SQL+.NET Documentation

Please visit https://www.sqlplus.net/Tutorials for videos on all the features of SQL+.NET!

SQL+.NET is a SQL first ecosystem. Write your SQL routines, add tags, generate services, done.

Semantic Tags

Everything starts with the tagging system. Available from the downloads page are code snippets for SQL Server Management Studio. These snippets making adding tags fool proof.

Routine Tags

The routine tag is the only tag that is required to generated code. It should be placed above the CREATE or ALTER PROCEDURE statement and follows the format:

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

```
*Note that the obsolete tag is optional*
```

SelectType: Choose one and only one of the following:

- **NonQuery** – the procedure does not execute a SELECT statement.
- **SingleRow** – the procedure executes a SELECT statement on a primary key, and will return a maximum of one row: SELECT WHERE PrimaryKey = @PrimaryKey.
- **MultiRow** – the procedure executes a SELECT statement on a non-primary key and will return an undetermined number of rows: SELECT WHERE LastName = @LastName.
- **JSON** – the procedure returns json: SELECT … FOR JSON PATH.

Comment: Comment for the generated method.

Author: The creator of the routine.

Obsolete: Choose one and only one of the following:

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

In addition to the error or warning options, a message must be included. This tags is used to inform other team members that a routine should no longer be used.

Parameter Validation Tags (Optional)

Parameter tags are used to add validation to input parameters. These tags are then consumed by the code generator, and validation becomes part of the service layer. Place one or more tags above the parameter declaration to achieve the desired validation. The following parameter tags are available:

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

Comment: The comment tag when applied to a parameter creates a comment for the property.

CreditCard: Indicates the value will be a credit card.

Currency: Indicates the value will be currency.

Default: The default value for a given property.

Email: Indicates the data will be an email address.

Enum: The name of the enumeration to substitute for the type.

Html: Indicates the data will be Html.

ImageUrl: Indicates the data will be an ImageUrl.

MaxLength: Maximum length for a string or array.

MinLength: Minimum length for a string or array.

Password: Indicates the value will be a password.

Phone: Indicates the value will be a phone number.

PostalCode: Indicates the value will be a postal code.

Range: Sets the minimum and maximum value allowed.

RegExPattern: Allows the use of a regular expression to validate the data.

Required: Indicates a required field.

StringLength: Indicates the length of the string must be between min and max values.

Upload: Indicates the field is utilized for uploads.

Url: Indicates format of the field is a URL.

Example:

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

```
--+Required
--+Email
--+Comment=Customers email to modify
@EmailAddress varchar (64)
```

And enforced as follows: the values is required, it must be a valid email, and a comment is presented to users of the service through intellisense.

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 tags can be utilized to control 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 required tag:

```
--+Required
--&ErrorMessage=Your custom error message
@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 resource file, 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 can make use of some or all of them:

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

Display: Gets or sets a value that is used for a label.

Prompt: Gets or sets a value that will be used to set the watermark for prompts in the UI.

ShortName: Gets or sets a value that is used for the grid column label.

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 name.

Example:

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

```
--+Display=User Name, not currently used.
@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
```

Return Tags

The return tag is used to enumerate return values. This tag provides the means to return meaningful information about the outcome of a given service call. Simply add the tag to each unique return value, and the return value in the service layer will be enumerated.

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

Example:

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

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

--+Return=Modified
RETURN 2;

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

Would create the following enumeration:

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

and the return value can then be compared using the enum:

```
if(output.ReturnValue == SampleProcedureOutput.Returns.Insert)
{
	//New Record distribution event to salesforce
}
```

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 and follow the on-screen prompts. Once installed the builder is run by right clicking on your project and choosing the SQL+.NET Build option from the menu.

Configuration

When the code generation utility runs, it inspects the contents of the SqlPlus.json file to determine what to build and from where. If the builder does not find this file, it will create it and include instructions for setting the values.

