QlikView Expressor Extension SDK

QlikView Expressor 3.9

CONTENTS

Accompanying this tutorial document are two ZIP files.

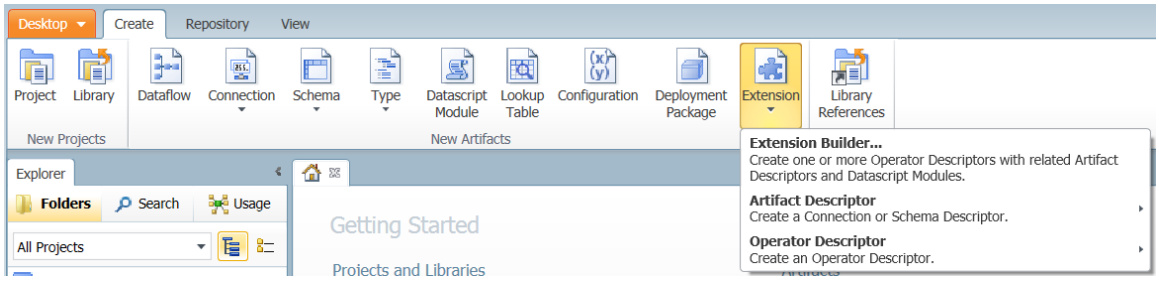
The archive extension_tutorial.zip is an export from Expressor Desktop that includes libraries implementing each of the examples discussed in this document.

The archive data.zip contains data files used when running the examples. You will need to create directories to hold these data files. Take note of the directory names in the tutorial document and create the directories as required.

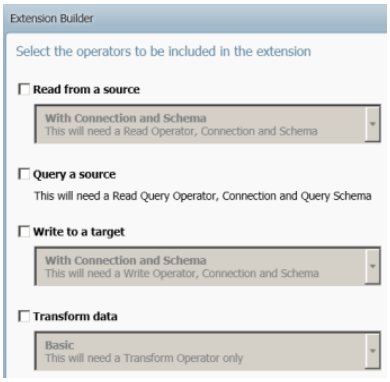
INTRODUCTION

The QlikView Expressor Software Development Kit (SDK) gives skilled Expressor developers the tools needed to create additional operators that can then be distributed to other developers and used in developing Expressor dataflows. Three types of extension operators may be developed: input, output, and transform. As most Expressor dataflows in a QlikView environment will write output to QVX files or database tables, you will find the ability to create new input and transform operators more useful than the ability to create output operators. Consequently, this document will include more information on developing input and transform operators than developing output operators.

There is no special licensing required to use the SDK. All QlikView Expressor licenses enable this functionality by setting the QEX_EXT_DEVELOPER entry to YES. However, the Extension SDK User Interface is not enabled by default. To enable this feature within a Workspace, click on the  icon in the upper right-hand corner of Expressor Desktop and select **Extension SDK User Interface…** from the drop down menu. When the Extension SDK User Interface is enabled, an **Extension** button appears on the **Create** tab of the Desktop ribbon bar.



Once you enable this functionality, libraries will include two additional subdirectories, **Artifact Descriptors** and **Operator Descriptors**, as extensions can only be created within a library (not in a project). As you can see from the screen shot above, you may create individual descriptors or you can use the Extension Builder wizard to generate the set of artifacts needed to develop an extension operator. This document will emphasize use of the Extension Builder as it simplifies the process by ensuring that all necessary artifacts are created and by establishing the required cross references between these artifacts.



As you can see from the screen shot at the left, a single extension can include more than one operator, just as the Expressor QlikView, Expressor Excel and Expressor SalesForce extensions include both input and output operators. Additionally extension operators may be of multiple types. For example, input and output extension operators may be designed to use a connection and schema, or simply a connection, or neither a connection nor a schema, and a transform extension operator may be designed around a connection or without the need for a connection.

This document will first discuss development of input extension operators, specifically an operator that can read one or more delimited files from a single file system directory location, with an option to use wildcard characters in specifying the file names. Moreover, the operator will process files that use any field delimiter character, e.g., comma, vertical bar, semicolon, or colon.

Additional input operators to read fixed width files and retrieve a file from an FTP server will be developed. These operators will provide functionality that does not currently exist in Expressor and therefore represent meaningful additions to the product.

As a final example, an extension transform operator will be developed that provides an easier approach to using the Expressor utility.custom_encrypt and utility.custom_decrypt functions.

READ CUSTOM OPERATOR

The input extension operators are derived from the Read Custom operator and it is possible to use this operator to read multiple files from a single directory. A simple implementation of this approach is shown in the following screen shot and in the ReadCustom project included in the extension_tutorial.zip archive file. If you want to examine the code in greater detail or run the dataflow, import the project into Expressor Desktop 3.9; be certain to create a directory to hold the input files, which are in the file data.zip.

Read Files in Directory

```
1  require "expressor.ScriptSupport"
2
3  file_handle = nil
4  header = {"place","lastname","firstname","party"}
5
6  function initialize()
7    -- read multiple files from same file system location
8    file_handle = io.popen(
9      "type C:\\data\\republicans.dat C:\\data\\democrats.dat C:\\data\\whigs.dat")
10 end
11
12 function read()
13   -- retrieve a line
14   line = file_handle:read("*l")
15   if line then
16     output = {}
17     values = {}
18     -- add field delimiter to line
19     line = string.concatenate(line,",")
20     -- parse line using comma as field delimiter
21     for value in string.iterate(line,"(.-),") do
22       values[#values+1] = value
23     end
24     -- initialize attributes in output
25     for index,value in ipairs(header) do
26       output[value] = values[index]
27     end
28     -- return output record
29     return expressor.ScriptSupport.OK, output
30   else
31     -- terminate processing
32     return expressor.ScriptSupport.FlowComplete
33   end
34 end
35
36 function finalize()
37   file_handle:close()
38 end
```

Outputs

- Output (Trash 1)
  - place
  - lastname
  - firstname
  - party

The processing logic should be clear.

- In the initialize function, a file handle is opened to the return from the Windows type utility, which concatenates the content of the three input files much like the UNIX/Linux cat utility.
- In the read function, records are read sequentially (line 14), appended with the field delimiter character (line 19), and parsed into individual fields with the string.iterate function (lines 21-23).
- The ipairs loop (lines 25-27) then initializes each of the attributes in the output record.
- The code assumes that the files will not include a header row.
  - If header rows exist, the code could be easily modified to skip those rows.

While improvements could be made to this code to make it more generic (for example, interrogating the directory for the names of the files to process), the problem still exists that you must manually define the attributes in the output and somehow provide the names of the header fields. Additionally, the field delimiter (the comma character) is hard coded and would need to be changed if the files use a different delimiter.

There are four issues with the simple Read Custom operator implementation.

1. The names and location of the files to be processed are hard coded.
2. The header field names are hard coded.
3. The field delimiter is hard coded.
4. The output record must be manually defined.

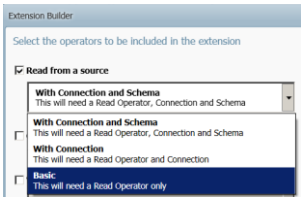
The extension operator to be developed in the next four sections will resolve all of these issues.

To avoid confusion and to make the development process more of a learning experience, the extension operator will be developed in a stepwise fashion, although as experience is gained all aspects of the operator could be tackled simultaneously.

- A first iteration will use an operator property to specify the name of a file, or a wildcard file name pattern, eliminating the need to include the file names in the code.
- A second iteration will enable the operator to use a file connection artifact so that the file system location will not need to be included in the code.
- The third iteration will integrate the operator with a schema so that the header field names and output record no longer need to be manually managed.
- And the fourth and final iteration will modify the schema and operator descriptors so that the field delimiter may be specified through an operator property rather than from within the code.

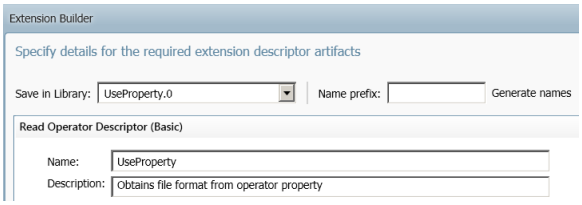
BASIC – READ FROM A SOURCE OPERATOR

Open a new Workspace and create a library named **UseProperty**.



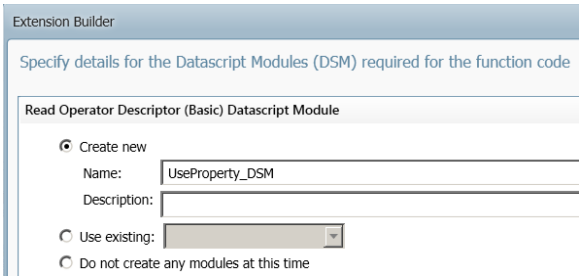
In the Extension Builder wizard,

- Check **Read from a source** drop down and select **Basic (This will need a Read Operator only)** from the dropdown menu.
- Click **Next**.



In the following window,

- Give the extension a name – **UseProperty** – and provide a brief description.
- Click **Next**.



And in the next window

- Indicate that the associated code should be included in a new datascript module – **UseProperty_DSM**.
- Click **Next** and **Finish** to close the wizard.

The Extension Builder has created a new operator icon named UseProperty in the Input grouping and two artifacts within the library.

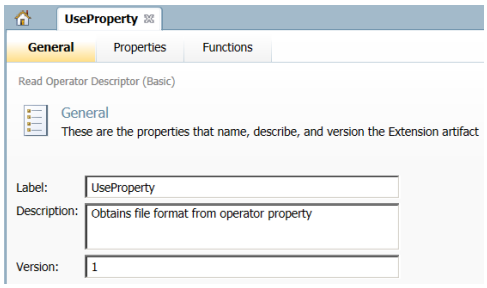
- An Operator Descriptor named UseProperty
- A Datascript Module named UseProperty_DSM

To complete the development process, a new operator property needs to be defined in the UseProperty operator descriptor, the ValidateOperator function needs to be implemented in the UseProperty_DSM datascript module, and code needs to be added to an instance of the UseProperty operator, which is then converted into an operator template.

The nice part about using the Extension Builder is that the operator descriptor already includes the necessary cross reference to the datascript module and it will be much easier to move between the descriptor and module, ensuring that you are working with the correct artifacts.

In the Desktop Explorer, open the UseProperty operator descriptor and note that set up information is distributed across three tabs.

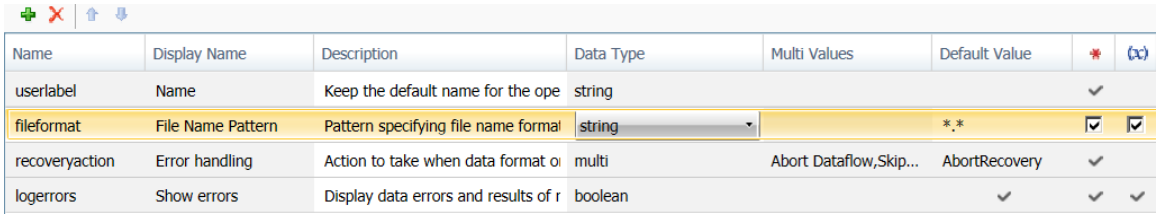
On the **General** tab, enter **UseProperty** as the label; this will become the name of the extension operator (equivalent to the operator names Read File or Read Table).



The **Properties** tab lists three default operator properties – Name, Error handling, Show errors – that cannot be modified.

To allow the Expressor developer to specify a file name or file name pattern, another operator property must be added. A control for entering a value into this property will subsequently appear in the UseProperty operator's Property panel.

- Click the + icon, which will add another line to the listing of properties.
- Define a string property.
    - Note that this property is both required * and parameterizable (x) and that *.* has been entered as the default value.
    - In the archive UseProperty.zip, the name of this property is fileformat, which will be labeled File Name Pattern in the operator's Property panel.

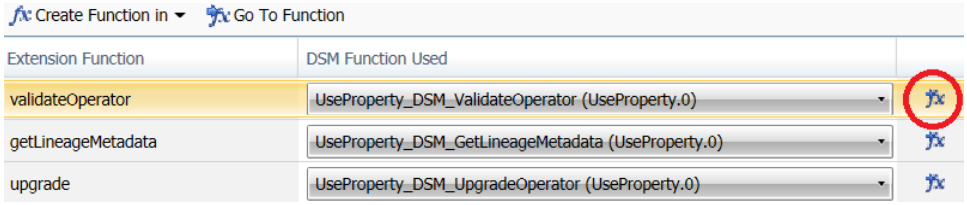


Name	Display Name	Description	Data Type	Multi Values	Default Value	*	(x)
userlabel	Name	Keep the default name for the ope	string			✓	
fileformat	File Name Pattern	Pattern specifying file name format	string		*.*	☑	☑
recoveryaction	Error handling	Action to take when data format o	multi	Abort Dataflow,Skip...	AbortRecovery	✓	
logerrors	Show errors	Display data errors and results of r	boolean		✓	✓	✓

The **Functions** tab lists the three required functions that must be included in the associated UseProperty_DSM datascript module. Note that the name of the datascript module has already been entered into the DSM Function Used column entry; this too is an effect of creating the descriptor and module through the functionality of the Extension Builder.

For this first version of the extension operator, it is only necessary to provide a non-default implementation for the ValidateOperator function. This code will check that an entry has been made into the File Name Pattern operator property.

Click the button adjacent to this function's entry in the listing, which opens the datascript module for editing.



Replace the default code with the following, which checks that an entry has been made in the File Name Pattern property and if not returns an error code and error message that is displayed in the Messages panel in Expressor Desktop.

