**Article**

**JOLT Transform Language Quick Reference**

**OBJECTIVE:**

Provide a quick-start guide for using the Jolt language within a NiFi JoltTransform.

**OVERVIEW:**

The NiFi JoltTransform uses the powerful Jolt language to parse JSON. Combined with the NiFi Schema Registry, this gives NiFi the ability to traverse, recurse, transform, and modify nearly any data format that can be described in AVRO or, using JSON as an intermediary step.

Although the language itself is open-source, and some documentation is available in the JavaDoc, this article can provide a starting point for understanding basic Jolt operations.

**PREREQUISITES:**

HDF 3.0 or later (NiFi 1.2.0.3 or later)

**BASICS OF JOLT:**

1. Simplified Overview
   1. The JoltTransform applies a set of transformations described in a JSON specification to an input JSON document and generates a new output JSON document.
2. Jolt Specification
   1. Overview  
      A Jolt Specification is a JSON structure that contains two root elements:
      1. operation (string): shift, sort, cardinality, modify-default-beta, modify-overwrite-beta, modify-define-beta, or remove
      2. spec (JSON): A set of key/value pairs of the form {“input-side search”: “output-side transformation”}.
   2. Simple: Select a single jolt transform type from the drop-down, then type or paste the specification JSON
   3. Chained: Multiple Jolt specifications can be chained together sequentially in an array of simple specifications
3. Stock Transforms
   1. Shift: Read values or portions of the input JSON tree and add them to specified locations in the output.
      1. I have a bunch of things in the breadbox that I want to move to the countertop. Let’s move everything in the breadbox to the countertop:

Input:

{

"breadbox": {

"loaf1": {

"type": "white"

},

"loaf2": {

"type": "wheat"

}

},

"fridge": {

"jar1": {

"contents": "peanut butter"

},

"jar2": {

"contents": "jelly"

}

}

}

Spec:

[

{

"operation": "shift",

"spec": {

"breadbox": "counterTop"

}

}

]

Output:

{

"counterTop": {

"loaf1": {

"type": "white"

},

"loaf2": {

"type": "wheat"

}

}

}

* 1. Default: Non-destructively adds values or arrays of values to the output JSON.
     1. I want to slice up loaf1 if it exists. Let’s add an array of slices to loaf1:

Input:

{  
 "counterTop": {  
 "loaf1": {  
 "type": "white"  
 },  
 "loaf2": {  
 "type": "wheat"  
 },  
 "jar1": {  
 "contents": "peanut butter"  
 },  
 "jar2": {  
 "contents": "jelly"  
 }  
 }

Spec:

[  
 {  
 "operation": "default",  
 "spec": {  
 "counterTop": {  
 "loaf1": {  
 "slices": [  
 "slice1",  
 "slice2",  
 "slice3",  
 "slice4"  
 ]  
 }  
 }  
 }  
 }  
 ]  
}

Output:

{

"counterTop" : {

"loaf1" : {

"type" : "white",

"slices" : [ "slice1", "slice2", "slice3", "slice4" ]

},

"loaf2" : {

"type" : "wheat"

},

"jar1" : {

"contents" : "peanut butter"

},

"jar2" : {

"contents" : "jelly"

}

}

}

* 1. Cardinality: Transforms elements in the input JSON to single values or to arrays (lists) in the output.
     1. I have too many slices of bread. No matter how many there are, I just want the first one in the array, but as a single value:

Input:

{

"counterTop": {

"loaf1": {

"type": "white",

"slices": [

"slice1",

"slice2",

"slice3",

"slice4"

]

},

"loaf2": {

"type": "wheat"

},

"jar1": {

"contents": "peanut butter"

},

"jar2": {

"contents": "jelly"

}

}

}

Spec:

[

{

"operation": "cardinality",

"spec": {

"counterTop": {

"loaf1": {

"slices": "ONE"

}

}

}

}

]

Output:

{

"counterTop" : {

"loaf1" : {

"type" : "white",

"slices" : "slice1"

},

"loaf2" : {

"type" : "wheat"

},

"jar1" : {

"contents" : "peanut butter"

},

"jar2" : {

"contents" : "jelly"

}

}

}

* 1. Remove: Remove elements if found in the input JSON.
     1. I don’t want loaf2 or jar1 (who needs whole wheat or peanut butter when you have jelly on pain bread!). Let’s remove loaf2 and jar2:

Input:

{

"counterTop": {

"loaf1": {

"type": "white",

"slices": "slice1"

},

"loaf2": {

"type": "wheat"

},

"jar1": {

"contents": "peanut butter"

},

"jar2": {

"contents": "jelly"

}

}

}

Spec:

[

{

"operation": "remove",

"spec": {

"counterTop": {

"loaf2": "",

"jar1": ""

}

}

}

]

Output:

{

"counterTop" : {

"loaf1" : {

"type" : "white",

"slices" : "slice1"

},

"jar2" : {

"contents" : "jelly"

}

}

}

* 1. Modify: Write calculated values to elements in the target JSON. Calculations include basic string and math operations (toLower, toUpper, concat, min/max/abs, toInteger, toDouble, toLong and can be applied to source JSON values.
     1. I really like jelly. Let’s make whatever’s in jar1 all caps to shout about it!

