Table of Contents

SQL+.NET Overview

Do we really need another ORM?

We looked at all the tools currently available, read countless reviews and rants, then used that information to define the requirements of our offering. What we ended up with was a SQL first ecosystem that empowers developers who are skilled with SQL. Services generated with SQL+ simply run faster and are more durable than that of any other tool, and the best part is, you get this all without sacrificing productivity.

SQL+ is a common sense approach to building data-services. You simply write your SQL to handle your CRUD operations, add semantic tags to define and fine tune your services, and then use the SQL+ Code Generation Utility to generate robust data services. It's a simple, proven approach that not only saves developer time but also reduces the load on your database saving you money down the line.

Please note this document is a reference guide. If you are just starting out with SQL+ we recommend you visit https://www.sqlplus.net/Home/Learn *and go through the video tutorials. Those hands-on tutorials are designed to guide you on your way and represent the fastest path to fully understanding and utilizing all the features and functionality provided by this tool.*

Welcome to the community and we look forward to hearing your success stories.

Two Different Approaches

SQL+ is a SQL first ecosystem, meaning you start with SQL, add comments, and generate code. It works with scalar functions, table functions, stored procedures and SQL files. Regardless of the approach, the generated code subscribes to the same patterns and practices. Additionally, the developer experience (consumers of your services) enjoys a consistent and familiar pattern, making your services easy to develop, and easy to use.

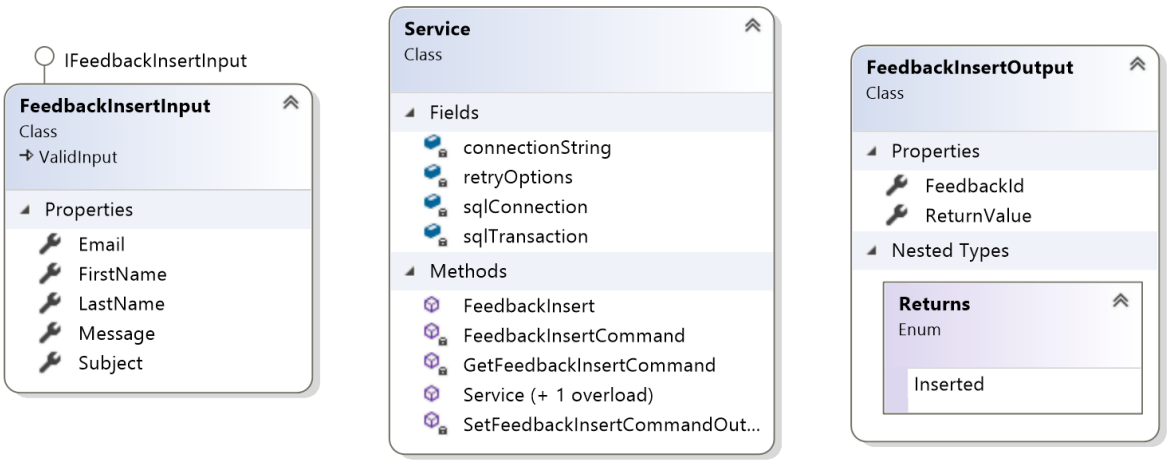
Stored Procedures and Functions

When using stored procedures or functions, you simply add comments to the stored procedure or function body, add the routine or schema to the build definition and run the code generator. The builder will query the database looking for those procedures, and generate your services accordingly. For more information on the build definition file see the section on the build definition.

Concrete Queries

By writing SQL statements and placing them in the appropriate folders within your project, the code generator will inspect those folders and generate services accordingly.

Generated Libary



The above diagram illustrates the generated class library for a simple insert statement. The input object (FeedbackInsertInput) encapsulates the parameters passed into the procedure and are passed to the service as an argument to the service call. The output object (FeedbackInsertOutput) shows the FeedbackId as a property which is an out parameter set to ScopeIdentity() as well as the return value which is enumerated (in this case a single value Inserted).