UseProperty_DSM *

```
module("UseProperty.UseProperty_DSM", package.seeall)

function ValidateOperator(propertyList)
  messages = {}
  if (is.null(propertyList.fileformat) or
    is.empty(string.trim(propertyList.fileformat.value))) then
      table.insert(messages,{type='Error',message='You need to specify a file name format'})
      return 1,messages
  end
  return 0
end
```

Next, create a dataflow and note the UseProperty operator icon in the Input operator category. Drag this operator onto the dataflow; observe that the File Name Pattern property shows the default entry *.*. Now open the rule editor and add the following code. Note that the string attributes – place, lastname, firstname, party – that define the output record must be manually added to the Outputs panel.

```
require "expressor.ScriptSupport"

files = {}
directory = "C:\\data\\"
file_list = ""
file_handle = nil
header = {"place","lastname","firstname","party"}

function initialize()
  -- obtain the file name format property value
  file_pattern = _expCurrentOperatorParameters.fileformat
  -- modify pattern to match Datascript/Lua pattern matching syntax
  characters = {}
  for i=1,#file_pattern do
    character = string.substring(file_pattern,i,i)
    if character == "." then
      characters[i] = "%."
    elseif character == "*" then
      characters[i] = ".+"
    else
      characters[i] = character
    end
  end
  file_pattern = table.concat(characters,"")
  file_pattern = string.concatenate("^",file_pattern)

  file_handle = io.popen(string.concatenate("dir /b ", directory))
  for value in file_handle:lines() do
    file = string.match(value,file_pattern)
    if file then
      files[#files+1] = string.concatenate(directory,value)
    end
  end
  file_handle:close()
  file_list = table.concat(files," ")
  file_handle = io.popen(string.concatenate("type ",file_list))
end
```

```
39 function read()
40   -- retrieve a line
41   line = file_handle:read("*l")
42   if line then
43     output = {}
44     values = {}
45     -- add field delimiter to line
46     line = string.concatenate(line,",")
47     -- parse line using comma as field delimiter
48     for value in string.iterate(line,"(.-),") do
49       values[#values+1] = value
50     end
51     -- initialize attributes in output
52     for index,value in ipairs(header) do
53       output[value] = values[index]
54     end
55     -- return output record
56     return expressor.ScriptSupport.OK, output
57   else
58     -- terminate processing
59     return expressor.ScriptSupport.FlowComplete
60   end
61 end
62
63 function finalize()
64   file_handle:close()
65 end
```

The code in the read and finalize functions has not changed from the implementation in the Read Custom operator. The processing in the initialize function has changed significantly.

- Lines 3-7 define variables that will be used in the code.
- On line 11, the _expCurrentOperatorParameters table, which is initialized and made available to the code by the Expressor engine, contains a collection of operator properties, including the value entered into the File Name Pattern property.
    - Retrieve the entry by referring to its property name: fileformat.
- Lines 13-25 convert the wildcard pattern entered into the operator's Property panel into the pattern matching syntax used by the datascript string functions.
    - Lines 16-17 escape any dot characters.
    - Lines 18-19 replace * with .+, which specifies any string of characters.
    - Line 25 prepends ^ to the pattern, indicating that the file name pattern comparison must start at on the first character.
- Lines 27-33 use the Windows dir utility to obtain a listing of the files in the directory (specified in line 4), compares each file name to the file name pattern, and creates a table containing the names of the files that are to be processed.
- Line 35 turns the table into a string containing a space separated listing of the file names that matched the file name pattern.
- Line 36 initializes a file handle to the return from the Windows type utility, which concatenates the contents of the source files.

Now when the dataflow runs, the entry in the File Name Pattern property determines which files in the source directory are processed. Remember, these files must be comma separated, have four fields corresponding to the attributes place, lastname, firstname, party, and **lack** a header row.

Once the code has been validated, create an operator template from the UseProperty operator on the dataflow. In the future, use this template rather than the UseProperty operator icon to add this extension operator to another dataflow. If you drag the UseProperty operator icon into another dataflow, it will not contain any code whereas the template contains the validated coding.

The library UseProperty in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation. Use the files democrats.dat, republicans.dat, and whigs.dat as source files; these files are in the file data.zip.

THE PROPERTYLIST TABLE

In the code in the validateOperator function, property values entered into the operator's Properties panel were retrieved from the propertyList table. Look at this code and note that the individual elements in this table contain nested tables. For example, propertyList.fileformat, where fileformat is the name of a property added to the operator's descriptor, is a table containing a string indexed element named value. Therefore, the statement

```
is.null(propertyList.fileformat)
```

determines whether this nested table exists and the statement

```
is.empty(string.trim(propertyList.fileformat.value))
```

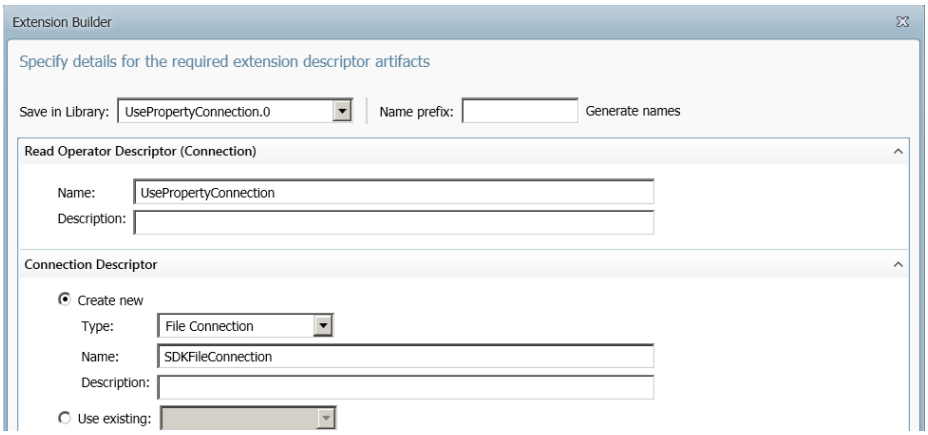
confirms that a non-space entry has been made into the control.

WITH CONNECTION – READ FROM A SOURCE OPERATOR

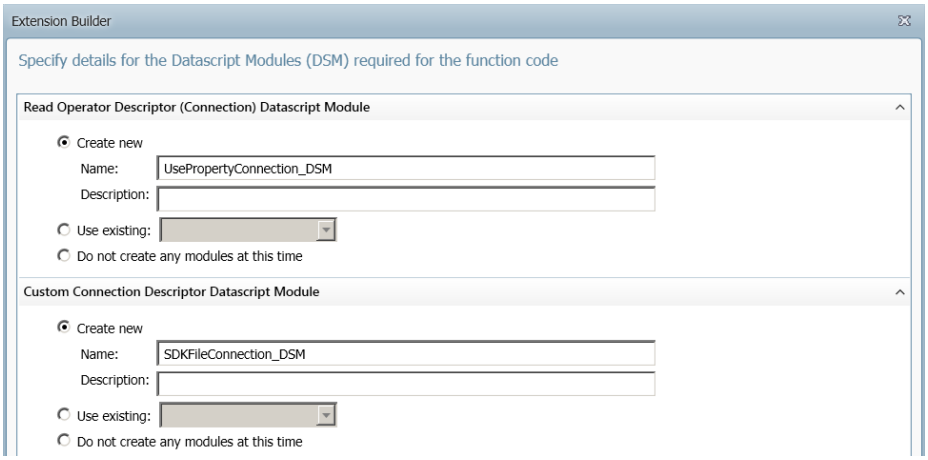
This extension operator will use a standard file connection artifact to locate the directory containing the source files. The library UsePropertyConnection in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation. Use the files democrats.dat, republicans.dat, and whigs.dat as source files; these files are in the file data.zip.

To begin this process, create a new library named **UsePropertyConnection**.

- In the Extension Builder select the **With Connection (This will need a Read operator and Connection)** entry from the Read from a source drop down list.
- In the Read Operator Descriptor (Connection) grouping, enter **UsePropertyConnection** as the name of the read operator descriptor.
- In the Connection Descriptor grouping, select File Connection from the Type drop down and enter **SDKFileConnection** as the name of the connection descriptor.



- Click **Next**.
- In the following window, accept the default name suggestions for the datascript modules that will be associated with each descriptor.

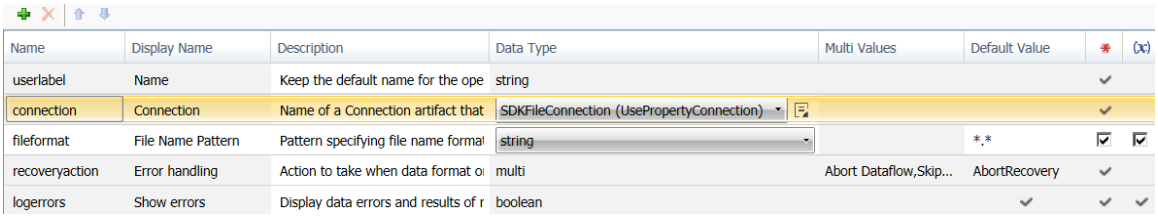


- Click **Next** and **Finish** to complete the process.

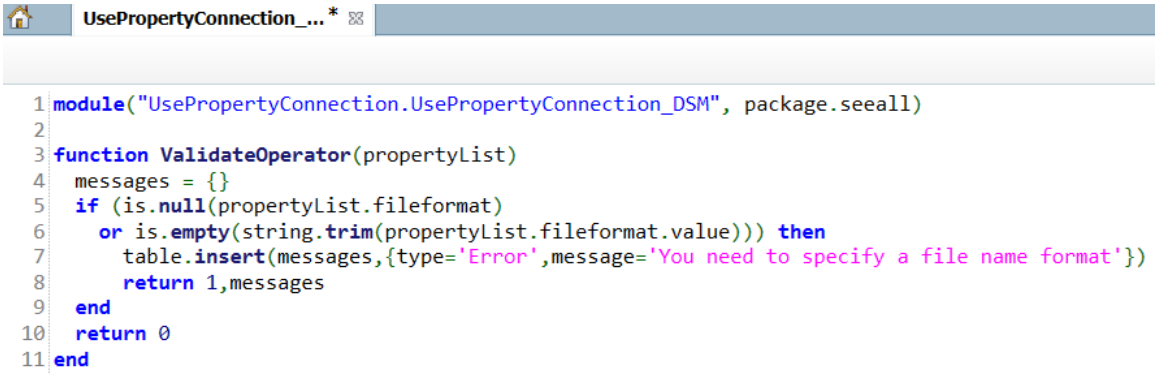
The wizard creates an Artifact Descriptor, Operator Descriptor, and two datascript modules, one associated with each descriptor.

For the Operator Descriptor – named UsePropertyConnection – enter **UsePropertyConnection** as the label value on the **General** tab

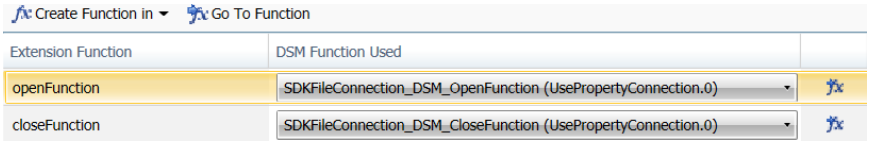
and then, as in the previous example, add the fileformat property to the listing in the **Properties** tab. Note that the connection property is already set to the datascript module corresponding to the connection Artifact Descriptor. This is an effect of using the Extension Builder to initiate the development effort.



Select the **Functions** tab, click on the button adjacent to the ValidateOperator function and add the same code to this function as in the previous example.



For the Artifact Descriptor – named SDKFileConnection – observe that on the **Functions** tab the associated datascript module is already entered into the DSM Function Used column.



The two functions are not invoked with an operator extension that just uses a connection artifact (they are invoked when the extension operator uses both a connection and schema artifact, which will be developed in the next example), so there is no need to open this datascript module and modify the default code.

Next, create a dataflow and note the UsePropertyConnection operator icon in the Input operator category. Drag this operator onto the dataflow; observe that the File Name Pattern property shows the default entry *.* and that there is now a drop down control labeled Connection. Create a standard File Connection, with its path pointing to the directory holding the source files, and enter its name into the Connection control.

Open the rule editor and add the following code. Again, the string attributes – place, lastname, firstname, party – that define the output record must be manually added to the Outputs panel.

```
require "expressor.ScriptSupport"

files = {}
directory = string.concatenate(_expCurrentOperatorParameters.Path ,"\\")
file_list = ""
file_handle = nil
fielddelimiter = ','
header = {"place","lastname","firstname","party"}

function initialize()
  -- obtain the file name format property value
  file_pattern = _expCurrentOperatorParameters.fileformat
  -- modify pattern to match Datascript/Lua pattern matching syntax
  characters = {}
  for i=1,#file_pattern do
    character = string.substring(file_pattern,i,i)
    if character ==". " then
      characters[i] = "%."
    elseif character == "*" then
      characters[i] = ".+"
    else
      characters[i] = character
    end
  end
  file_pattern = table.concat(characters,"")
  file_pattern = string.concatenate("^",file_pattern)

  file_handle = io.popen(string.concatenate("dir /b ", directory))
  for value in file_handle:lines() do
    file = string.match(value,file_pattern)
    if file then
      files[#files+1] = string.concatenate(directory,value)
    end
  end
  file_handle:close()
  file_list = table.concat(files," ")
  file_handle = io.popen(string.concatenate("type ",file_list))
end
```

```
40 function read()
41   -- retrieve a line
42   line = file_handle:read("*l")
43   if line then
44     output = {}
45     values = {}
46     -- add field delimiter to line
47     line = string.concatenate(line,fielddelimiter)
48     -- parse line using comma as field delimiter
49     pattern = string.concatenate("(.-)",fielddelimiter)
50     for value in string.iterate(line,pattern) do
51       values[#values+1] = value
52     end
53     -- initialize attributes in output
54     for index,value in ipairs(header) do
55       output[value] = values[index]
56     end
57     -- return output record
58     return expressor.ScriptSupport.OK, output
59   else
60     -- terminate processing
61     return expressor.ScriptSupport.FlowComplete
62   end
63 end
64
65 function finalize()
66   file_handle:close()
67 end
```

The code in the finalize function has not changed from either of the earlier implementations. The processing in the initialize function is the same as in the UseProperty example. The processing before the initialize function and in the read function has changed slightly.

