Code First Stored Procedures – V2.1

Introduction

Version 2.1 of the CodeFirst stored procedures modules introduces more functionality and a completely new interface. The new interface is more in line with the CodeFirst paradigm of using POCO objects and separating declaration from usage. The "StoredProc" object has both generic and non-generic versions, and represents the stored procedure. Meta information needed to determine the types and properties for creating SQL Parameters are represented as attributes that can be added to properties or, in some cases, classes.

Declaring a Simple Stored Procedure

While this is a fairly simple stored procedure, it does make use of several of the more common data interchanges for stored procedures, using input and output parameters and returning a result set of data.

```
-- Stored procedure with input and output parameters, and a single result set
create proc testone @in varchar(5), @out int out
as
begin
	select table_name, column_name from INFORMATION_SCHEMA.COLUMNS
	set @out = @@ROWCOUNT
end
go
```

The first step in calling this stored procedure from code using "Code First Stored Procedures" is to create a class to hold the parameters for the stored procedure as its properties. The properties are decorated with attributes to define values used to create the proper Sql parameter for that property. For Example:

```
/// <summary>
/// Parameters object for the 'testone' stored procedure
/// </summary>
public class testone
{
    // Override the parameter name. The parameter name is "in", but that's
    // not a valid property name in C#, so we must name the property
    // something else and provide an override to set the parameter name.
    [StoredProcAttributes.Name("in")]
    [StoredProcAttributes.ParameterType(System.Data.SqlDbType.VarChar)]
    public String inparm { get; set; }

    // This time we not only override the parameter name, we're also setting
    // the parameter direction, indicating that this property will only
    // receive data, not provide data to the stored procedure. Note that
    // we include the size in bytes.
    [StoredProcAttributes.Name("out")]
    [StoredProcAttributes.Direction(System.Data.ParameterDirection.Output)]
    [StoredProcAttributes.Size(4)]
    public Int32 outparm { get; set; }
}
```

Each property will be matched to a parameter in the stored procedure. You can use the “NotMapped” attribute defined in ‘System.ComponentModel.DataAnnotations’ to make properties ‘invisible’ to this process. Attributes are used to define how the property will be mapped to a SqlParameter, defining the database type and characteristics of the property. In this case, the “Output” property defines the second property as receiving data from the stored procedure parameter that it is mapped to. “In” and “out” – the stored procedure parameter names – are not valid C# property names so we use the Name to override the ‘property name equals parameter name’ default.

The second step is to create a class to hold the returned result set. For each row in the result set, an object will be created and property names will be matched to column names for storing data.

```
/// <summary>
/// Results object for the 'testone' stored procedure
/// </summary>
public class TestOneResultSet
{
    [StoredProcAttributes.Name("table_name")]
    public string table { get; set; }

    [StoredProcAttributes.Name("column_name")]
    public string column { get; set; }
}
```

The “Name” attribute can be use to override the default property – column name mapping. The string given in the “Name” attribute is the column that will be mapped to this property.

The stored procedure that uses this parameter object and results object can be defined using the generically typed StoredProc object.

```
// simple stored proc
public StoredProc<testone> testoneproc = new StoredProc<testone>(typeof(TestOneResultSet));
```

This stored proc definition identifies the object 'testone' as the source of the stored procedure parameters and that the stored procedure will return a list of type 'TestOneResultSet'. By default, the type name of the parameters class is the name of the stored procedure called. This can be overridden either in the constructor, using a fluent API style interface, or by attributes on the class definition. While this object could be created anywhere it's needed, my preference is to declare these in my DbContext object, where they will reside with all the other database objects.

To call the stored procedure, once you have all the definitions created, you just create your parameter object and call the stored procedure.

```
using (testentities te = new testentities())
{
    //---------------------------------------------------------------
    // Simple stored proc
    //---------------------------------------------------------------
    testone parms1 = new testone() { inparm = "abcd" };
    ResultsList results1 = te.CallStoredProc<testone>(te.testoneproc, parms1);
    var r1 = results1.ToList<TestOneResultSet>();
}
```

In this example, we create an instance of the parameters object and provide a value to the input property. Then we execute the stored procedure using the 'CallStoredProc' routine, giving it the stored procedure object and the parameters object. This call returns a 'ResultsList' object (defined below) which is a list of result sets. The generically typed 'ToList' method allows the ResultsList to identify and return the requested result set by its type. The output property 'outparm' (defined above in the testone type) is automatically set to the output value returned from the mapped stored procedure parameter 'out'.

