific features related to updating or publishing data).

```
DataConnection connection = new DataConnection(PublicationScope.Published,
new CultureInfo("da-DK"));
```

Listing 5: Data connection with a specific scope and a specific locale

This constructor creates a DataConnection instance with a specific scope and a specific locale and is the combination of the above two.

1.4 Classes behind Data Types

Each data type available in the C1 CMS data system is represented by its own class that implements the IData interface and is generated by C1 CMS automatically.

You can access a data item as an instance of such a data type class by referring to it by the full name of the data type, which includes its namespaces.

```
Demo.Users myUser = DataConnection.New<Demo.Users>();
```

Listing 6: Creating an instance of the Demo.Users data type

In the above example, you create a data type instance by calling the **DataConnection**’s static member method **New**.

You can then access fields of the data type items as properties to set and get values.

```
myUser.Name = "John Doe";
```

Listing 7: Setting the Name property of the Demo.Users instance

Data operations are available for both a single instance of a data type and multiple instances represented by an enumerable list of instances that implement the IData interface.

2 How to Query Data Using LINQ

To query data in the Data store, you should use the Get method of DataConnection specifying an existing data type. The method returns an IQueryable instance of the IData interface.

```
public IQueryable<TData> Get<TData>()
where TData : class, IData
```

Listing 8: The DataConnection.Get method

You can use the instance to further query it by using LINQ.

To query data in the Data store:

1. Connect to the data system.
2. Get the IQueryable instance of the data type you need.
3. Query data by using LINQ.

```
using (DataConnection connection = new DataConnection())
{
   var myUsers =
      from d in connection.Get<Demo.Users>()
      where d.Name == "John Doe"
      select d;
}
```

Listing 9: Querying data using LINQ

Page folder data and page meta data also have the PageId property. Use this property to get folder data or meta data for a specific page. To filter data by the current page's ID, use SitemapNavigator.CurrentPageId.

The code will be the same for both page folder data and page meta data:

```
using (DataConnection connection = new DataConnection())
{
   var myFolderData =
      from d in connection.Get<My.Folder.Type>()
      where d.PageId == SitemapNavigator.CurrentPageId
      select d;
}
```

Listing 10: Querying page folder data

```
using (DataConnection connection = new DataConnection())
{
   var myMetaData =
      from d in connection.Get<My.Meta.Type>()
      where d.PageId == SitemapNavigator.CurrentPageId
      select d;
}
```

Listing 11: Querying page meta data

For information on SitemapNavigator, please see http://api.composite.net/html/P_Composite_Data_DataConnection_SitemapNavigator.htm

For information about joining data from two data types, please see "Joining Data".

3 How to Add Data

There are two overloaded **Add** methods to add data to the default Data storage. Before you add new data items, you should create its instance by using the static **New** method of **DataConnection**.

3.1 Adding a Single Data Item

To a single data item to the Data store:

1. Connect to the data system.
2. Create a new instance of the data type you need.
3. Set its fields to values.
4. Add the item to the data system.

```
using (DataConnection connection = new DataConnection())
{
   Demo.Users myUser = DataConnection.New<Demo.Users>();
   myUser.Id = Guid.NewGuid();
   myUser.Name = "John Doe";
   myUser = connection.Add<Demo.Users>(myUser);
}
```

Listing 12: Adding a single data item

3.2 Add Multiple Data Items

To add multiple data items to the Data store:

1. Connect to the data system.
2. Create a new instance of enumerable list of the data type items you need.
3. Iterate the list and set its fields to values.
4. Add the list to the data system.

```
using (DataConnection connection = new DataConnection())
{
   List<Demo.Users> myUsers = new List<Demo.Users>();

   for (int i = 0; i < 10; i++)
   {
      Demo.Users myUser = DataConnection.New<Demo.Users>();
      myUser.Id = Guid.NewGuid();
      myUser.Name = "John Doe";
      myUser.Number = i;
      myUsers.Add(myUser);
   }

   connection.Add<Demo.Users>(myUsers);
}
```

Listing 13: Adding multiple data items

4 How to Update Data

There are two overloaded **Update** methods to update data in the Storage. Before updating existing data items, you need to get their instances (for example, by querying the data store).

4.1 Updating a Single Data Item

To update a data item in the Data store:

1. Connect to the data system.
2. Get an existing item of the data type you need by querying the data store.
3. Access its fields and change their values.
4. Commit the changes in the updated item to the data system.

```
using (DataConnection connection = new DataConnection())
{
   Demo.Users myUser =
      (from d in connection.Get<Demo.Users>()
       where d.Name == "John Smith"
       select d).First();

   myUser.Name = "John Doe";

   connection.Update<Demo.Users>(myUser);
}
```

Listing 14: Updating a single data item

4.2 Updating Multiple Data Items

To add multiple data items to the Data store:

1. Connect to the data system.
2. Get an enumerable list of existing items of the data type you need by querying the data store.
3. Iterate the list and change values in the fields of its items.
4. Commit the changes in the updated items to the data system

```
using (DataConnection connection = new DataConnection())
{
   var myUsers = connection.Get<Demo.Users>().Where (d => d.Number <
10).ToList();

   foreach (Demo.Users myUser in myUsers)
   {
      myUser.Number += 10;
   }

   connection.Update<Demo.Users>(myUsers);
}
```

Listing 15: Updating multiple data items

5 How to Delete Data

There are two overloaded **Delete** methods to permanently delete data in the Storage. Before deleting existing data items, you need to get their instances (for example, by querying the data store).

5.1 Deleting a Single Data Item

To delete a data item from the Data store:

1. Connect to the data system.
2. Get an existing item of the data type you need by querying the data store.
3. Delete the item.

```
using (DataConnection connection = new DataConnection())
{
   Demo.Users myUser =
      (from d in connection.Get<Demo.Users>()
       where d.Name == "John Doe"
       select d).First();

   connection.Delete<Demo.Users>(myUser);
}
```

Listing 16: Deleting a single data item

5.2 Deleting Multiple Data Items

To delete multiple data items from the Data store:

1. Connect to the data system.
2. Get an enumerable list of existing items of the data type you need by querying the data store.
3. Delete the items in the list.

```
using (DataConnection connection = new DataConnection())
{
   var myUsers = connection.Get<Demo.Users>().Where (d => d.Number >
15).ToList();

   connection.Delete<Demo.Users>(myUsers);
}
```

Listing 17: Deleting multiple data items

6 Joining Data

If you have two data types, one of which ("child") has a field reference to the other one ("parent"), you can access data in both by joining with the "parent" type when querying the "child" type.

Let's assume you have these data types with the following fields:

Demo.Parent:

- Id (GUID)
- Name (String)

Demo.Child

- Id (GUID)
- Name (String)
- Parent (A reference to the Demo.Parent)

When querying the Demo.Child data type, you also want to get the Demo.Parent type's Name field value. This is where you can join the types.

To join data from two data types when querying:

1. Connect to the data system.
2. Get the IQueryable instance of each data type.
3. Join the "child" type with the "parent" type on the corresponding field using LINQ.
4. Query data by using LINQ.

```
using (DataConnection connection = new DataConnection())
{
   var myList =
      from c in connection.Get<Demo.Child>()
      join p in connection.Get<Demo.Parent>()
      on c.Parent equals p.Id
      select new {ParentName = p.Name, ChildName = c.Name};
}
```

Listing 18: Joining data from two data types

Now you can use the list with both the "child" and "parent" data combined:

```
foreach (var myItem in myList)
{
       Log.LogInformation("Demo", myItem.ParentName + ": " +
myItem.ChildName);
}
```

Listing 19: Using joined data