Talend User Components tJSONDoc*

Purpose

This bundle of components is dedicated to work with JSON documents in the most flexible and unlimited way.
Following components exists:

Component	Purpose
tJSONDocOpen	Holds the root of the json document and can be initially loaded from various sources Validate the document with json schema
tJSONDocInput	Selects objects via JSON-path and reads attribute values
tJSONDocExtractFields	Selects objects via JSON-path and reads attribute values but can take the JSON document from an incoming flow. It follows the Talend component pattern of the t*ExtractFields – components and has beside of this the same features as tJSONDocInput
tJSONDocOutput	Builds JSON objects or arrays and sets their attributes
tJSONDocSave	Renders the final JSON tree pretty formatted as String Validate the document with json schema
tJSONDocInputStream	Reads large JSON files and uses a stream parser to read the values or objects.
tJSONDocValidationInput	Provides the validation result as flow. No output records mean no problems (or no validation was done)
tJSONDocMerge	Merge one json document (and here parts of it) in another json document
tJSONDocDiff	Compare 2 json documents and provide a flow with details to the differences

The idea behind these components is to read, validate or build complex JSON documents in a fain grained way. Means you read or write in sub jobs only parts of the documents and the components references its parent nodes and enhance them.

This way you can build or read JSON documents in any way. You can of course read and write similar – means you can perform any transformation.

These components use the JSON path syntax:
http://goessner.net/articles/JsonPath/
To check your json path expressions you can use online evaluators like this:
http://jsonpath.com/

But for special features like automatically creation of missing nodes in a hierarchy, only the dot-notation is currently supported.

Talend-Integration

You find these components in the studio in the palette under JSON.

Component tJSONDocOpen

This component is the root of the JSON document.
This component can create a new empty root (as Object or as Array) or can read the initial JSON document from a source:

- A file
- The input field containing a Java String or plain text representing a JSON document
- A column of an input flow

Because this component carries the necessary library, this component is always necessary in any use case.
Where you place this component decides about the overall document structure.
If you place it at the beginning of a job, this means, in your job you build one document.
If you place within an iteration or flow (this is always possible) it means you create as much documents as you have rows in your flow (e.g. per request or database record)

Basic settings

Property	Content
Setup the document as/from	Choose here how to create the initial nodes.
Validate input	This option appears if you have setup a json schema in the advanced settings.
Die on validation errors	This option allows to decide should the component die if the json-schema-validation fails or not.

Various ways to build the initial document

Create an empty ObjectNode
This creates an empty node: {}

Create an empty ArrayNode
This creates an empty array node: []

Read from input flow column
This is especcially useful if the document has to be created(initiated) within a flow, e.g. every request of a tRESTRequest have to build its own new document or build for every database record one document. In this case decide in which column of the incoming schema the initial json content has be read.

Read from file
Point here to a file containing the json content you want to read.

Read from input field below as Java Code
The now visible input field expects Java Code creating the json content as String. This helps in case you need some dynamic in this initial content.
Here an example how it can look like:

```
"{\n"
    + "  \"level-1\" : {\n"
    + "    \"level-2\" : [ {\n"
    + "      \"id\" : \"abc\",\n"
    + "      \"level-3\" : [ {\n"
    + "        \"integer-value\" : 10,\n"
    + "        \"jsonString\" : {\n"
    + "          \"a1\" : \"v1\"\n"
. . .
    + "}"
```

The option: “Simplified line breaks” means you can put here content in the way you usually do e.g. in the database input components for the SQL. In this case the line breaks will be added automatically, and you do not need to chain the content with Java String operation, and you do not need to quote every line.
Here an example of the simplified notation:

```
"{
   \"level-1\" : {
     \"level-2\" : [ {
       \"id\" : \"abc\",
       \"level-3\" : [ {
         \"integer-value\" : 10,
         \"float_val\" : 1.1,
         \"double_val\" : 1.2
       } ]
     ] }
   }
}"
```

You have to escape the double quotas within the content, and it needs a double quota at the start and the end.
This is also the place where you put the context variable containing your json content.
E.g.:
`context.json`