We choose this object-oriented design to facilitate calling services directly, as well as supporting data binding in various UI clients. In addition, the input objects with escalated validation tags provide the IsValid() method which means invalid data is prevented from ever making it to the database.

Semantic Tags

The only difference between ordinary SQL, and SQL+, is the use of semantic tags. These tags are represented by comments embedded in the SQL and fall into one of two categories:

1) Primary tags in the format (--+Token)
2) Supplemental tags in the format (--&Token)

Semantic tags are used to define your service, and optionally provide validation to parameters, indicate display properties, enumerate return values, specify parameter modes, and identify multi result set queries all within your SQL. This makes the SQL a single source of truth where semantic tags are escalated into the generated code, and therefore enforced in the service layer. This makes those generated services true gatekeepers of pristine data, all within a highly productive workflow.

There are code snippets available from the downloads page at https://www.sqlplus.net/Home/Downloads to simplify the process of adding tags and reduce the learning curve.

The following sections provide additional information for each of the semantic tag categories.

Routine Tag

The routine tag is the only tag that is required and is placed at the beginning of the procedure or statement. This tag provides the flexibility to tailor your service to the exact results you wish to achieve. You specify the select type (see below) as well as provide a comment and author name. In addition, there is an obsolete tag (optional) that can be used for graceful deprecation and transition to newer versions.

```
--+SqlPlusRoutine
  --&SelectType=NonQuery or SingleRow or MultiRow or JSON or XML or MultiSet
  --&Comment=Comment
  --&Author=Author
  --&CommandTimeout=seconds
  --&Obsolete=Error or Warning,Message
--+SqlPlusRoutine

*Note that the command timeout and obsolete tags are optional*
```

SelectType: By specifying a select type we tailor the output of our service. Choose from the following:

- **NonQuery** – the output of the call will not contain a result set.
- **SingleRow** – the procedure executes a SELECT that will return a maximum of one row. Used when selecting by a primary key. Since this will return a single row, the result set is mapped to a single instance of a (result object), not a list.
- **MultiRow** – the procedure executes a SELECT statement that will return an undetermined number of rows. The multi-row result is mapped into a list of (result object) not a single entity.
- **JSON** – the procedure executes a SELECT … FOR JSON PATH. The result string is mapped to a property - JsonResult.
- **MultiSet** – use this in combination with **Query** tags for services that return multiple result sets. See Multiple Result Sets and Query Tags for additional information.
- **XML** – the procedure executes a SELECT … FOR XML. The result string is mapped to a property - XmlResult.

Comment: Comment for the generated service available to other developers using the service. This comment will be displayed via IntelliSense to the users of your class library.

Author: The creator of the SQL routine. This information provides other developers the means to quickly track and resolve issues about the SQL, you can also provide an email here if so desired.

Obsolete: Used to deprecate routines in a less abrupt way. It gives other developers the opportunity to seek alternatives during a transition. Choose one and only one of the following and supply a message:

- **Error** – the method will generate an error at compile time.
- **Warning** – the method will generate a warning at compile time.

CommandTimeout: Used to override the default timeout for commands. Enter the number of seconds for the allowed execution time.

Parameter SectionTags

When generating services with the concrete query approach, use this tag to surround any variables that should be included as parameters to your query. You place the opening tag above the DECLARE statement, and the closing tag after the last variable you want to include. Any values assigned to the variables within the parameters tags will be ignored by the builder. This provides a convenient way to test the procedures in isolation.

```
--+Parameters
--+Parameters
```

Example:

The UserId variable is enclosed in the parameters tag, therefore it will be included as a property of the input object. The TempValue variable will not, since it falls outside the parameter sections tag.

```
--+Parameters

DECLARE
@UserId int;

--+Parameters

DECLARE
@TempValue int;
```

Parameter Validation Tags (Optional)

Parameter validation tags are used to add functionality beyond what a database supports natively. Place one or more tags above the parameter declaration to achieve the desired results. The following parameter validation tags are available:

```
--+Comment=Comment
--+CreditCard
--+Default=DefaultValue
--+Display=Name,Description
--+Email
--+Enum=EnumerationName
--+Explicit=ExplicitValue
--+Html
--+MaxLength=MaximumLength
--+MinLength=MinimumLength
--+Password
--+Phone
--+PostalCode
--+Range=MinimumValue,MaximumValue
--+RegExPattern=RegularExpression
--+Required
--+StringLength=MinimumLength,MaximumLength
--+Url
```

Comment: The comment tag although not a validator, is applied to a parameter to create a comment for the generated property and is available to other developers using your service via Intellisense.