- Line 4 retrieves the path from the connection artifact through an entry in the _expCurrentOperatorParameters table.
    - The path separator is then appended to the path so the absolute path to each file can be generated.
- Line 7 sets a variable named fielddelimiter to the comma character.
- Line 49 uses the value in the fielddelimiter variable (line 7) to define the pattern that will be used to parse fields from each record (lines 50-52).
    - In the UseProperty example, the comma character was hard coded into this pattern.

Now when the dataflow is run, the path to the source directory is obtained from the file connection artifact.

Once the code has been validated, create an operator template from the UsePropertyConnector operator on the dataflow. In the future, use this template rather than the UsePropertyConnector icon to add this extension operator to another dataflow

The library UsePropertyConnector in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation. Use the files democrats.dat, republicans.dat, and whigs.dat as source files; these files are in the file data.zip.

WITH CONNECTION AND SCHEMA – READ FROM A SOURCE OPERATOR

In the previous examples, each operator descriptor was associated with a single datascript module that included implementations of the required functions ValidateOperator, GetLineageMetadata, and Upgrade, although only the default code in the ValidateOperator function was replaced. For an extension operator that includes a schema, there will be two datascript modules associated with the operator descriptor.

- A datascript module implementing the required ValidateOperator, GetLineageMetadata, and Upgrade functions.
- A datascript module implementing the operator functions initialize, read and finalize.
    - Unlike the previous approaches to developing an extension operator, the operator functions are included in a module rather than within the rule editor of the operator.
    - The operator's rule editor will not open.

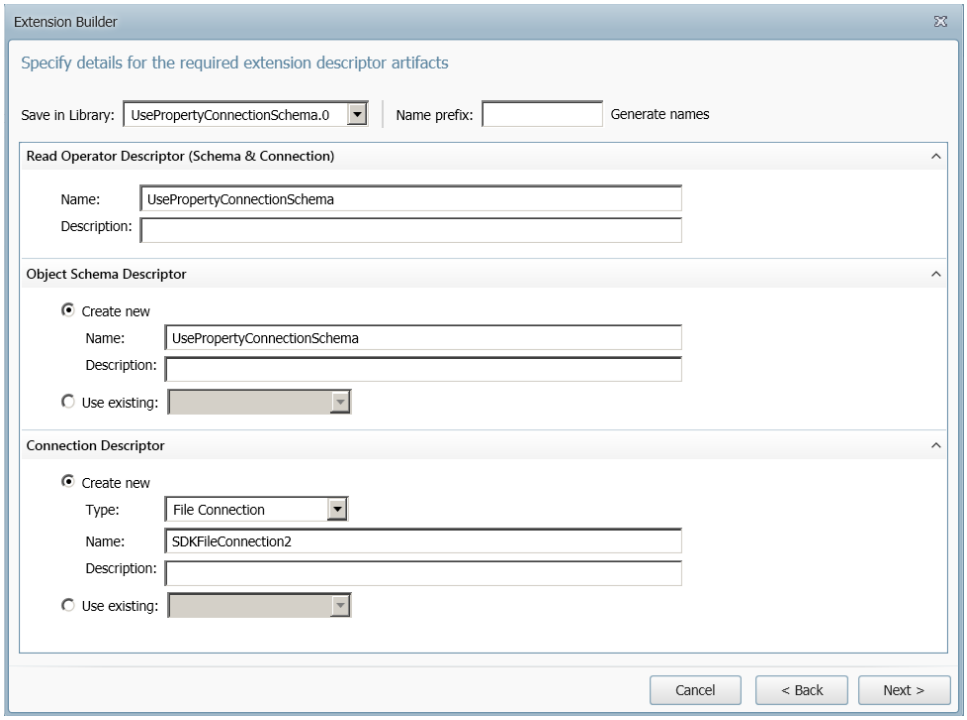
There will also be an artifact descriptor that describes the properties and behaviors of the schema artifact and an associated datascript module that implements the GetObjectList and GetFieldList functions, which dynamically create a schema from a representative data file.

And, as in the previous example, there will be an artifact descriptor for the connection, associated with a datascript module that implements the OpenConnection and CloseConnection functions.

The library UsePropertyConnectionSchema in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation. Use the files democrats.txt, republicans.txt, and whigs.txt as source files. These files **include** a header row and are in the file data.zip.

To begin this process, create a new library named **UsePropertyConnectionSchema**.

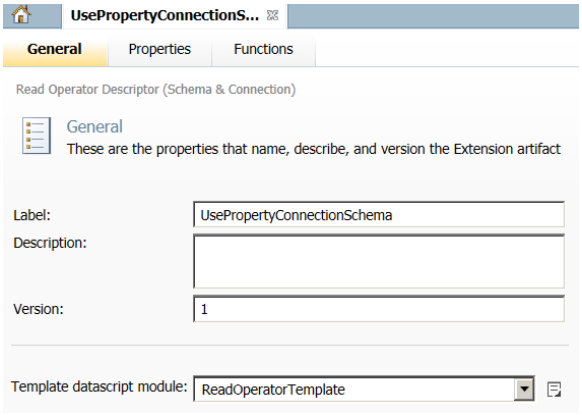
- In the Extension Builder select the **With Connection and Schema (This will need a Read operator, Connection and Schema)** entry from the Read from a source drop down list.
- In the Read Operator Descriptor (Schema & Connection) grouping, enter **UsePropertyConnectionSchema** as the name of the read operator descriptor.
- In the Object Schema Descriptor grouping, enter **UsePropertyConnectionSchema** as the name of the object schema descriptor.
- In the Connection Descriptor grouping, select File Connection from the Type drop down and enter **SDKFileConnection2** as the name of the connection descriptor.



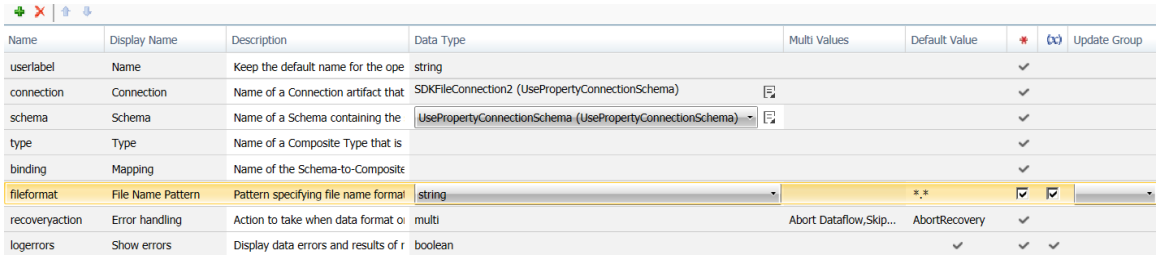
- Click **Next**.
- In the following window, accept the default name suggestions for the datascript modules that will be associated with each descriptor.
    - UsePropertyConnectionSchema_DSM, which is associated with the object schema descriptor.
    - SDKFileConnection2_DSM, which is associated with the connection descriptor.
    - ReadOperatorTemplate_DSM, which is associated with the read operator descriptor.
    - ReadOperatorTemplate, into which the operator's coding will be added.
- Click **Next** and **Finish** to complete the process.

The wizard creates two Artifact Descriptors, an Operator Descriptor, and the four datascript modules.

For the Operator Descriptor – named UsePropertyConnectionSchema – enter **UsePropertyConnectionSchema** as the label value on the **General** tab, noting that the ReadOperatorTemplate datascript module is identified as the location of the operator code.



Then add the fileformat property to the listing in the **Properties** tab as in the previous example. Note that the connection and schema properties are already set to the datascript modules corresponding to the Artifact Descriptor. This is an effect of using the Extension Builder to initiate the development effort.



Name	Display Name	Description	Data Type	Multi Values	Default Value	*	(x)	Update Group
userlabel	Name	Keep the default name for the ope	string			✓		
connection	Connection	Name of a Connection artifact that	SDKFileConnection2 (UsePropertyConnectionSchema)			✓		
schema	Schema	Name of a Schema containing the	UsePropertyConnectionSchema (UsePropertyConnectionSchema)			✓		
type	Type	Name of a Composite Type that is				✓		
binding	Mapping	Name of the Schema-to-Composite				✓		
fileformat	File Name Pattern	Pattern specifying file name format	string		*.*	✓	✓	
recoveryaction	Error handling	Action to take when data format o	multi	Abort Dataflow,Skip...	AbortRecovery	✓		
logerrors	Show errors	Display data errors and results of r	boolean		✓	✓	✓	

Select the **Functions** tab, click on the button adjacent to the ValidateOperator function (this function is in the ReadOperatorTemplate_DSM datascript module) and add the same code to this function as in the previous examples.

```
function ValidateOperator(mappingSet, propertyList, recordSpecial, query)
  messages = {}
  if (is.null(propertyList.fileformat)
    or is.empty(string.trim(propertyList.fileformat.value))) then
      table.insert(messages,{type='Error',message='You need to specify a file name format'})
      return 1,messages
  end
  return 0
end
```

For the Connection Descriptor – named SDKFileConnection2 – non-default implementations for OpenFunction and CloseFunction must be added. These functions will be called when the Expressor developer creates an instance of the schema that is defined in the object descriptor. Since an extension file connection is based on the standard file connection, it doesn't have any additional properties, so this descriptor has only two tabs: General and Functions.

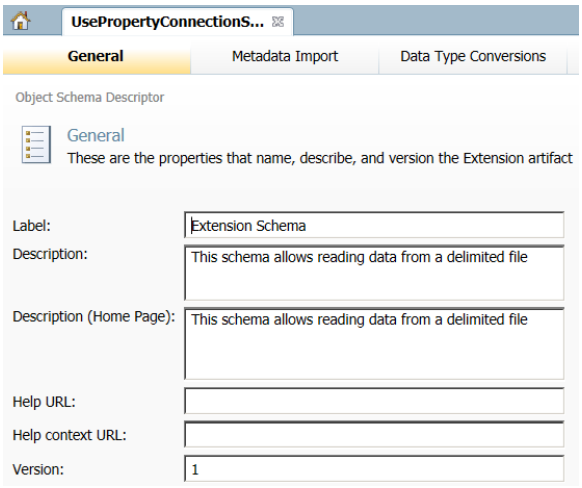
Select the **Functions** tab, click on the button adjacent to either function and add the following code.

```
1  module("UsePropertyConnectionSchema.SDKFileConnection2_DSM", package.seeall)
2
3
4  function OpenFunction(paramList)
5    session = {pathToFile=paramList.Path,fullFileName=paramList.sourceFileName}
6    return 0,session
7  end
8
9  function CloseFunction(session)
10   return 0
11 end
```

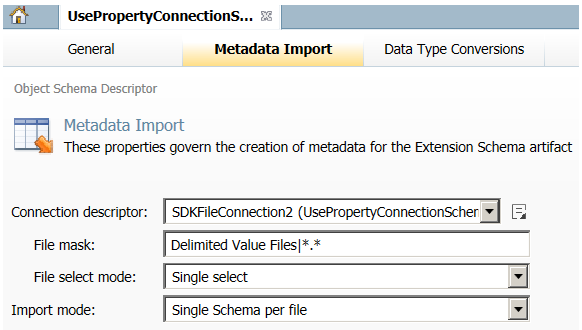
The session object returned from OpenFunction is a datascript table that includes all information needed to use the opened connection. With a file connection artifact, this information includes only the path to, and name of, the file used when creating the schema.

Now open the object schema descriptor named UsePropertyConnectionSchema. This descriptor's set up is distributed across five tabs: General, Metadata Import, Data Type Conversions, Schema Editing, Functions.

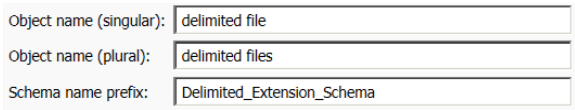
On the **General** tab, enter **Extension Schema** as the label and suitable descriptions.



On the **Metadata Import** tab, the file connection descriptor is already identified; again an effect of using the Expression Builder to initiate the development effort. Enter **Delimited Value Files|*.*** into the file mask control and choose Single select and Single Schema per file from the file select mode and import mode drop down controls.

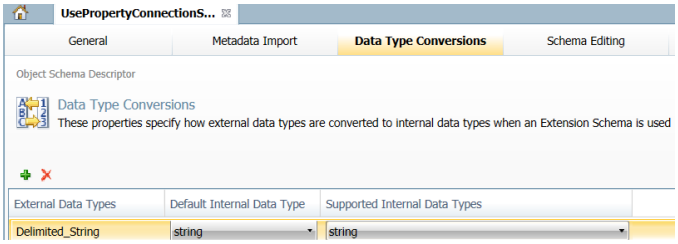


Scroll down the tab and enter delimited file, delimited files, and Delimited_Extension_Schema into the Object name (singular), Object name (plural), and Schema name prefix controls.



Leave all other controls on this tab blank or set to their default entries.

On the **Data Type Conversions** tab, enter a name for the data type(s) of the external data. Any meaningful name can be entered. Then select string as the default and supported internal Expressor data types. Within Expressor, data corresponding to this external type will be converted to the specified Expressor type.



Do not change any settings on the **Schema Editing** tab.