Read from input field below as plain text
The same content as abow but now as real plain json content without any Java language parts.
This is also a very good help while testing your job. Simply pleace here your test document if you have to parse it.

```
{
  "level-1" : {
    "level-2" : [ {
      "id" : "abc",
      "level-3" : [ {
        "integer-value" : 10,
        "float_val" : 1.1,
        "double_val" : 1.2,
        "bigDec_value" : 1.3,
        "bool_val" : true,
        "date_val" : "14-06-2016",
        "jsonString" : {
          "a1" : "v1"
```

```
},
```

Advanced Settings

Property	Content
JSON schema	Choose if you want to use a json schema to validate the initial document and where the schema should be taken. **No JSON schema**: no schema avaliable, no validation possible **Read from plain text**: This option adds an input box (mandatory) below and here you can put in the text of the schema without any java code and escape chars. **Read freom java code**: This option adds an input box (mandatory) below which expects java code to setup the schema. It means the schema is either a String literal or you add here a context or globalMap variable containing the schema text. This schema can be used within the tJSONDocOpen and the tJSONDocSave component. Both are referencing to exact this json-schema! If you choose anything else than “No JSON schema” then in the basic settings, you will get the input check fields to enable json schema validation.

Return values

Return value	Content
ERROR_MESSAGE	If the parsing will fail, the error message goes here.
CURRENT_NODE	This is the root JsonNode in this case.
CURRENT_PATH	The current path to the current node (always $ in this component). This return value here exists because of simplifiying the process.
COUNT_OBJECTS_WITHIN_ROOT	Number of objects in the root node (at the very first level, no recursive count)
NB_VALIDATION_PROBLEMS	Number problems found in json-schema validation. 0=no problems.

Component tJSONDocInput

This component is used to read values from the JSON document.
It can build an hierachy of components (also with tJSONDocOutput) to reflect the JSON document structure.

Basic settings

Property	Content
Parent JSON Document	Choose here thet JSONDoc* component which current processing node should be the starting point to read.
Take the values from the referenced parent object	If you want to read (reuse) the json object from the parent component (because of other attribute set etc.) check this option. The option below will disappear.
JSON path to loop	This is the path to the json document which surfes as loop element. You can set here an absolute json-path or a relative attribute path. **If the path starts with an $ the path will be parsed with the original json-path methology**. **If the path does not start with an $, it means it is simple a chain of attributes describing the way to the target loop node** but starting from a particular node. This goes typically with a parent JSON document like tJSONDocInput or tJSONDocOutput. **The value "." (standalone) means, the source for the attributes is the referenced parent object.** This way 2 components can read from the same object but e.g. different attributes with different constraints (not null or missing). You could also simply check the option above! **If the addressed node is an object node,** the component reads the attributes from this node and provides only one row. **If the addressed node is an array node** which the component reads the attributes of the addressed child nodes and provides one row per child node. **If the addressed node is a value array**, the component provides per element in the array a new row and set the value into the given schema columns. Actually, it does not make sense to have a schema with more than one column in this mode.
Send an empty record if the parent does not exist	With this option you can solve use cases in which you have to provide default values even if the parent object does not exist. Otherwise there is no record and therefore you cannot apply a “missing attribute”-value.
Die if attribute does not exist	If the JSON path does not exists but the existance of the objects withing the path is mandatory you can let the component die with a meaningful error message.
Die if parent object is empty	In case of the parent object is an array and there should exists objects within the array, this option detects that and let the component die.
Attributes	Configure here the attributes you want to read from the json objects. **Column:** the schema column **Alternative Name:** You can set here the name of the attribute if it is not the same as the schema column name. This way you can set names which are not compatible with Java conventions like attribute names with a minus e.g. If this expression results in an empty or null name, the schema name will be used instead. **Use Column:** Decide here which column you want to fill from the json object. **Allow missing:** Set this option if the attribute can be missed. The default option is off. **Value if attribute is missing:** Set here a replacement value for missing attributes. This value could be used later in the job to detect if the attribute was null or missing.
Schema	The default schema editor. Please use the Date pattern to parse the Date strings in the json document. JSON actually does not have a standard date pattern or even such a data type. It depends on string parsing to work with dates and timestamps. The component use a internal default pattern "yyyy-MM-dd’T’HH:mm:ss:SSS" if you do not provide a date pattern in the schema.