CreditCard: Enforces credit card validation for the mapped property in the generated service.

Default: Provides a default value for the mapped property in the generated service.

Email: Enforces email validation for the mapped property in the generated service.

Explicit: Verbatim parameter annotation for use with your custom annotations.

Enum: The name of the enumeration to substitute for the type. (See Enumerations section below)

Html: Enforces Html validation for the mapped property in the generated service.

MaxLength: Enforces max length validation for the mapped property in the generated service.

MinLength: Enforces min length validation for the mapped property in the generated service.

Password: Indicates the value will be a password. Alters display when bound to UI elements.

Phone: Enforces liberal phone validation for the mapped property in the generated service.

PostalCode: Enforces liberal postal code validation for the mapped property in the generated service.

Range: Enforces range validation for the mapped property in the generated service.

RegExPattern: Enforces regex validation for the mapped property in the generated service.

Required: Enforces required (non-nullable) validation for the mapped property in the generated service.

StringLength: Enforces min/max string length validation for the mapped property in the generated service.

Url: Enforces the format of the field to be a fully qualified URL.

Example:

A routine that would update a non-nullable column that holds a value for the customers primary email address could be validated in the service layer with the following tags:

```
--+Required
--+Email
--+Comment=Customers Primary Email
@PrimaryEmail varchar (64)
```

And enforced as follows:
The value is required, it must be a valid email, and a comment 'Customers Primary Email' is presented to users of the service through IntelliSense.

Validation Tag Supplemental Values

The error messages generated by invalid data are provided through data annotations in C#, and the default values are acceptable most of the time. However, if custom messages are preferred, the following supplemental tags can be utilized to customize error messages.

```
--&ErrorMessage=ErrorMessage
--&ErrorResource=ResourceType,ResourceName
```

ErrorMessage: Provides a custom error message to the associated validation tag.

ErrorResource: Provides the ability to map a resource file and key to the associated validation tag.

Examples:

To override the default error message and specify a custom message for the required tag:

```
--+Required
--&ErrorMessage=Email is required
@Email varchar(64)
```

To override the default error message and specify a resource file and key:

```
--+Required
--&ErrorResource=ErrorMessages,RequiredMessage
@Email varchar(64)
```

Where ErrorMessages is the name of a resource file (resource type) and RequiredMessage is the key within that file.

Parameter Display Tags (Optional)

Display tags are utilized when input objects are bound to UI elements. There are many options available to this tag and different application platforms like MVC or WPF can make use of some or all of them. In practice, the Display name is the most useful with or without resources files. When using the display name in MVC applications the description is ignored. It is the responsibility of the UI framework to consume the annotations.

```
--+Display=Name,Description
  --&ShortName=ShortName
  --&Prompt=Prompt
  --&Group=Group
  --&Resource=ResourceType
```

Display: (Name) gets or sets a value that is used for a label, (Description) could be used for hovers or watermarks etc.

Prompt: Gets or sets a value that will be used as a hover or watermark etc.

ShortName: Could be used as an abbreviated name.

Group: Gets or sets a value that is used to group fields in the UI.

Resource: Specifies the resource file to use in tandem with the display property where the Display=Name becomes the key in the resource file.