On the **Functions** tab, note that the name of the datascript module containing implementations of the extension functions is already selected in the DSM Function Used column. Click the button adjacent to the GetObjectList function and enter the following code into the datascript module.

```
function GetObjectList(session, schemaImportOptions)
    return 0, {{name = session.fullFileName, description = 'The delimited file'}}
end
```

This function returns a datascript table that lists the objects available from the data source (for example, a listing of table names in a relational database or a file name).

Then scroll down to the GetFieldList function and enter the following code.

```
function GetFieldList(
    session, objectNames, schemaImportOptions, context, constructionMetadata)
  local fielddelimiter = ','
  local schemas = {}
  local discoveredFields = {}
  --take a full file name saved in session in the OpenConnection function
  local fullFileName = session.fullFileName
  --open a file
  local file = io.open(fullFileName, 'r')
  if file ~= nil then
    --read the first line with a header:
      local headerLine = file:read("*l")
      if headerLine ~= nil then
        --split record by field delimiter
        headerLine = string.concatenate(headerLine,fielddelimiter)
        pattern = string.concatenate("(.-)",fielddelimiter)
        headerColumns = {}
        for value in string.iterate(headerLine,pattern) do
          headerColumns[#headerColumns+1] = value
        end

        for index, value in ipairs(headerColumns) do
          --create a field that was discovered.
          --Note that all properties in all fields are the same excepting a field name.
          table.insert(discoveredFields,
                {
                  name = string.trim(value),
                  dataTypeName = 'Delimited_String',
                  size = 0,
                  allownulls = false,
                  customizable = false,
                  fieldSpecial = { }
                }
                )
        end
```

```
        --create a single schema.
        local csvSchema = {
          sourceName = session.fullFileName,
          customizable = false,
          recordSpecial = {},
          fields = discoveredFields
        }
        --add schema to schemas set
        table.insert(schemas, csvSchema)
        --close a file
        file:close()
        --return success code and schemas  set
        return 0, schemas

      else
        return 2, "error reading CSV header from the CSV file"
      end
  else
    return 1, "Error: the selected file cannot be opened"
  end

end
```

The coding in this function is less complex than it seems. The processing reads the first line of the file, the header row, and parses it (by looking for the comma field delimiters) into a numerically indexed Datascript table (named headerColumns) such that each element contains the name of a field.

```
if headerLine ~= nil then
  --split record by field delimiter
  headerLine = string.concatenate(headerLine,fielddelimiter)
  pattern = string.concatenate("(.-)",fielddelimiter)
  headerColumns = {}
  for value in string.iterate(headerLine,pattern) do
    headerColumns[#headerColumns+1] = value
  end
```

Then, for each field, a Datascript table describing its name, data type (always Delimited_String), and other characteristics is created and added to a Datascript table named discoveredFields.

```
for index, value in ipairs(headerColumns) do
  --create a field that was discovered.
  --Note that all properties in all fields are the same excepting a field name.
  table.insert(discoveredFields,
        {
          name = string.trim(value),
          dataTypeName = 'Delimited_String',
          size = 0,
          allownulls = false,
          customizable = false,
          fieldSpecial = { }
        }
        )
end
```

Finally, a schema description is defined within a Datascript table and this table is inserted into another table containing a collection of schemas.

```
--create a single schema.
local csvSchema = {
  sourceName = session.fullFileName,
  customizable = false,
  recordSpecial = {},
  fields = discoveredFields
}
--add schema to schemas set
table.insert(schemas, csvSchema)
```

Next, open the datascript module **ReadOperatorTemplate** and add the following code.

```
require "expressor.ScriptSupport"

files = {}
directory = string.concatenate(_expCurrentOperatorParameters.Path ,"\\")
file_list = ""
file_handle = nil
header = {}
fielddelimiter = ','
function initialize()
  -- obtain the file name format property value
  file_pattern = _expCurrentOperatorParameters.fileformat
  -- modify pattern to match Datascript/Lua pattern matching syntax
  characters = {}
  for i=1,#file_pattern do
    character = string.substring(file_pattern,i,i)
    if character "." then
      characters[i] = "%."
    elseif character == "*" then
      characters[i] = ".+"
    else
      characters[i] = character
    end
  end
  file_pattern = table.concat(characters,"")
  file_pattern = string.concatenate("^",file_pattern)

  file_handle = io.popen(string.concatenate("dir /b ", directory))
  for value in file_handle:lines() do
    file = string.match(value,file_pattern)
    if file then
      files[#files+1] = string.concatenate(directory,value)
    end
  end
  file_handle:close()
  file_list = table.concat(files," ")
  file_handle = io.popen(string.concatenate("type ",file_list))

  headerLine = file_handle:read("*l")
  -- add field delimiter to line
  headerLine = string.concatenate(headerLine,fielddelimiter)
  -- parse line using comma as field delimiter
  pattern = string.concatenate("(.-)",fielddelimiter)
  for value in string.iterate(headerLine,pattern) do
    header[#header+1] = value
  end
end
-- Compile substutution block

--__MAPPING_SET__
--__RECORD_SPECIAL__
--__EXTENSION_VERSION__

-- End of compile substitution block
```

```
57 function read()
58   -- retrieve a line
59   line = file_handle:read("*l")
60   if line then
61     pattern = string.concatenate("^",header[1])
62     if is.null(string.match(line,pattern)) then
63       output = {}
64       values = {}
65       -- add field delimiter to line
66       line = string.concatenate(line,fielddelimiter)
67       -- parse line using comma as field delimiter
68       pattern = string.concatenate("(.-)",fielddelimiter)
69       for value in string.iterate(line,pattern) do
70         values[#values+1] = value
71       end
72       for index,value in ipairs(values) do
73         fieldName = string.trim(header[index])
74         field = mappingSet.fields[fieldName]
75         if is.null(field) then
76           error(string.concatenate(
77             "Field ",header[index]," cannot be found"))
78         end
79         output[field.attrName] = value
80       end
81       -- return output record
82       return expressor.ScriptSupport.OK, output
83     else
84       return expressor.ScriptSupport.SkipRecord
85     end
86   else
87     return expressor.ScriptSupport.FlowComplete
88   end
89 end
90
91 function finalize()
92   file_handle:close()
93 end
```

The coding in the initialize function should look familiar as it is similar to the previous example. However, lines 39-46 read the first line of the first file and parse the field names in this header row into a Datascript table named header.

In the read function, line 61 determines if the line just read is a header row, in which case it is skipped (line 84). If the line is not a header row, it is parsed into its individual values (lines 66-71). Finally, lines 72-80 initialize each of the attributes in the output record, which is emitted from the operator (line 82).

This logic probably needs a bit more discussion.

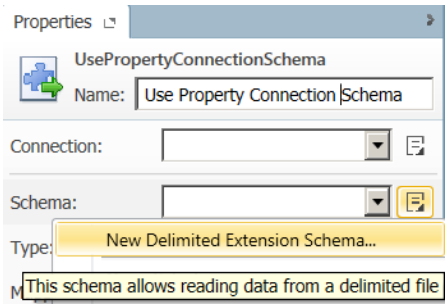
The ipairs loop (lines 72-80) iterates through a Datascript table in which each element contains the value of a field in the record being processed. Since the order of these values is the same as the order of field names in the header, line 73 can retrieve the field name from the Datascript table named header (created in lines 44-46 of the initialize function) and then use this field name to retrieve the name of the corresponding attribute in the schema's composite type (line 74 and line 79). In line 74, a field description is extracted from the mappingSet table and from this description the attribute name is obtained in line 79. Reviewing the structure of the mappingSet table will make this clearer.

```
mappingSet = {
       object='…',
       query='…',
       fields={                          -- table of mapped fields to attributes
              ID = {                     -- schema field name, for example, ID
                     type='…',           -- schema field type
                     attrName='AcctID',  -- composite type attribute name
                     attrType='…',       -- composite type attribute type
                     fieldSpecial={…},
                     position=#          -- position of field in schema
              },
              -- other field descriptions, no comma after last description
       } -- end fields table
} -- end mappingSet
```

Line 74 uses the field name as an index into the table mappingSet.fields to retrieve the element corresponding to a specific field (for example, ID). Then line 79 uses the attrName index to retrieve the name of the composite type attribute (for example, AcctID) corresponding to the field.

Finally, create a dataflow and note the UsePropertyConnectionSchema icon in the Input operator category. Drag this operator onto the dataflow; observe that the File Name Pattern property shows the default entry *.* that there is a drop down control labeled Connection, and controls for Schema, Type, and Mapping. Now:

- Create a standard File Connection, with its path pointing to the directory holding the source files, and make an entry into the Connection control.
- Create a Delimited Extension Schema.
    - Click the  button to the right of the Schema drop down control and select **New Delimited Extension Schema** from the popup menu.



    - In the next window, select a file on which to base the schema.
        - Click **Next**.
    - In the last window, accept or change the name of the schema and click **Finish**.
    - Entries appear in the Schema, Type, and Mapping controls.

Now when the dataflow is run, the entry in the File Name Pattern property determines which files in the source directory are processed. Remember, these files must be comma separated, have four fields corresponding to the attributes place, lastname, firstname, party, **and include a header row**. And the path to the source directory is obtained from the file connection artifact.

Once the code has been validated, create an operator template from the UsePropertyConnectorSchema operator on the dataflow. In the future, use this template rather than the UsePropertyConnectorSchema icon to add this extension operator to another dataflow

The library UsePropertyConnectorSchema in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation. Use the files democrats.txt, republicans.txt, and whigs.txt as source files; these files **include** a header row and are in the file data.zip.

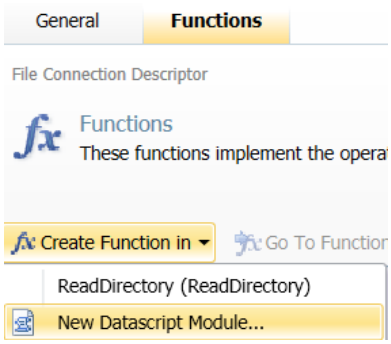
READ DIRECTORY EXTENSION OPERATOR

Now it's time to create the reusable extension operator that will add meaningful functionality to Expressor. The operator created in the last exercise was close to what is required, but it lacks one feature – the ability to specify the field delimiter. This exercise will reproduce the operator developed in the last exercise, adding this last feature. However, this exercise will create each descriptor and datascript module individually rather than using the Extension Builder.

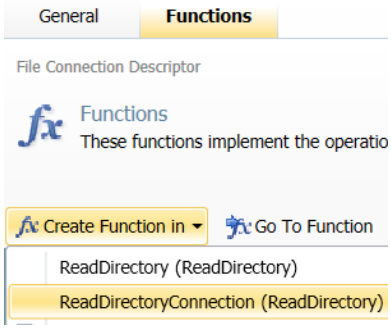
The library ReadDirectory in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation. Use the files democrats.txt, republicans.txt, and whigs.txt as source files. These files **include** a header row and are in the file data.zip.

To begin this process, create a new library named **ReadDirectory**.

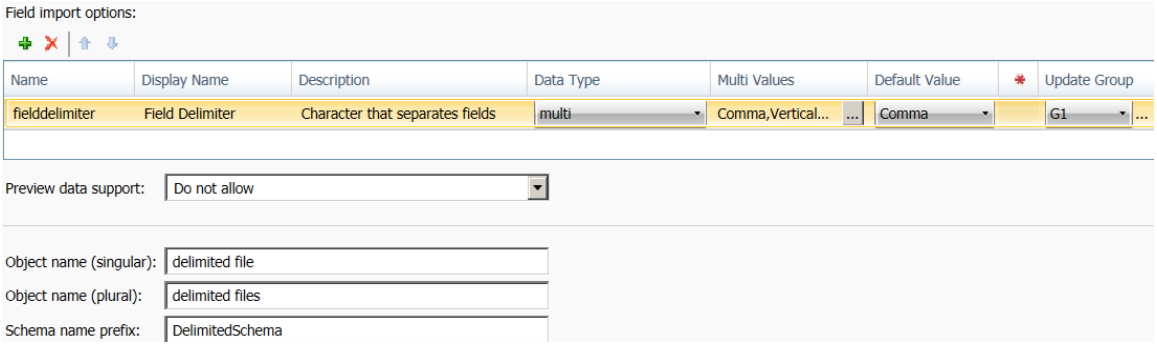
- Create a file connection descriptor:
  - Either click the **Create > Extension** ribbon bar button and selecting **Artifact Descriptor > File Connection Descriptor** from the popup menu, or
    - Or right-click on **ReadDirectory > Artifact Descriptors** in the **Explorer > Folders** tab and selecting **New > File Connection Descriptor** from the popup menu.
  - Name the descriptor **ReadDirectoryConnection** and confirm that it will be saved into the ReadDirectory library.
  - Open in the artifact descriptor editor.
  - On the **Functions** tab:
    - Highlight the **OpenFunction** entry and click the **Create Function in > New Datascript Module…** menu item.



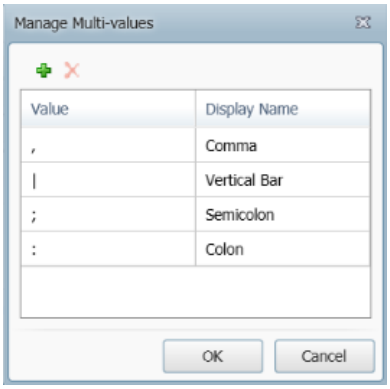
      - Name the module **ReadDirectoryConnection**.
    - Highlight the **CloseFunction** entry, click the **Create Function in > ReadDirectoryConnection** menu item.