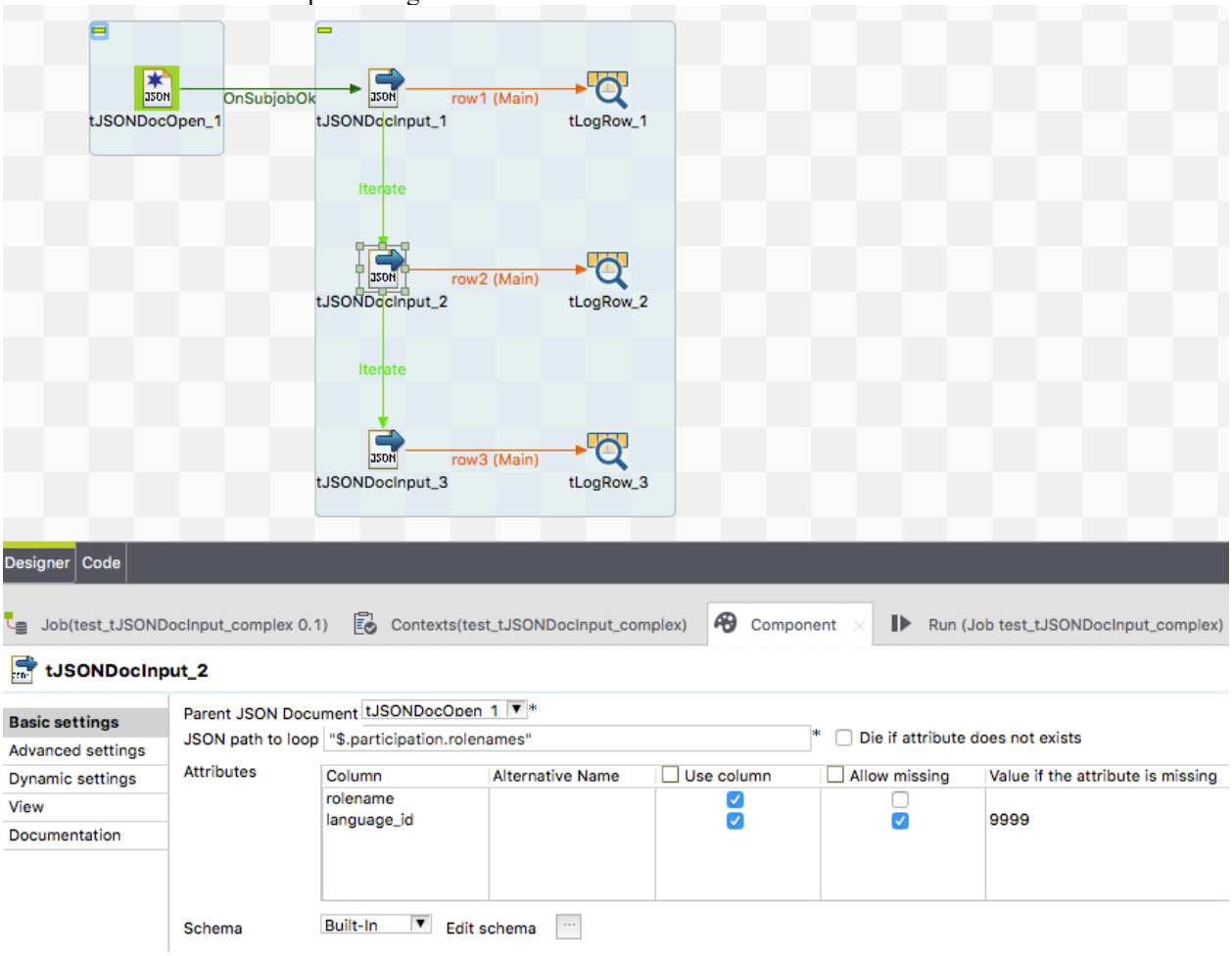
Provide values as comma separated list	If you get a single attribute from the source, you can get with this option a comma separated list of all values (also optionally formatted as SQL list). Such a list is great if you use this as where condition for a next level query. This list is a return value VALUES_AS_COMMA_SEP_LIST

