Nix Library Hybrid

This provides several helpers for flow processing.

Flow processing is geared towards handling a sequence of items one at a time, typically to conserve resources. Sources are typically presented as IEnumerables<T>, allowing Linq processing, and also the Flow Operators detailed below. Targets typically have an Add or AddAsync method.

In general, IEnumerables are sources,and final output will typically be to a list, file or database table. In between, Flow Methods can be used to filter, split, combine, expand and monitor the chain of elements being processed.

Sources are IEnumerables that are candidates for chained operations: Targets are used to persist results.

This generic library provides simplification wrappers for handling

- blobs
- files/folders
- Dates/Times
- strings (simple text, csv, arrays, property lists, dictionary, json)
- simple value parsing
- MS Sql
- Flows between sources and targets.

For example,

- a series of dates may be used to generate
- a number of folder names which are used to retrieve
- multiple blobs to
- local file storage which are then
- scanned for content
- filtered
- used to generate objects
- pushed to a data table

Chaining these operations allows the data to be processed iteratively with the minimum resource requirements.

BlobFlows: simplify listing, finding, opening or copying blobs

`BlobFlows` may be created with a full connection string - `UsaBlowFlows` and `EuropeBlobFlows` are self-initialized for standard Visier reports locations.

Sources

`IEnumerable<string>` **`Find`**`(string beginsWith, string contains = "", string endsWith = "")` returns a sequence of blob paths with optional filtering

Methods

`Task<BlobFlows.BlobInfo>` **`InfoAsync`**`(string path)` returns blob metadata

`Task<string>` **`ReadAsync`**`(string path)` returns blob content as a string

`Task<string>` **`PullAsync`**`(string blobPath, string filePath)` copies a blob to a local file

FileFlows: simplify listing files and folders, reading or writing lines

Sources

`static IEnumerable<string>` **`GetFileNames`**`(string path, bool recurse = true)` returns a list of file paths with or without recursion

`static IEnumerable<string>` **`GetFolderNames`**`(string path, bool recurse = true)` returns a list of folders with or without recursion

`static IEnumerable<string>` **`ReadLines`**`(string path)` returns an enumerable collection of lines

Targets

`FlowFile``<T>(path, append = true)` creates a file to which lines can be added using `Add(T item)`

`FlowList``<T>(ref List<T> list)` creates a list to which elements can be appended using `Add(T item)`

(`FlowFile` and `FlowList` are interchangable)

Methods

`static bool` **`EnsureFolder`**`(string path)` ensures that a folder exists (returns false if unable)

TimeFlows: simplify the generation of time strings or DateTimes

Sources

`static IEnumerable<string> **GetDates**(string earlier, string later, string format = "yyyy-MM-dd")` returns a sequence of formatted date strings

`static IEnumerable<string> **GetDatesReversed**(string earlier, string later, string format = "yyyy-MM-dd")` returns a sequence of formatted date strings

`static IEnumerable<string> **GetDates**(DateTime earlier, DateTime later, string format = "yyyy-MM-dd")` returns a sequence of formatted date strings

`static IEnumerable<string> **GetDatesReversed**(DateTime earlier, DateTime later, string format = "yyyy-MM-dd")` returns a sequence of formatted date strings

`static IEnumerable<DateTime> **GetDateSequence**(DateTime earlier, DateTime later)` returns a sequence of dates

`static IEnumerable<DateTime> **GetDateSequenceReversed**(DateTime earlier, DateTime later)` returns a sequence of dates

`static IEnumerable<DateTime> **GetTimeSequence**(DateTime earlier, DateTime later, TimeSpan delta)` returns a sequence of dates

`static IEnumerable<DateTime> **GetTimeSequenceReversed**(DateTime earlier, DateTime later, TimeSpan delta)` returns a sequence of dates

Methods

`static DateTime **Next**(this DateTime current, DayOfWeek dayOfWeek)` returns the date of the next specified day-of-the-week

`static DateTime **Previous**(this DateTime current, DayOfWeek dayOfWeek)` returns the date of the previous specified day-of-the-week

`static DateTime **ClearTime**(this DateTime dateTime)` reverts the time to midnight 00:00:00

`static DateTime **ClearDate**(this DateTime dateTime)` reverts the date portion to 0001-01-01

Text handler (Tx): simplify parsing and converting strings, string arrays and string-based property lists.

The text object can be used for a string, or an array of strings, or an array of strings with labels. It simplifies string parsing, and also allows conversion between those forms.

For example, delimited values can be separated using the '/' operator, then provided with a label array, and converted to and from from json or joined again using different delimiters using the "*" operator.