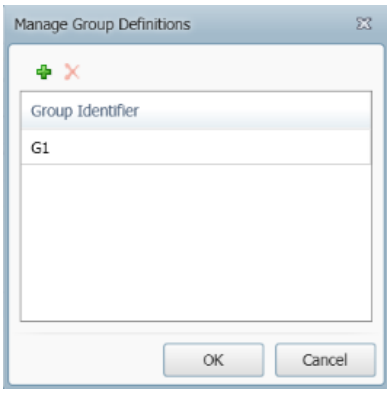
        - The controls under the DSM Function Used column are filled in.
    - Open the ReadDirectoryConnection datascript module and provide implementations for OpenFunction and CloseFunction.
        - Simply copy the code from the SDKFileConnection2_DSM datascript module in the UsePropertyConnectionSchema library.
- Create an object schema descriptor:
    - Either click the **Create > Extension** ribbon bar button and selecting **Artifact Descriptor > Object Schema Descriptor** from the popup menu, or
        - Or right-click on **ReadDirectory > Artifact Descriptors** in the **Explorer > Folders** tab and selecting **New > Object Schema Descriptor** from the popup menu.
    - Name the descriptor **ReadDirectorySchema** and confirm that it will be saved into the ReadDirectory library.
    - Open in the artifact descriptor editor.
    - On the **General** tab, enter **Delimited Extension Schema** into the label control and suitable entries into the description controls.
    - On the **Metadata Import** tab, select **ReadDirectoryConnection** from the drop down Connection descriptor control and enter **Delimited Value Files|*.*** into the File mask control.
        - Create an entry in the Field import options grouping.
            - Click the + icon to insert a row into the listing.



            - To make entries in the Multi Values column, click on the ellipsis (…) to open the Manage Multi-values window. Click the + icon to add each entry.

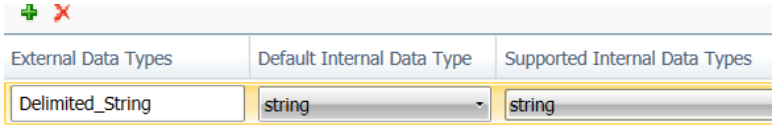


            - To make entries in the Update Group column, click on the ellipsis (…) to open the Manage Group Definitions window. Click the + icon to add an entry.



- Enter any character string as the group identifier.
- The Default Value column will convert to a drop down control once you save and reopen the descriptor.

- On the **Data Type Conversions** tab, click the + icon and add an entry for the **Delimited_String** type.

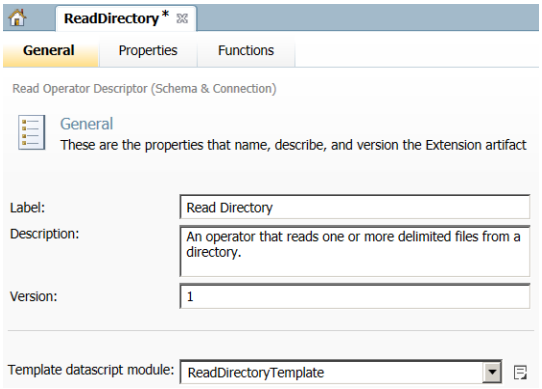


- Do not change any of the entries in the **Schema Editing** tab.
- On the **Functions** tab:
  - Highlight the **GetObjectList** entry and click the **Create Function in > New Datascript Module…** menu item.
    - Name the module **ReadDirectorySchema**.
  - Highlight the **GetFieldList** entry, click the **Create Function in > ReadDirectorySchema** menu item.
  - Highlight the **Upgrade** entry, click the **Create Function in > ReadDirectorySchema** menu item.
- Open the ReadDirectorySchema datascript module and provide implementations for GetObjectList and GetFieldList.
  - Start by copying the code from the UpdatePropertyConnectionSchema_DSM datascript module in the UsePropertyConnectionSchema library.
  - Change the first line of code in the GetFieldList function, retrieving the field delimiter from the schemaImportOptions argument.

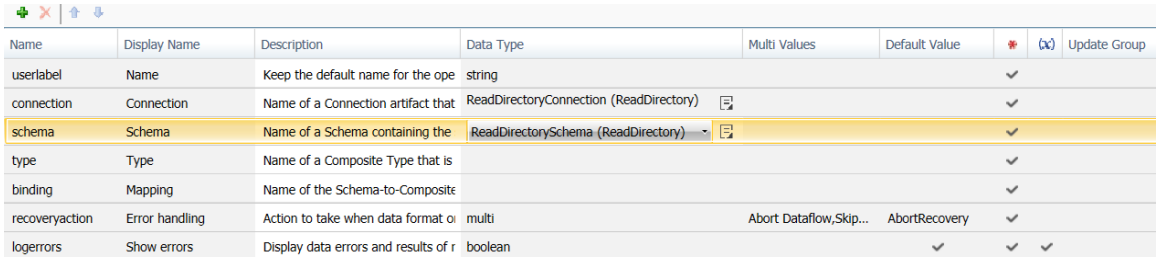
```
function GetFieldList(session, objectNames, schemaImportOptions, context, constructionMetadata)
  local fielddelimiter = string.trim(schemaImportOptions[1].fielddelimiter)
```

- Create an operator descriptor:
  - Either click the **Create > Extension** ribbon bar button and selecting **Operator Descriptor > Read Operator Descriptor (Schema & Connection)** from the popup menu, or
    - Or right-click on **ReadDirectory > Operator Descriptors** in the **Explorer > Folders** tab and selecting **New > Read Operator Descriptor (Schema & Connection)** from the popup menu.
  - Name the descriptor **ReadDirectory** and confirm that it will be saved into the ReadDirectory library.
  - Open in the operator descriptor editor.

- On the **General** tab, enter **Read Directory** into the label control, write a suitable description and click the button to create a new datascript module named **ReadDirectoryTemplate**, which will ultimately contain implementations of the initialize, read, and finalize functions.



- On the **Properties** tab, click in the Data Type cell in the schema row and select ReadDirectorySchema from the drop down control.
    - The Data Type cell in the connection row immediately displays ReadDirectoryConnection.



    - Next, create two additional properties: fileformat and fielddelimiter.

        - Entries in the Update Group drop down control for the fielddelimiter entry may not appear until you save, close and reopen the artifact.

- On the **Functions** tab:
    - Highlight the **Compile** entry and click the **Create Function in > New Datascript Module…** menu item.
        - Name the module **ReadDirectory**.
    - Highlight the **ValidateOperator** entry, click the **Create Function in > ReadDirectory** menu item.
    - Highlight the **GetLineageMetadata** entry, click the **Create Function in > ReadDirectory** menu item.
    - Highlight the **Upgrade** entry, click the **Create Function in > ReadDirectory** menu item.
- Open the **ReadDirectory** datascript module and provide an implementation for ValidateOperator.
    - Simply copy this code from the ReadOperatorTemplate_DSM datascript module in the UsePropertyConnectionSchema library.

- Open the datascript module **ReadDirectoryTemplate** and provide code for the initialize, read, and finalize functions.
    - Start by copying the code from the ReadOperatorTemplate datascript module in the UsePropertyConnectionSchema library.
    - Change the eighth line in the code, extracting the field delimiter from the _expCurrentOperatorParameters table.

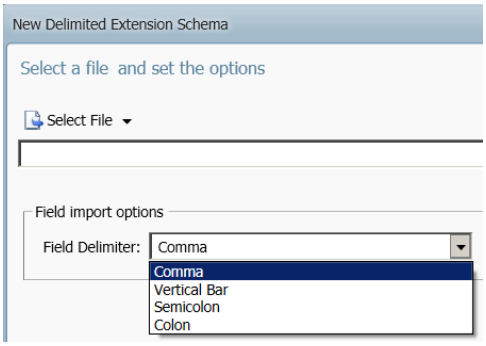
```
8 fielddelimiter = _expCurrentOperatorParameters.fielddelimiter
```

Now when the dataflow is run, the entry in the File Name Pattern property determines which files in the source directory are processed. Remember, these files must have four fields corresponding to the attributes place, lastname, firstname, party, and **include a header row**. And the path to the source directory is obtained from the file connection artifact.

Once the code has been validated, create an operator template from the Read Directory operator on the dataflow. In the future, use this template rather than the Read Directory icon to add this extension operator to another dataflow

The library Read Directory in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation. Use the files democrats.txt, republicans.txt, and whigs.txt as comma delimited source files, and the files democrats.pipe, republicans.pipe, and whigs.pipe as vertical bar delimited source files. These files **include** a header row and are in the file data.zip.

THE GETFIELDLIST FUNCTION



This function is called when a new schema described by the ReadDirectorySchema artifact is created. In the example developed in this section, the schemaImportOptions argument is queried to extract the field delimiter character specified in the schema wizard.

```
local fielddelimiter = string.trim(
      schemaImportOptions[1].fielddelimiter)
```

What is the schemaImportOptions argument? The schemaImportOptions table is a numerically indexed table where each element is a string indexed table. The selected field delimiter is extracted from the element identified by the index value fielddelimiter. Why was the first element in the schemaImportOptions table selected, and are there other elements in the nested table besides fielddelimiter?

These are not easy questions to answer as this function is called while developing a schema, not while the dataflow runs. The nested table should contain the field import and object import options entered into the Metadata Import tab in the ReadDirectorySchema descriptor. But demonstrating this cannot be achieved through simple calls to the logging functions. Code that interrogates this table and writes to a file must be added to the GetFieldList function.

In the GetFieldList function, immediately following the function declaration statement, enter the following code.

```
39 function GetFieldList(session, objectNames, schemaImportOptions, context, constructionMetadata)
40   io.output("C:\\info\\schemaStuff.txt")
41   for k,v in ipairs(objectNames) do
42     io.write(k,'\t',v,'\n')
43   end
44   for k,v in ipairs(schemaImportOptions) do
45     for x,y in pairs(v) do
46       io.write(x,'\t',y,'\n')
47     end
48   end
49   io.close()
```

This code will iterate through both the objectNames and schemaImportOptions tables. The objectNames table contains a single entry that identifies the file used to generate the schema and the schemaImportOptions table contains the entries made into the field and object import listings in the artifact descriptor.

```
1       C:\data\presidents.csv
fielddelimiter      ,
```

With this information, the statement that retrieves the field delimiter character can be written.

Note: Field import options are available from the schemaImportOptions argument in the GetFieldList function while Object import options are available from the schemaImportOptions argument in the GetObjectList function.

THE _EXPCURRENTOPERATORPARAMETERS TABLE

In the last three implementations, code in the ReadDirectoryTemplate datascript module extracted information – the delimiter character and the file name pattern –from a variable named _expCurrentOperatorParameters. This variable refers to a string indexed Expressor Datascript table that contains the values of the various parameters of the extension operator. Depending on the properties created for the operator, the content of this table will change. In order to see what properties are documented in this table, use the pairs function to iterate through the table, displaying each element's index and value.

The most suitable place in the code to place this simple loop is in the read function (lines 86-88 below) just before the return statement (line 90) that shuts down the operator. Then run the dataflow and observe in the log the properties and their current values.

```
85   else
86     for k,v in pairs(_expCurrentOperatorParameters) do
87       log.notice(string.concatenate(k,"\t",v))
88     end
89     -- terminate processing
90     return expressor.ScriptSupport.FlowComplete
```

READ FIXED EXTENSION OPERATOR

They say that repetition is the way to learn, so let's create another extension operator, one that reads files with fixed width fields. This is another input operator that would expand the capabilities of Expressor.

Fortunately, developing this operator is quite similar to the Read Directory operator developed in the last exercise, so the effort should go quite smoothly.

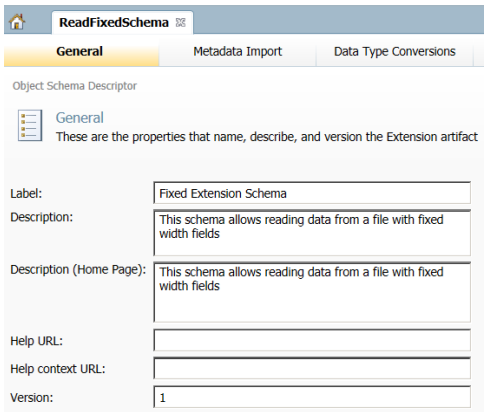
The library ReadFixed in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation. Use the files democrats.fix, republicans.fix, and whigs.fix as source files. These files **include** a header row and are in the file data.zip. The width of the fields are place, 6; lastname, 20; firstname, 15; party, 30.

To begin this process, create a new library named **ReadFixed**.

