Code First Stored Procedures – V3.3

Code First Stored Procedures has received a much needed sprucing up to include features that were introduced in recent versions of Entity Framework. Version 3 adds support for asynchronous procedure calls and Streaming of data to and from SQL Server. It also adds a new call pattern to clean up the interface and make it more intuitive. Note nothing in the new interface will break any of your existing Stored Procedure calling code. Version 3.1 includes the option to enroll your stored procedure call into a transaction on the database, allowing you to group updates and rollback changes. Version 3.3 contains mostly internal changes for working with the Glimpse.EF6 library.

Declaring and Calling 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:

Input Parameters Object

```
/// <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.

Results Object

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 used 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.

Declare the StoredProc Property

The third step is to declare a StoredProc property in your DbContext, and call “InitializeStoredProcs” in the DbContext constructor to populate the property with an actual object. Note: If you have existing StoredProc code, it will continue to work, however the method described here is preferred.

```
// Simple Stored Proc declaration in DbContext
public class testing : DbContext
{
    // StoredProc object to call a proc named "testone"
    [StoredProcAttributes.Name("testone")]
    [StoredProcAttributes.ReturnTypes(typeof(TestOneResultSet))]
    public StoredProc<testone> test { get; set; }

    // Constructor
    public testing()
    {
        // Creates and initializes StoredProc properties in DbContext
        // Must be called prior to using any StoredProc objects
        this.InitializeStoredProcs();
    }
}
```

This StoredProc property definition identifies a property in the DbContext that can be used to call a stored procedure. The type “<testone>” identifies the type of the input paramaters object. The attributes declare the name of the stored procedure to call and that the stored procedure is expected to return a set of “TestOneResultSet” objects. The stored procedure name, schema owner and return types can be set through attributes, as shown here, or via a fluent API interface.

The name of the stored procedure to be called is set by the “Name” attribute on the StoredProc property in DbContext; the “Name” attribute can also be placed on the input type (“testone” in this case”) in the type declaration. If neither of these are provided, the type name of the input type object is used as the stored procedure name.

Note that the extension method “InitializeStoreProcs” must be called to initialize all the StoredProc properties declared in the DbContext object, prior to any use.

Calling the Stored Procedure

To call a stored procedure, use the “CallStoredProc” method on the StoredProc<T> object, passing in the parameters object.

```
// Example of calling a simple stored proc
using (testing db = new testing())
{
    // create input parameters object
    var test1parms = new testone() { inparm = "abcd" };
    // call the stored proc
    var test1results = db.test.CallStoredProc(test1parms);
    // get our results into a usable object
    List<TestOneResultSet> results = test1results.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 on the stored procedure object and pass in the parameters object.

The CallStoredProc method returns a ‘ResultsList’ object (defined below) which is a list of result sets having the types identified in the “ReturnTypes” attribute in the object declaration. The order of types listed must match the order of kinds of result sets returned by the stored procedure. The generically typed ‘ToList’ method allows the ResultsList to identify and return the requested result set by its type.

Returned values are matched to properties in the same manner as input parameters. If a “Name” attribute does not specify a returned column name, the properties are matched on property name. In this case, 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’ based on matching return column name and the name specified in the “Name” attribute.

Stored Procedures without Parameters

Stored procedures that do not take parameters can be defined and called using the non-generic ‘StoredProc’ class.

```
// Simple Stored Proc declaration in DbContext using Fluent API instead of Attributes
public class testing : DbContext
{
    // Stored Proc with no parms & two return types
    public StoredProc test3 { get; set; }

    // Constructor
    public testing()
    {
        // Must be called prior to using any stored proc objects
        this.InitializeStoredProcs();

        test3.HasName("testthree")
            .HasOwner("dbo")
            .ReturnsTypes(typeof(TestOneResultSet), typeof(TestTwoResultSet));
    }
}
```

In this example, the object's necessary properties are defined using a Fluent API style interface instead of attributes. In addition to declaring the stored procedure name and schema owner, this example indicates that the stored procedure will return multiple result sets. The first result set is a list of 'TestOneResultSet' and the second is a list of 'TestTwoResultSet'. Calling the stored procedure is straightforward, just invoke the "CallStoredProc" method , without any parameters.

Accessing Results Sets

Returned result sets from calling a stored procedure are saved in the ResultsList object, returned by the CallStoredProc method. Stored procedures can return multiple results sets. These are accessed from the ResultsList object, using the returned type declarations, as shown below.

```
using (testentities te = new testentities())
{
    // Stored proc with no parameters and multiple result sets
    var results3 = te.test3.CallStoredProc();
    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 procedure, 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 procedure.

