Variable Substitution

All triggers and actions contain properties that affect their behavior and execution. Substitution allows for some of these properties to be replaced at runtime with variables from the executing action, flow context, flow chart, or runtime configuration. This behavior allows for dynamic property loading and convenience when using variables on a global, flow chart, scale.

For example, suppose after detecting files with a File Exist Trigger, those files must be deleted. This behavior may be implemented by retrieving the results from the File Exist Trigger, mapping them to the flow context, then using that variable for the Simple File Source of a File Delete Action.

There are three types of substitution, Global, Runtime, and Date substitution. These types of substitution are all explained below in the proceeding sections.

NOTE: Substitution may not be used on any properties whose syntax is verified at build time including, but not limited to, Listeners and Join Expressions.

In this section:

Global Substitution

Global substitution allows you to access the value of a variable in an action property, from either the action, flow context, or flow chart variable manager.

Global substitution uses the following syntax:

```
${<variable name>}
```

Where "<variable name>" is the name of a variable in one of the supported variable managers.

When global substitution is used, the syntax above is replaced with the value of the variable at runtime of the flow chart – as though you had directly entered the variable's value when defining the flow chart. This allows your triggers, actions, and flow charts to easily use data that is dynamically generated at runtime.

This global substitution syntax can be used in any action or flow chart property in Flux that accepts one of the supported data types listed further below on this page.

Global substitution works in precedence order. The variable name the property uses is first searched for in the action's variable manager, then flow context, and finally the flow chart's variable manager. If no instance of the variable is found in these scopes, the property will be used in its present condition.

Concatenating Several Variables in an Action's Property

Since variables themselves can't contain references to other variables in their definition (i.e. /variable_name=SomeValue/${MyVariable}); there are a few ways to concatenate and combine several variables in one action/trigger field. The easiest way to accomplish this is to have all the variables defined (be it Runtime Variables, Workflow Variables or Flow Context Variables) and then calling them from the field you want the variables to be substituted. For example, let's assume you have a runtime variable called 'Variable1' that makes up part of a file path; and the rest of the path is held in a flow context variable called 'Variable2'. You can use those two variables to call the entire file path like so:

```
/${runtime Variable1}/${Variable2}
```

This will concatenate the values of the variables and add the slashes '/'; making it simple to combine values saved in the Runtime Configuration and Runtime Variables with values that are held in the Flow Context.

Accessing Variable Fields

If your variable is a complex data type (any Object whose Class is not listed above), and the variable contains a public field of one of the supported data types listed further below on this page, you can access that field directly using variable substitution. Like global substitution, this can be done with any variable in an action, flow context, or flow chart variable manager (and follows the same rules for precedence).

In order to access public fields on a variable directly, you can use the following syntax:

```
${<variable name>.<public field name>}
```

Where "<variable name>" is the name of the variable, and "<public field name>" the name of a field on that variable.

For example, consider the FileInfo class, a complex data type used in the Flux file triggers to store information about the files found by the trigger. This data type contains several public fields that might be useful to obtain using variable substitution, including "filename", "url", and "size".

In order to access these fields directly using variable substitution, you can use the following syntax (assuming that "fileInfoVariable" is the name of a variable of type *FileInfo* located in one of the supported variable managers):

filename:

```
${fileInfoVariable.filename}
```

url:

```
${fileInfoVariable.url}
```

size:

```
${fileInfoVariable.size}
```

Calling Methods on Variables

If your variable is an Object with publicly available methods that return one of the supported data types listed further below on this page, you can use variable substitution to call those methods. Like global substitution, this can be done with any variable in an action, flow context, or flow chart variable manager (and follows the same rules for precedence).

You can call a method and access its result using the following syntax:

```
${<variable name>.<methodSignature>}
```

Where "<variable name>" is the name of the variable in one of the supported variable managers, and "<methodSignature>" the signature of the method you are trying to call (and access the result).

For example, suppose you had a variable of the data type *MyClass*, and this class had a method *myMethod()* that returned a String. You can access the value that would be returned from *myMethod()* directly in your action or trigger property by using the following syntax (assuming that "myClassVariable" is the name of a variable of type *MyClass*):

```
${myClassVariable.myMethod()}
```

This would call the method *myMethod()*, then replace the syntax above with the return value of the method at runtime.