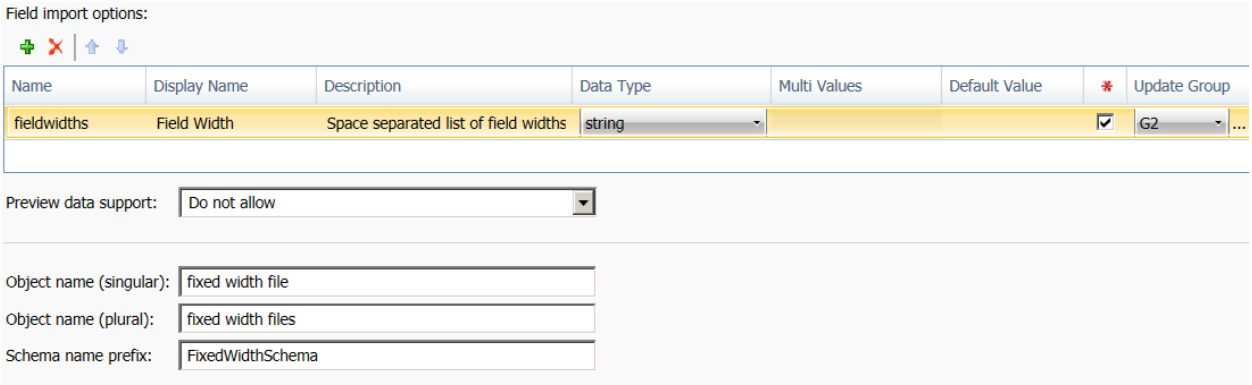
- Use the Extension Builder.
- Check **Read from a source** drop down and select **With Connection and Schema (This will need a Read Operator, Connection and Schema)**.
    - Click **Next**.
- In the following window,
    - Enter **ReadFixed** for the Name in the Read Operator Descriptor (Schema & Connection) grouping.
    - Select Create New and enter **ReadFixedSchema** for the Name in the Object Schema Descriptor grouping.
    - Select Create New, choose File Connection and enter **ReadFixedConnection** for the Name in the Connection Descriptor grouping.
    - Click **Next**.
- In the following window,
    - Select Create New and enter **ReadFixed_DSM** for the Name in the Read Operator Descriptor Datascript Module grouping.
    - Select Create New and enter **ReadFixedTemplate** for the Name in the Read Operator Descriptor Template Code Datascript Module grouping.
    - Select Create New and enter **ReadFixedSchema_DSM** for the Name in the Object Schema Descriptor Datascript Module.
    - Select Create New and enter **ReadFixedConnection_DSM** for the Name in the Custom Connection Descriptor Datascript Module.
    - Click **Next**.
- In the last window, review the listing of artifacts and datascript modules.
    - Click **Finish**.
- Open the **ReadFixedConnection** artifact descriptor.
    - On the **Functions** tab, click on the button that opens the ReadFixedConnection_DSM file and provide implementations for OpenFunction and CloseFunction.
        - Copy the function implementations from the ReadDirectoryConnection datascript module in the ReadDirectory library.

```
module("ReadFixed.ReadFixedConnection_DSM", package.seeall)

function OpenFunction(paramList)
  session = {pathToFile=paramList.Path,fullFileName=paramList.sourceFileName}
  return 0,session
end

function CloseFunction(session)
    return 0
end
```

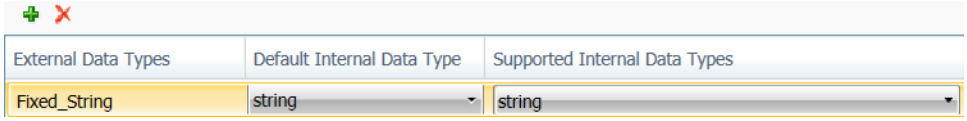
- Open the **ReadFixedSchema** artifact descriptor.
    - On the **General** tab, enter **Fixed Extension Schema** as the label and provide meaningful descriptions.



    - On the **Metadata Import** tab add an entry to the **Field import options** grouping to hold a listing of the field widths.
        - Make suitable entries into the Object name (singular), Object name (plural), and Schema name prefix controls.
        - Create an Update Group.



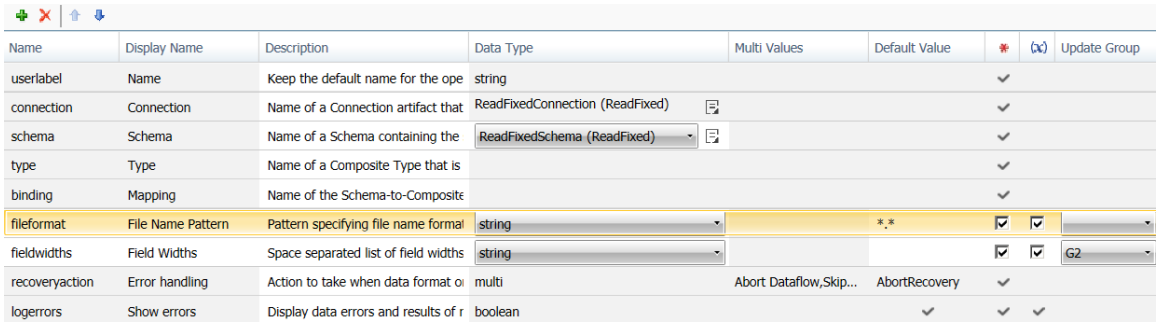
    - On the **Data Type Conversions** tab, add an entry for fixed width string fields (**Fixed_Width_String**).



    - Do not make any changes to the settings on the **Schema Editing** tab.
    - On the **Functions** tab, click on the button that opens the ReadFixedSchema_DSM file and provide implementations for GetObjectList and GetFieldList.

```lua
function GetObjectList(session, schemaImportOptions)
  return 0, {{name = session.fullFileName, description = 'The fixed width file'}}
end

function GetFieldList(session, objectNames, schemaImportOptions, context, constructionMetadata)
  local fieldwidths = string.concatenate(string.trim(schemaImportOptions[1].fieldwidths)," ")
  -- extract field widths
  local fields = {}
  for field in string.iterate(fieldwidths,"(%d+)%s+") do
    fields[#fields+1] = field
  end

  local schemas = {}
  local discoveredFields = {}
  --take a full file name saved in session in the OpenConnection function
  local fullFileName = session.fullFileName

  --open a file
  local file = io.open(fullFileName, 'r')
  if file ~= nil then
    --read the first line with a header:
      local headerLine = file:read("*l")
      if headerLine ~= nil then
        --split record by field widths
        headerColumns = {}
        for k,v in ipairs(fields) do
          headerColumns[#headerColumns+1] = string.trim(string.substring(headerLine,1,v))
          headerLine = string.substring(headerLine,v+1)
        end

        for index, value in ipairs(headerColumns) do
          --create a field that was discovered.
          --Note that all properties in all fields are the same excepting a field name.
          table.insert(discoveredFields,
                {
                  name = string.trim(value),
                  dataTypeName = 'Fixed_Width_String',
                  size = 0,
                  allownulls = false,
                  customizable = false,
                  fieldSpecial = { }
                }
                )
        end
        --create a single schema.
        local csvSchema = {
          sourceName = session.fullFileName,
          customizable = false,
          recordSpecial = {},
          fields = discoveredFields
        }
        --add schema to schemas set
        table.insert(schemas, csvSchema)
        --close a file
        file:close()
        --return success code and schemas  set
        return 0, schemas

      else
        return 2, "error reading header from the fixed width file"
      end
  else
    return 1, "Error: the selected file cannot be opened"
  end
end
```

- Open the **Read Fixed** operator descriptor.
    - On the **General** tab, enter Read Fixed as the label and ensure that **ReadFixedTemplate** is the selection in the Template datascript module control.
    - On the **Properties** tab,
        - Select **ReadFixedSchema** in the Data Type drop down control for the schema property.
            - The ReadFixedConnection entry appears for the type property.
        - Create entries for the fileformat and fieldwidths properties.



Name	Display Name	Description	Data Type	Multi Values	Default Value	*	(x)	Update Group
userlabel	Name	Keep the default name for the ope	string			✓		
connection	Connection	Name of a Connection artifact that	ReadFixedConnection (ReadFixed)			✓		
schema	Schema	Name of a Schema containing the	ReadFixedSchema (ReadFixed)			✓		
type	Type	Name of a Composite Type that is				✓		
binding	Mapping	Name of the Schema-to-Composite				✓		
fileformat	File Name Pattern	Pattern specifying file name format	string		*.*	☑	☑	
fieldwidths	Field Widths	Space separated list of field widths	string			☑	☑	G2
recoveryaction	Error handling	Action to take when data format o	multi	Abort Dataflow,Skip...	AbortRecovery	✓		
logerrors	Show errors	Display data errors and results of r	boolean		✓	✓	✓	

    - On the **Functions** tab, confirm that the ReadFixed_DSM is identified as the code location in the DSM Function Used column for all four functions.
        - Click on the  button adjacent to the ValidateOperator function to open this datascript module and add the following code.

```
function ValidateOperator(mappingSet, propertyList, recordSpecial, query)
  messages = {}
  if (is.null(propertyList.fileformat)
    or is.empty(string.trim(propertyList.fileformat.value))) then
      table.insert(messages,{type='Error',message='You need to specify a file name format'})
  end
  if (is.null(propertyList.fieldwidths)
    or is.empty(string.trim(propertyList.fieldwidths.value))) then
      table.insert(messages,{type='Warning',message='You need to specify field widths'})
  end
  if #messages>0 then return 1,messages else return 0 end
end
```

- Open the **ReadFixedTemplate** datascript module and add the following code.

```
require "expressor.ScriptSupport"

files = {}
directory = string.concatenate(
  _expCurrentOperatorParameters.Path ,"\\")
file_list = ""
file_handle = nil
header = {}
fieldwidths = string.concatenate(string.trim(
  _expCurrentOperatorParameters.fieldwidths)," ")
fields = {}

function initialize()
  -- obtain the file name format property value
  file_pattern = _expCurrentOperatorParameters.fileformat
  -- modify pattern to match Datascript/Lua pattern matching syntax
  characters = {}
```

```
  for i=1,#file_pattern do
    character = string.substring(file_pattern,i,i)
    if character ==". " then
      characters[i] = "%."
    elseif character == "*" then
      characters[i] = ".+"
    else
      characters[i] = character
    end
  end
  file_pattern = table.concat(characters,"")
  file_pattern = string.concatenate("^",file_pattern)

  file_handle = io.popen(string.concatenate("dir /b ", directory))
  for value in file_handle:lines() do
    file = string.match(value,file_pattern)
    if file then
      files[#files+1] = string.concatenate(directory,value)
    end
  end
  file_handle:close()
  file_list = table.concat(files," ")
  file_handle = io.popen(string.concatenate("type ",file_list))

  for field in string.iterate(fieldwidths,"(%d+)%s+") do
    fields[#fields+1] = field
  end

  headerLine = file_handle:read("*l")
  for k,v in ipairs(fields) do
    header[#header+1] = string.trim(string.substring(headerLine,1,v))
    headerLine = string.substring(headerLine,v+1)
  end
end

-- Compile substutution block

--__MAPPING_SET__
--__RECORD_SPECIAL__
--__EXTENSION_VERSION__

-- End of compile substitution block

function read()
  -- retrieve a line
  line = file_handle:read("*l")
  if line then
    pattern = string.concatenate("^",header[1])
    if is.null(string.match(line,pattern)) then
      output = {}
      values = {}
      for k,v in ipairs(fields) do
        values[#values+1] = string.trim(string.substring(line,1,v))
        line = string.substring(line,v+1)
      end

      for index,value in ipairs(values) do
        fieldName = string.trim(header[index])
        field = mappingSet.fields[fieldName]
        if is.null(field) then
          error(string.concatenate(
            "Field ",header[index]," cannot be found in the selected mapping set"))
        end
        output[field.attrName] = value
      end
      -- return output record
      return expressor.ScriptSupport.OK, output
    else
      return expressor.ScriptSupport.SkipRecord
    end
  else
    -- terminate processing
    return expressor.ScriptSupport.FlowComplete
  end
end

function finalize()
  file_handle:close()
end
```

Now when the dataflow is run, the entry in the File Name Pattern property determines which files in the source directory are processed. Remember, these files must have four fields corresponding to the attributes place, lastname, firstname, party, and **include a header row**. And the path to the source directory is obtained from the file connection artifact.

Once the code has been validated, create an operator template from the Read Fixed operator on the dataflow. In the future, use this template rather than the Read Fixed icon to add this extension operator to another dataflow

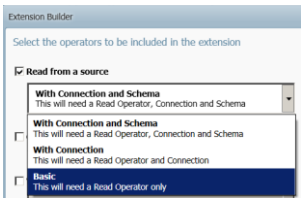
The library ReadFixed in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation. Use the files democrats.fix, republicans.fix, and whigs.fix as fixed width data files. These files **include** a header row and are in the file data.zip. The field widths are 6, 20, 15, and 30.

READ FTP EXTENSION OPERATOR

The ability to read a file from an FTP server is another feature that Expressor 3.9 does not include. No problem, an extension operator can provide this functionality.

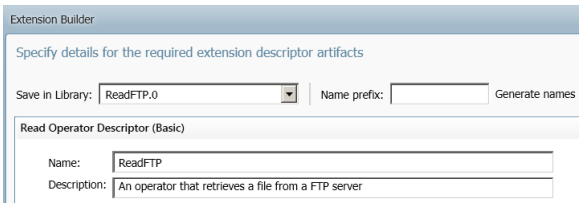
This implementation will be derived using the Basic – Read from a source approach, so a connection and schema will not be necessary. Once the file has been retrieved, the dataflow can use a transform operator to parse each line into its constituent fields.

Create a library named **ReadFTP**.



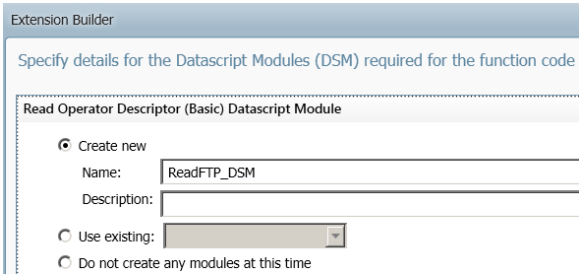
In the Extension Builder wizard,

- Check **Read from a source** drop down and select **Basic (This will need a Read Operator only)** from the dropdown menu.
- Click **Next**.



In the following window,

- Give the extension a name – **ReadFTP** – and provide a brief description.
- Click **Next**.



