Scope Tutorial > Extending Scope

Author: Saveen Reddy

10/25/2014

# Table of Contents

[Introduction 1](#_Toc402027166)

[Extending Scope with C# 1](#_Toc402027167)

[Custom Extractors 2](#_Toc402027168)

[Custom Outputters 2](#_Toc402027169)

[Important 3](#_Toc402027170)

[User-Defined Functions 3](#_Toc402027171)

[User-Defined Types (UDTs) 3](#_Toc402027172)

[User-Defined Operators 5](#_Toc402027173)

[User-Defined Aggregates 5](#_Toc402027174)

[Processors 7](#_Toc402027175)

[Combiners 10](#_Toc402027176)

[Debug Streams 11](#_Toc402027177)

[Debugging UDO Code Locally 13](#_Toc402027178)

[Additional Resources 13](#_Toc402027179)

[Changes 13](#_Toc402027180)

# Introduction

This document is one part of the Scope Tutorial (<http://aka.ms/ScopeTutorial>). This document focused on extending Scope with C# code.

# Extending Scope with C#

* You can create
  + User-Defined Functions
  + User-Defined Types
  + User-Defined Operators
    - Aggregates
    - Extractors
    - Outputters
    - Processors
    - Reducers
    - Combiners

# Custom Extractors

You can build your own extractors with C#.

Below is a sample extractor that performs a simple extraction for TSV text files (for illustrative purposes only, in real life you would just use **DefaultTextExtractor**)

public class **MyTsvExtractor** : **Extractor**

{

public override Schema **Produces**(string[] requested\_columns, string[] args)

{

return new Schema(requested\_columns);

}

public override IEnumerable<Row> **Extract**(StreamReader reader, Row output\_row, string[] args)

{

char delimiter = '\t';

string line;

while ((line = reader.ReadLine()) != null)

{

var tokens = line.Split(delimiter);

for (int i = 0; i < tokens.Length; ++i)

{

output\_row[i].UnsafeSet(tokens[i]);

}

yield return output\_row;

}

}

}

Below you can see how we use the extractor.

searchlog =

EXTRACT IId:int, UId:int, Start:DateTime, Market:string, Query:string, DwellTime:int, Results:string, ClickedUrls:string

FROM @In\_SearchLog

**USING MyTsvExtractor()**;

PROTIP: Just because you can doesn’t mean you should. Always check whether the functionality you need is already provided by Scope or a shared library before rolling out your own.

# Custom Outputters

You can create your own outputters. A simple example that outputs in CSV format is shown below.

public class MyCsvOutputter: Outputter

{

public override void Output(RowSet input, StreamWriter writer, string[] args)

{

foreach (Row row in input.Rows)

{

int c = 0;

for (int i = 0; i<row.Count; i++)

{

if (c > 0)

{

writer.Write(",");

}

row[i].Serialize(writer);

c++;

}

writer.WriteLine();

writer.Flush();

}

}

}

To use this outputter:

OUTPUT searchlog TO @Out\_SearchLog USING MyCsvOutputter();

# Important

This tutorial shows you how you \*can\* write custom objects, but that doesn’t mean you \*should\*! In particular, you should try to avoid writing your own Aggregates, Extractors, Combiners, etc. – inbuilt Scope constructs and UDOs are very expressive while shared libraries exist for common scenarios not natively handled by Scope.

User-defined functions and user-defined types are relatively common and safe.

# User-Defined Functions

This was covered earlier in the document.

# User-Defined Types (UDTs)

Besides the normal .NET types, Scope also supports the creation of custom types for columns. These are very simple to create.

One key limitation: Scope output will only write out the natively-supported Scope types and does not support writing user-defined types to streams.