Stored procedures that do not take parameters can be defined using the non-generic 'StoredProc' class. In this example, we see first the declaration of the stored procedure object (as a property in this case), followed by the creation of the actual object in the parent object's constructor.

```
// stored proc with no parameters and multiple result sets
public StoredProc testthree { get; set; }

// use fluent API to set values in the parent object constructor
testthree = new StoredProc()
    .HasName("testthree")
    .ReturnsTypes(typeof(TestOneResultSet), typeof(TestTwoResultSet));

// alternate and equally valid declaration for testthree
testthree = new StoredProc("testthree", typeof(TestOneResultSet), typeof(TestTwoResultSet))
```

The object's necessary properties are defined using a Fluent API style interface. This particular example explicitly declares the stored procedure name and that it will return multiple result sets. The first result

set is a list of 'TestOneResultSet' and the second is a list of 'TestTwoResultSet'. Accessing multiple result sets is straightforward.

```
using (testentities te = new testentities())
{
    //------------------------------------------------------------
    // Stored proc with no parameters and multiple result sets
    //------------------------------------------------------------
    var results3 = te.CallStoredProc(te.testthree);
    var r3_one = results3.ToList<TestOneResultSet>();
    var r3_two = results3.ToArray<TestTwoResultSet>();
}
```

The type specifier on the 'ToList' and 'ToArray' methods of the returned ResultsList object identify which of the multiple result sets to return. If there is more than one result set of the same type (perfectly proper, by the way) only the first is returned through this method. To access a second or third result set of the same type, use an array indexer '[]' or an enumerator to process all the result sets.

Table Valued Parameters

One of the real powers of T-SQL stored procedures is the ability to pass an entire table as an input parameter to a stored procedure. Within the stored proc, this table can be treated pretty much the same as any other table, participating in joins, subqueries, etc. On the database side of things, a table 'type' must be created, which can then be used as a parameter type in the definition of the stored proc.

```
-- Create Table variable
create type [dbo].[testTVP] AS TABLE(
 [testowner] [nvarchar] (50) not null,
 [testtable] [nvarchar] (50) NULL,
 [testcolumn] [nvarchar](50) NULL
)
GO

-- Create procedure using table variable
create proc testfour @tt testTVP readonly
as
begin
       select table_schema, table_name, column_name from INFORMATION_SCHEMA.COLUMNS
       inner join @tt
       on table_schema = testowner
       where (testtable is null or testtable = table_name)
       and (testcolumn is null or testcolumn = column_name)
end
go
```

On the .Net side of things, this table definition must be recreated so that the data being transmitted can be type checked and formatted properly. The table and its columns are defined by attributes on the class that holds data rows to be passed as the table parameter. Here's an example of defining a class to pass data to a Table Valued Parameter.

```
/// <summary>
/// Class representing a row of data for a table valued parameter.
/// Property names (or Name attribute) must match table type column names
/// </summary>
[StoredProcAttributes.Schema("dbo")]
[StoredProcAttributes.TableName("testTVP")]
public class sample
{
    [StoredProcAttributes.Name("testowner")]
    [StoredProcAttributes.ParameterType(SqlDbType.VarChar)]
    [StoredProcAttributes.Size(50)]
    public string owner { get; set; }

    [StoredProcAttributes.ParameterType(SqlDbType.VarChar)]
    [StoredProcAttributes.Size(50)]
    public string testtable { get; set; }

    [StoredProcAttributes.ParameterType(SqlDbType.VarChar)]
    [StoredProcAttributes.Size(50)]
    public string testcolumn { get; set; }
}
```

The ‘schema’ and ‘tablename’ attributes form the qualified name of the database table type that is the stored procedure parameter. The attributes on the class properties must contain column definitions that mirror the actual table type definition. To use this in a parameters object, create a List or Array parameter with this class as the underlying type and decorate it with the “SqlDbType.Structured” parameter type.

```
/// <summary>
/// Parameter object for 'testfour' stored procedure
/// </summary>
[StoredProcAttributes.Name("testfour")]
public class testfourdata
{
    [StoredProcAttributes.ParameterType(SqlDbType.Structured)]
    [StoredProcAttributes.Name("tt")]
    public List<sample> tabledata { get; set; }
}
```

As with parameters classes and result set classes, the Name attribute can be used to override the default ‘property name equals column name’.

