Apache NiFi 3

Apache NiFi Expression Language Guide

Date of Publish: 2018-12-18

https://docs.hortonworks.com/

Contents

Overview

All data in Apache NiFi is represented by an abstraction called a FlowFile. A FlowFile is comprised of two major pieces: content and attributes. The content portion of the FlowFile represents the data on which to operate. For instance, if a file is picked up from a local file system using the GetFile Processor, the contents of the file will become the contents of the FlowFile.

The attributes portion of the FlowFile represents information about the data itself, or metadata. Attributes are key-value pairs that represent what is known about the data as well as information that is useful for routing and processing the data appropriately. Keeping with the example of a file that is picked up from a local file system, the FlowFile would have an attribute called filename that reflected the name of the file on the file system. Additionally, the FlowFile will have a path attribute that reflects the directory on the file system that this file lived in. The FlowFile will also have an attribute named uuid, which is a unique identifier for this FlowFile.

However, placing these attributes on a FlowFile do not provide much benefit if the user is unable to make use of them. The NiFi Expression Language provides the ability to reference these attributes, compare them to other values, and manipulate their values.

Structure of a NiFi Expression

The NiFi Expression Language always begins with the start delimiter ${ and ends with the end delimiter }. Between the start and end delimiters is the text of the Expression itself. In its most basic form, the Expression can consist of just an attribute name. For example, ${filename} will return the value of the filename attribute.

In a slightly more complex example, we can instead return a manipulation of this value. We can, for example, return an all upper-case version of the filename by calling the toUpper function: ${filename:toUpper()}. In this case, we reference the filename attribute and then manipulate this value by using the toUpper function. A function call consists of 5 elements. First, there is a function call delimiter :. Second is the name of the function - in this case, toUpper. Next is an open parenthesis (), followed by the function arguments. The arguments necessary are dependent upon which function is being called. In this example, we are using the toUpper function, which does not have any arguments, so this element is omitted. Finally, the closing parenthesis ()) indicates the end of the function call. There are many different functions that are supported by the Expression Language to achieve many different goals. Some functions provide String (text) manipulation, such as the toUpper function. Others, such as the equals and matches functions, provide comparison functionality. Functions also exist for manipulating dates and times and for performing mathematical operations. Each of these functions is described below, in the Fuctions section, with an explanation of what the function does, the arguments that it requires, and the type of information that it returns.

When we perform a function call on an attribute, as above, we refer to the attribute as the subject of the function, as the attribute is the entity on which the function is operating. We can then chain together multiple function calls, where the return value of the first function becomes the subject of the second function and its return value becomes the subject of the third function and so on. Continuing with our example, we can chain together multiple functions by using the expression ${filename:toUpper():equals('HELLO.TXT')}. There is no limit to the number of functions that can be chained together.

Any FlowFile attribute can be referenced using the Expression Language. However, if the attribute name contains a "special character", the attribute name must be escaped by quoting it. The following characters are each considered "special characters":

- $ (dollar sign)
- | (pipe)
- { (open brace)
- } (close brace)
- ((open parenthesis)

-) (close parenthesis)
- [(open bracket)
-] (close bracket)
- , (comma)
- : (colon)
- ; (semicolon)
- / (forward slash)
- * (asterisk)
- ' (single quote)
- (space)
- \t (tab)
- \r (carriage return)
- \n (new-line)

Additionally, a number is considered a "special character" if it is the first character of the attribute name. If any of these special characters is present in an attribute is quoted by using either single or double quotes. The Expression Language allows single quotes and double quotes to be used interchangeably. For example, the following can be used to escape an attribute named my attribute: ${"my attribute"} or ${'my attribute'}.

In this example, the value to be returned is the value of the "my attribute" value, if it exists. If that attribute does not exist, the Expression Language will then look for a System Environment Variable named "my attribute." If unable to find this, it will look for a JVM System Property named "my attribute." Finally, if none of these exists, the Expression Language will return a null value.

There also exist some functions that expect to have no subject. These functions are invoked simply by calling the function at the beginning of the Expression, such as ${hostname()}. These functions can then be changed together, as well. For example, ${hostname():toUpper()}. Attempting to evaluate the function with subject will result in an error. In the Functions section below, these functions will clearly indicate in their descriptions that they do not require a subject.