## Walkthrough

First, let’s create a simple type. In this case, assume it’s important for us to parse out the ClickedUrls field into a list. To illustrate a UDT we’ll create a wrapper around the .NET **List<T>** class.

public class **UrlList**

{

public List<string> **Items**;

private static char [] sepchars = new char[] {','};

public **UrlList**(string s)

{

this.Items = new List<string>(s.Split(sepchars));

}

}

PROTIP: You should override **Equals()** and **GetHashCode()** in any UDT you write. Here is some guidance on approaches you can use: <http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an-overridden-system-object-gethashcode>

First, we do our normal **EXTRACT:**

searchlog =

EXTRACT IId:int, UId:int, Start:DateTime, Market:string, Query:string, DwellTime:int, Results:string, **ClickedUrls:string**

FROM @"SampleInputs\SearchLog.txt"

USING DefaultTextExtractor();

Now we create another rowset that creates a column filled with **UrlList** objects:

searchlog2 =

SELECT IId, UId, Start, Market, Query, DwellTime, Results, **new UrlList (ClickedUrls) AS CLickedUrlsList**

FROM searchlog;

Now, to prove that this script really uses this new object, let’s just find the number of **ClickedUrls**:

searchlog3 =

SELECT IId, UId, Start, Market, Query, DwellTime, Results, **CLickedUrlsList.Items.Count AS ClickedUrlsCount**

FROM searchlog2;

## UDT Serialization and Deserialization

To convert the original string into our UDT, the UrlList constructor served the role of a deserializer, converting a string into our UDT.

If you want to **OUTPUT** that UDT to stream, you will need to provide your own serializer, converting it to something that Scope can output natively. Typically this is done simply by adding a **Serialize()** method to the UDT.

As shown below, the **Serialize()** method does the opposite job of the constructor:

public class UrlList

{

public List<string> Items;

private static char [] sepchars = new char[] {','};

public UrlList(string s)

{

this.Items = new List<string>(s.Split(sepchars));

}

public static UrlList Create(string s)

{

return new UrlList(s);

}

**public string Serialize()**

**{**

**return string.Join(";", this.Items);**

**}**

}

Now we can output the UDT to a stream.

searchlog =

EXTRACT IId:int, UId:int, Start:DateTime, Market:string, Query:string, DwellTime:int, Results:string, **ClickedUrls:string**

FROM @"SampleInputs\SearchLog.txt"

USING DefaultTextExtractor();

searchlog2 =

SELECT IId, UId, Start, Market, Query, DwellTime, Results, **new UrlList (ClickedUrls) AS CLickedUrlsList**

FROM searchlog;

searchlog3 =

SELECT IId, UId, Start, Market, Query, DwellTime, Results, **CLickedUrlsList.Serialize() AS ClickedUrls**

FROM searchlog2;

OUTPUT searchlog3 TO @"D:\output-searchlog.txt";

# User-Defined Operators

User-Defined Operators UDOs are the most powerful way of extending scope.

Using C# code you can create:

* Extractors
* Outputters
* Processors
* Reducers
* Combiners
* Aggregators

# User-Defined Aggregates

User-defined aggregates allow you to create your own aggregates operators in C#. Let’s start by implementing the **SUM** aggregation operator. We’ll call our version **MySum**.

We will use it just like the normal **SUM** aggregate:

SELECT Market, **MySum(Duration)** AS TotalDuration

FROM searchlog

GROUP BY Market;

The code to create **MySum** is shown below:

class **MySum\_Double** : **Aggregate1<double, double>**

{

double sum = 0;

public override void **Initialize**()

{

sum = 0;

}

public override void **Add**(double y)

{

sum += y;

}

public override double **Finalize**()

{

return sum;

}

}

Key Points:

* The class is called **MySum\_Double**, not **MySum**. This is significant. Scope requires the name to specify which type the aggregator will eventually yield.
* **MySum\_Double** inherits from **Aggregate1<double, double>**. The first double indicates what input to aggregate, the second the type of the final aggregate.