```
-- 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 a 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>
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 procedure based on this class:

```
public class testentities : DbContext
{
    // StoredProc object to call a proc named "testfour"
    [StoredProcAttributes.Name("testfour")]
    [StoredProcAttributes.ReturnTypes(typeof(testfourreturn))]
    public StoredProc<testfourdata> test4 { get; set; }

    // Constructor
    public testentities()
    {
        // Must be called prior to using any StoredProc objects
        this.InitializeStoredProcs();
    }
}

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

    // 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.test4.CallStoredProc(four);
    var retdata = ret4.ToList<testfourreturn>();
}
```

Since all the messy definition is handled by attributes in the declaration of classes and properties, 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.

Command Timeout

The underlying SQL interface has an oddity – each call to execute a stored procedure has its own, independent, timeout value. In order to set this value for an execution instance, the "CallStoredProc" method has an optional parameter "CommandTimeout".

```
// call stored procedure with an extended timeout value
var ret4 = te.test4.CallStoredProcAsync(4000, four);
var retdata = ret4.ToList<testfourreturn>();
```

In this case, we're telling the underlying SQL call to use 4 seconds as the Command Timeout value.

Transactions

CodeFirstStoredProcs must be given a transaction to enroll in when calling a stored procedure within a transaction context. The transaction is a property of the StoredProc object and can be set directly, or a fluent interface style can be used. Alternatively, the transaction can be included in the call to execute the stored procedure. NOTE: Please note that the type of the internally stored transaction has changed as of V3.3 from “SqlTransaction” to “DbTransaction”. This change is to allow this library to work with Glimpse.EF6.

Transactions can be generated from multiple sources – from the connection object, the database object, or by using the generic “TransactionScope” object defined in “System.Transactions”.

```
// Options for enrolling a stored procedure call in a Transaction
var conn = db.Database.Connection;
using (DbTransaction tran = conn.BeginTransaction())
{
    // Manually enroll in transaction
    te.test4.Transaction = tran;
    var ret4 = te.test4.CallStoredProc(4000, four);
    te.test4.Transaction = null;

    // … or …

    // Fluent API to enroll in transaction
    var ret4 = te.test4
        .UseTransaction(tran)
        .CallStoredProc(4000, four);
    te.test4.dbTransaction = null;

    // … or …

    // Direct API to enroll in transaction
    var ret4 = te.test4
        .CallStoredProc(4000, tran, null, null);
}
```

Enrolling the stored procedure call in the transaction can be done by manually setting the stored procedure objects ‘Transaction’ property, by using the fluent API style method call or by adding the transaction object to the “CallStoredProc” method execution. The StoredProc object will stay ‘enrolled’ in the provided transaction until the “Transaction” property is set to “null”. Note that not enlisting the stored procedure call in the current transaction can generate an error.

In this next example, we're enrolling the stored procedure call in a transaction that is started at the database object that is a member of the current database context.

```
// Enrolling a stored proc in DbContextTransaction
using (testentitiesnew db = new testentitiesnew())
{
    using (var dbtran = db.Database.BeginTransaction())
    {
        // Enroll call in current transaction
        var resultstran1 = db.trantest1
            .UseTransaction(dbtran)
            .CallStoredProc(new tran_test() { a = "1234", b = 1234 });

        dbtran.Commit();
    }
}
```

Again, the Stored Proc object will remain 'enrolled' in the transaction until the 'Transaction' property is reset to null. Note that not enlisting the stored procedure call in the current transaction can generate an error.

Using a TransactionScope object to manage transactions is slightly different. The stored procedure call will be automatically enrolled in the current transaction.

```
// use transactionscope
using (testentitiesnew db = new testentitiesnew())
{
    using (var ts = new TransactionScope())
    {
        // No manual transaction enrollment required
        var resultstran1 = db.trantest1
            .CallStoredProc(new tran_test() { a = "1234", b = 1234 });

        // "cancel" the transaction
        ts.Dispose();
    }
}
```

Asynchronous Processing

CodeFirstStoredProcs has been expanded to include support for asynchronous processing. The interface elements that support async processing have “Async” appended to the method name, and return Tasks that can be waited on .

```
// call stored procedure and await response
var ret4 = await te.test4.CallStoredProcAsync(four);
var retdata = ret4.ToList<testfourreturn>();
```

The Async versions of the procedures support all the same interface and processes as their non-async counterparts and use the async / await calling pattern. Support of cancellation tokens is included.

```
// Call StoredProc with cancellation token having a one second timeout
CancellationTokenSource tokensource = new CancellationTokenSource();
tokensource.CancelAfter(1000);
try
{
    var ret4 = await te.test4.CallStoredProcAsync(tokensource.Token, four);
}
catch (Exception ex)
{
    …
}
```

The StoredProc object has a property ‘commandBehavior’ that can be set to help performance. The use of ‘CommandBehavior.SequentialAccess’ can, under some circumstances, improve the performance for async data processing. As always, testing is recommended.