If / Else Statements

Variable Naming - Do not use ${} syntax in these examples

Note that in the following examples, and your own code, that you do not use the ${} around your variables in the condition of the if statement itself. If you are using flow context variables, simply use their name in the condition of the if/else statements. Usually in Flux actions you would enter a variable name into a script or text field as "${myvariable}". In if/else conditions you just use "myvariable".

You can use if statements within your action or triggers properties to allow conditional substitution based on the values of the variables in your flow chart / flow context.

This provides powerful logic directly within your action and trigger properties – in essence, this allows you to say "if <condition A> matches, then this property should have <value A>, and if <condition B> matches, then this property should have <value B>."

You can also use else or even elseif statements, to check other conditions if one condition does not match, or to substitute a particular value if all substitutions fail.

The syntax for if statements is as follows:

```
<#if [condition]>
Value for Substitution
<#elseif [condition]>
Value for Else If Substitution or ${aVariable}
<#else>
Value for Else Substitution or ${aVariable}
</#if>
```

The if statement is opened with the special syntax "<#if", indicating the start of an if statement, followed by a condition and the closing character ">". Directly following the closing tag is the value that should be substituted if the condition matches. This is then following by the optional <#elseif> tag, which you can use to check a further condition, and finally the (optional) <#else> tag, which contains the value that should be substituted if no other condition matches. The if statement ends with the closing tag "</#if>".

Conditions should have the simple format "<variable name> == <test value>", where "<variable name>" is a variable in one of the supported variable managers, and "<test value>" the value you are testing for. You can test numerical values:

```
<#if MY_VARIABLE == 1>
Test was true!
</#if>
```

String values:

```
<#if MY_VARIABLE == 'Some Text'>
Test was true!
</#if>
```

Or boolean values (boolean values do not require a test value, as just listing the variable name implies that you are testing for a 'true' value):

```
<#if MY_VARIABLE>
Test was true!
</#if>
```

You can use normal substitution within the if statement as well:

```
<#if MY_VARIABLE>
${SUBSTITUTE_ME}
</#if>
```

The if statement can even be used within a property's value to select a small subset of text. For example, the following will insert the letter 'a' into the middle of the property value if "MY_VARIABLE" is true:

```
property value start <#if MY_VARIABLE>
a
</#if> property value end.
```

You can also optionally use one or more elseif statements and, also optionally, end with a final else statement if no other condition matches (only a single else statement can be used). Elseif statements can also use different variables than the if statement they are qualifying:

```
<#if MY_VARIABLE == 1>
Test was true!
<#elseif MY_OTHER_VARIABLE == 'Some Text'>
Second test was true!
<#elseif MY_VARIABLE == 2>
Third test was true!
<#else>
No tests were true!
</#if>
```

You can even use nested if statements to check for multiple conditions (complete with nested if or elseif statements):

```
<#if MY_VARIABLE == 1>
<#if MY_OTHER_VARIABLE == 1>
Both conditions match!
<#else>
</#if>
First condition matched!
<#else>
No conditions matched!
</#if>
```

Note that each nested if statement must be closed by its own corresponding "</#if>" tag, to allow Flux to clearly differentiate between different levels of if statements.

If statements also support "and" (&&) operators:

```
<#if MY_VARIABLE == 1 && MY_OTHER_VARIABLE == 2>
${MY_VARIABLE}
</#if>
```

As well as the "or" (||) operator:

```
<#if MY_VARIABLE == 1 || MY_OTHER_VARIABLE == 1>
${MY_VARIABLE}
</#if>
```

For a full list of operators allows in the if condition, see the table at the end of this section.

Note also that although the examples above use multiple lines for clarity, if statements can be used on a single line of text:

```
<#if MY_VARIABLE == 1>Test matched!<#else>Test did not match</#if>
```

This means you can still use if statements for properties on triggers and actions that do not allow more than a single line of text input.

As an example, suppose you have a Process Action running a script called "myscript.bat" that takes a configuration file as input. Now suppose that, depending on the value of a boolean variable called MY_VARIABLE (that is generated during a previous step in the flow chart), the process action should use one of two different configuration files: "file1.config" (if "MY_VARIABLE" is true) and "file2.config" (if not).