```
{
  "RenderUrl": "https://sqlplusrender.azurewebsites.net/api",
  "ConnectionString": "Server = SERVER; Initial Catalog = INITIAL_CATALOG; User ID =
USER; Password = PASSWORD;",
  "DatabaseType": "MSSQLServer",
  "BuildSchemas": [
    {
      "Schema": "SCHEMA",
      "Namespace": "NAMESPACE"
    },
    {
      "Schema": "SCHEMA",
      "Namespace": "NAMESPACE"
    }
  ],
  "BuildRoutines": [
    {
      "Schema": "SCHEMA",
      "Namespace": "NAMESPACE",
      "Routine": "ROUTINE"
    },
    {
      "Schema": "SCHEMA",
      "Namespace": "NAMESPACE",
      "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>"
    },
       {
      "Name": "<name for the enum>",
      "Table": "<source table of enum>",
      "NameColumn": "<column for the name>",
      "ValueColumn": "<column for the value>",
      "Filter": "<query filter>"
    },
 ],
  "Template": "DotNetFramework",
  "UserName": "USERNAME",
  "Password": "PASSWORD"
}
```

RenderUrl: This is the location that the builder communicates with, and the default value applies to most users https://sqlplusrender.azurewebsites.net/api . For self-hosted organizations, point this to your private location.

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

DatabaseType: Currently only MSSQLServer is supported.

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 in the destination project.

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.
- **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.

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 backwards compatible.

UserName: Your user name from https://www.sqlplus.net

Password: Your password from https://www.sqlplus.net

Enum Queries

Enumerations will be created for all EnumQueries defined in the SqlPlus.json file. This is done by building an adhoc query from the entered values. 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
{
    EnumName = EnumValue,
    EnumName = EnumValue
    …
}
```

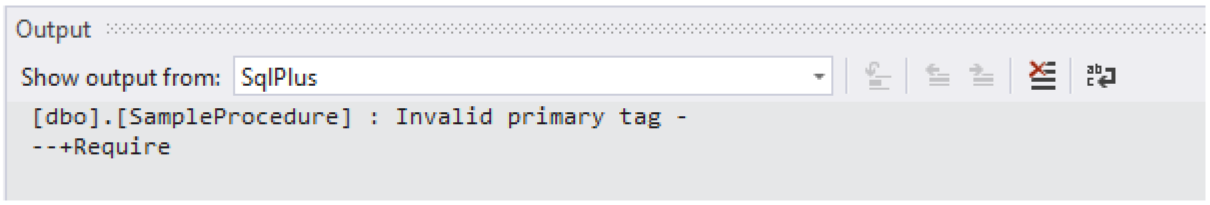
When enumerations are created in this manner, they can also be assigned to input parameters using the enum tag and specifying the name of the enum query.

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

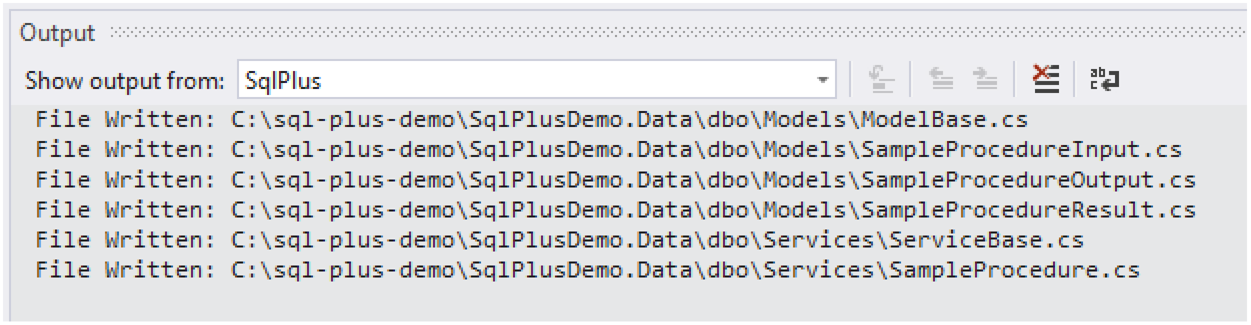
A word of caution, this is a build time process, and changes to underlying data are not automatically propagated to deployed code. The builder must be run, the project must be compiled, and any code in production would require a redeployment. This can be managed with continuous build continuous integration pipelines, so it is really up to your team to decide if the nature of the data is suitable to enumerating, and the benefits outweigh the drawbacks. When in doubt, go without.

