Article
Iryna Mykhailova · Oct 30, 2018 11m read

Caché eXTreme for .NET - direct access to globals from C#

InterSystems states that Caché supports at least three data models – relational, object and hierarchical (globals). On can work with data presented in relational model in a program written on C# the same way one works with any other relational DB. To work with data presented by object model in C# one needs to use .NET Managed Provider or some kind or ORM. And starting with version 2012.2 one can work directly with globals (or use direct access to hierarchical data) via Caché eXTreme for .NET. Documentation says:

Caché eXTreme is a set of technologies that enable Caché to be leveraged as a high performance persistence storage engine optimized for XTP (Extreme Transaction Processing) applications.

Unlike the standard .NET binding, the eXTreme APIs do not use TCP/IP to communicate with Caché. Instead, they use a fast in-memory connection (implemented via standard .NET and the Caché Callin API), and run in the same process as the Caché instance. Although the Caché server and the .NET application must be on the same machine, the application can still use Caché ECP to access data on remote machines.

So how can it be used?

First of all, you have to set / check environmental variables of Windows:

1. The variable GLOBALSHOME should contain the full path to the directory where you put the DBMS. In my case, this is C:\InterSystems\Cache\
2. The PATH variable must contain the full path to the Bin directory. In my case, this is C:\InterSystems\Cache\Bin

Beware, if you have several versions of Caché on the same machine, the one that occurs in this variable first will be used in Caché eXTreme.
Secondly, you have to add references to two libraries to your project:

- InterSystems.CacheExtreme.dll
- InterSystems.Data.CacheClient.dll

Both libraries can be found in the folder C:\InterSystems\Cache\dev\dotnet\bin\vXXX , where vXXX – version of .NET you want to use.

Naturally, in the corresponding module in the using section you need to add:

```
using InterSystems.Globals;
```

```
using InterSystems.Data.CacheClient;
```

After all the preliminary work is done, you can start actual programming.

Since Caché eXTreme allows you to record as subscripts four data types: int, double, string, long, and six data types as values: int, double, long, string, bytes [], ValueList, I tried to come up with a structure for the global, in which there will be as many types as possible. I took the data on bank card transactions as a subject area.

In Studio we can create a global with such structure:

```
 set ^CardInfo(111111111111) = "Smith,John"

 set ^CardInfo(111111111111, "DBO546") = "Some bank 1"
 set ^CardInfo(111111111111, "DBO546", 29244825509100) = 28741.35
 set ^CardInfo(111111111111, "DBO546", 29244825509100, 2145632596588547) = "Smith,Joh
n/1965"
 set ^CardInfo(111111111111, "DBO546", 29244825509100, 2145632596588547, 1) = $lb(0,
29244225564111, "John Doe", 500.26, "Payment for goods from ToysRUs")
 set ^CardInfo(111111111111, "DBO546", 29244825509100, 2145632596588547, 2) = $lb(0,
26032009100100, "John Smith", 115.54, "Transfer to own account in different bank")

 set ^CardInfo(111111111111, "DXO987") = "Some bank 2"
 set ^CardInfo(111111111111, "DXO987", 26032009100100) = 65241.24
 set ^CardInfo(111111111111, "DXO987", 26032009100100, 6541963285249512) = "Smith Joh
n||1965"
 set ^CardInfo(111111111111, "DXO987", 26032009100100, 6541963285249512, 1) = $lb(1,
29242664509184, "Jane Doe", 500.26, "Recurring payment to Amazon")
 set ^CardInfo(111111111111, "DXO987", 26032009100100, 6541963285249512, 2) = $lb(0,
26548962495545, "John Doe", 1015.10, "Payment for delivery")

 set ^CardInfo(111111111111, "DXJ342") = "Some bank 3"
 set ^CardInfo(111111111111, "DXJ342", 26008962495545) = 126.32
 set ^CardInfo(111111111111, "DXJ342", 26008962495545, 4567098712347654) = "John Smit
h 1965"
 set ^CardInfo(111111111111, "DXJ342", 26008962495545, 4567098712347654, 1) = $lb(0,
29244825509100, "John Smith", 115.54, "Transfer to own account in different bank")
 set ^CardInfo(111111111111, "DXJ342", 26008962495545, 4567098712347654, 2) = $lb(1,
26032009100100, "John Smith", 1015.54, "Transfer to own account in different bank")
```

So John Smith has accounts in 3 different banks with 1 card linked to each account and 2 operations done from each account.