Return values

Return value	Content
ERROR_MESSAGE	If the parsing will fail, the error message goes here.
CURRENT_NODE	This is the current JsonNode from which the attributes are currently read.
NB_LINE	The number of outgoing rows.
NB_NOT_NULL_ATTRIBUTES	The number of attributes which are not missing and not null in the current node
CURRENT_PATH	The JSON-Path to the current node as String
COUNT_OBJECTS	Count objects to iterate through. The number of objects is known by the parent object.
PARENT_ATTRIBUTE_EXISTS	The parent attribute (the attribute to address the current object) exists -> true, otherwise false
VALUES_AS_COMMA_SEP_LIST	String with comma separated list of all values.

Scenario: Reading a document with multiple levels:

This shows how to chain the processing with Iterate flows.



The json path points to the node which is the loop element.
Because we have a json path starting with $ we read just from the root element. But is could also be possible to reference the tJSONDocInput_1 and use a relative path which takes the current node from tJSONDocInput_1 as starting point to find the node(s) to read from.

Take note of the value 9999 in the column Value if attribute is missing. This value will be send if the attribute is missing at all. A null value is NOT a missing attribute!

Scenario: Reading document with multiple nested arrays

The goal is to have values from the higher levels and the details of the lowest levels in one flow.
There are 2 multile ways: This scenario describes the way by addressing the objects.

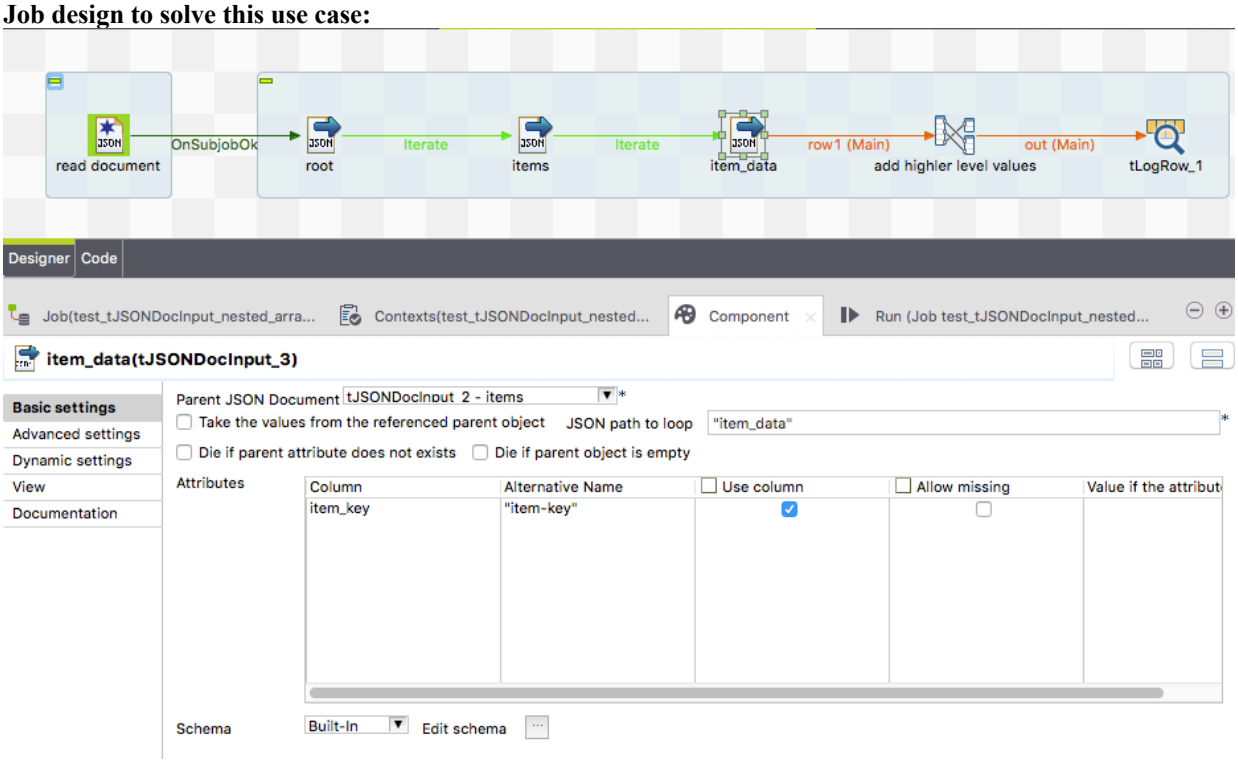
Here the input:

```
[
{
    "header": "global header1",
    "items": [
        {"group_header": "group_header1",
         "item_data": [
               {"item-key": 1},
               {"item-key": 2},
               {"item-key": 3}
         ]
        },
        {"group_header": "group_header2",
         "item_data": [
               {"item-key": 1},
               {"item-key": 2},
               {"item-key": 3}
         ]
        }
    ]
},
{
    "header": "global header2",
    "items": [
        {"group_header": "group_header1",
         "item_data": [
               {"item-key": 1},
               {"item-key": 2},
               {"item-key": 3}
         ]
        },
       {"group_header": "group_header2",
         "item_data": [
               {"item-key": 1},
               {"item-key": 2},
               {"item-key": 3}
         ]
        }
    ]
}
]
```