Example:

When a UI element is bound to this property, it will display **User Name** vs **UserName**. (Space)

```
--+Display=User Name, X
@UserName varchar(32)
```

When a UI element is bound to this property, it will display the value from the resource file (UIPrompts) with the key UserName.

```
--+Display=UserName, not currently used.
--&Resource=UIPrompts
```

Parameter Mode (Direction) Tags

By default, procedures in T-SQL are inputs. When out is specified for the parameter, it becomes an in/out parameter in the parameters collection. In SQL+ parameters are input by default, and exclusively out when out is specified in the procedure. To everride the default behavior mode tags are available.

```
--+Input
--+InOut
--+Output
```

Input: Used with procedure parameters to make a parameter input and output.

InOut: Used with concrete queries to make a parameter input and output.

Output: Used with concrete queries to make a parameter output.

Procedure Example:

In the following procedure, the UserId parameter is intended to be both an input and an output. Since by default parameters are input, we mark the parameter using the SQL out keyword. To make this parameter input and output we add the input tag. Additional validation tags are applied to the input.

```
ALTER PROCEDURE dbo.TESTINPUT
(
   --+Input
   @UserId int out
)
```

Concrete Query Example:

In the following SQL statement, the UserId is an input, the name is an input-output, and the return value is an output. In addition, when using the special variable name @ReturnValue with the output tag, the return value can be enumerated. (See also return tags.)

```
@UserId int,

--+InOut
@Name varchar(32),

--+Output
@ReturnValue int
```

Return Tags (Optional)

The return tag is used to enumerate return values. Enumerating return values allows clear information about the outcome of a given service call to be passed back to developers using your service. Simply add the tag to each unique return value, and an enumeration will be generated and associated with the value of the return. In addition, it can be used in conjunction with the output tag and variable name *@ReturnValue* when using concrete queries.

```
--+Return=EnumeratedValue
```

Example:

If a routine supported an upsert (insert or update) the return value could be used to provide specific information related to the exact crud operation that was executed.

Procedures:

```
--+Return=Inserted
RETURN 1;

--+Return=Modified
RETURN 2;

--+Return=NotFound
RETURN 3;
```

Statements:

```
--+Return=Inserted
SET @ReturnValue = 1;

--+Return=Modified
SET @ReturnValue = 2;

--+Return=NotFound
SET @ReturnValue = 3;
```

Would create the following enumeration:

```
public enum Returns
{
    Inserted = 1,
    Modified = 2,
    NoAction = 3
}
```

and the return value can then be evaluated as follows:

```
if(output.ReturnValue == SampleProcedureOutput.Returns.Inserted)
```

Multiple Result Sets and Query Tags

```
--+QueryStart=Name,SelectType
--+QueryEnd
```

When procedures or statements return multiple result sets, each query is wrapped with query tags to indicate a name for the result set property as well as the select type that applies to that specific query.

```
--+QueryStart=Name,SelectType
SELECT … WHERE … GROUP BY … etc.
 --+QueryEnd

--+QueryStart=Name,SelectType
SELECT … WHERE … GROUP BY … etc.
 --+QueryEnd

--+QueryStart=Name,SelectType
SELECT … WHERE … GROUP BY … etc.
 --+QueryEnd
```

QueryStart: Marks the beginning of the query.

Name: Indicates the name for the query, this will be the name of the class defined by the query result sets

SelectType: See select types in the routine tag section.

QueryEnd: Marks the end of the query and should be placed after the entire select statement including any filter clauses or aggregates.

Installation and Setup

To install the SQL+.NET code generation utility, head to the downloads page at https://www.SQLPLUS.net/Home/Downloads and follow the link for the visual studio extension. At the visual studio market place, click the download button, then double click the VSIX file. Follow the onscreen prompts to complete the installation. Once installed the builder is run by right-clicking on your project and choosing the SQL+.NET Build option from the menu.

If this is the first time running the builder, the builder will create the folders and files required to configure your build, as well as the root folder for placing concrete queries. There are two configuration files created in the SQL+ folder.