To begin working with data, you need to connect to the DB first.

It was mentioned earlier that the application runs in the same stream as the database. So in the process there can only be one connection and all C# objects use this particular connection. To get it, use the ConnectionContext.GetConnection() method. To check whether the connection is open or not, the IsConnected() method is used. To open a connection, use the Connect() method, to close - Close().

The program will look like this:

```
class Program
    {
        static Connection Connect()
        {
            //getting conection
            Connection myConn = ConnectionContext.GetConnection();
            //check if the connection is opened
            if (!myConn.IsConnected())
```

```
            {
                Console.WriteLine("Connectiong...");
                //if connection is not opened, then connect
                myConn.Connect("User", "_SYSTEM", "SYS");
            }

            if (myConn.IsConnected())
            {
                Console.WriteLine("Successfully connected");
                //if OK then return opened connection
                return myConn;
            }
            else { return null; }
        }

        static void Disconnect(Connection myConn)
        {
            if (myConn.IsConnected())
                myConn.Close();
        }

        static void Main(string[] args)
        {
            try
            {
                Connection myConn1 = Connect();
                //ToDo: work with globals
                Disconnect(myConn1);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }

            Console.ReadKey();
        }
    }
```

After the connection to DB is established there are several ways to save info to DB.

The first way is building a tree by adding subscripts (deepening from the root).

In order to start building a tree, you need to fix its root. In our case, this is the ^CardInfo value. This operation is performed using the CreateNodeReference() method:

```
NodeReference nodeRef = myConn1.CreateNodeReference("CardInfo");
```

Once you have fixed the pointer to the root of the tree, you can begin to build it. The nodeRef variable will always store a pointer to the node in the tree that is currently active.

To add a new subscript to a variable, use the AppendSubscript() method , in which you can pass a double, int, long, or string value. To insert a value in the global node, use the Set() method. It can accept values of types byte [], double, int, long, string, ValueList as the first parameter. The second parameter will be considered further, now it is not needed yet. If everything is clear with byte [], double, int, long, string, this is a standard byte array, a real value, an integer, long integer value and a string, then the last data type is a special data type for working with Caché system lists in COS using the $ListBuild function (also known as $lb).

In general, until the Set() method is executed, the data will not be written to the database.

```
node.AppendSubscript("111111111111");
node.Set("Smith,John");
node.AppendSubscript("DBO546");
node.Set("Some bank 1");
node.AppendSubscript(29244825509100);
node.Set(28741.35);
node.AppendSubscript(2145632596588547);
string slip = "Smith,John/1965";
byte[] bytes = System.Text.Encoding.GetEncoding(1251).GetBytes(slip);
node.Set(bytes);
node.AppendSubscript(1);
ValueList myList = myConn.CreateList();
myList.Append(0, 29244225564111, "John Doe", 500.26, "Payment for goods from ToysRUs"
);
node.Set(myList);
myList.Close();
```

With each " step " the pointer to the current node will move to the newly added subscript.

However, we have another transaction. To add it in the same way, you need to go up a level. In order to do this (and generally to jump to any other level), the SetSubscriptCount() method is used, to which the number of node subscripts (level number) is passed where the transition should be made.

```
node.SetSubscriptCount(4);
  node.AppendSubscript(2);
  myList = myConn.CreateList();
  myList.Append(0, 26032009100100, "John Smith", 115.54, "Transfer to own account in
different bank");
  node.Set(myList);
  myList.Close();
```

Second way to write data is to set the values explicitly.

The previous section mentioned that the Set() method can take two parameters. If only one is transmitted, then the value is inserted into the current global node. The second parameter is intended for a list of subscripts relative to the current node (to which the reference is stored in a variable of NodeReference type) where the value should be inserted.