You could set the "Command" property of the action to:

```
myscript.bat <#if MY_VARIABLE>file1<#else>file2</#if>.config
```

If the variable "MY_VARIABLE" is true, the actual command executed will be:

```
myscript.bat file1.config
```

And if the variable "MY_VARIABLE is false, the actual command executed will instead be:

```
myscript.bat file2.config
```

The follow table contains a complete list of operators available when using if statements:

Operator	Description	Example
==	"equals" – the item on the left is equal to the item on the right	<#if MY_VAR == 1>True!</#if>
!=	"not equal" – the item on the left is not equal to the item on the right	<#if MY_VAR ?!= 2>True!</#if>
<	"less than" – the item on the left is less than the item on the right	<#if MY_VAR < 10>True!</#if>
>	"greater than" – the item on the left is greater than the item on the right	<#if MY_VAR > 1>True!</#if>
&&	"and" – both conditions must match	<#if MY_VAR == 1 && MY_OTHER_VAR == 2>True!</#if>
||	"or" – one of the conditions must match	<#if MY_VAR == 1 || MY_OTHER_VAR == 1>True!</#if>
+	"add" – allows you to add to a variable value used for comparison (for numeric variables, this function adds the numbers together; for string variables, this function concatenates the variables)	<#if MY_VAR + 1 == MY_OTHER_VAR>True!</#if>
-	"subtract" – allows you to subtract from a variable value used for comparison	<#if MY_VAR -1 == MY_OTHER_VAR>True!</#if>
*	"multiply" – allows you to multiply variable values used for comparison	<#if MY_VAR * 2 == MY_OTHER_VAR>True!</#if>
/	"divide" – allows you to divide variable values used for comparison	<#if MY_VAR / 2 == MY_OTHER_VAR>True!</#if>

%	"modulo" – divide the variable by a value and retrieve the remainder of the operation (i.e., 4 % 2 = 0)	<#if MY_VAR % 2 == 0>True!</#if>

Flux's Built-in Operations for Variables

There are also several built-in operations available for different variable data types. Sometimes use of a built-in operation is required. The Flux substitution engine substitutes numbers and strings without requiring a format. For booleans and dates and other objects one needs to add a format (e.g., ${<variable_name>?<format>}.

For example, the outputting of the contents of a FileInfo object from a File Trigger into a Console message could be defined as follows (where i is a FileInfo object in the flowContext):

```
Complete FileInfo Object = ${i}
Filename = ${i.filename}
Path to file = ${i.parent}
Is Readable = ${i.read?string("yes","no")}
Is Writable = ${i.read?string("yes","no")}
Last Modified = ${i.lastModified?datetime}
size = ${i.size}
url = ${i.url}
```

The corresponding output looks like this:

```
Complete FileInfo Object = FileInfo (read=true, write=true, filename='temp.xxx', lastModified=Sat Oct 18 18:18:47
CDT 2014, parent='c:\', size=27, url=file:/c:/temp.xxx)
Filename = temp.xxx
Path to file = c:\
Is Readable = yes
Is Writable = yes
Last Modified = Oct 18, 2014 6:18:47 PM
size = 27
url = file:/c:/temp.xxx
```

The following tables list the specific operations available for each type:

String (text) variables

Operation	Usage	Description	Example	Example Variable Value	Result
substring	<variable name>?substring(*from*)	A substring of the variable, beginning with the character at index *from*.	${MY_STRING?substring(7)}	MY_STRING = "Hello, World!"	World!
substring	<variable name>?substring(*from*, *to*)	A substring of the variable, beginning with the character at index *from*. The index *to* should be one number higher than the index of the last character to include in the substring.	${MY_STRING?substring(0, 5)}	MY_STRING = "Hello, World!"	Hello
cap_first	<variable name>?cap_first	Capitalize the first letter of the variable value.	${LOWER_STRING?cap_first}	LOWER_STRING = "hello, world!"	Hello, world!
uncap_first	<variable name>?uncap_first	The opposite of cap_first – lower-cases the first letter of the variable value.	${MY_STRING?uncap_first}	MY_STRING = "Hello, World!"	hello, World!
capitalize	<variable name>?capitalize	Capitalize each word in the variable (assuming a space separates each word).	${LOWER_STRING?capitalize}	LOWER_STRING = "hello, world!"	Hello, World!
date	<variable name>?date("<date formatter>")	Convert the variable to a readable date. See the date substitution section below for acceptable date formatter values.	${DATE_STRING?date("MM/dd/yyyy")}	DATE_STRING = "4/4/2011"	Apr 4, 2011
time	<variable name>?time("<date formatter>")	Convert the variable to a readable time. See the date substitution section below for acceptable date formatter values.	${TIME_STRING?time("HH:mm:ss")}	TIME_STRING = "17:08:35"	5:08:35 PM
datetime	<variable name>?datetime("<date formatter>")	Convert the variable to a readable date and time. See the date substitution section below for acceptable date formatter values.	${DATETIME_STRING?datetime("yyyy-MM-dd hh:mm")}	DATETIME_STRING = "2011-4-4 17:08"	Apr 4, 2011 5:08:00 PM
ends_with	<variable name>?ends_with(*string*)	Evaluates to true if the variable value ends with the given *string*. Can be used within if statements to evaluate conditions.	<#if MY_STRING?ends_with("World!")>true</#if>	MY_STRING = "Hello, World!"	true

html	<variable name>?html	The string as HTML markup ("<" replaced with "<", ">" replaced with ">", etc.)	${HTML_STRING?html}	HTML_STRING = "<html>"	`&lt; html &gt;`
index_of	<variable name>?index_of(*string*)	Gets the first index of *string* as it occurs in the variable. This is typically used with "substring" to obtain the part of the variable that occurs before *string*.	${ABC_STRING?substring(0, ABC_STRING?index_of("def"))}	ABC_STRING = "abcdef"	abc
j_string	<variable name>?j_string	Escape the string using the escaping rules of Java String literals.	${JAVA_STRING?j_string}	JAVA_STRING = "My "String" string."	My \" String\" string
js_string	<variable name>?js_string	Escape the string using the escaping rules of JavaScript string literals.	${JAVASCRIPT_STRING?js_string}	JAVASCRIPT_STRING = "Javascript's "String" value."	Javascript \'s \" String\" value.
last_index_of	<variable name>?last_index_of(*string*)	Gets the last index of *string* as it occurs in the variable. This is typically used with "substring" to obtain the part of the variable that occurs before the last instance of *string*.	${REPEAT_STRING?substring(0, REPEAT_STRING?last_index_of("text"))}	REPEAT_STRING = "textonetexttwotextthree"	textonetexttwo
length	<variable name>?length	Gets the number of characters in the variable. Typically used with the "substring" command to remove a certain number of characters from the end of the string.	${FILENAME_STRING?substring(0, FILENAME_STRING?length - 4)}	FILENAME_STRING = "myfile.txt"	myfile
lower_case	<variable name>?lower_case	Convert the variable to lower case.	${MY_STRING?lower_case}	MY_STRING = "Hello, World!"	hello, world!
contains	<variable name>?contains(*string*)	Evaluates to true if the variable contains the given *string*.	<#if MY_STRING?contains("World!")>true</#if>	MY_STRING = "Hello, World!"	true
matches	<variable name>?matches(*regex*)	Evaluates to true if the variable matches the given regular expression *regex*.	<#if MY_STRING?matches("Hello.*")>true</#if>	MY_STRING = "Hello, World!"	true
number	<variable name>?number(*string*)	Converts the specific *string* to a number. Useful when a variable is stored as a string but is used for a numeric property, like a port number.	${NUM_STRING?number}	NUM_STRING = 1	1
replace	<variable name>?replace(*string*, *replacement*)	Replace all instances of *string* in the variable with the substitute string *replacement*.	${MY_STRING?replace("one", "two")}	MY_STRING = "one two one three"	two two two three
starts_with	<variable name>?starts_with(*string*)	Evaluates to true if the variable starts with the given *string*.	<#if MY_STRING?starts_with("Hello")>true</#if>	MY_STRING = "Hello, World!"	true
trim	<variable name>?trim	Removes any white space before the beginning of the text content of the variable, and after the end.	${SPACE_STRING?trim}	SPACE_STRING = " trim me "	trim me
upper_case	<variable name>?upper_case	Convert the variable to upper case.	${MY_STRING?upper_case}	MY_STRING = "Hello, World!"	HELLO, WORLD!