Database Connection (DatabaseConnection.json)

Use this file to define the database connection. The database type and the connection string should point to the source database, that is the database you are building from. This is required regardless of the approach taken, concrete queries or procedures.

```
{
  "DatabaseType": "MSSQLServer",
  "ConnectionString": "Enter the connections string for the source database"
}
```

DatabaseType: Currently only MSSQLServer is supported. We hope to support others in the future.

ConnectionString: Edit as necessary providing the server (your server), initial catalog (database name), user id, and password. This user must have access to system views, typically dbo. This connection string is for the source database.

Since this file contains sensitive information, it should be excluded from and source control.

Build Configuration (BuildConfiguration.json)

When the code generation utility runs, it inspects the contents of the BuildConfiguration.json file to determine what procedures, enumerations, and static data to build.

```
{
  "Template": "DotNetFramework | DotNetCore | Documentation",
  "BuildSchemas": [
    {
      "Schema": "SCHEMA",
      "Namespace": "NAMESPACE or + for project root"
    }
  ],
  "BuildRoutines": [
    {
      "Schema": "SCHEMA",
      "Namespace": "NAMESPACE or + for project root ",
      "Routine": "ROUTINE"
    }
  ],
  "EnumQueries": [
    {
      "Name": "<name for the enum>",
      "Table": "<source table of enum>",
      "NameColumn": "<column for the name>",
      "ValueColumn": "<column for the value>",
      "Filter": "<query filter>"
    }
  ],
  "StaticQueries": [
    {
      "Name": "<name for the static query>",
      "Query": "<Select Statement for the query>",
    }
  ],
}
```

Template: Choose the appropriate template for the destination project, currently SQL+.NET supports DotNetFramework and DotNetCore. In the future, DotNetCore may be further separated into different versions depending on features that may not be backward compatible.

BuildSchemas: Enter the list of database schemas that you would like to build. Only routines within those schemas will be built.

- **Schema**: The actual name of the schema in the database.
- **Namespace**: The desired namespace (folder) in the destination project.
  Note that if you want your models and services to be built in the application root folder, substituted the symbol + for the namespace.

BuildRoutines: Enter the list of database routines that you would like to build. Only routines entered will be built.

- **Schema**: The actual name of the schema in the database.
- **Namespace**: The desired namespace in the destination project.
  Note that if you want your models and services to be built in the application root folder, substituted the symbol + for the namespace.
- **Routine**: The name of the routine to build.

Note that build schemas and build routines can be used in any combination, so long as the same routine is not identified in both places. This provides the flexibility to create customized libraries dedicated to units of functionality.

EnumQueries: See the section on enumerations.

StaticQueries: See the section on static data.

Enum Queries

Enumerations will be created for all EnumQueries defined in the build definition file. This is done by building an Adhoc query from the entered values. Each enumeration query will generate a file and those files will be placed in the Enumerations folder. For instance:

```
"EnumQueries": [
    {
      "Name": "AccountStatuses",
      "Table": "dbo.AccountStatus",
      "NameColumn": "AccountStatusName",
      "ValueColumn": "AccountStatusId",
      "Filter": "Active = 1",
    }],
```

Will generate the following SQL:

```
SELECT AccountStatusName AS EnumName, AccountStatusId AS EnumValue FROM dbo.AccountStatus
WHERE Active = 1
```

And the following enumeration will be generated:

```
public enum AccountStatuses
{
    Active = 1,
    Pending = 2
    …
}
```

When enumerations are created in this manner, they can be assigned to input parameters using the enum tag and specifying the name of the of the enum query form the Enum Queries collection.

```
--+Enum=AccountStatuses
@AccountStatusId int
```

A word of caution, this is a build time process, and changes to underlying data are not automatically propagated to deployed code. The builder must be run, the project must be compiled, and any code in production would require a redeployment. This can be managed with continuous build continuous integration pipelines, so it is really up to your team to decide if the nature of the data is suitable to enumerations, and if the benefits outweigh the drawbacks.