Thus, to add account data in the next bank, you must first return to the level of a single index. To do this, use the SetSubscriptCount() method with parameter 1. And further, using the Set() method, simply add subscripts. In this case the pointer in nodeRef after each operation will still be at the first level.

```
node.SetSubscriptCount(1);
  node.Set("Some bank 2", "DXO987");
  node.Set(65241.24, "DXO987", 26032009100100);
  string slip = "Smith John||1965";
  byte[] bytes = System.Text.Encoding.GetEncoding(1251).GetBytes(slip);
  node.Set(bytes, "DXO987", 26032009100100, 6541963285249512);
  ValueList myList = myConn.CreateList();
  myList.Append(1, 29242664509184, "Jane Doe", 500.26, "Recurring payment to Amazon")
;
  node.Set(myList, "DXO987", 26032009100100, 6541963285249512, 1);
  myList.Close();
  myList = myConn.CreateList();
  myList.Append(0, 26548962495545, "John Doe", 1015.10, "Payment for delivery");
```

```
  node.Set(myList, "DXO987", 26032009100100, 6541963285249512, 2);
  myList.Close();
```

You may notice that this approach closely resembles COS code for creating globals. In the same way, every time we write all the subscripts relative to some base one and add its value to be saved in the database.

The third way to write data is to explicitly set the number of indexes to create a value at this level of the hierarchy.

The last approach to creating nodes is to explicitly set the level at which you want to add a new value. At the same time, unlike the previous approach, the pointer moves through the tree.

To indicate the level of the tree, the SetSubscript() method is used, to which we pass two parameters: the required number of subscripts and the value of the new subscript. To insert values, the Set() method is used, in which only one parameter is passed - the value of the node.

Apart from writing data into DB one needs to also be able to read existing data from the DB. Since globals are similar to trees and in the example we did not skip indexes, we can safely recursively go around the whole global and output its values.

To get all the values we will use the familiar methods SetSubscript() and AppendSubscript() . In addition to them, we will use the methods:

- NextSubscript() - returns the value of the next index at the same level, similar to the $Next and $Order functions in COS.
- GetSubscriptCount() - returns the number of indexes at the current tree level.
- HasData() - checks if there is a value in this node, i.e. does it exist.
- HasSubnodes() - checks if the current index has sub-indices.

```
node.SetSubscriptCount(0);
node.AppendSubscript("");
string subscr = node.NextSubscript();
while (!subscr.Equals(""))
{
   node.SetSubscript(node.GetSubscriptCount(), subscr);
   if (node.HasData())
   {
      Console.WriteLine(" ".PadLeft(node.GetSubscriptCount() * 4, '-') + subscr);
      //ToDo: print node value
   }
   if (node.HasSubnodes())
   {
      ReadData(node);
   }
   subscr = node.NextSubscript();
}
```

We see that all our indexes are beautifully displayed. Now you need to add the node values instead of ToDo.

Considering that all data is stored in Caché as strings, the programmer will either need to know at which level of subscripts what type of data is stored or check each chunk of data and make conversions. The GetInt(), GetDouble(), GetLong(), GetString(), GetBytes(), GetList() and GetObject() functions are used to get global node values when you already know the type of data that is stored in this particular node. They return a value of type string and immediately convert it to type int, double, longInt, string, bytes [], ValueList, respectively. Last GetObject() function returns an object whose type can be checked and converted to the value of the desired type.

As already noted, all data is returned as a string. If, in fact, the data type is numeric, the system can determine this. But the system always defines lists, arrays of bytes and the strings themselves as strings. Therefore, the processing of such data must take into account at what level what type of data is stored. This is the reason why it is

considered a good idea to store the same type of data on the same level. Moreover, the system does not return an error when trying to display a list using the method of working with strings. Instead, a beautiful (but incomprehensible) text will be displayed:

So instead of ToDo in the previous code we can add the lines which will return the value of the node.

```
            Object value = node.GetObject();
            if (value is string)
            {
                if ((node.GetSubscriptCount() == 1) || (node.GetSubscriptCount() == 2
))
                {
                    Console.WriteLine(value.ToString());
                }
                else if (node.GetSubscriptCount() == 5)
                {
                    ValueList outList = node.GetList();
                    outList.ResetToFirst();

                    for (int i = 0; i < outList.Length - 1; i++)
                    {
                        Console.Write(outList.GetNextObject() + ", ");
                    }
                    Console.WriteLine(outList.GetNextObject());
                    outList.Close();
                }
                else if (node.GetSubscriptCount() == 4)
                {
                    string tempString = Encoding.GetEncoding(1251).GetString(node.Get
Bytes());
                    Console.WriteLine(tempString);
                }
            }
            else if (value is double)
            {
                Console.WriteLine(value.ToString());
            }
            else if (value is int)
            {
                Console.WriteLine(value.ToString());
            }
```

After we compose all these parts of code into one program, we will get the result:

The whole .NET project is on GitHub.

If you have comments or questions please don't hesitate to ask.

#Caché #.NET