xhtml	<variable name>?xhtml	The variable as XHTML text. Similar to XML substitution, but escapes the ' character as: `&#39` To support older browsers that are not able to interpret: `&apos;` All of the following are replaced: `< with &lt;` `> with &gt;` `& with &amp;` `" with &quot;` `' with &#39;`	${XML_STRING?xhtml}	XML_STRING = "<te'xt>"	`&lt;te&#39;xt&gt;`
xml	<variable name>?xhtml	The variable as XML text. The only difference between XML and XHTML substitution is that XML escapes the ' character as: `&apos;` Instead of: `&#39;`	${XML_STRING?xml}	XML_STRING = "<te'xt>"	`&lt;te&apos;xt&gt;`

Numeric Variables

The following operations can be used for any variable that contains a numeric data type:

Operation	Usage	Description	Example	Example Variable Value	Result
round	<variable name>?round	Round to the nearest whole number	${MY_NUMBER?round}	MY_NUMBER = 1.5	2
floor	<variable name>?floor	Round the number downward (to the next-lowest whole number)	${MY_NUMBER?floor}	MY_NUMBER = 2.9	2
ceiling	<variable name>?ceiling	Round the number upward (to the next-highest whole number)	${MY_NUMBER?ceiling}	MY_NUMBER = 15.2	15

All Variable Types

The following operations can be used on any variable to determine whether the variable matches a certain type. Usage for all variables is the same – for example, to test if a variable is a string using the is_string operation, you can use:

```
<#if MY_VARIABLE?is_string>true<#else>false</#if>
```

Operation	Evaluates to true if variable...	Example
is_string	is a string	<#if MY_VARIABLE?is_string>MY_VARIABLE is a string!<#else>MY_VARIABLE is not a string.</#if>

is_number	is a number	<#if MY_VARIABLE?is_number>MY_VARIABLE is a number!<#else>MY_VARIABLE is not a number.</#if>
is_boolean	is a boolean	<#if MY_VARIABLE?is_boolean>MY_VARIABLE is a boolean!<#else>MY_VARIABLE is not a boolean.</#if>
is_date	is a date (all types, including date, datetime, and time only)	<#if MY_VARIABLE?is_date>MY_VARIABLE is a date!<#else>MY_VARIABLE is not a date.</#if>
is_collection	is a collection	<#if MY_VARIABLE?is_collection>MY_VARIABLE is a collection!<#else>MY_VARIABLE is not a collection.</#if>
??	exists	<#if MY_VARIABLE??>MY_VARIABLE exists!<#else>MY_VARIABLE does not exist.</#if>

For Loops ("List" command)

You can use a for loop within an action or trigger property to access each value in a collection (that is, a list, array, or set data type).

The syntax for for loops is as follows:

```
<#list [collection variable name] as [variable to create]>
Value for Substitution
</#list>
```

The for loop is opened with the special syntax "<#list", indicating the start of a for loop (also called a "list"). This is followed by the name of a variable stored in one of the supported variable manager, which (as noted above) must be a list, array, or set data type. This is then followed by the special phrase " as ", then the name of a new variable that should be created to iterate over each item in the list. As the loop progresses through the collection, the value of the current item in the collection will be stored in a new variable using the name supplied here (this can be any name you like as long as it does not conflict with an existing variable in your flow chart or flow context). All of this is then closed with the character ">", then the value for substitution, followed by the closing tag "</#list>" indicating the end of the loop.

When the action or trigger actually executes, it will loop through each item in the collection, set the next item's value to the name of the variable supplied in "[variable to create]", then substitute in whatever value is between the <#list> and </#list> tags. This will happen once for each item in the collection before the loop exits.

Like if statements (described above), for loops can be used across multiple lines or contained on a single line of text, allowing you to use them for action properties that do not support more than one line of text.