Static Queries

Class definitions, as well as populated lists of static data will be created for each static data element in the build definition file. Each item will be represented as a partial class and place in the StaticData folder. For instance:

```
"StaticData": [
  {
    "Name": "AccountStatus",
    "Query": "Select * from dbo.AccountStatuses",
  }]
```

Would create a class that represents the rows from the query:

```
public class AccoutStatus
{
  public AccountStatus(type ColumnName, …)
  {
    this.ColumnName = ColumnName;
    …
  }
  public type ColumnName { get; set;}
  …

}
```

The rows from the result will then be utilized to create a static list as follows:

```
public List<AccountStatus> AccountStatusList = new List<AccountStatus>
{
  new AccountStatuses( “ColumnValue”,…),
  …

}
```

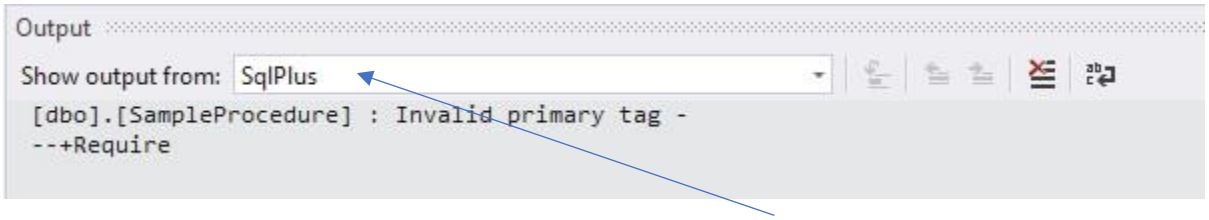
A word of caution, this is a build time process, and changes to underlying data are not automatically propagated to deployed code. The builder must be run, the project must be compiled, and any code in production would require a redeployment. This can be managed with continuous build continuous integration pipelines, so it is really up to your team to decide if the nature of the data is suitable to be represented as static data and if the benefits outweigh the drawbacks.

Building Your Project

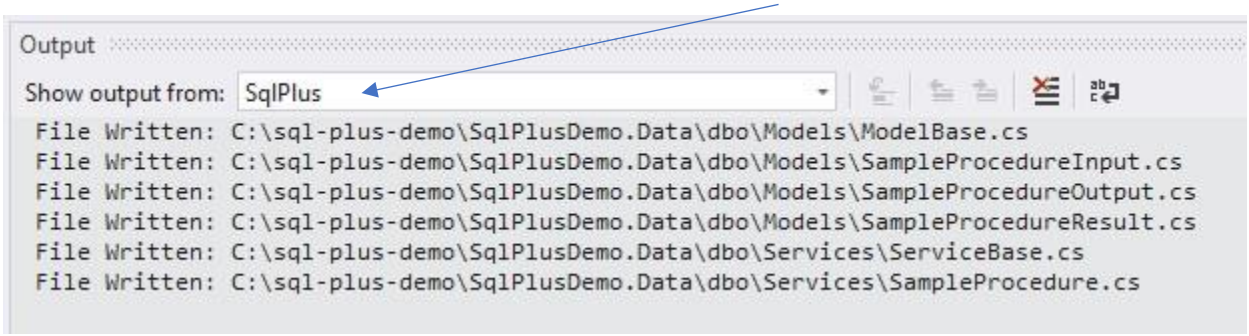
To run the builder, right-click on the project and choose the menu option SQL+.NET Build. The builder will use the information in the build definition file as well as collect all the concrete queries to get the full set of objects for the current build. During the build, a dialog will be displayed showing the progress.

Builder Results

The output window provides all the feedback from the build process. If any routine fails to parse, has an invalid select type, or contain invalid tags, errors will be reported along with the routine name where the error was found, otherwise, the list of files created and written to the project will be displayed. In the following case, the required tag was misspelled.