And in the next window

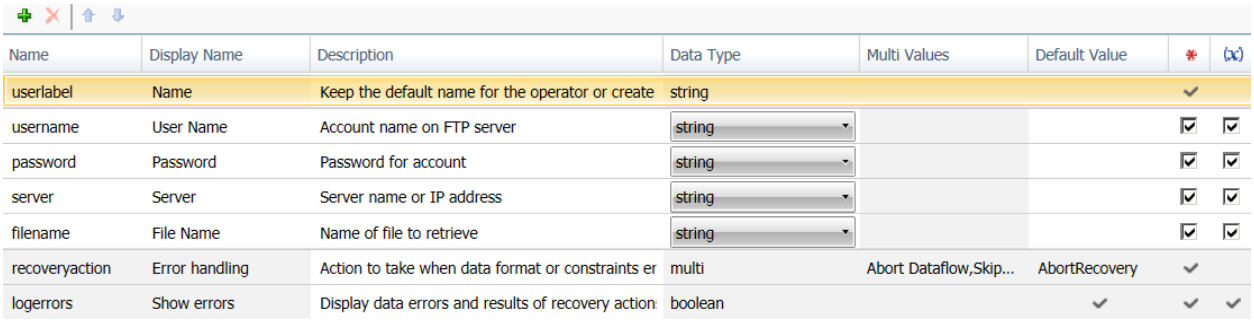
- Indicate that the associated code should be included in a new datascript module – **ReadFTP_DSM**.
- Click **Next** and **Finish** to close the wizard.

To complete the development process, additional operator properties need to be defined in the ReadFTP operator descriptor, the ValidateOperator function needs to be implemented in the ReadFTP_DSM datascript module, and code needs to be added to an instance of the Read FTP operator, which is then converted into an operator template.

In the Desktop Explorer, open the ReadFTP operator descriptor.

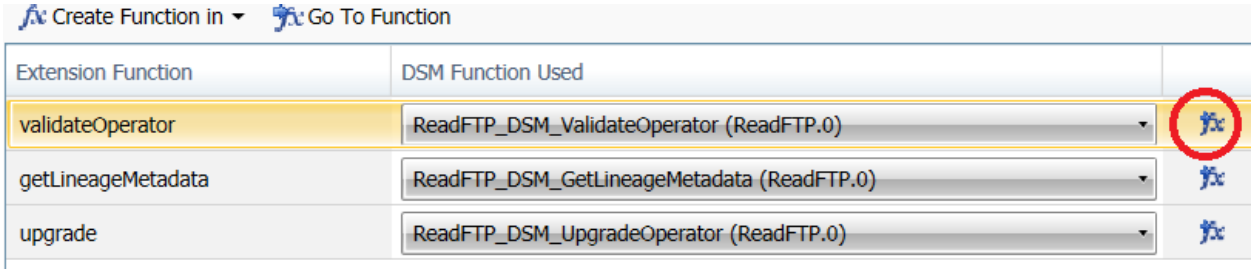
On the **General** tab, enter **Read FTP** as the label (note the space character).

On the **Properties** tab, add properties for the account credentials (username and password), the server name or IP address, and the name of the file to retrieve. All these properties are required strings and parameterizable.



Name	Display Name	Description	Data Type	Multi Values	Default Value	*	(x)
userlabel	Name	Keep the default name for the operator or create	string			✓	
username	User Name	Account name on FTP server	string			☑	☑
password	Password	Password for account	string			☑	☑
server	Server	Server name or IP address	string			☑	☑
filename	File Name	Name of file to retrieve	string			☑	☑
recoveryaction	Error handling	Action to take when data format or constraints er	multi	Abort Dataflow,Skip...	AbortRecovery	✓	
logerrors	Show errors	Display data errors and results of recovery action	boolean		✓	✓	✓

On the **Functions** tab, note that the datascript module ReadFTP_DSM already includes boiler plate implementations for the three required functions.



Click on the button to open the datascript module and add the following code to the validateOperator function.

```
function ValidateOperator(propertyList)
  messages = {}
  if (is.null(propertyList.server) or is.empty(string.trim(propertyList.server.value))) then
    messages[#messages+1] = {type='Warning',message='Server name or IP address is required'}
  end
  if (is.null(propertyList.username) or is.empty(string.trim(propertyList.username.value))) then
    messages[#messages+1] = {type='Warning',message='User name is required'}
  end
  if (is.null(propertyList.password) or is.empty(string.trim(propertyList.password.value))) then
    messages[#messages+1] = {type='Warning',message='User password is required'}
  end
  if (is.null(propertyList.filename) or is.empty(string.trim(propertyList.filename.value))) then
    messages[#messages+1] = {type='Warning',message='Must supply name of file to retrieve'}
  end

  if #messages>0 then return 1, messages else return 0 end

end
```

Create a dataflow and drag an instance of the Read FTP operator onto the design panel. Open the operator's rules editor and add the following code and a string output attribute named line.

Note that the code uses a datascript module named dscurl, which is an undocumented and unsupported feature within Expressor. Consequently, you will not find any coverage of this module in the product documentation.

Read FTP

```
flag = true
curl = require "dscurl"
results = {}
c = nil

function setCurl()
  results = {}
  c=curl.new()
  credentials = string.concatenate(
    _expCurrentOperatorParameters.username,":",_expCurrentOperatorParameters.password)
  c:setopt(curl.OPT_USERPWD,credentials)
  c:setopt(curl.OPT_WRITEDATA,results)
  c:setopt(curl.OPT_WRITEFUNCTION,
    function(results,buffer)
      table.insert(results,buffer)
      return #buffer
      end)
end

function initialize()
  setCurl()
end


function read()
  output = {}
  if flag then
    flag=false
    url = string.concatenate(
      "ftp://",_expCurrentOperatorParameters.server,"/",_expCurrentOperatorParameters.filename)
    c:setopt(curl.OPT_URL,url)
    c:perform()

    output.line=table.concat(results)
    return output
  else
    return true
  end
end

function finalize()
  c:close()
end
```

To confirm that the operator is successfully retrieving the desired file, connect the Read FTP extension operator to a Write File operator and write the downloaded content to a text file. Once the operator's functionality has been confirmed, create a template.

In an actual application, use the template rather than the Read FTO operator then place a transform operator immediately downstream of the extension operator and parse each line from the file into its individual values.

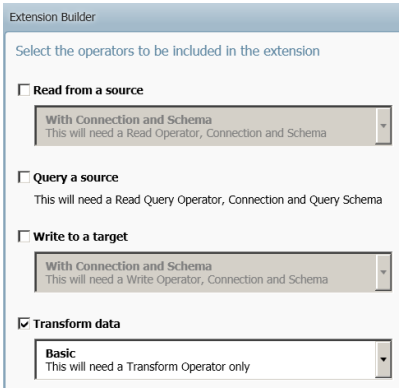
The library ReadFTP in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation.

BASIC – TRANSFORM DATA OPERATOR

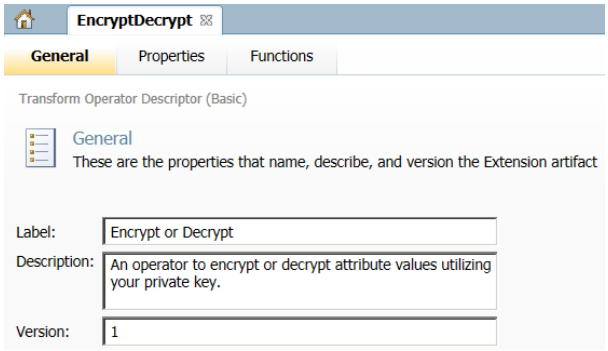
The SDK also supports development of extension Transform operators. This section describes the development of an extension operator that makes the Expressor encryption and decryption functions (utility.custom_encrypt and utility.custom_decrypt) easy to use. The necessary configuration details are coded into the extension operator and properties are used to enter the private key and specify the type of encryption paradigm and key size. This implementation assumes that the private key is always provided as a base64 encoded character string of a key encrypted with the utility.encrypt_custom_key function.

To begin this process, create a new library named **EncryptDecrypt**.

- Start the Extension Builder.
- Check **Transform data** drop down and select **Basic (This will need a Transform Operator only)** from the dropdown menu.



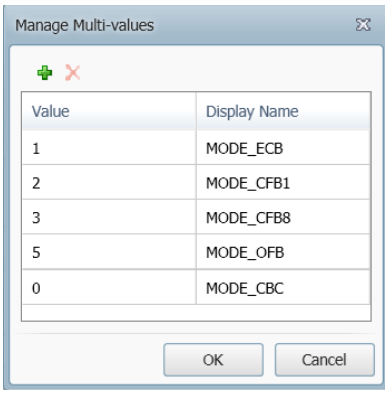
- Click **Next**.
- On the next page, name the operator **EncryptDecrypt** and provide a meaningful description.
- Click **Next**.
- On the following page, accept the suggested name for the new datascript module, then click **Next** and **Finish** to complete the process.
- Open the operator descriptor named EncryptDecrypt.
    - On the **General** tab, enter **Encrypt or Decrypt** into the label control and a description.

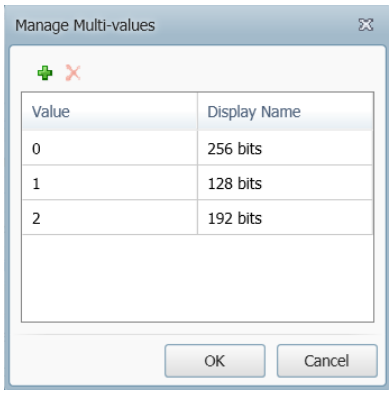


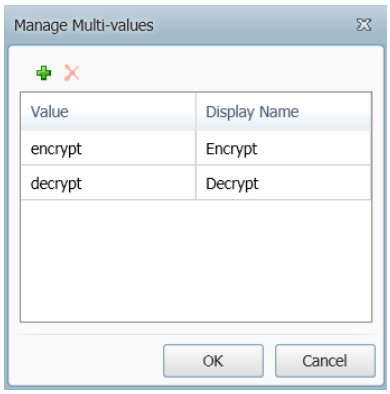
    - On the Properties tab, add five new properties: privatekey, iv, mode, keysize, and encryptdecrypt.
        - Note that these added properties are parameterizable.

Name	Display Name	Description	Data Type	Multi Values	Default Value	*	(x)
userlabel	Name	Keep the default name for the ope	string			✓	
privatekey	Encrypted Private Key	Your private key, encrypted, base	string			☑	☑
iv	Initialization Vector	Base 64 encoded	string			☐	☑
mode	Mode	Block cipher mode of action	multi	MODE_ECB,MOD...	MODE_ECB		☑
keysize	Key Size	Bit size of private key	multi	256 bits,128 bits,...	256 bits		☑
encryptdecrypt	Encrypt or Decrypt	Whether to encrypt or decrypt attr	multi	Encrypt,Decrypt	Encrypt		☑
recoveryaction	Error handling	Action to take when data format o	multi	Abort Dataflow,Skip...	AbortRecovery	✓	✓
logerrors	Show errors	Display data errors and results of r	boolean		✓	✓	✓
partitionCount	Partition count	The number of parallel streams int			0	✓	

- Set up the three multi controls as shown in the following screen shots.



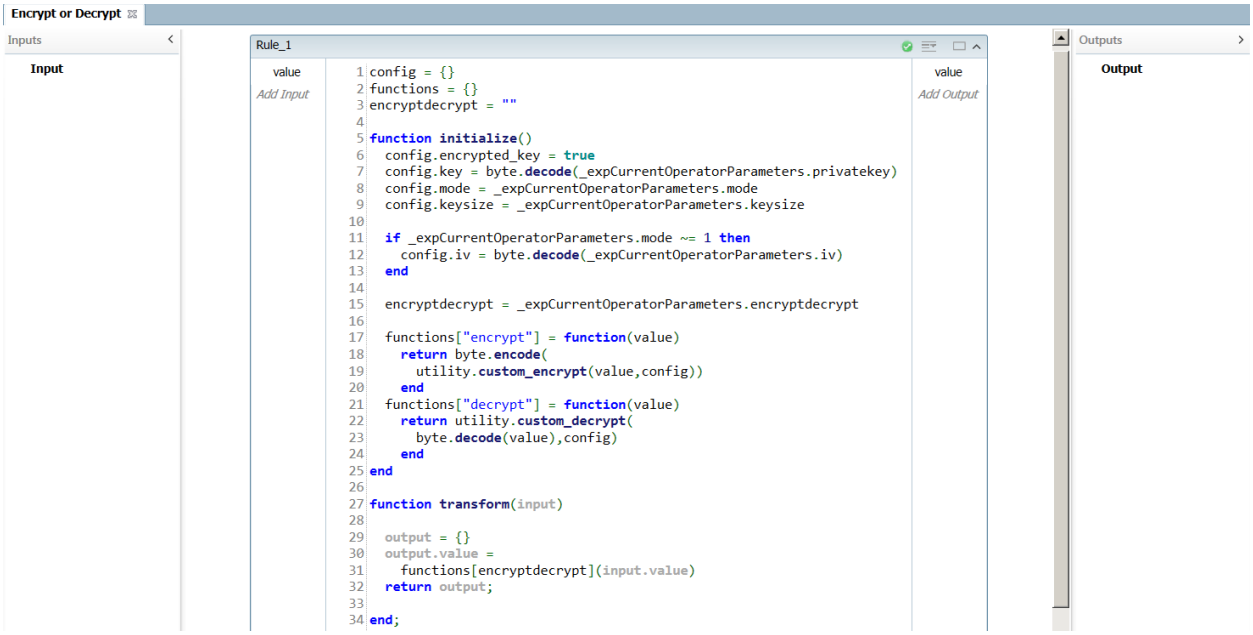




  - On the Functions tab, open the datascript module to the validateOperator function and enter the following code.

```
function ValidateOperator(propertyList)

  local messages = {}
  if (is.null(propertyList.privatekey)
    or is.empty(string.trim(propertyList.privatekey.value)))then
      table.insert(messages,{type='Error',message='You must supply an encrypted, base64 encoded private key'})
  end

  if #messages>0 then return 1, messages else return 0 end

end
```

  - Save both the operator descriptor and the datascript module.
- Create a dataflow and drag the **Encrypt or Decrypt** operator onto the canvas.
  - Open this operator's rules editor.
    - Add input and output parameters named **value**.
    - Enter the following code.
      - The input and output parameters – value – are string types.