## Aggregators That Take Multiple Inputs

The previous aggregator always added up one number. But sometimes you’ll need to feed in multiple arguments to the aggregation. The skeleton code below shows what this would look like. It accepts an integer and double and aggregates them into a string.

class MyAgg2\_IntegerDouble : Aggregate2<int, double, string>

{

System.Text.StringBuilder sb;

public override void Initialize()

{

sb = new System.Text.StringBuilder();

}

public override void Add(int count, double value)

{

for (int i=0; i<count; i++)

{

sb.Append(value.ToString());

}

}

public override string Finalize()

{

return sb.ToString();

}

}

# Processors

Processors are a way of extending Scope with C#. A processor allows you to programmatically transform a rowset. Processors can modify the values of a rowset, add columns, remove columns, remove rows, and create new rows. There’s a lot of power in processors.

Before we get to some of their applications we will start get familiar with the mechanics of creating and using one.

## A Processor that Copies a Rowset

We’ll start off with the simplest processor. This one does nothing to the rowset; every row of input is sent back as output.

Below is how the script calls a processor called **CopyProcessor**. The rowset called rs2 will be equivalent to rs1.

rs1 = SELECT Market, Results

FROM searchlog;

rs2 = **PROCESS** rs1

**PRODUCE** Market, Results

**USING** CopyProcessor;

Now let’s look at the code for **CopyProcessor**.

public class **CopyProcessor** : Processor

{

public override Schema **Produces**(string[] requested\_columns, string[] args, Schema input\_schema)

{

var output\_schema = input\_schema.CloneWithSource();

return output\_schema;

}

public override IEnumerable<Row> **Process**(RowSet input\_rowset, Row output\_row, string[] args)

{

foreach (Row input\_row in input\_rowset.Rows)

{

input\_row.CopyTo(output\_row);

yield return output\_row;

}

}

}

Key points:

* A processor inherits from **ScopeRuntime.Processor.**
* A processor implements two methods: **Produces()** and **Process().**
* **Produce()** defines the schema of the output.
* **Process()** generates the output based on the input.

In this example, **Produces()** simply duplicates the input schema.

Notice that **Process()** simply duplicates each row.

## A Processor that Modifies a Column

The first processor we built uses the **CopyTo()** method to create the output row. In the example below we will modify an existing value. For example, let’s prepend “FOO” to the **Market** field. Notice that in this case, the processor has knowledge of the schema of the input rowset.

rs1 = SELECT Market, Results

FROM searchlog;

rs2 = PROCESS rs1

PRODUCE Market, Results

USING MyProcessor;

public class **MyProcessor** : Processor

{

public override Schema Produces(string[] requested\_columns, string[] args, Schema input\_schema)

{

var output\_schema = input\_schema.Clone();

return output\_schema;

}

public override IEnumerable<Row> Process(RowSet input\_rowset, Row output\_row, string[] args)

{

foreach (Row input\_row in input\_rowset.Rows)

{

input\_row.CopyTo(output\_row);

**string market = input\_row[0].String;**

**output\_row[0].Set( "FOO" + market );**

yield return output\_row;

}

}

}

## A Processor that Modifies a Schema

In the previous, example we modified the **Market** column. Now, instead of modifying it, we will create a new column and put the new value there.

First notice that **PRODUCE** clause identifies the new field:

rs1 = SELECT Market, Results

FROM searchlog;

rs2 = PROCESS rs1

**PRODUCE** Market, Results, **Market2**

USING MyProcessor;

The code for the processor now looks like this:

public class MyProcessor : Processor

{

public override Schema Produces(string[] requested\_columns, string[] args, Schema input\_schema)

{

var output\_schema = input\_schema.Clone();

**var newcol = new ColumnInfo("Market2", typeof(string));**

**output\_schema.Add(newcol);**

return output\_schema;

}

public override IEnumerable<Row> Process(RowSet input\_rowset, Row output\_row, string[] args)

{

foreach (Row input\_row in input\_rowset.Rows)

{

input\_row.CopyTo(output\_row);

**string market = input\_row[0].String;**

**output\_row[2].Set( "FOO" + market );**

yield return output\_row;

}

}

}

## A Processor that Generates Rows

The examples so far have a 1-to-1 mapping between input rows and output rows. Processers can easily return more rows. For example, we are aware that the Results field is string that contains a comma-separated list or URLs. What we want is to “break apart” a row into multiple rows so that each url is on a separate row.