Note that SqlPlus is selected in the show output from drop down



Executing Your Code

The common pattern is to create an input to supply input parameters. That object is validated and passed into the service call, which returns an output object containing out parameters, result set data, and (optionally enumerated) return value. When a procedure does not require input parameters the service is called without an input object. Given the following procedure:

```
--+SqlPlusRoutine
    --&Author=Alan Hyneman
    --&Comment=Selects single row from dbo.Feedback table by identity column.
    --&SelectType=SingleRow
--+SqlPlusRoutine
ALTER PROCEDURE [dbo].[FeedbackById]
(
    --+Required
    --+Comment=FeedbackId
    @FeedbackId int
)
AS
BEGIN

    SET NOCOUNT ON;

    SELECT
        FeedbackId,
        LastName,
        FirstName,
        Email,
        Subject,
        Message,
        Created
    FROM
        dbo.Feedback
    WHERE
        FeedbackId = @FeedbackId;

    IF @@ROWCOUNT = 0
    BEGIN
        --+Return=NotFound
        RETURN 0;
    END;

    --+Return=Ok
    RETURN 1;

END;
```

The following code illustrates how the generated code would be called.

```
/// <summary>
/// Illustrating a valid call for select by id
/// Utilizing dbo.FeedbackById
/// </summary>
[Test]
[Order(7)]
0 references
public void ById()
{
    var input = new FeedbackByIdInput
    {
        FeedbackId = 1
    };
    Assert.IsTrue(input.IsValid());

    //Call the service to get the output object
    var output = ServiceFactory.Data().FeedbackById(input);

    //We can test the return value against the enumeration for Ok
    Assert.IsTrue(output.ReturnValue == FeedbackByIdOutput.Returns.Ok);

    //Check the result data aligns with the values from test 2
    Assert.IsTrue(output.ResultData.Email == "SomeoneElse@SQLPLUS.net");

    var feedback = output.ResultData;
    Console.WriteLine(feedback.Created);
    Console.WriteLine(feedback.Email);
    Console.WriteLine(feedback.FeedbackId);
    Console.WriteLine(feedback.FirstName);
    Console.WriteLine(feedback.LastName);
    Console.WriteLine(feedback.Message);
    Console.WriteLine(feedback.Subject);
}
```

Multiple Result Sets

Multiple result set services (services where the select type is MultiSet and the Query tags are utilized) behave in much the same way as every other service, however, the result object will have additional properties for each of the queries.

Given the following procedure:

```
--+SqlPlusRoutine
    --&SelectType=MultiSet
    --&Comment=Gets the order and order details for the given OrderId
    --&Author=Alan Hyneman
--+SqlPlusRoutine
ALTER PROCEDURE dbo.OrderById
(
    @OrderId int
)
AS
BEGIN

    SET NOCOUNT ON;

    --+QueryStart=Order,SingleRow
    SELECT
        OrderId,
        CustomerId
    FROM
        dbo.[Order]
    WHERE
        OrderId = @OrderId;
    --+QueryEnd

    IF @@ROWCOUNT = 0
    BEGIN
        --+Return=NotFound
        RETURN 0;
    END;

    --+QueryStart=OrderDetail,MultiRow
    SELECT
        OrderDetailId,
        ProductId
    FROM
        dbo.OrderDetail
    WHERE
        OrderId = @OrderId;
    --+QueryEnd

    --+Return=Ok
    RETURN 1;

END;
```

The service is called and results are presented as follows:

```
[TestMethod]
0 references
public void OrderHappy()
{
    //Create the input
    OrderByIdInput input = new OrderByIdInput
    {
        OrderId = 1
    };

    //Validate the input and write any errors
    if (!input.IsValid())
    {
        Utilities.WriteValidationErrors(input.ValidationResults);
        Assert.Fail();
    }
    else
    {
        //Call the service and get the output
        OrderByIdOutput output = Services.TestService.OrderById(input);

        //--+QueryStart=Order,SingleRow yeilds a result object
        OrderByIdResult.Order order = output.ResultData.OrderResult;

        //--+QueryStart=OrderDetail,MultiRow yeilds a list
        List<OrderByIdResult.OrderDetail> orderDetails = output.ResultData.OrderDetailResult;

        //Write out the order information
        Console.WriteLine($"Order - Id:{order.OrderId} " +
                          $"CustomerId:{order.CustomerId}");

        //Write out each of the order details information
        foreach(OrderByIdResult.OrderDetail orderDetail in orderDetails)
        {
            Console.WriteLine($"OrderDetail - Id:{orderDetail.OrderDetailId} " +
                              $"ProductId:{orderDetail.ProductId}");
        }
    }
}
```