… and here the desired output:

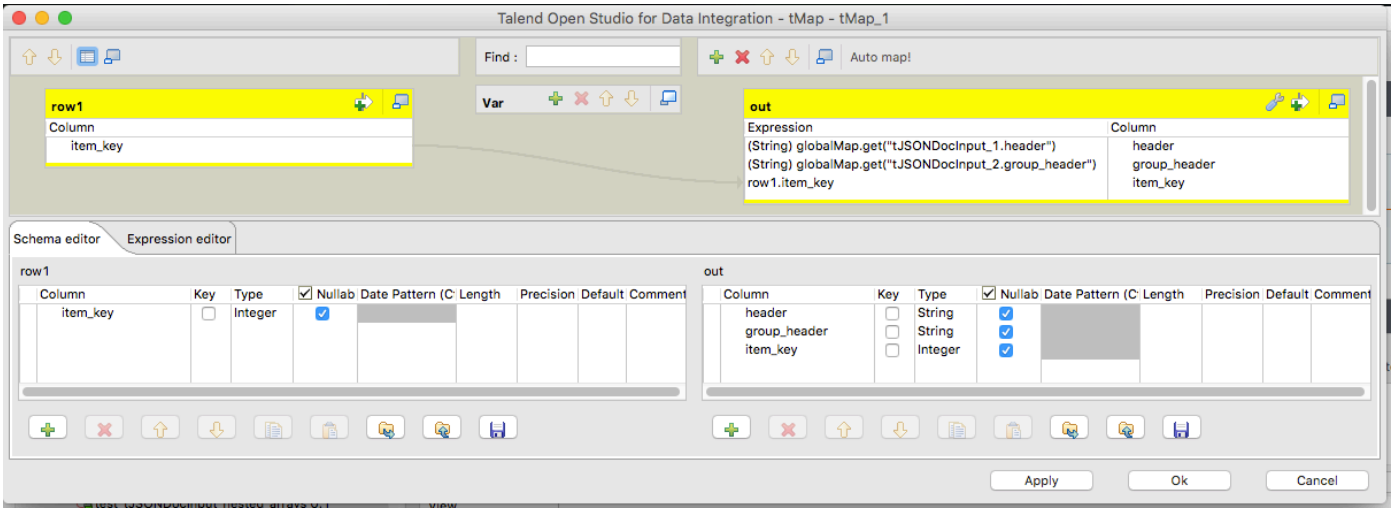
```
header          |group_header |item_key
-----------------------------------------------------------
global header1|group_header1|1
global header1|group_header1|2
global header1|group_header1|3
global header1|group_header2|1
global header1|group_header2|2
global header1|group_header2|3
global header2|group_header1|1
global header2|group_header1|2
global header2|group_header1|3
global header2|group_header2|1
global header2|group_header2|2
global header2|group_header2|3
```

Job design to solve this use case:



The first tJSONDocInput references as parent the tJSONDocOpen component. All other references the current predecessor.

To provide the higher-level values we have to use a tMap and adds these values into the output flow.
All components put their current values into the global map with the key: <unique-component-id>.<schema-column-name>
Example refer to the picture below.