For example, for this input:

|  |  |
| --- | --- |
| Market | Results |
| en-us | A;B;C |
| en-gb | D;E;F |

We want this as output:

|  |  |
| --- | --- |
| Market | Results |
| en-us | A |
| en-us | B |
| en-us | C |
| en-gb | D |
| en-gb | E |
| en-gb | F |

NOTE: It is preferred to do a transformation like the above with the **CROSS APPLY** operator. See the **CROSS APPLY** section of this document for an example.

rs1 = SELECT Market, Results

FROM searchlog;

rs2 = PROCESS rs1

**PRODUCE** Market, Results

USING MyProcessor;

public class MyProcessor : Processor

{

public override Schema Produces(string[] requested\_columns, string[] args, Schema input\_schema)

{

var output\_schema = input\_schema.Clone();

return output\_schema;

}

public override IEnumerable<Row> Process(RowSet input\_rowset, Row output\_row, string[] args)

{

**var splitchars = new [] {';'};**

foreach (Row input\_row in input\_rowset.Rows)

{

**input\_row.CopyTo(output\_row);**

**string results = input\_row[1].String;**

**var urls = results.Split(splitchars);**

**foreach (string url in urls)**

**{**

**output\_row[1].Set(url);**

**yield return output\_row;**

**}**

}

}

}

Notice now that the **yield return** is in a loop. For each input row it will yield multiple times.

# Combiners

Combiners are an advanced topic, they are included here only for completeness. Although there could be valid reasons for creating a combiner, we suggest you always seek alternatives that are native Scope constructs.

Combiners take two rowsets as input and *combine* them into one output based on some matching criteria. Below is an example of the **COMBINE** clause used with a custom combiner called MyCrossJoinCombiner.

employees =

EXTRACT EmpName:string, DepID:string

FROM @"SampleInputs\Employees.txt"

USING DefaultTextExtractor();

departments =

EXTRACT DepID:string, DepName:string

FROM @"SampleInputs\departments.txt"

USING DefaultTextExtractor();

rs0 = **COMBINE** employees **WITH** departments

**ON** employees.DepID == departments.DepID

**USING** **MyCrossJoinCombiner** ();

OUTPUT rs0 TO "output.txt";

Now let’s look at what MyCrossJoinCombiner does. It is the simplest combiner possible: it merges the two rowset schemas and yields every combination of rows it receives.

public class **MyCrossJoinCombiner**: **Combiner**

{

public override Schema **Produces**(

string[] requestedColumns, string[] args,

Schema leftSchema, string leftTable,

Schema rightSchema, string rightTable)

{

var tokens = new List<string>();

foreach (var col in leftSchema.Columns)

{

string prefix = rightSchema.Contains(col.Name) ? leftTable : "";

tokens.Add(string.Format("{0}{1}:{2}", prefix, col.Name, col.CLRType));

}

foreach (var col in rightSchema.Columns)

{

string prefix = leftSchema.Contains(col.Name) ? rightTable : "";

tokens.Add(string.Format("{0}{1}:{2}", prefix, col.Name, col.CLRType));

}

var schemastring = string.Join(",", tokens);

return new Schema(schemastring);

}

public override IEnumerable<Row> **Combine**(RowSet left, RowSet right, Row outputRow, string[] args)

{

var \_rowList = new RowList();

\_rowList.Load(right); // Load the right RowSet into memory

foreach (Row leftRow in left.Rows)

{

leftRow.CopyTo(outputRow); // Copy the data from the leftRow to the output

// Copy the data from the rightRow to the output

foreach (Row rightRow in \_rowList.Rows)

{

for (int i = 0; i < rightRow.Count; ++i)

{

rightRow[i].CopyTo(outputRow[i + leftRow.Count]);

}

yield return outputRow;