Often times, we will need to compare the values of two different attributes to each other. We are able to accomplish this by using embedded Expressions. We can, for example, check if the filename attribute is the same as the uuid attribute: ${filename:equals(${uuid})}. Notice here, also, that we have a space between the opening parenthesis for the equals method and the embedded Expression. This is not necessary and does not affect how the Expression is evaluated in any way. Rather, it is intended to make the Expression easier to read. White space is ignored by the Expression Language between delimiters. Therefore, we can use the Expression ${ filename : equals(${ uuid}) } or ${filename:equals(${uuid})} and both Expressions mean the same thing. We cannot, however, use ${file name:equals(${uuid})}, because this results in file and name being interpreted as different tokens, rather than a single token, filename.

Expression Language in the Application

The Expression Language is used heavily throughout the NiFi application for configuring Processor properties. Not all Processor properties support the Expression Language, however. Whether or not a Property supports the Expression Language is determined by the developer of the Processor when the Processor is written. However, the application strives to clearly illustrate for each Property whether or not the Expression Language is supported.

In the application, when configuring a component property, the User Interface provides an Information icon () next to the name of the Property. Hovering over this icon with the mouse will provide a tooltip that provides helpful information about the Property. This information includes a description of the Property, the default value (if any), historically configured values (if any), and the evaluation scope of this property for expression language. There are three values and the evaluation scope of the expression language is hierarchical: NONE # VARIABLE_REGISTRY # FLOWFILE_ATTRIBUTES.

- NONE - expression language is not supported for this property

- VARIABLE_REGISTRY is hierarchically constructed as below:
  - Variables defined at process group level and then, recursively, up to the higher process group until the root process group.
  - Variables defined in custom properties files through the nifi.variable.registry.properties property in nifi.properties file.
  - Environment variables defined at JVM level and system properties.
- FLOWFILE_ATTRIBUTES - will use attributes of each individual flow file, as well as those variables defined by the Variable Registry, as described above.

Escaping Expression Language

There may be times when a property supports Expression Language, but the user wishes to use a literal value that follows the same syntax as the Expression Language. For example, a user may want to configure a property value to be the literal text Hello ${UserName}. In such a case, this can be accomplished by using an extra $ (dollar sign symbol) just before the expression to escape it (i.e., Hello $${UserName}). Unless the $ character is being used to escape an Expression, it should not be escaped. For example, the value Hello $$User$$Name should not escape the $ characters, so the literal value that will be used is Hello $$User$$Name.

If more than two $ characters are encountered sequentially before a {, then each pair of $ characters will be considered an escaping of the $ character. The escaping will be performed from left-to-right. To help illustrate this, consider that the variable abc contains the value xyz. Then, consider the following table of Expressions and their corresponding evaluated values:

Expression	Value	Notes
${abc}	xyz	
$${abc}	${abc}	
$$${abc}	$xyz	
$$$${abc}	$${abc}	
$$$$${abc}	$$xyz	
I owe you $5	I owe you $5	No actual Expression is present here.
You owe me $$5 too	You owe me $$5 too	The $ character is not escaped because it does not immediately precede an Expression.
Unescaped $$${5 because no closing brace	Unescaped $$${5 because no closing brace	Because there is no closing brace here, there is no actual Expression and hence the $ characters are not escaped.
Unescaped $$${5} because no closing brace	<Error>	This expression is not valid because it equates to an escaped $, followed by ${5} and the ${5} is not a valid Expression. The number must be escaped.
Unescaped $$${'5'} because no closing brace	Unescaped $ because no closing brace	There is no attribute named 5 so the Expression evaluates to an empty string. The $$ evaluates to a single (escaped) $ because it immediately precedes an Expression.

Expression Language Editor

When configuring the value of a Processor property, the NiFi User Interface provides help with the Expression Language using the Expression Language editor. Once an Expression is begin by typing ${, the editor begins to

highlight parentheses and braces so that the user is easily able to tell which opening parenthesis or brace matches which closing parenthesis or brace.