For example, consider a variable named MY_ARRAY, which is an array containing the text values "A", "B", and "C". If you have an action property with the following for loop:

```
<#list MY_ARRAY as THIS_VALUE>${THIS_VALUE} </#list>
```

Then the action or trigger property will be set to the actual value:

```
A B C
```

As the newly created variable THIS_VALUE is substituted once for each item in the list (followed by a space).

Note that the variable you create in this list is only active while the for loop actually executes. That means that once the action or trigger completes, you will not be able to use the variable THIS_VALUE in any subsequent triggers or actions (although the collection itself will remain unchanged and is not modified or cleared by the for loop).

You can use nested if statements within for loops as well. For example, the following will iterate over a collection and substitute every value that is divisible by 2:

```
<#list MY_COLLECTION as THIS_ITEM>
<#if THIS_ITEM % 2 == 0>%{THIS_ITEM}</#if>
</#list>
```

You can also access the index (position in the collection) of the current item by using the special syntax "<current item variable>_index", like so:

```
<#list MY_COLLECTION as THIS_ITEM>
${THIS_ITEM_index}. ${THIS_ITEM}
</#list>
```

Or determine whether the current item is the last in the collection, using "<current item variable>_has_next":

```
<#list MY_COLLECTION as THIS_ITEM>
${THIS_ITEM}<#if THIS_ITEM_has_next>, </#if>
</#list>
```

These for loops can be used to easily iterate over elements in an action or trigger result that are returned as a collection.

For example, consider a flow chart with a File Exist Trigger that locates three files, named "fileA.ext", "fileB.ext", and "fileC.ext". Now suppose that this trigger is followed by a process action that needs to pass these three file names as parameters for a single command-line invocation, looking something like:

```
myprogram.exe fileA.ext fileB.ext fileC.ext
```

To accomplish this, you could simply add a normal flow from the File Exist Trigger to the Process Action, then set the Process Action's "Command" property to:

```
myprogram.exe <#list RESULT.filename_matches as THIS_FILENAME>${THIS_FILENAME} </#list>
```

"filename_matches" is a result property of the File Exist Trigger, and is returned from the trigger as a list. This means that it can be iterated over using the "list" property to access each filename one at a time, substituting it into the command after the last filename is entered. As a result, at runtime the actual command will be set to the following as expected:

```
myprogram.exe fileA.ext fileB.ext fileC.ext
```

Iterating through Sub-lists

If you have a list of lists (for example – a list containing all the rows returned by a database action) to iterate through, Flux's variable substitution also supports nested lists. For example, suppose you have a list named "mylist", where each entry is a list representing a row in a database result set, and each entry in that sub-list is a column in the current row. You can iterate through the entire list and print out the value of each column like so:

```
<#list mylist as thisrow>
<#list thisrow as thiscolumn>
<#if thiscolumn??>
${thiscolumn}
</#if>
</#list>
</#list>
```

Date Substitution

Date substitution allows you to access the date that a trigger or action runs and substitute that (with precise control over the formatting or presentation of the date) within the action or trigger's properties. This can be used, for example, to get the current date in a file trigger or action's file criteria, allowing you to search for, create, or delete files based on the date when the action or trigger actually runs.

You can also apply a time expression against the date, allowing you to retrieve some other date relative to the date your action or triggers runs (for example, to subtract one month from the date).

To use date substitution, enter something like the following in your action or trigger property:

```
${date (<optional time expression>) <date formatter>}
```

The special keyword "date" indicates that you are invoking date substitution. This is followed by an optional time expression (if you are planning to access a relative value) and, finally, a formatter indicating how the date output should be formatter (see the table below for a complete list of date formatter options).

For example, if the following date substitution were used on 28 March 2011:

```
${date dd-MMM-yyyy}
```

The actual date output would be:

```
28-Mar-2011
```

Which would be substituted into the action or trigger property where the date substitution was entered.

And if the following substitution were used:

```
${date (-1d) dd-MMM-yyyy}
```

The date output would be:

```
27-Mar-2011
```

Or one day before the action or trigger ran (as the "-1d" time expression indicates "one day previous to the current date").

As an example, suppose that you run a program which generates log files automatically every night. The program creates a file with the name "myprogram-<date>.log", where "<date>" is the date of the log file's generation in the format "dd-MMM-yyyy". Now suppose that you would like to create a Flux flow chart to delete those log files automatically after one month.