}

}

}

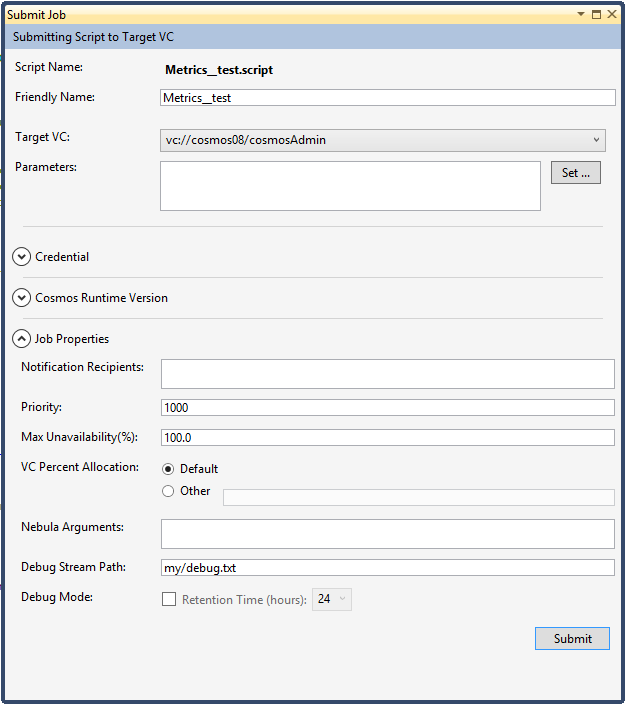
}

# Debug Streams

You can enable the Debug Streams feature that will enable you to use a a special **DebugStream.WriteLine()** method as shown below:

ScopeRuntime.Diagnostics.DebugStream.WriteLine("Hello World");

To turn on the Debug Stream feature set the **Debug Stream Path** value when you submit a job.



Below is a more complete example for an extractor that logs any lines it has trouble with.

Using ScopeRuntime.Diagnostics;

public class MyTsvExtractor : Extractor

{

public override Schema Produces(string[] requested\_columns, string[] args)

{

return new Schema(requested\_columns);

}

public override IEnumerable<Row> Extract(StreamReader reader, Row output\_row, string[] args)

{

while ((line = reader.ReadLine()) != null)

{

…

try { … }

catch (SomeException exc)

{

ScopeRuntime.Diagnostics.DebugStream.WriteLine(exc.Message);

}

…

yield return output\_row;

}

}

}

To examine the debug stream output, we’ve built a built-in extractor for you:

Debuglines = EXTRACT \*

FROM "my/debugstream.txt"

USING DefaultDebugStreamExtractor();

There is much more to the feature, so please read the documentation: <https://microsoft.sharepoint.com/teams/Cosmos/_layouts/15/WopiFrame.aspx?sourcedoc=/teams/Cosmos/Documents/Scope/Scope-User-Code-Debug-Streams.docx>

# Debugging UDO Code Locally

If you develop a Scope class such as an Extractor, a Reducer or a User Defined Function, you can debug it faster by running the query under the Visual Studio (VS) debugger.

There, you can set breakpoints in your code, examine variables at runtime, and generally find more bugs than you would by simply running the query locally.

## In Scope Studio

To do this in Scope Studio, you can set break points (F9) in a script, C# behind file, and reference C# project to do step-by-step debugging (F10), as well as step into the methods called in the statement(F11). Also, you can add a RowSet name to the watch window and preview the content with visualizer.

If you have trouble using F11 to step into methods, please make sure that "Enable Just My Code" has been selected in Visual Studio>Debug >Options and Settings.

If the output preview window is not opened automatically, you can click the Local Stream Preview button under the Scope menu to open it.

# Changes

* 2014-06-19 – Cleanup
* 2014-06-03 – Proofread and fixed errors
* 2013-12-15 – Added a note about GetHashCode(), Enhanced the section on debug streams, and cleaned up formatting
* 2012-04-21 – Clean up
* 2012-04-02 – Mentioned Debug Streams
* 2012-10-15 – Split from Main Scope Tutorial