Streaming Data to SQL Server

To enable streaming of data to SQL Server, a few things must be put into place. First, the data element must be of type (or derived from type) Stream, and the data property "Size" attribute is set to '-1'.

```
public class testfileupdata
{
    [StoredProcAttributes.ParameterType(System.Data.SqlDbType.NVarChar)]
    public String filename { get; set; }

    // set size=-1 to tell sql to stream data
    [StoredProcAttributes.ParameterType(System.Data.SqlDbType.VarBinary)]
    [StoredProcAttributes.Size(-1)]
    public Stream filedata { get; set; }
}
```

To upload data, the Stream must be created, and then the StoredProc called normally.

```
// create our parameters object with a stream of data
testfileupdata f = new testfileupdata()
{
    filename = "testing",
    filedata = new MemoryStream(Encoding.UTF8.
    GetBytes("123498;llk;asf010uro;njewfjiru093rcn8ury9tyt4nu8qo38423845743poiwej"))
};

// send streamed data to database
Db.testfileup.commandbehavior = System.Data.CommandBehavior.SequentialAccess;
var fileupresults = db.testfileup.CallStoredProc(f);
```

Be sure to set the StoredProc object's "commandBehavior" property to "System.Data.CommandBehavior.SequentialAccess" to ensure streaming to the host.

Streaming Data from SQL Server

Streaming data back from SQL Server takes a bit more setup. Two Stream types are supported: FileStream and MemoryStream. FileStreams require the name of the file to be included in the row of data returned along with the actual stream and will create files for the returned data. MemoryStreams can be defined to read data into properties that are Byte[] and Strings, in addition to a Stream. Buffering, if indicated, means that the internal Stream used to read data will be wrapped in a BufferedStream object.

```
public class testfiledownresults
{
    [StoredProcAttributes.ParameterType(System.Data.SqlDbType.NVarChar)]
    public String filename { get; set; }

    [StoredProcAttributes.StreamToFile(Buffered = true, Location="C:\\Temp",
        FileNameField="filename", LeaveStreamOpen=false)]
    public String filedata { get; set; }
}
```

To use file streaming, the row of data returned must include a name field that can be referenced to create the receiving FileStream. The “Location” field identifies the directory or UNC where the files will be written, the “FileNameField” tells CodeFirstStoredProcs which returned property to look in to find the name of the file. The Location and file name are combined to create a fully qualified file name and are passed to the FileStream constructor. After being written, the FileStream is rewound, and by default is closed. The “LeaveStreamOpen” flag can be set to tell CodeFirstStoredProcs to not close the stream. If this flag is set, the caller is responsible for closing the stream.

MemoryStreams can be also be used to receive streamed data from SQL Server. MemoryStreams also support base property types of Byte[], and String.

```
public class testfiledownresults
{
    [StoredProcAttributes.ParameterType(System.Data.SqlDbType.NVarChar)]
    public String filename { get; set; }

    // Get data back in a stream, leave the stream open so we can access it
    [StoredProcAttributes.StreamToMemory(Buffered = true, LeaveStreamOpen=true)]
    public Stream filedataStream { get; set; }

    // Streamed data will be read into a byte array
    [StoredProcAttributes.StreamToMemory(Buffered = true)]
    public Byte[] filedataByteArray { get; set; }

    // Streamed data will be read into a string using the indicated encoding
    [StoredProcAttributes.StreamToMemory(Buffered = true, Encoding = "UTF8")]
    public String filedataString { get; set; }
}
```

Obviously, with large data sets, streaming to memory can cause problems, so use with caution. When streaming to a String, the “Encoding” attribute parameter is used to interpret the returned data. The value passed to “Encoding” should be the name of one of the properties on the System.Text.Encoding object that returns an Encoding object. Currently, valid values include “ASCII”, “BigEndianUniCode”, ”Default”, ”Unicode”, “UTF32”, “UTF7”, and “UTF8”.

StoredProcs that receive streamed data are called normally, and async processing is supported.

```
// get data streamed from sql server
testfiledowndata getfile = new testfiledowndata()
{
    filename = "testing"
};

db.testfiledown.commandbehavior = System.Data.CommandBehavior.SequentialAccess;
var filedownresults = await db.testfiledown.CallStoredProcAsync(getfile);
```

To ensure that data is streamed, set the StoredProc object’s ‘commandBehavior’ property to ‘SequentialAccess’.

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 result set from the stored procedure call, in order of listing. The list of types may be null. Depricated. StoredProc objects should be properties in a DbContext object and initialized by calling "InitializeStoredProcs" in the DbContext object constructor.

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<T> UseTransaction(SqlTransaction)
StoredProc<T> UseTransaction(DbTransaction)
StoredProc<T> UseTransaction(DbContextTransaction) – Specify the transaction in which to enroll the stored procedure call.