Creation:

string or string enumerable extenders:

- `static Tx t(this string s)` for a single Value
- `static Tx t(this IEnumerable<string> ss)` for one or more Values
- `static Tx t(this string s, Func<Tx, Tx> function)` makes a Tx from a `string` using the provided function to generate the Values
- `static Tx t(this IEnumerable<string> s, Func<Tx, Tx> function)` makes a `Tx` from an `IEnumerable<string>` using the provided function to generate the Values

constructors:

- **`Tx`**`(string value)`
- **`Tx`**`(Tx t)`
- **`Tx`**`(string value, string label)`
- **`Tx`**`(enumerable<string> values)`
- **`Tx`**`(IEnumerable<string> values, IEnumerable<string> labels)`
- **`Tx`**`(params string[] values)`
- **`Tx FromJson`**`()` converts a json Value to Labels and Values
- `Tx` **`FromInnerJsonObject`**`()` ditto, uses innermost json object
- `Tx` **`Clone`**`(Tx tx)` makes a copy

Properties:

- **`s`** shorthand to convert to string as Value[0]
- **`int Scale`** = the number of values

- `string` **`Value`**
- `string[]` **`Values`**
- `string[]` **`Labels`**
- `string` **`Delimiter`**

Indexing:

- `string this`**`[int index]`** (does not throw out of range)
- `string this`**`[string label]`** gets or sets the value associated with a label, inserting if necessary

Methods:

- `int` **`IndexOfLabel`**`(string label)` returns the index of a label (or -1)
- `Tx` **`NthMatch`**`(int count, string regex)` (count is 1-based, negative works from end)
- `string` **`AsJson`**`()` if labeled, returns a flat property list otherwise an array of values

Operators:

These mainly act on the Value and allow fluent definitions of string operations.

Adjust:

- **`+a`** Upper case initial
- **`-a`** Lower case initial
- **`!a`** Decrypt
- **`~a`** Encrypt
- **`a++`** To Upper
- **`a--`** To Lower
- `a` **`+`** `b` Concatenate
- `a` **`-`** `b` remove from end by string length
- `a` **`%`** `n` remove from start (remainder)

Search:

- `a` **`<`** `b` Keep up to match
- `a` **`<=`** `b` Keep up to match, including
- `a` **`>`** `b` Keep after match
- `a` **`>=`** `b` Keep after match, including
- `a` **`<<`** `b` Regex match from start
- `a` **`>>`** `b` Regex match from end

Operate on parts:

- `a` **`==`** `e` select out parts using a regex to find
- `a` **`!=`** `e` remove parts using regex
- `a` **`/`** `b` divide a into multiple parts by any character in (string) b
- `a` **`/`** `c` divide into parts using a regex to delimit on (char) c
- `a` **`*`** `s` recombine using the connecting string

IfParses: simplify generic parsing of values

Generic parser returns true or false, and also passes either the parsed value or the original object respectively to functions to handle the success and failure outcomes. Can parse enums too.

bool **IfParses**<T>(action<T> onSuccess, action<object>onFailure) attempts to parse the object on which it operates to the `T`. If successful, it executes the `onSuccess` action, passing the parsed result as a parameter; if unsuccessful, it passes the original object to the (optional) `onFailure` action. This returns `true` if `onSuccess` is invoked.

Unlike normal parsing, which requires an out parameter, either action can do anything with the parameter, including calling methods and setting variables within scope.

SqlClient: simplify object storage and retrieval

Constructor

SqlClient(string connection String, string IdentityFieldName = "Id") creates the disposable instance, preferably in a using statement. The Identity Field name is used to automate table creation for storing of objects by type (flat objects only)

Data definition methods

Task **CreateTableAsync**<T>(string tableName, int stringLength = 256) creates a table with nvarchar columns for the data and a big int (long) identity field (by default Id). This table would be suitable for use as a target.

Task **DropTableAsync**(string tableName) simply drops a table.

Async Query methods

Task<int> **InsertAsync**<T>(string tableName, T data) inserts a row into a table containing the matching properties and an Identity key, returning the Id of the inserted row

Task<int> **ExecuteSqlAsync**(string sql) executes a sql statement, typically returning the number of affected rows

Task<int> **InsertManyAsync**<T>(string tableName, T rows) where T : IEnumerable inserts multiple rows from an IEnumerable of the object of type T

Task<int> **ApplyWhereInAsync**(string sql, long[] Ids) executes sql "where in ()" using the list of ids

Task<int> **ResetTableAsync**(string tableName, long newBase = 1) truncates a table and resets the Identity counter

Task<long> **SpInsertAsync**(string spName, params (string name, object value)[] parameters) executes a stored procedure inserting an object and returning the identity

Task<long> **SpExecuteAsync**(string spName, params (string name, object value)[] parameters) executes a stored procedure typically returning the rows affected count