To do that, you could simply create a File Delete Action, set up the file criteria in the normal way, and use date substitution in the File Delete Action's include:

```
myprogram-${date dd-MMM-yyyy}.log
```

When the trigger actually ran, the substitution would be performed and the include would become something like:

```
myprogram-27-Mar-2011.log
```

The substitution would be re-evaluated for each execution of the File Delete Action, so in a looping flow chart the action would correctly update its include for each day the the action ran.

The following table shows the complete formatter syntax available for date substitution (if a character does not appear in the table below, it will be translated into the date output literally – so, for example, the character "-" will be included in the date output exactly as you enter it into your formatter).

Date Formatter Symbols

Date Formatter	Result
yyyy	2006
yy	06
MMMM	June
MMM	Jun
MM	06
M	6
dd	01
d	1
DD	Thursday
D	Thurs
HH	08
H	8
mm	05
m	5
ss	06
s	6

Runtime Configuration Substitution

Runtime configuration substitution behaves similarly to global substitution except that runtime configuration substitution deals with a broader spectrum. Given the proper syntax, runtime configuration substitution retrieves a specified variable’s value from the engine’s runtime configuration and substitutes it into a property. Runtime configuration substitution is helpful when a variable must be used in more than one flow chart.

To specify that a runtime configuration variable must be retrieved, use the following syntax:

```
${runtime <variable name>}
```

Where "<variable name>" is the name of the variable in the runtime configuration whose value you want to substitute.

For more information on the runtime configuration, see Runtime Configuration.

Supported Data Types

Variable substitution can be used in any flow chart or action property in Flux as long as it is one of the following three data types:

- **java.lang.String**. This includes textual values (i.e. "mytext").
- **java.lang.Number**. This includes numerical values supported by Java ("10", "5.3", "1.001", etc.).
- **java.net.URL**. URLs such as "http://www.fluxcorp.com" or "ftp://user@host/path/to/file".

There are no restrictions on the data type of the variables themselves (the only restriction is that, if you are using variable field or method substitution, the field or method must be publicly accessible in the Java class of the variable).

Handling of Missing and Null Values

An error will occur and abort the variable substitution if you try to access a missing variable. By default, the variable string itself is returned if it cannot be substituted. However Flux provides two special operators so you can suppress this error and handle the problematic situation. These operators handle the situation when a method call doesn't return a value (from the viewpoint of Java programmers: it returns null or it's return type is void), so it's more correct to say that these operators handle missing values in general, rather than just missing variables.

Java Concept of Null

Flux's variable substitution treats Java nulls as missing values. Variable substitution doesn't know the concept of null. For example, if you have a bean that has a maidenName property, and the value of that property is null, then that's the same as if there were no such property at all as far as the substitution is concerned. The result of a method call that returns null is also treated as a missing variable.

If you need to provide a default value to handle cases of a missing value or null, simply change the variable to look like either of the following:

- **${<variable_name>!<default_value>}** - Here any instance of a missing value is replaced by the <default_value>. Examples would include: ${row .get(0)!"Some value"} or or ${row.get(0)!false} or ${row.get(0)!0}
- **${<variable_name>!}** - If the default value is omitted (e.g., ${row.get(0)!}), then the variable substitution will return an empty string and empty sequence and empty hash at the same time. Note the consequence is that you can't omit the default value if you want it to be 0 or false.

When is Variable Substitution Performed?

Variable substitution occurs each time the Flux engine attempts to access the property where variable substitution was used. Properties that use variable substitution are also re-evaluated each time they are accessed, so in a looping flow chart, an action or trigger can always access the latest copy of a variable.

For trigger and action properties that use variable substitution, this means that the substitution is performed each time the trigger or action executes. For action and trigger properties, variable substitution is done after the prescript (if any) executes, so any variable changes made in a prescript can be picked up by variable substitution.

Ensure runtime variables are configured and loaded before workflow executes

Ensure that all required runtime variables are configured and loaded before any workflow runs that rely on these variables. Not doing so will cause the workflow to fail with a null pointer exception or other error since the variable cannot be resolved. Note that you can create workflows with runtime variables that are not configured at the time the workflow is designed, but the variable must be configured and loaded before the workflow executes.