Input:

{

"counterTop": {

"loaf1": {

"type": "white",

"slices": "slice1"

},

"jar2": {

"contents": "jelly"

}

}

}

Spec:

[

{

"operation": "modify-overwrite-beta",

"spec": {

"counterTop": {

"jar2": {

"contents": "=toUpper"

}

}

}

}

]

Output:

{

"counterTop" : {

"loaf1" : {

"type" : "white",

"slices" : "slice1"

},

"jar2" : {

"contents" : "JELLY"

}

}

}

* 1. Sort: Sorts all arrays and maps from the input JSON into the output. Sort cannot be configured beyond this all-or-nothing sort. Let’s put the jelly first to make it easier to spread on the bread later:

Input:

{

"counterTop": {

"loaf1": {

"type": "white",

"slices": "slice1"

},

"jar2": {

"contents": "JELLY"

}

}

}

Spec:

[

{

"operation": "sort"

}

]

Output:

{

"counterTop" : {

"jar2" : {

"contents" : "JELLY"

},

"loaf1" : {

"slices" : "slice1",

"type" : "white"

}

}

}

* 1. Custom Transforms: (Custom Transforms are out of scope for this tutorial)

1. Wildcards and Operators
   1. Input-side (lefthand side)  
      Input-side wildcards retrieve a value or JSON tree from the input JSON.
      1. \* (asterisk)
         1. The asterisk wildcard traverses and reads each element in the source JSON at the level of the preceding search specification. Typically, the asterisk wildcard will return an array of elements.
            1. Rather than just one source element, such as the breadbox, let’s grab everything, no matter what element it’s in, and put it on the counter

Input:

{

"breadbox": {

"loaf1": {

"type": "white"

},

"loaf2": {

"type": "wheat"

}

},

"fridge": {

"jar1": {

"contents": "peanut butter"

},

"jar2": {

"contents": "jelly"

}

}

}

Spec:

[

{

"operation": "shift",

"spec": {

"\*": "counterTop"

}

}

]

Output:

{

"counterTop" : [ {

"loaf1" : {

"type" : "white"

},

"loaf2" : {

"type" : "wheat"

}

}, {

"jar1" : {

"contents" : "peanut butter"

},

"jar2" : {

"contents" : "jelly"

}

} ]

}

* + - 1. The asterisk wildcard can be used with other string characters to parse data within an input JSON element (we’ll use the $ wildcard notation here – see below for an explanation of that).
         1. Let’s take a look at the expiration date on the jelly. I am not a stickler for expiration dates, so I just want to check the year:

Input:

{

"breadbox": {

"loaf1": {

"type": "white"

},

"loaf2": {

"type": "wheat"

}

},

"fridge": {

"jar1": {

"contents": "peanut butter"

},

"jar2": {

"contents": "jelly",

"expiration": "25-APR-2019"

}

}

}

Spec:

[

{

"operation": "shift",

"spec": {

"fridge": {

"jar2": {

"expiration": {

"\*-\*-\*": {

"$(0,3)": "expiry.year"

}

}

}

}

}

}

]

Output:

{

"expiry" : {

"year" : "2019"

}

}

* + 1. @ (“at” or arobase)
       1. The “at” wildcard traverses backwards up the source JSON and returns the entire tree or value at the specified position.
          1. @ or @0 (return value or tree of the matched key from the input JSON)

Let’s say I take a look at the jelly in jar2, and it has spoiled – We can use @contents to just toss the jelly into the garbage, but if the jelly is terribly bad, we can use @ or @0 to throw out everything in the jar, @1 to throw out everything in the fridge, or @2 to toss the whole kitchen into the garbage!

Input:

{

"breadbox": {

"loaf1": {

"type": "white"

},

"loaf2": {

"type": "wheat"

}

},

"fridge": {

"jar1": {

"contents": "peanut butter"

},

"jar2": {

"contents": "jelly",

"expiration": "25-APR-2019"

}

}

}

Spec:

[

{

"operation": "shift",

"spec": {

"fridge": {

"jar2": {

"contents": "garbage1",

"@0": "garbage2",

"@1": "garbage3",

"@2": "garbage4"

}

}

}

}

]

Output:

{

"garbage0" : {

"contents" : "jelly",

"expiration" : "25-APR-2019"

},

"garbage1" : "jelly",

"garbage2" : {

"contents" : "jelly",

"expiration" : "25-APR-2019"