Task<T> **ReadSingleValueAsync**<T>(string sql) executes sql and returns a single value (scalar)

Sources

- `IEnumerable<T> ReadData<T>(string tableName, string whereOrderClause = "")` reads from a table row by row, returning a deserialized object of type T each time
- `IEnumerable<string> ReadRowsAsJson<T>(string tableName, string whereOrderClause = "")` reads from a table row by row, returning json
- `IEnumerable<ValueCount> GetValueCounts(string tableName, string columnName)` returns a `ValueCount` enumerable, each having a `Value` (the unique string) and `Count` (the number of references)
- `IEnumerable<KeyReferences> GetKeyReferences(string tableName, string keyName, string orderByPhrase = "")` returns a list of `KeyReferences`, each having a `Key` and an array of `References`. Typically this is used to collect one-to-many relationships grouped by the key
- `IEnumerable<T> QueryAsType<T>(string sql)` returns an enumerable set of deserialized objects using a sql query
- `IEnumerable<string> QueryAsJson(string sql, int lookAhead = 4096)` returns objects as Json. The look-ahead length might need to be increased in order to accommodate longer data lines
- `IEnumerable<string[]> ReadDataRaw(string sql)` returns data as an enumerable of string arrays
- `IEnumerable<string[]> ReadRawFromSp(string spName, params (string name, object value)[] parameters)` returns data as an enumerable of string arrays using a stored procedure
- `IEnumerable<T> SpAsType<T>(string spName, params (string name, object value)[] parameters)` returns deserialized objects using a stored procedure
- `IEnumerable<string> SpAsJson(string spName, params (string name, object value)[] parameters)` returns json using a stored procedure.

Targets

`Target<T> GetTarget<T>(string tableName)` makes a target into which output can be dumped using `target.AddAsync(T data)`

Flow Methods: simplify fluent processing

These are extenders on `IENumerable<T>`, and typically take and return `IEnumerable<T>` and support Linq methods.

- **FlowFrom**`<T>(this T singleObject)` allows starting from a single object instead of an enumerable.

Same-type flows:

- **FlowAll**`<T>(this IEnumerable<T> source, params IEnumerable<T>[] inputs)` draws from multiple sources sequentially
- **FlowMesh**`<T>(this IEnumerable<T> source, params IEnumerable<T>[] inputs)` draws from multiple sources in rotation
- **FlowMerge**`<T>(this IEnumerable<T> source, IEnumerable<T> secondarySource, Predicate<(T ChoiceA, T ChoiceB)> test, Func<T, T> function)` draws from 2 sources with a test to determine the sequence
- **FlowDone**`<T>(this IEnumerable<T> source, Action<int> onDone)` calls a final method with the final count
- **FlowResetIf**`<T>(this IEnumerable<T> source, Predicate<(T Element, int Index)> test)` conditionally restarts the flow
- **FlowStopIf**`<T>(this IEnumerable<T> source, Predicate<(T Element, int Index)> test)` conditionally stops the flow
- **FlowAct**`<T>(this IEnumerable<T> source, Action<(T Element, int Index)> action)` does a read-only action without changing the flow
- **FlowActOn**`<T>(this IEnumerable<T> source, Predicate<(T Element, int Index)> test, Action<(T Element, int Index)> onTrue, Action<(T Element, int Index)> onFalse)` conditionally does a read-only action without changing the flow
- **FlowIf**`<T>(this IEnumerable<T> source, Predicate<(T Element, int Index)> test)` conditionally skips or passes elements

Optionally type-changing flows:

- **FlowWith**`<T, T1, T2>(this IEnumerable<T> source, IEnumerable<T1> secondarySource, Func<T, T1, T2> function)` draws from 2 sources at the same time and passes both to a function
- **FlowThrough**`<T, T2>(this IEnumerable<T> source, Func<(T Element, int Index), IEnumerable<T2>> nestedFunction)` passes each element to a function that itself returns an IEnumerable of some type
- **FlowInitial**`<T, T2>(this IEnumerable<T> source, Func<(T Element, int Index), T2> onInitial, Func<(T Element, int Index), T2> onRest)` calls different functions for the first and subsequent elements
- **FlowDo**`<T, T2>(this IEnumerable<T> source, Func<(T Element, int Index), T2> function)` Does a function
- **FlowDoOn**`<T, T2>(this IEnumerable<T> source, Predicate<(T Element, int Index)> test, Func<(T Element, int Index), T2> onTrue, Func<(T Element, int Index), T2> onFalse)` conditionally does one or another function

May be used to start the flow if no enumerating function is called:

`int` **Flow**`()` at the end of a fluent sequence, starts iterating and returns the final count.