Building Your Project

To run the builder, right click on the project and choose the menu option SQL+.NET Build. The output window provides all the feedback from the build process. If any routines contain invalid tags, errors will be reported along with the routine name where the invalid tag was found, otherwise, the list of files created and written to the project will be displayed.



```
[dbo].[SampleProcedure] : Invalid primary tag -
--+Require
```

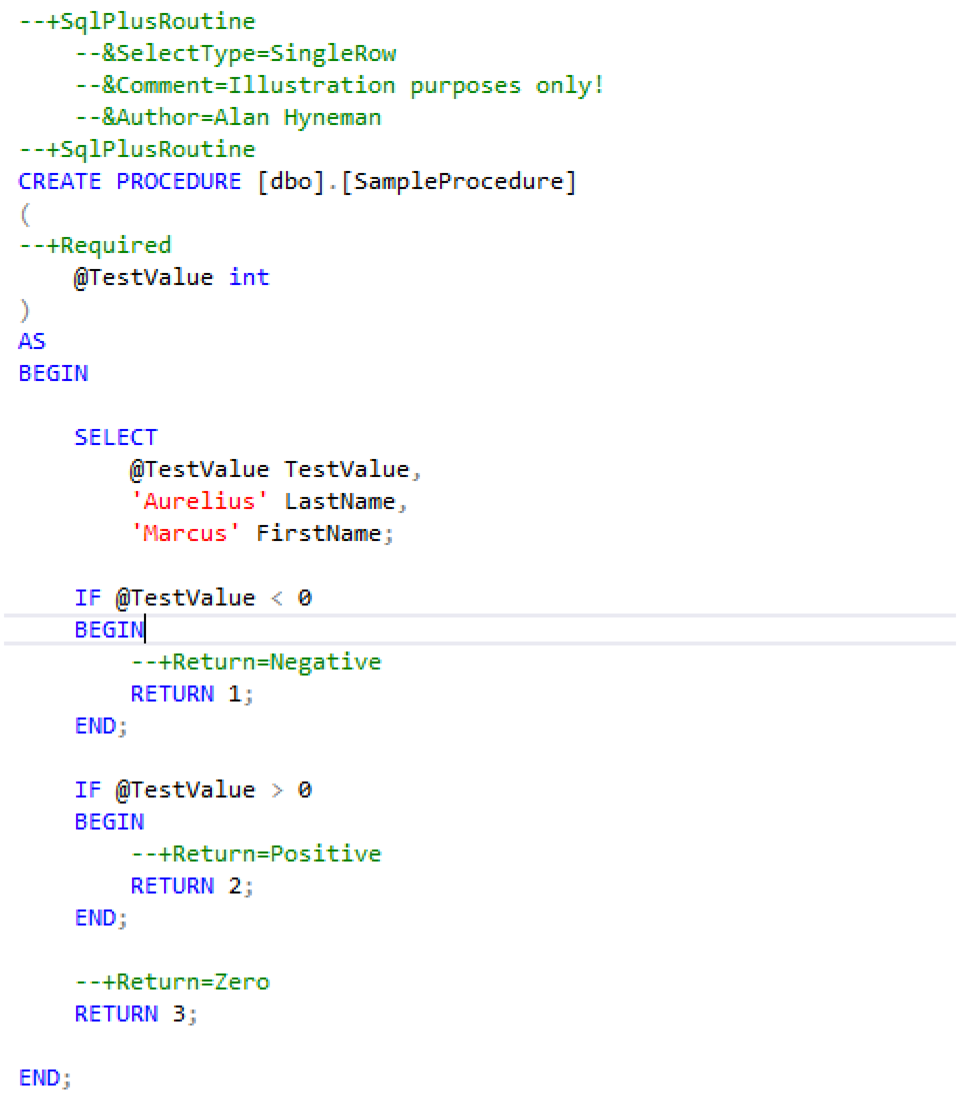


```
File Written: C:\sql-plus-demo\SqlPlusDemo.Data\dbo\Models\ModelBase.cs
File Written: C:\sql-plus-demo\SqlPlusDemo.Data\dbo\Models\SampleProcedureInput.cs
File Written: C:\sql-plus-demo\SqlPlusDemo.Data\dbo\Models\SampleProcedureOutput.cs
File Written: C:\sql-plus-demo\SqlPlusDemo.Data\dbo\Models\SampleProcedureResult.cs
File Written: C:\sql-plus-demo\SqlPlusDemo.Data\dbo\Services\ServiceBase.cs
File Written: C:\sql-plus-demo\SqlPlusDemo.Data\dbo\Services\SampleProcedure.cs
```