},

"garbage3" : {

"jar1" : {

"contents" : "peanut butter"

},

"jar2" : {

"contents" : "jelly",

"expiration" : "25-APR-2019"

}

},

"garbage4" : {

"breadbox" : {

"loaf1" : {

"type" : "white"

},

"loaf2" : {

"type" : "wheat"

}

},

"fridge" : {

"jar1" : {

"contents" : "peanut butter"

},

"jar2" : {

"contents" : "jelly",

"expiration" : "25-APR-2019"

}

}

}

}

* 1. Output-side (righthand side)  
     Output-side wildcards return a single value that can be used in a target JSON key, key path or value.
     1. & (ampersand)
        1. The ampersand wildcard traverses backwards up the source JSON tree, beginning at the level of the preceding match. It returns only the value or key name (not the tree). The ampersand can be used in three ways:
           1. & or &0 (return the name of the matched key from the input JSON)

Let’s look at jar2 more closely, but I only care about what’s in it. We’ll just put the value of “contents” into the same element’s name (@0)

Input:

{

"breadbox": {

"loaf1": {

"type": "white"

},

"loaf2": {

"type": "wheat"

}

},

"fridge": {

"jar1": {

"contents": "peanut butter"

},

"jar2": {

"contents": "jelly",

"expiration": "25-APR-2019"

}

}

}

Spec:

[

{

"operation": "shift",

"spec": {

"fridge": {

"jar2": {

"contents": "&0"

}

}

}

}

]

Output:

{

"contents" : "jelly"

}

* + - * 1. &n (walk back up the tree ‘n’ levels and return the key name from the specified level)

Since the extra “contents” key is a bit superfluous, let’s just use the name of the parent element (&1) instead:

Input:

(same as above)

Spec:

[

{

"operation": "shift",

"spec": {

"fridge": {

"jar2": {

"contents": "&1"

}

}

}

}

]

Output:

{

"jar2" : "jelly"

}

* + - * 1. &(n,x) (walk back up the tree ‘n’ levels and return the key name from the xth child of the key at that level)

We really just want to know where to look for jelly, no matter what container it’s in. Let’s look at the top-level parent’s name instead (&(2,0))

Input:

(same as above)

Spec:

[

{

"operation": "shift",

"spec": {

"fridge": {

"jar2": {

"contents": "&(2,0)"

}

}

}

}

]

Output:

{

"fridge" : "jelly"

}

* + 1. @ (“at” or arobase)
       1. The “at” wildcard traverses backwards up the source JSON and returns the entire tree or value at the specified position.

Same functionality as on the input side, above: @, @(n), @(keyName), and @(n,keyName) forms

* + - * 1. Let’s see what we have in all the jars in our refrigerator. We want to match everything with a name starting with “jar” (jar\*) and return the contents of each element we find (@0,contents):

Input:

(same as above)

Spec:

[

{

"operation": "shift",

"spec": {

"\*": {

"jar\*": {

"@(0,contents)": "Things in jars"

}

}

}

}

]

Output:

{

"Things in jars" : [ "peanut butter", "jelly" ]

}

* + 1. $ (dollar sign)
       1. The dollar sign traverses backwards up the source JSON and returns only the value at the specified position.
          1. Same functionality as @, above: $, $(n), and $(n,x) forms

Input:

(same as above)

Spec:

[

{

"operation": "shift",

"spec": {

"\*": {

"jar\*": {

"$0": "List of jars"

}

}

}

}

]

Output:

{

"List of jars" : [ "jar1", "jar2" ]

}

1. “Temp” workspace  
   In a chained Jolt specification, it is possible to create a temporary structure as a workspace within the output JSON. This temporary structure can be useful for making multi-pass transformations or for holding a copy of the original input JSON during destructive transformations. They can then be removed from the output JSON within the same chained specification before the output JSON is produced.

For an example, see the JOLT transform for this article:   
<https://community.hortonworks.com/articles/232333/image-data-flow-for-industrial-imaging.html>

In this example spec, three “shift” operations are chained together. The “particles-orig” element is created to back up the original data in “particles,” then three passes are attempted because there may be a variable number of semicolon-delimited values in the “particles” value. When the three passes are complete, the successful pass is written to the output as “particles” and the backup is removed with a “remove” operation.