CallStoredProc(T data, params Type[] types)
CallStoredProc(int? CommandTimeout, T data, params Type[] types)
CallStoredProc(int? CommandTimeout, DbTransaction transaction, T data, params Type[] types) – Call the stored procedure. The CommandTimeout parameter allows setting of execution instance specific timeouts. The transaction parameter identifies a transaction in which to enroll the stored procedure call. Input values in the data parameter are set to the stored procedure and the optional types parameter can be used to define return types, though setting return types through a "ReturnTypes" attribute on the object declaration is preferred.

CallStoredProcAsync(T data, params Type[] types)
CallStoredProcAsync(int? CommandTimeout, T data, params Type[] types)
CallStoredProcAsync(CancellationToken token, T data, params Type[] types)
CallStoredProcAsync(DbTransaction transaction, T data, params Type[] types)
CallStoredProcAsync(CancellationToken token, int? CommandTimeout, T data, params Type[] types)
CallStoredProcAsync(CancellationToken token, int? CommandTimeout, DbTransaction transaction, T data, params Type[] types) – Call the stored procedure asynchronously; returns a Task<ResultsList> that can be waited on. The CancellationToken can be used to cancel a process.

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.

CommandBehavior commandBehavior – Property that sets the data readers' "CommandBehavior" option. Default value is System.Data.CommandBehavior.Default. Required to be set to CommandBehavior.SequentialAccess for streaming.

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

SqlTransaction SqlTransaction { get; set; } – Property access to transaction object, deprecated and kept for backwards compatibility.

StoredProc<T> UseTransaction(SqlTransaction)
StoredProc<T> UseTransaction(DbTransaction)
StoredProc<T> UseTransaction(DbContextTransaction) – Specify the transaction in which to enroll the stored procedure call.

StoredProc()
StoredProc(String name)
StoredProc(String name, params Type[] types)
StoredProc(String owner, String name, params Type[] types) – Constructors for the StoredProc object. Depricated. StoredProc objects should be properties in a DbContext object and initialized by calling "InitializeStoredProcs" in the DbContext object constructor.

CallStoredProc(params Type[] types)
CallStoredProc(int? CommandTimeout, params Type[] types)
CallStoredProc(int? CommandTimeout, DbTransaction transaction, params Type[] types) – Call the stored procedure. The CommandTimeout parameter allows setting of execution instance specific timeouts. The transaction parameter identifies the transaction in which to enroll the stored procedure call. The optional types parameter can be used to define return types, though setting return types through a "ReturnTypes" attribute on the object declaration is preferred.

CallStoredProcAsync(params Type[] types)

CallStoredProcAsync(CancellationToken token, params Type[] types)
CallStoredProcAsync(int? CommandTimeout, params Type[] types)
CallStoredProcAsync(DbTransaction transaction, params Type[] types)
CallStoredProcAsync(CancellationToken token, int? CommandTimeout, DbTransaction transaction, params Type[] types) – Call the stored procedure asynchronously; returns a Task<ResultsList> that can be waited on. The CancellationToken can be used to cancel a process.

Attributes

Schema – Overrides the default schema owner of 'dbo' for stored procedure and table valued parameter declarations. This attribute should be placed on the StoredProc property declaration in the DbContext object for stored procedures and on the underlying table object for table valued parameters.

Name – Overrides the default property name for use with SQL Server. When placed on a data property, it indicates the column name (for result sets and table valued parameter input classes). When placed on the StoredProc object declaration in the DbContext object, it sets the name of the stored procedure to be called. It can also be placed on the input data object to set the stored procedure name.

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.

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

ParameterType – The SqlDbType of the parameter.

TypeName – User defined type name for the parameter.

FileStream – Declares that this data property is to be streamed from SQL Server into a file, defined in the attribute parameters.

MemoryStream – Declares that this data property is to be streamed from SQL Server and kept in memory. Stream, Byte[] and String are valid data types for a property with this attribute.

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

(These items are depricated and may be removed in a future release)

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. Depricated.

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. Depricated.

ResultsList CallStoredProc <T>(this DbContext context, StoredProc procedure, int CommandTimeout, T data) – Type specific routine uses values from the 'data' object as parameters for the stored procedure. The CommandTimeout value is passed to the command object executing the stored proc, allowing for stored procedures that take longer than the default amount of time. Depricated.

ResultsList CallStoredProc (this DbContext context, StoredProc procedure, int CommandTimeout, IEnumerable<SqlParameter> parms = null) – Non-type specific routine sends values from the input SqlParameters to the stored procedure. The CommandTimeout value is passed to the command object executing the stored proc, allowing for stored procedures that take longer than the default amount of time. Depricated.