**Introduction**

**JOLT** (*JsOn Language for Transform*) is a transformation library, written in Java that allows a developer to convert one JSON structure to another. Jolt provides a set of transformation types, each with their own DSL (called specifications), that define the new structure for outgoing JSON data. In this tutorial we will explore some JOLT transformations and write related JOLT specifications.

**What is Jolt Transformation?**

**Jolt** Transformation is a lightweight and versatile JSON-to-JSON transformation library and framework. It is designed to simplify the process of transforming structured data, particularly JSON data, from one format to another. Jolt provides a declarative way to specify how the transformation should occur, making it a powerful tool for data manipulation and conversion tasks.

**JOLT Library Features**

Jolt provides a robust solution for JSON transformation tasks, offering a combination of simplicity, flexibility, and performance. Whether you need to reshape JSON data for analytics, data migration, or API integration, Jolt offers a versatile toolkit for achieving your transformation objectives.

**JSON-to-JSON Transformation**

Jolt is primarily used for transforming JSON data from one structure to another. It allows you to define transformation rules in a declarative manner using a JSON-based specification known as the Jolt Spec.

**Declarative Transformation Rules**

Jolt transformation rules are defined using a set of JSON objects called “Jolt Spec.” The Jolt Spec describes how input JSON should be transformed to produce the desired output JSON. Transformation rules are expressed using key-value pairs, where keys represent output JSON paths, and values define transformation operations.

**Wide Range of Transformation Operations**

Jolt supports a variety of transformation operations, including:

* Moving data from one location to another.
* Set default values for missing keys.
* Removing specific keys from the output JSON.
* Modifying the values of keys based on predefined rules.
* Flattening or nest JSON structures.
* Combining values of multiple keys into a single key.
* Filtering out unwanted data from the output JSON.

**Flexibility and Extensibility**

Jolt provides flexibility in defining complex transformation rules through chaining of multiple operations. It allows you to create custom transformations by extending Jolt’s capabilities using custom Java code.

**Ease of Use and Integration**

Jolt is easy to learn and use, making it accessible to developers with varying levels of expertise in JSON manipulation. It integrates well with various programming languages and frameworks, including Java, Python, and JavaScript. Jolt can be seamlessly integrated into data processing pipelines and workflows to perform JSON transformations at scale.

**Performance and Scalability**

Jolt is designed for performance and scalability, making it suitable for handling large volumes of JSON data efficiently. It leverages parallel processing and optimized algorithms to minimize processing times and resource utilization.

**Jolt LHS and RHS**

In Jolt, “**LHS**” and “**RHS**” stand for “**Left-Hand Side**” and “**Right-Hand Side**,” respectively. They refer to the structure of the input JSON (LHS) and the desired structure of the output JSON (RHS) in a Jolt transformation.

LHS and RHS are used in Jolt transformation specifications to describe the structure of the input JSON data (LHS) and the desired structure of the output JSON data (RHS). By defining transformation rules that map LHS paths to corresponding RHS paths, you can reshape, manipulate, and transform JSON data efficiently.

**Left-Hand Side (LHS)**

The  “**Left-Hand Side**”  (LHS) represents the structure of the input JSON data. It defines the paths and keys in the input JSON that you want to transform or manipulate. The LHS consists of the input JSON structure that you specify in the transformation rules. You use dot notation (`.`) and other special characters to navigate through the input JSON structure and access specific keys or values.

Example LHS: “name“, “address.city“, “contacts.\*.type“, etc.

**Right-Hand Side (RHS)**

The “**Right-Hand Side**”  (RHS) the desired structure of the output JSON data. It defines where the data from the input JSON should be moved or transformed to create the output JSON. The RHS consists of the output JSON structure that you define in the transformation rules. You specify the keys and paths in the output JSON where you want to place the transformed or manipulated data from the input JSON.

Example RHS: “fullName“, “location“, “details.age“, etc.

**JOLT Operations**

Jolt provides various operations to manipulate and transform JSON data. The following types of operations are the most commons:

* **shift**
* **default**
* **remove**
* **sort**
* **cardinality**
* **modify-default-beta**
* **modify-overwrite-beta**

Below is a detailed description of these Jolt operations:

To perform the commands covered in this tutorial you can either connect with your sandbox using SSH (mobaXterm) or create a Zeppelin note and use the **shell** interpreter (**%sh**).