Transactions

The following illustrates how to execute multiple calls in a single transaction using the generated services:

```
[TestMethod]
0 references
public void TransactionTest()
{
    //Create and open a connection object
    using(SqlConnection cnn = new SqlConnection(connectionString))
    {
        cnn.Open();

        //Use the open connection to create a transaction
        SqlTransaction transaction = cnn.BeginTransaction();

        //Create the service passing in the connection and the transaction
        Service service = new Service(cnn, transaction);
        try
        {
            //Exectute all methods - input validation removed for illustration
            service.SampleProcedure(new SampleProcedureInput { TestValue = 1 });
            service.SampleProcedure(new SampleProcedureInput { TestValue = 1 });

            //Commit the transaction
            transaction.Commit();
        }
        catch(Exception ex)
        {
            //In the event of an error rollback the transaction
            transaction.Rollback();
            Console.Write(ex.ToString());
            Assert.Fail();
        }
    }
}
```

Note that transactions cannot be used in tandem with retry options.

Transient Errors and Retry Options

If you're new to transient errors, in short, a transient error is an error that has an underlying cause that is self-resolving. For instance, a network-related error has nothing to do with your SQL, so retry logic can compensate for that situation. On the other hand, an index violation will continue to fail no matter how many times it's tried.

Transient errors and retries are managed by SQL+.NET, by providing retry options in the constructor of the service. You create a custom retry object by deriving from the abstract class retry options, and provide the list of error numbers you deem transient, express the intervals for retries, and supply an optional logging component:

```
//Transient Logger Implemeting ITransientLogger
1 reference
public class MyTransientLogger : ITransientLogger
{
    3 references
    public void Log(SqlException sqlException)
    {
        //Log the exception
    }
}

//Class Deriving from Retry Options
1 reference
public class MyRetryOptions : RetryOptions
{
    0 references
    public MyRetryOptions() :
        base(
            //Transient Error Numbers
            new List<int> { 2, 4060, 40197, 40501, 40613, 49918, 49919, 49920, 11001 },
            //Retry intervals
            new List<int> { 1000, 2000, 5000 },
            //Optional ITransientLogger
            new MyTransientLogger()
            )
    { }
}
```

To utilize the MyRetryOptions:

```
//Pass retry options to the constructor
Service service = new Service(connectionString, new MyRetryOptions());

//Call retries according to MyRetryOptions
service.SampleProcedure(input);
```

In practice, you will normally create several variations of retry options for specific purposes. UI might be very short and fewer retries, back end processes would be longer intervals and more retries.

More Information

Please visit https://www.sqlplus.net/Home/Learn *for videos on all the features of SQL+.NE and to stay up to date with new releases and new functionality!*

Support

Contact Form: https://www.sqlplus.net/Home/Contact

LinkedIn: https://www.linkedin.com/in/alanhynemandev/

Stack Overflow Tag: sql-plus-dot-net

Please be sure to go through the getting started series, it takes about an hour and will take you through a series of hands-on exercises designed to familiarize you with every feature of SQL+.NET.

Note that support is not included with community edition.

Conclusion

Fellow SQL Professionals,

SQL+.NET is a enterprise grade tool that features a SQL first ecosystem and is preferential to those developers who have invested their time to become fluent with SQL. For those individuals, SQL+.NET will leverage those skills and give you the ability to instantly convert stored procedures or SQL statements into high performance, robust data services. I hope you find the tools to be a useful addition to your toolset.

Your comments, suggestions, problems, etc., are all welcome and will help shape future versions of the product.

Sincerely,

Alan Hyneman
SQL+.NET Founder and CEO