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

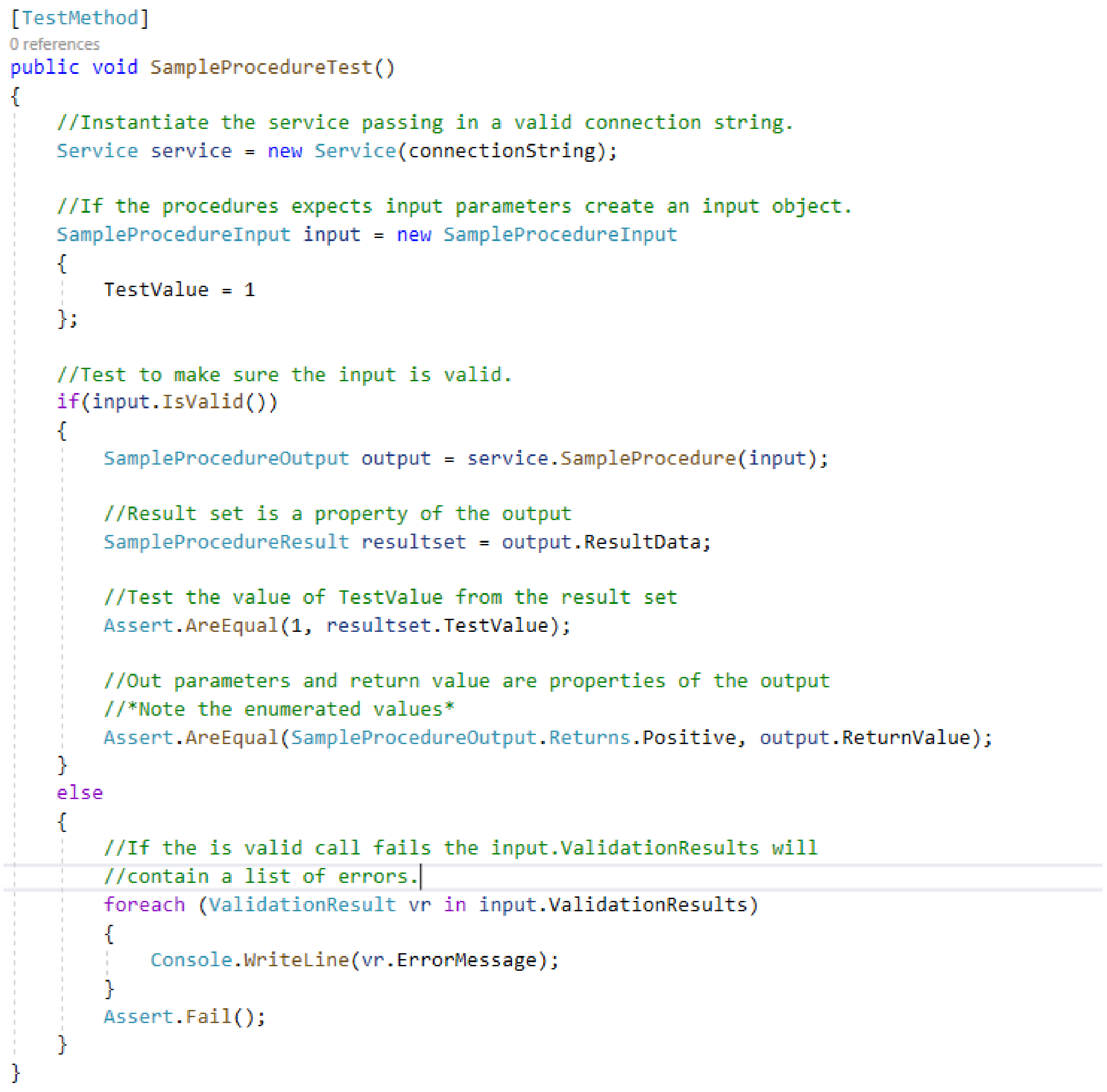
Executing Your Code

The common pattern is to create an input object to supply input parameters. That object is validated and passed into the service call, and an output containing out parameters, result data, and return value is returned. Given the following procedure:



```sql
--+SqlPlusRoutine
    --&SelectType=SingleRow
    --&Comment=Illustration purposes only!
    --&Author=Alan Hyneman
--+SqlPlusRoutine
CREATE PROCEDURE [dbo].[SampleProcedure]
(
--+Required
    @TestValue int
)
AS
BEGIN

    SELECT
        @TestValue TestValue,
        'Aurelius' LastName,
        'Marcus' FirstName;

    IF @TestValue < 0
    BEGIN
        --+Return=Negative
        RETURN 1;
    END;

    IF @TestValue > 0
    BEGIN
        --+Return=Positive
        RETURN 2;
    END;

    --+Return=Zero
    RETURN 3;

END;
```