%sh

jolt transform <jolt-spec> <json-input>

**Shift**

The **shift** operation is used to move data from one location to another in the output JSON. It supports basic mapping of keys and values, as well as nested structures. Example:

Consider the following input JSON file:

{

"name": "John",

"age": 30,

"address": {

"city": "New York",

"country": "USA"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"fullName" : "John",

"city" : "New York",

"country" : "USA",

"age" : 30

}

JOLT Specification:

[

{

"operation": "shift",

"spec": {

"name": "fullName",

"address": {

"city": "city",

"country": "country"

},

"age": "age"

}

}

]

In this example, values from keys `**name**`, `**address.city**`, `**address.country**`, and `**age**` are shifted to `**fullName**`, `**city**`, `**country**`, and `**age**` keys in the output JSON, respectively.

**Default**

The **default**operation sets default values for missing keys in the output JSON. It ensures that all keys present in the output JSON have values, even if they are missing in the input JSON.

Example: Consider the following input JSON file:

{

"name": "John",

"address": {

"city": "New York"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"name" : "John",

"address" : {

"city" : "New York",

"country" : "Unknown"

},

"contacts" : [ {

"type" : "email",

"value" : "john@example.com"

}, {

"type" : "phone",

"value" : "123-456-7890"

} ],

"age" : 0

}

JOLT Specification:

[

{

"operation": "default",

"spec": {

"address": {

"city": "Unknown",

"country": "Unknown"

},

"age": 0

}

}

]

In this example, if the `**address**` or `**age**` keys are missing in the input JSON, default values of  “**Unknown**” and `**0**` will be assigned to them in the output JSON, respectively.

**Remove**

The  **remove** operation removes specific keys or paths from the output JSON. It helps to filter out unwanted data from the output.

From the following input JSON file we want to remove the **contacts** node from the output.

The input JSON:

{

"name": "John",

"address": {

"city": "New York"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"name" : "John",

"age" : 30,

"address" : {

"city" : "New York",

"country" : "USA"

}

}

JOLT Specification:

[

{

"operation": "remove",

"spec": {

"contacts": ""

}

}

]

In this example, the `**contacts**` key and its value will be removed from the output JSON.

**Sort**

The  **sort** operation sort fields name and objects in a JSON in alphabetical order. This operation does not sort the nested attributes.

The input JSON contains information about Customers:

{

"name": "John",

"address": {

"city": "New York"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"address": {

"city": "New York",

"country": "USA"

},

"age": 30,

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

],

"name": "John"

}

JOLT Specification:

[

{

"operation": "sort"

}

]

In this example, all the fields in the output are sorted alphabetically.

**Cardinality**

The  **cardinality** operation is used to transform simple fields and objects into lists of objects and vice-versa.

Usage **RHS**

Valid Cardinality Values:

* ‘**ONE**’: If the input value is a List, grab the first element in that list, and set it as the data for that element. For all other input value types, no-op.
* ‘**MANY**’: If the input is not a List, make a list and set the first element to be the input value. If the input is “null”, make it an empty list. If the input is a list, no-op.

Considering the following input JSON:

{

"colors": [

{

"white": {

"red": 33,

"blue": 11,

"green": 59

}

},

{

"black": {

"cyan": 40,

"magenta": 60,

"purpule": 40

}

}

]

}

No matter how many colors there are, We just want the first one in the array, but as a single value:

Desired output JSON:

{

"colors" : {

"white" : {

"red" : 33,

"blue" : 11,

"green" : 59

}

}

}

JOLT Specification:

[

{

"operation": "cardinality",

"spec": {

"colors": "ONE"

}

}

]

In this example, we limited the output to the first element in the array as a single value.

**Modify-overwrite-beta**

**Modify-default-beta**

The `**modify-overwrite-beta**` and `**modify-default-beta`** operations modifie the values of keys in the output JSON based on predefined rules. These operations allow you to apply transformations such as concatenation, formatting, or arithmetic operations to values.

While **modify-default-beta** will assign a value to a field if it doesn’t exist, **modify-overwrite-beta** will overwrite any value even if the field already exists.

**modify-overwrite-beta** also allows us to apply functions to our JSON.

They are:

* **String**  
  toLower, toUpper, concat, join, split, substring, trim, leftPad e rightPad
* **Number**  
  min, max, abs, avg, intSum, doubleSum, longSum, intSubtract, doubleSubtract, longSubtract, divide e divideAndRound
* **Type**  
  toInteger, toDouble, toLong, toBoolean, toString, recursivelySquashNulls, squashNulls, size
* **List**  
  firstElement, lastElement, elementAt, toList, sort

The input JSON:

{

"name": "John",

"address": {

"city": "New York"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"name" : "john",

"age" : 30,

"address" : {

"city" : "New York",

"country" : "USA"

},

"contacts" : [ {

"type" : "email",

"value" : "john@example.com"

}, {

"type" : "phone",

"value" : "123-456-7890"

} ]

}

JOLT Specification:

[

{

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

"spec": {

"name": "=toLower"

}

}

]

​

In this example, the value of  `**name**` key will be converted to lower case in the output JSON.