        - Note that it is not necessary to specify input and output attributes. These will appear when the operator is connected to upstream and downstream operators.
- To test the operator, read some data from a file and encrypt one of the values, writing the encrypted value to a file.
    - Then create a second dataflow that reads the file emitted by the first dataflow and decrypts the encrypted value.
        - Again write to an output file and confirm that the encrypted value has been accurately decrypted.
- Once fully tested, disconnect the operator from the Read File and Write File operators and save as a template in the EncryptDecrypt library.

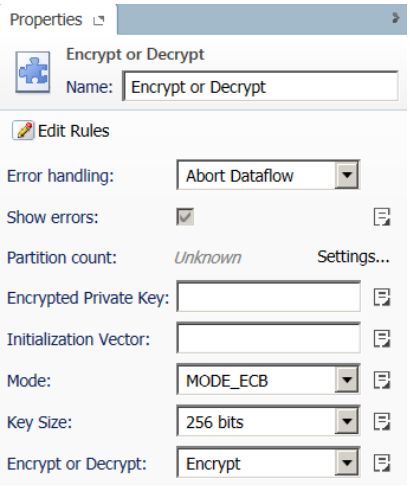
When you use this extension operator template in a dataflow, you can add additional input and output parameters if it is necessary to encrypt or decrypt multiple values. Then duplicate the code in lines 30-31 as appropriate.

```
27 function transform(input)
28
29   output = {}
30   output.value =
31     functions[encryptdecrypt](input.value)
32   output.value2 =
33     functions[encryptdecrypt](input.value2)
34   return output;
35
36 end;
```

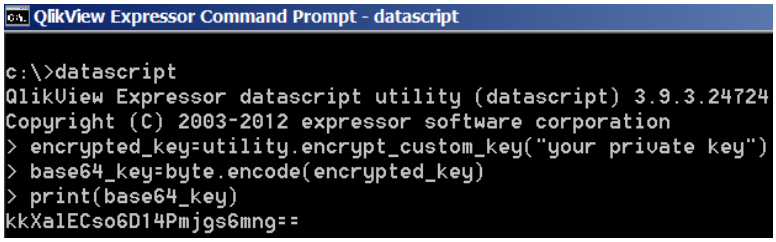
To understand how the Expressor encryption and decryption functions work, read the knowledge base article Using the QlikView Expressor Encryption & Decryption Functionality (http://community.qlikview.com/docs/DOC-3359).

In this implementation, the initialize function is used to extract the required encryption information from the entries made into the operator's Property sheet controls (lines 6-15). A string indexed Datascript table – functions – is then created to hold the actual invocations of the utility.custom_encrypt and utility.custom_decrypt functions (lines 17-24).

In the transform function, note how the keys in this Datascript table (encrypt and decrypt) are used to select the appropriate function call (line 30).



Examine the code above and note that the private key must be encrypted using the Expressor utility.encrypt_custom_key function and then, after encryption, converted into a base64 representation with the byte.encode function. This is a simple two statement process that is executed from within an Expressor command window using the datascript utility. Copy/paste the encoded encrypted key into the Properties panel control.



If the initialization vector entry is needed (depends on the choice of mode), then this value must also be converted into a base64 representation (but do not encrypt) using the byte.encode function.

Base64 representations are necessary as many keys and initialization vectors contain unprintable characters, which would make it impossible to enter as text into the controls on the Properties panel.

The library EncryptDecrypt in the Expressor Desktop export file extension_tutorial.zip includes this extension operator implementation.

WAIT, THERE'S MORE

In the UsePropertyConnectionSchema, ReadDirectory, and ReadFixed operator extensions, the schema was created by interrogating a representative source file and the field names extracted from the header row. As part of this process, on the artifact descriptor's Data Type Conversions tab an external data type was defined. This type could have any name. In the UsePropertyConnectionSchema and ReadDirectory extensions, Delimited_String was specified as the type's name; in the Read Fixed extension, Fixed_Width_String was specified. In all three examples, the corresponding default internal data type was string and the supported internal data types listing included only string.

All of these operators will read each value as a string and the data type of the corresponding attribute in the schema will be string. But the primary distinguishing characteristic of an Expressor schema is the ability to change the structure of the composite type by either assigning an existing composite type or by assigning a shared atomic type to an attribute. In either case, the data type of an attribute may change. For example, an employee identifier's data type might be changed from string to integer.

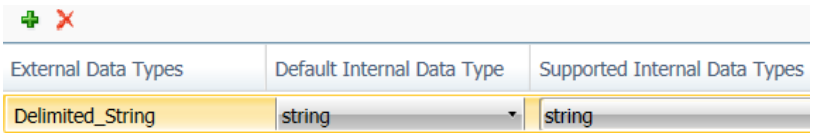
Open one of the schemas from these examples and observe that the mapping between a field and its corresponding attribute cannot be edited. Since the mapping is critical to informing the Expressor engine how to convert a string data type to another type, it appears as if the ability to alter attribute types is compromised when dealing with extension schemas. But this is not so. The type conversions are simply performed within the operator's code.

There are actually two approaches to this task. The first approach allows the structure of the header row and the field names to remain unknown to the developer, while the second approach depends on knowledge of the data in the file.

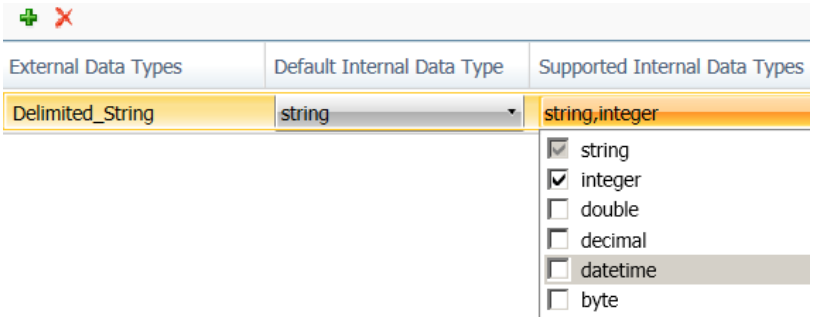
APPROACH ONE

Let's use the UsePropertyConnectionSchema extension to investigate this approach.

- Open the UsePropertyConnectionSchema artifact descriptor and select the Data Type Conversions tab.



  - Click in the Supported Internal Data Types drop down control and check integer.



  - This adds a second supported type to the listing.

In the extension schema's composite type, either string or integer may now be selected as an attribute's type.

- Now open the datascript module – ReadOperatorTemplate – that contains the operator's code.
  - Scroll down to the following statement.

```
output[field.attrName] = value
```

  - Replace with the following code.

```
if field.attrType=="integer"
  then output[field.attrName] = tointeger(value)
else
  output[field.attrName] = value
end
```

  - The conditional clause determines what data type has been assigned to the schema attribute and:
    - If the type is integer, the value is converted into an integer and assigned to the corresponding output field.
    - If the type is string, the value does not need to be converted and can be directly assigned.

OK. Integers are simple enough, but what about a value that requires transformation before it can be converted and assigned? For example: a datetime.

In this situation, the coding must use the string.datetime function to transform the string representation into a datetime type and the modified code would be similar to the following, where format describes how the date is written (for example, MM/DD/CCYY).

```
if field.attrType=="datetime"
  then output[field.attrName] = string.datetime(value,"format")
else
  output[field.attrName] = value
end
```

Similarly, a value to be represented as a decimal might need editing (possibly removing the dollar sign) before its type could be converted.

The point to note is that the code must be prepared to handle each record value's conversion to a different data type.

APPROACH TWO

To illustrate this approach, use the ReadDirectory extension.

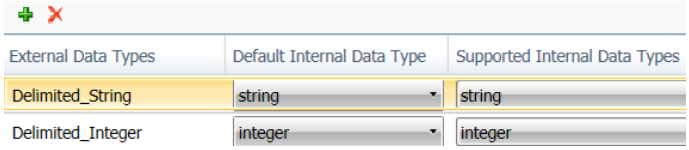
So far, only one Data Type Conversion entry has been added to the artifact descriptor – Delimited_String. But additional types could be added, for example, a Delimited_Integer or Delimited_Datetime entry. But to utilize these additional types requires adding more information to the schema artifact descriptor and editing the code in the datascript module associated with this operator extension.

In addition, a file containing the metadata description of the records being processed needs to be created and placed into a file system location from which it can be easily read.

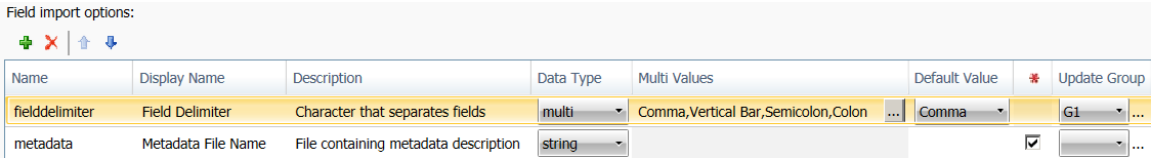
- Create a file that describes the metadata (that is the data type) of each field in the records being processed.
    - The most appropriate file system location in which to locate this file is in the same directory with the file(s) to be processed.
    - Add the following content to this file.
        - Note how each row sets the external data type for a field.
            - The arrangement of square braces and equal signs is important; the order of entries is not important.

```
{
[ [=[party]=] ] = [=[Delimited_String]=],
[ [=[firstname]=] ] = [=[Delimited_String]=],
[ [=[lastname]=] ] = [=[Delimited_String]=],
[ [=[place]=] ] = [=[Delimited_Integer]=]
}
```

        - Save the file with a meaningful name, for example, **metadata.data**.
            - It is best to be certain that the extension for this file is not one typically used for data files.
- Open the artifact descriptor.
    - Select the **Data Type Conversions** tab.
        - Add an external data type representing integer values.



    - Select the **Metadata Import** tab.
        - Add a required field import option that will hold the name of a file that contains the metadata description of the record being processed.



- Open the datascript module ReadDirectorySchema.
    - In the GetFieldList function, scroll down to the code that initializes the Datascript table discoveredFields.
        - Note that each field is represented by a nested Datascript table that contains the name of the external data type.

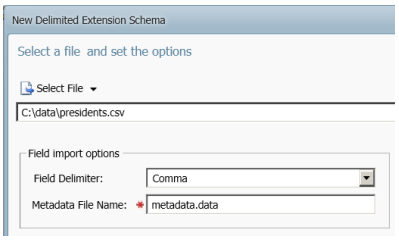
```
for index, value in ipairs(headerColumns) do
  table.insert(discoveredFields,
        {
          name = string.trim(value),
          dataTypeName = 'Delimited_String',
          size = 0,
          allownulls = false,
          customizable = false,
          fieldSpecial = { }
        }
  )
end
```

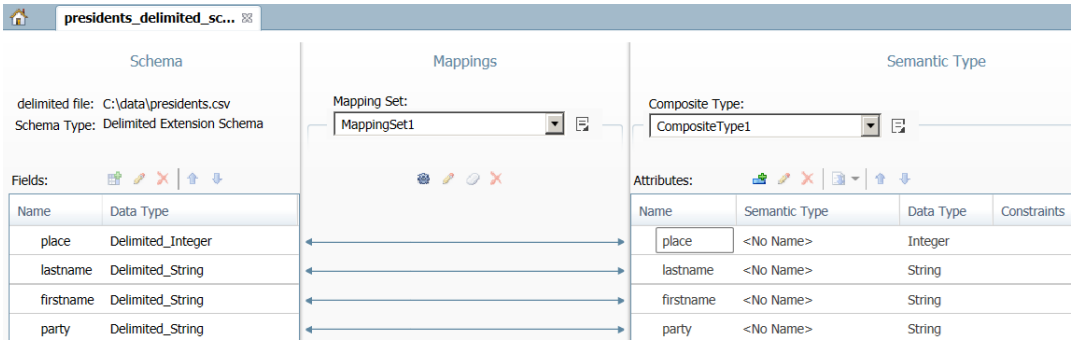
- While the table headerColumns contains an entry for each header name, the code doesn't know what these values are. But assuming that the header names and the data types are known to the developer who can prepare the metadata file, the code can be edited such that the appropriate type can be assigned.[1]

- In the GetFieldList function, construct a Datascript table that holds the metadata description of the records being processed.[2]

```
require "expressor.tableutils"
metadata_file = string.concatenate(
  session.pathToFile,"\\",schemaImportOptions[1].metadata)
io.input(metadata_file)
md = io.read("*a")
io.close()

metadata = expressor.tableutils.deserialize(md)

for index, value in ipairs(headerColumns) do
  value = string.trim(value)
  table.insert(discoveredFields,
        {
          name = value,
          dataTypeName = metadata[value],
          size = 0,
          allownulls = false,
          customizable = false,
          fieldSpecial = { }
        }
  )
end
```

  - In this example, the header field name is used to set the data type to Delimited_Integer rather than Delimited_String. In the resulting schema, the type of the corresponding attribute will be integer.





[1] This approach is only an example to illustrate the concept.

[2] See the knowledge base article (http://community.qlikview.com/docs/DOC-3252) for details of the expressor.tableutils datascript module.

AND, STILL MORE

What about constraints? Can they be used in an extension schema? The answer is yes. Although some constraint specifications could be set as part of the code that initializes the Datascript table discoveredFields, for example, by allowing or disallowing null values or setting the maximum length of a string value, the easier approach is to simply associate constraints with the schema attributes as with any other schema.

What happens if a record can't be processed? For example, a value that should be converted into a datetime is corrupted and the string.datetime function call fails. When this happens with the Read File or Read Table operator, an error condition is raised, which is handled as specified by the operator's Error handling property. Well, the same approach works with an extension operator. Simply set the operator's Error handling property and skip or reject the record as appropriate.