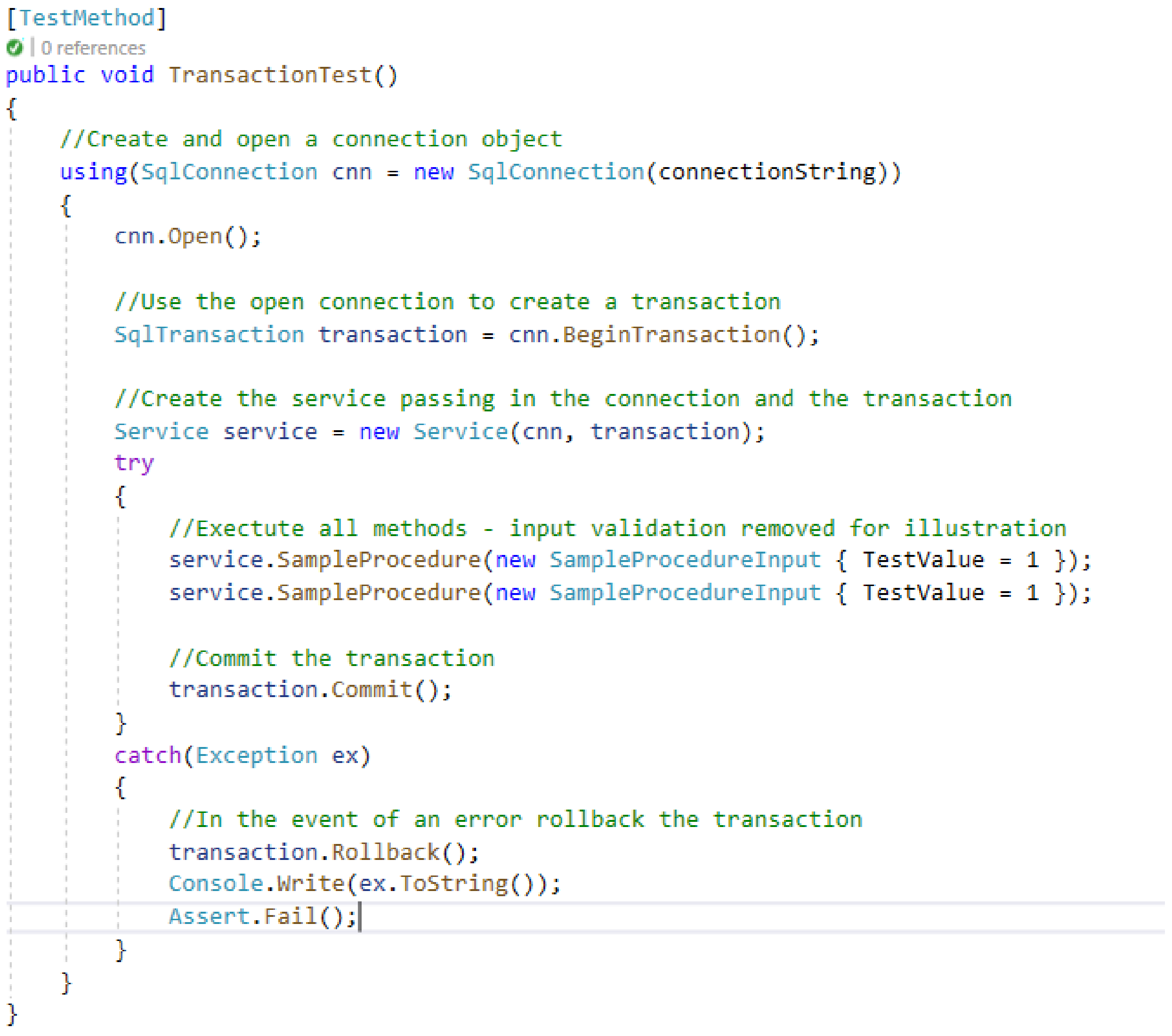
The following code illustrates how the generated code would be called.



```
[TestMethod]
0 references
public void SampleProcedureTest()
{
    //Instantiate the service passing in a valid connection string.
    Service service = new Service(connectionString);

    //If the procedures expects input parameters create an input object.
    SampleProcedureInput input = new SampleProcedureInput
    {
        TestValue = 1
    };

    //Test to make sure the input is valid.
    if(input.IsValid())
    {
        SampleProcedureOutput output = service.SampleProcedure(input);

        //Result set is a property of the output
        SampleProcedureResult resultset = output.ResultData;

        //Test the value of TestValue from the result set
        Assert.AreEqual(1, resultset.TestValue);

        //Out parameters and return value are properties of the output
        //*Note the enumerated values*
        Assert.AreEqual(SampleProcedureOutput.Returns.Positive, output.ReturnValue);
    }
    else
    {
        //If the is valid call fails the input.ValidationResults will
        //contain a list of errors.
        foreach (ValidationResult vr in input.ValidationResults)
        {
            Console.WriteLine(vr.ErrorMessage);
        }
        Assert.Fail();
    }
}
```