Here’s an example of calling a stored proc based on this class:

```
using (testentities te = new testentities())
{
    // new parameters object for testfour
    testfour four = new testfour();

    // load data to send in to the table valued parameter
    four.tabledata = new List<sample>()
    {
        new sample() { owner = "tester" },
        new sample() { owner = "dbo" }
    };

    // call stored proc
    var ret4 = te.CallStoredProc<testfour>(te.testfour, four);
    var retdata = ret4.ToList<testfourreturn>();
}
```

Since all the messy definition is handled by attributes in the class declaration, using the table valued parameter is relatively simple. We simply create a list of data, assign it to the property of the parameters class and call the stored proc.

Reference

StoredProc<T>

This object defines a stored procedure, and uses a Type specifier to identify the source of parameters for the stored procedure. Inherits from the class StoredProc.

StoredProc(params Type[] types) – Constructor, takes a series of types as parameters. Each type is used as the type of a resultset from the stored procedure call, in order of listing. The list of types may be null.

StoredProc<T> HasOwner(String owner) – Allows setting the schema or owner of the stored procedure in the database.

StoredProc<T> HasName(String name) – Allows setting the name of the stored procedure to be called.

StoredProc<T> ReturnsTypes(params Type[] types) – Allows defining or adding to the list of result set types that will be used to process result sets returned from the stored procedure. The list of types may be null.

StoredProc

Contains properties for defining the stored procedure that will be called.

String schema { get; set; } – Property containing the schema, or owner, of the stored procedure.

String procname { get; set; } – Contains the name of the stored procedure to call.

StoredProc HasOwner(String owner) – Fluent API style interface, assigns the 'schema' property to the input value.

StoredProc HasName(String name) – Fluent API style interface, assigns the 'procname' property to the input value.

StoredProc ReturnsTypes(params Type[] types) – Fluent API style interface, assigns the series of types of result sets returned by the stored procedure.

StoredProc()
StoredProc(String name)
StoredProc(String name, params Type[] types)
StoredProc(String owner, String name, params Type[] types) – Constructors for the StoredProc object.

Attributes

Schema – Overrides the default schema of 'dbo' for stored procedure and table valued parameter declarations.

TableName – Overides the default table name (the class name) for table valued parameter declarations.

Name – Overrides the default property name to parameter / column name matching. Allows setting the parameter name (for input to the stored proc) or column name (for result sets and table valued parameter input classes).

Size – Specifies the size in bytes of available storage for return values for output parameters.

Precision – Specifies the precision value for Decimal data being passed to SQL.

Scale – Specifies the scale value for Decimal data being passed to SQL.

Direction – Value is from the enum "ParameterDirection" and identifies whether this parameter is sending data to the stored procedure, receiving data from the stored procedure, or both.

ParameterType – The SqlDbType of the parameter.

TypeName – User defined type name for the parameter.

ResultsList

List of result sets from the stored procedure, implements the IEnumerable interface for processing multiple result sets. Also provides some helper routines for accessing result sets.

Void Add(List<object> list) – Adds a result set to the object.

IEnumerator GetEnumerator() – Gets an enumerator that iterates over the result sets. The current result set item returned by the enumerator is of type List<object>.

Int32 Count – returns a count of the number of result sets saved from the call to the stored procedure.

List<object> this[int index] – Accessor property that returns the nth result set returned from the stored procedure. Throws an exception if there is no nth result set item.

List<T> ToList<T>() – Accessor method that returns the first result set containing items of type T, cast to a List of type T.

Array<T> ToArray<T>() – Accessor method that returns the first result set containing items of type T, cast to an Array of type T.

Calling and Reading Data from Stored Procedures

The method that actually does all the work is created as an extension method off of the DbContext object. This allows the ReadFromStoredProcedure routine access to the current connection information.

ResultsList CallStoredProc <T>(this DbContext context, StoredProc procedure, T data) – Type specific routine uses values from the 'data' object as parameters for the stored procedure.

ResultsList CallStoredProc (this DbContext context, StoredProc procedure, IEnumerable<SqlParameter> parms = null) – Non-type specific routine sends values from the input SqlParameters to the stored procedure.