The editor also supplies context-sensitive help by providing a list of all functions that can be used at the current cursor position. To activate this feature, press Ctrl+Space on the keyboard. The user is also able to type part of a function name and then press Ctrl+Space to see all functions that can be used that start with the same prefix. For example, if we type into the editor ${filename:to and then press Ctrl+Space, we are provided a pop-up that lists six different functions: toDate, toLower, toNumber, toRadix, toString, and toUpper. We can then continue typing to narrow which functions are shown, or we can select one of the functions from the list by double-clicking it with the mouse or using the arrow keys to highlight the desired function and pressing Enter.

Functions

Functions provide a convenient way to manipulate and compare values of attributes. The Expression Language provides many different functions to meet the needs of a automated dataflow. Each function takes zero or more arguments and returns a single value. These functions can then be chained together to create powerful Expressions to evaluate conditions and manipulate values.

Data Types

Each argument to a function and each value returned from a function has a specific data type. The Expression Language supports four different data types:

- String: A String is a sequence of characters that can consist of numbers, letters, white space, and special characters.
- Number: A Number is an whole number comprised of one or more digits (0 through 9). When converting to numbers from Date data types, they are represented as the number of milliseconds since midnight GMT on January 1, 1970.
- Decimal: A Decimal is a numeric value that can support decimals and larger values with minimal loss of precision. More precisely it is a double-precision 64-bit IEEE 754 floating point. Due to this minimal loss of precision this data type should not be used for very precise values, such as currency. For more documentation on the range of values stored in this data type refer to this https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3. The following are some examples of the forms of literal decimals that are supported in expression language (the "E" can also be lower-case):
  - 1.1
  - .1E1
  - 1.11E-12
- Date: A Date is an object that holds a Date and Time. Utilizing the Dates and Type Cast functions this data type can be converted to/from Strings and numbers. If the whole Expression Language expression is evaluated to be a date then it will be converted to a String with the format: "<Day of Week> <Month> <Day of Month> <Hour>:<Minute>:<Second> <Time Zone> <Year>". Also expressed as "E MMM dd HH:mm:ss z yyyy" in Java SimpleDateFormat format. For example: "Wed Dec 31 12:00:04 UTC 2016".
- Boolean: A Boolean is one of either true or false.

After evaluating expression language functions, all attributes are stored as type String.

The Expression Language is generally able to automatically coerce a value of one data type to the appropriate data type for a function. However, functions do exist to manually coerce a value into a specific data type.

Hex values are supported for Number and Decimal types but they must be quoted and prepended with "0x" when being interpreted as literals. For example these two expressions are valid (without the quotes or "0x" the expression would fail to run properly):

- ${literal("0xF"):toNumber()}

- ${literal("0xF.Fp10"):toDecimal()}

Boolean Logic

One of the most powerful features of the Expression Language is the ability to compare an attribute value against some other value. This is used often, for example, to configure how a Processor should route data. The following functions are used for performing boolean logic, such as comparing two values. Each of these functions are designed to work on values of type Boolean.

String Manipulation

Each of the following functions manipulates a String in some way.

Encode/Decode Functions

Each of the following functions will encode a string according the rules of the given data format.

Searching

Each of the following functions is used to search its subject for some value.

Mathematical Operations and Numeric Manipulation

For those functions that support Decimal and Number (whole number) types, the return value type depends on the input types. If either the subject or argument are a Decimal then the result will be a Decimal. If both values are Numbers then the result will be a Number. This includes Divide. This is to preserve backwards compatibility and to not force rounding errors.

Date Manipulation

Type Coercion

Subjectless Functions

While the majority of functions in the Expression Language are called by using the syntax ${attributeName:function()}, there exist a few functions that are not expected to have subjects. In this case, the attribute name is not present. For example, the IP address of the machine can be obtained by using the Expression

${ip()}. All of the functions in this section are to be called without a subject. Attempting to call a subjectless function and provide it a subject will result in an error when validating the function.

Evaluating Multiple Attributes

When it becomes necessary to evaluate the same conditions against multiple attributes, this can be accomplished by means of the and and or functions. However, this quickly becomes tedious, error-prone, and difficult to maintain. For this reason, NiFi provides several functions for evaluating the same conditions against groups of attributes at the same time.