[

{

"operation": "shift",

"spec": {

"particles": ["particles-orig",

"particles-0",

"particles-1",

"particles-2",

"particles-3",

"particles-4"],

"timestamp": "ts",

"\*": "&"

}

},

{

"operation": "shift",

"spec": {

"particles-orig": "particles-orig",

"particles-0": {

"\*;\*;\*;\*;\*": {

"$(0,1)": "tmp.particle1[]",

"$(0,2)": "tmp.particle2[]",

"$(0,3)": "tmp.particle3[]",

"$(0,4)": "tmp.particle4[]",

"$(0,5)": "tmp.particle5[]"

}

},

"particles-1": {

"\*;\*;\*;\*": {

"$(0,1)": "tmp.particle1[]",

"$(0,2)": "tmp.particle2[]",

"$(0,3)": "tmp.particle3[]",

"$(0,4)": "tmp.particle4[]"

}

},

"particles-2": {

"\*;\*;\*": {

"$(0,1)": "tmp.particle1[]",

"$(0,2)": "tmp.particle2[]",

"$(0,3)": "tmp.particle3[]"

}

},

"particles-3": {

"\*;\*": {

"$(0,1)": "tmp.particle1[]",

"$(0,2)": "tmp.particle2[]"

}

},

"particles-4": "tmp.particle1[]",

"\*": "&"

}

},

{

"operation": "shift",

"spec": {

"tmp": {

"\*": {

"0": {

"\*,\*,\*,\*": {

"@(4,runid)": "particles.[#4].runid",

"@(4,ts)": "particles.[#4].ts",

"$(0,1)": "particles.[#4].Xloc",

"$(0,2)": "particles.[#4].Yloc",

"$(0,3)": "particles.[#4].Xdim",

"$(0,4)": "particles.[#4].Ydim"

}

}

}

},

"\*": "&"

}

},

{

"operation": "remove",

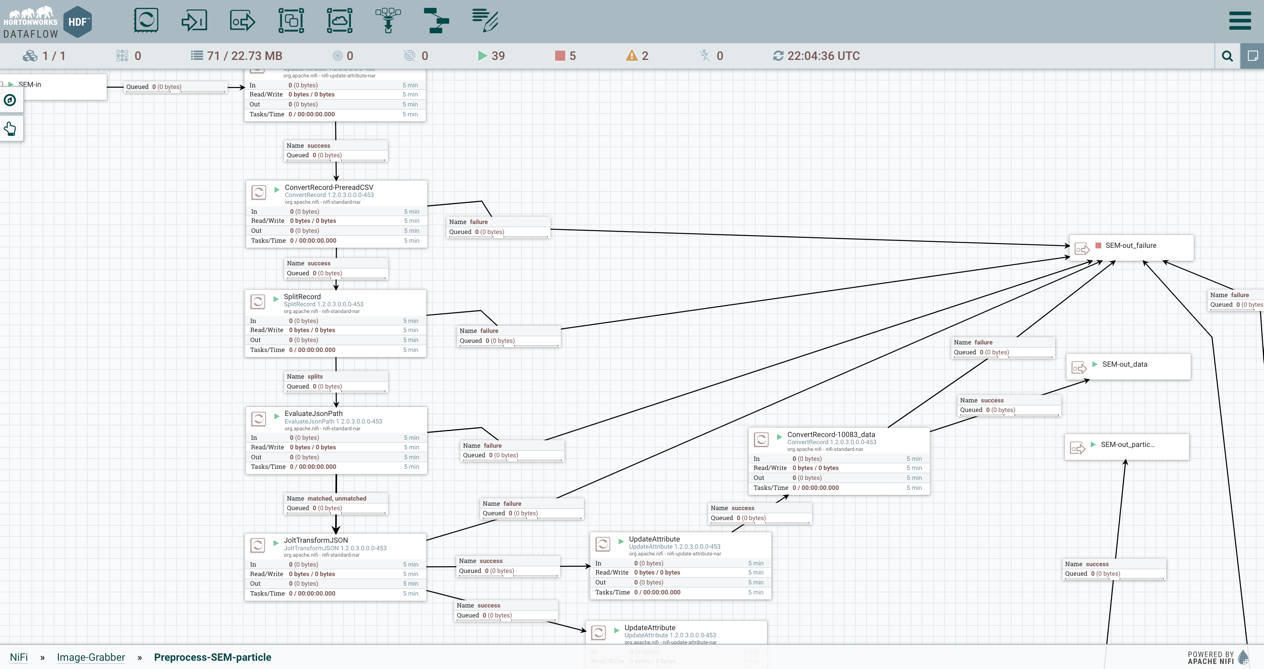
"spec": {

"particles-orig": ""

}

}

]



**REFERENCES:**

* [Jolt transform language GitHub](https://github.com/bazaarvoice/jolt)
* [Jolt Demo site (like JSfiddle, but for Jolt)](http://jolt-demo.appspot.com/)
* [Complete Jolt transform readme at GitHub)](https://github.com/bazaarvoice/jolt/blob/master/jolt-core/src/main/java/com/bazaarvoice/jolt/Shiftr.java)

**RELATED POSTS:**

* [Image Data Flow for Industrial Imaging](https://community.hortonworks.com/articles/232333/image-data-flow-for-industrial-imaging.html)