Transactions

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



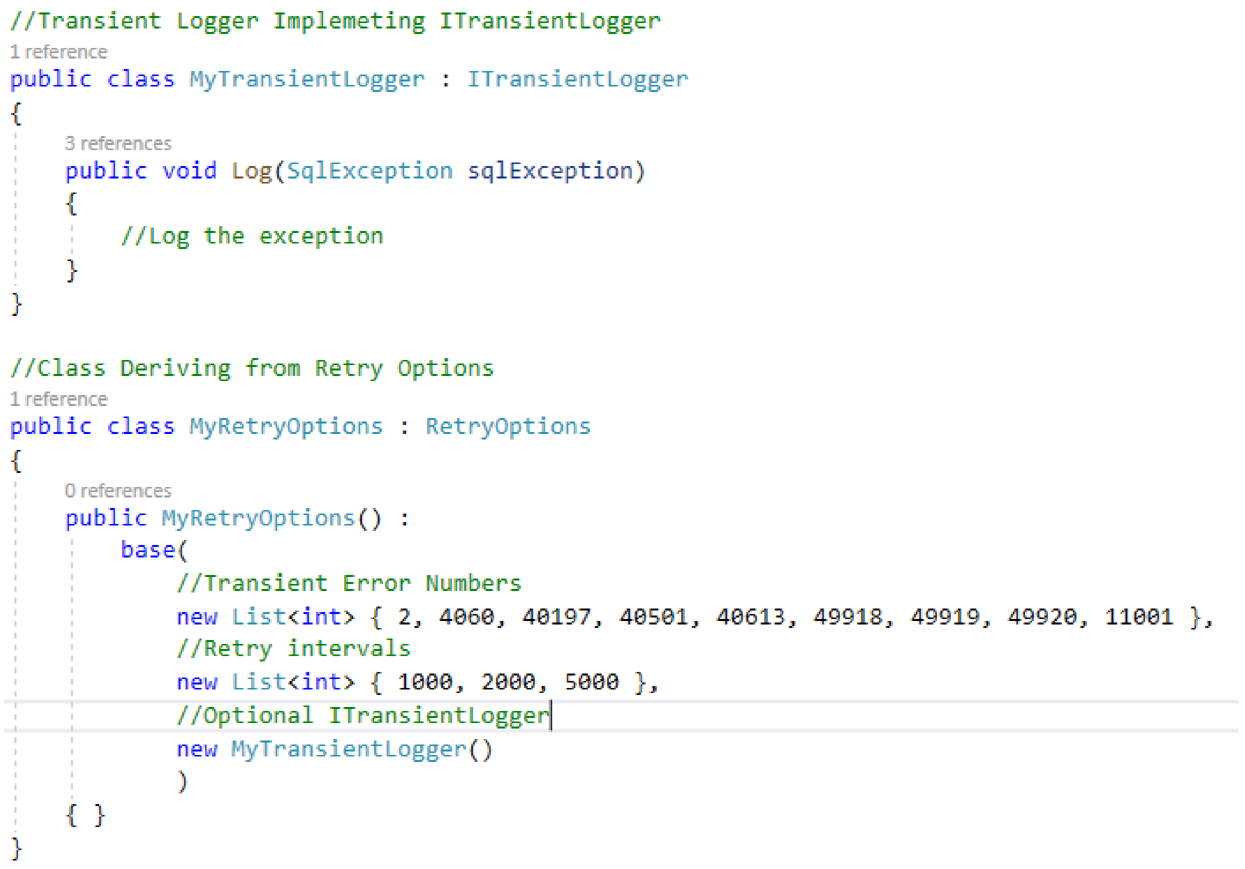
```
[TestMethod]
public void TransactionTest()
{
    //Create and open a connection object
    using(SqlConnection cnn = new SqlConnection(connectionString))
    {
        cnn.Open();

        //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

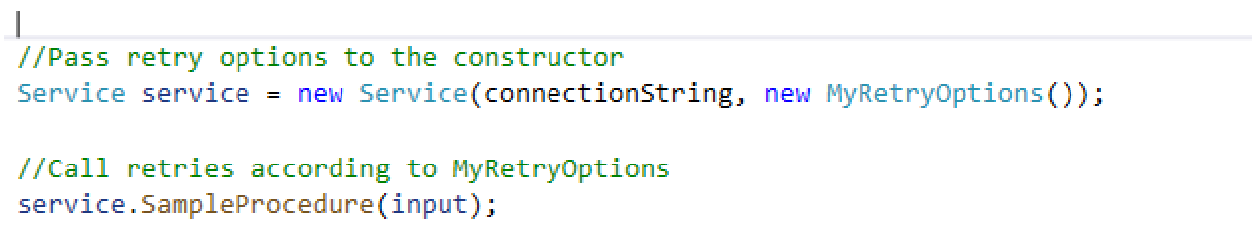
If your new to transient errors, in short, a transient error is an error that has an underlying cause that is self-resolving. For instance, a network related error has nothing today with your SQL, so retry logic can compensate for that situation. On the other hand, an index violation will continue to 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/Tutorials for videos on all the features of SQL+.NET!

Support

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

Email: Alan@SqlPlus.net

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

Conclusion

Fellow SQL Professionals,

My goal is to put data access development back in the hands of SQL professionals. Side by side, SQL+.NET out performs entity framework in every capacity, and we need your help to educate people to the fact. Your support is essential to the success of these tools, and is greatly appreciated.

Sincerely,

Alan Hyneman
SQL+.NET Founder and CEO