**Conditional Shift**

The `**Conditional Shift**` operation allows you to conditionally move data based on certain criteria. It supports if-else logic to handle different cases.

Consider the following input JSON:

{

"name": "John",

"address": {

"city": "New York"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"address" : {

"city" : "New York",

"citizenship" : "American"

}

}

JOLT Specification:

[

{

"operation": "shift",

"spec": {

"address": {

"city": "address.city",

"country": {

"USA": {

"#American": "address.citizenship"

},

"\*": {

"#Foreigner": "address.citizenship"

}

}

}

}

}

]

In this example, we manage to establish the client’s citizenship from his country of origin. On the above transformation, we use the IF-ELSE principle to check the “**country**” value. If it is “**USA**“, we complete the “**citizenship**” field with the “**American**” value. If the “**country**” value is a country other than “**Brazil**“, we complete the “**citizenship**” field with the “**Foreigner**” value.

**JOLT Special Characters**

In Jolt, special characters are used to define transformation rules and perform advanced operations during JSON data transformation. Here’s a detailed description of some commonly used special characters in Jolt:

**Dot (`.`)**

The dot (`**.**`) is used to access nested keys within JSON objects. It allows you to specify the hierarchy of keys in the input JSON to navigate to a specific value.

Consider the following input JSON file:

{

"name": "John",

"age": 30,

"address": {

"city": "New York",

"country": "USA"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"client" : {

"name" : "John"

}

}

JOLT Specification:

[

{

"operation": "shift",

"spec": {

"name": "client.name"

}

}

]

In this example, the value of the `**name**` key  will be moved to the `**client.name**` key in the output JSON.

**Asterisk (`\*`)**

The asterisk (`**\***`) is used as a wildcard character to match any value or key. It is often used in conjunction with other special characters to perform operations on multiple keys or values simultaneously.

Usage: **LHS**  
Operations: shift, remove, cardinality, modify-default-beta and modify-overwrite-beta.

{

"name": "John",

"age": 30,

"address": {

"city": "New York",

"country": "USA"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"name" : "John",

"email" : "john@example.com",

"phone" : "123-456-7890"

}

JOLT Specification:

[

{

"operation": "shift",

"spec": {

"name": "name",

"contacts": {

"\*": {

"type": {

"email": {

"@(2,value)": "email"

},

"phone": {

"@(2,value)": "phone"

}

}

}

}

}

}

]

In this example, the asterisk (`**\***`) acts as a wildcard character, matching any value. It is used to iterate over each item in the `**contacts**` array. Depending on the value of the `**type**` key (`**email**` or `**phone**`), the value of the `value` key is moved to either the `email` or `phone` key in the output JSON.

**At Sign (`@`)**

The At Sign (`**@**`) References the value of a field or object contained in the input JSON, but has different effects depending on its usage.

Usage: **LHS** and **RHS**  
Operations: shift (LHS and RHS), modify-overwrite-beta (RHS) and modify-overwrite-beta (RHS).

Consider the following input JSON file:

{

"name": "John",

"age": 30,

"address": {

"city": "New York",

"country": "USA"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"name" : "John",

"country" : "USA",

"city" : "New York",

"fulladdress" : "New York ,USA"

}

JOLT Specification:

[

{

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

"spec": {

"address": {

"fulladdress": "=concat( @(1,city),' ,',@(1,country))"

}

}

},

{

"operation": "shift",

"spec": {

"name": "name",

"address": {

"@(0,country)": "country",

"@(0,city)": "city",

"@(0,fulladdress)": "fulladdress"

}

}

}

]

In this example, we are chaining two Jolt transformations. The first one will concatenate two fields (**city**and **country**) to create a new one (**fulladdress**). Then in the second Jolt transformation we select the fields which will appear in the output.

The at sign (`**@**`) is used to reference the current position or parent object in the JSON structure. It concatenates the value of the `**country**` key and the `**city**` key with a comma and adds the new key `**fulladdress**` to he `**address**` key in the output JSON.

In “**@ (1, city)**” and “**@ (1, country)**” we are taking the value of the “city” and “country” fields to be used as part of the concatenation. The field that will receive that value is the “fulladdress“. The use of **@**in both LHS and RHS involves declaring the level at which we are seeking information and counting levels from level 1 onwards.

In this case, the “**city**” field is on the same level as the “**country**” field, so we use the number 1.

The usage of **@** in LHS follows the same way as in RHS.

**Hash (`#`)**

The hash (`**#**`) is used to refer to the root object of the input JSON. It allows you to access keys or values from the root level of the JSON structure. the hash sign can be used either in input or output sides and has different meaning in each side.

Usage: **LHS** and **RHS**  
Operations: shift

* In input side (**LHS**), it has the function of entering values ​​manually in the output JSON.
* In output side (**RHS**), on the other hand, it is applicable only to create lists and has the function of grouping certain content of the input JSON within the list to be created.

Consider the following input JSON file:

{

"name": "John",

"age": 30,

"address": {

"city": "New York",

"country": "USA"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"name" : "John",

"address" : {

"code" : "1",

"city" : "New York",

"country" : "USA"

},

"communicate" : [ "email", "phone" ]

}

JOLT Specification:

[

{

"operation": "shift",

"spec": {

"name": "name",

"address": {

"\*": "address.&",

"#1": "address.code"

},

"contacts": {

"\*": {

"type": "communicate.[#2]"

}

}

}

}

]

In this example, we are using the Hash (‘**#**‘)  character in LHS and RHS in the Jolt specification. In the LHS it is used to add a default key to the output. While in the RHS it is used to organize the output ‘**communicate**‘ list.

The value contained after the wildcard **#** will always be assigned to the field declared in the RHS, which in our case is the “**code**” field within the “**address**” object.

The use of **#** in RHS involves declaring the level where we are seeking information. The declaration **[#2]** represents the creation of a list (**[ ]**) and that it must group (**#**) all the information that is found **2** levels above. We need this declaration to guarantee the correct grouping of each product with its respective “code” and “price”.

That is, in **“type”: “communicate.[# 2]”** we are taking the value from the “**type**” field and throwing it to another field within a list named “**communicate**“.

However, when creating the list “**communicate**” we will look **2** levels above (level of the list “contacts“) and the way that each item is grouped in the previous “**communicate**” list, this grouping will be maintained in the new “**communicate**” list.

**Ampersand (`&`)**

The Ampersand (`**&**`) uses the content of what is declared in the LHS to compose the structure of the output JSON, without the need to make this content explicit in the transformation. This wildcard is based on the navigation made during the transformation.

Usage: **RHS**  
Operation: shift

{

"name": "John",

"age": 30,

"address": {

"city": "New York",

"country": "USA"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"address" : {

"city" : "New York",

"country" : "USA"

}

}

JOLT Specification:

[

{

"operation": "shift",

"spec": {

"address": "&"

}

}

]

In this example, we take the values of all the fields of the “**address**” key then we extracted to the same object name “**address**” in the output Json. Thus we create a new JSON keeping the input field structure of the input JSON.

**Dollar (`$`)**

The Dollar (`**$**`) references the name of a field or object contained in the input JSON to be used as the value of a field or object in the output JSON.

Usage: **LHS**  
Operations: shift

{

"name": "John",

"age": 30,

"address": {

"city": "New York",

"country": "USA"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"address" : [ "city", "country" ]

}

JOLT Specification:

[

{

"operation": "shift",

"spec": {

"address": {

"\*": {

"$": "address[]"

}

}

}

}

]

In this example, we take the values of all the fields of the “**address**” key then we extracted to the same object name “**address**” in the output Json. Thus we create a new JSON keeping the input field structure of the input JSON.

**Pipe (`|`)**

The Pipe (`**|**`) allows referencing multiple fields or objects of an input JSON so that, regardless of the name of the field or object, its value is allocated to the same destination in the output JSON.

Usage: **LHS**  
Operations: shift

{

"name": "John",

"age": 30,

"address": {

"city": "New York",

"country": "USA"

},

"contacts": [

{

"type": "email",

"value": "john@example.com"

},

{

"type": "phone",

"value": "123-456-7890"

}

]

}

Desired output JSON:

{

"first\_name" : "John"

}

JOLT Specification:

[

{

"operation": "shift",

"spec": {

"name|fname": "first\_name",

"contact": "contact.&"

}

}

]

In this example, in the input JSON, there is the possibility that the field “**name**” comes with the name “**fname**“. We need the transformation to be prepared to recognize the two possibilities and produce only one single field in the output Json (“**name**“).

**Summary**

Jolt Transformation stands as a valuable tool in the world of JSON data manipulation. Its declarative approach to transforming data simplifies the process of mapping, filtering, and reshaping JSON structures. It excels at converting JSON data from one format to another without the need for custom code, ensuring consistency and efficiency in data processing.

Open the Zepplin note

[Introduction To Jolt Transformations](http://localhost:19995/#/notebook/2JY6JME54)