Component tJSONDocOutput

This component is dedicated to creating and write all kind of json nodes.
It can be chained with other tJSONDoc components and work relatively on top of the current node of the referenced parent component.
It is also possible to use foreign key columns to mount objects to higher level objects.

Basic settings

Property	Content
Parent JSON Document	Choose here the JSONDoc* component which current processing node should be the starting point to read.
Use foreign key column to address parent node	This option enables the foreign key assignment feature. This feature allows to add ther current nodes to a higher-level node. Example an order item to an order and the matching order will be addressed by an order_id got from the order-item (foreign key).
Foreign key column	Choose from the incoming flow the column which points to the higher-level object in the parent json object. This option appears if you check the option above.
Use the referenced parent object	Use this option if the referenced parent component has already the obeject as current node you want to write in here.
JSON path for the current parent	This is the path (in dot-notation) to the current element you want to create and write. The last part of the path is the attribute under which the nodes are created or be written. This option appears if you uncheck the option above.
Output structure	Setup here which kind of output structure you want: **Array of Object nodes:** The component creates or uses an ArrayNode and add to this array node for every incoming row a new object node and set their attributes. **One single object node:** The component creates or uses a single ObjectNode and set its attributes. In this mode multiple incoming rows actually does not make sense. The last record will determine the content. **Array of simple values:** The component creates a value array and takes every incoming row as one value element. If the schema has multiple schema columns all column values are written as their own array value. It is actually more meaningful to have only one schema column. Actually, in this mode the schema should have only one column or only one column should be used here. **Array of arrays:** The component creates per incoming row an array (every schema column carries one value in the array), and these arrays will be added to an array referenced by the component json path (just like in the option “Array of Object nodes”. This mode is also known as CSV mode. Not a really well designed json document but unfortunately sometimes necessary to be compatible with old CSV interfaces.
Attributes	Configure here the attributes you want to read from the json objects. **Column:** the schema column **Alternative Name:** You can set here the name of the attribute if it is not the same as the schema column name. This way you can set names which are not compatible with Java conventions like attribute names with a minus e.g. If this expression results in an empty or null name, the schema name will be used instead. **Use Column:** Decide here which column you want to fill from the json object. **Is a JSON object/array:** Check this option if the content of the schema column is a JSON node, otherwise the content will be treated as value and the content will be escaped to be compatible with json strings. If the schema column is of an Object type the content will be taken as JsonNode object, if the content is of String type, the content will be parsed to a JsonNode. **Omit attribute if value is null:** With this option you can prevent writing the attribute if the value is null
Schema	The default schema editor. Please use the Date pattern to parse the Date strings in the json

	document. JSON actually does not have a standard date pattern or even such a data type. It depends on string parsing to work with dates and timestamps. The component use a internal default pattern "yyyy-MM-dd'T'HH:mm:ss:SSS" if you do not provide a date pattern in the schema.

Advanced settings

Property	Content
SQL code in case of there are no records and therefore no keys	This option is part of the foreign key feature. In the return values you can get an SQL like String containing all the keys fetched in this component as list to use this in a where clause for the child records. In case you do not have get any rows in this component you need a where condition which takes care you do not get any child records. You could measure of course the number of incoming records and prevent the next parts of the job (reading and ading child objects) but you can also simplify the job design with a where condition which prevents the job from getting children for none existing parents.

Return values

Return value	Content
ERROR_MESSAGE	If the parsing will fail, the error message goes here.
CURRENT_NODE	This is the current JsonNode from which the attributes are currently written.
NB_LINE	The number of incoming rows
CURRENT_PATH	The path to the current node as JSON-Path String
KEYS_AS_SQL_IN_CLAUSE	The primary keys fetched from the incoming flow. Example: `in (234,9341,632)`

How foreign key assignment works

The great advantage of this mode is you reduce the queries in the source data drastically. In the best case you have to query the data as much you have levels and not as much you have parent instances.

Imagine, you have an order and order-item relation and your object contains about 100 orders.

With the foreign key mode, you need only 2 queries, otherwise you have to select the order-items once per order.

First of all, read the root objects and take care you mark ONE column as key column (you do not need to use this column in your json object). This procedure works only with one column marked as key column!

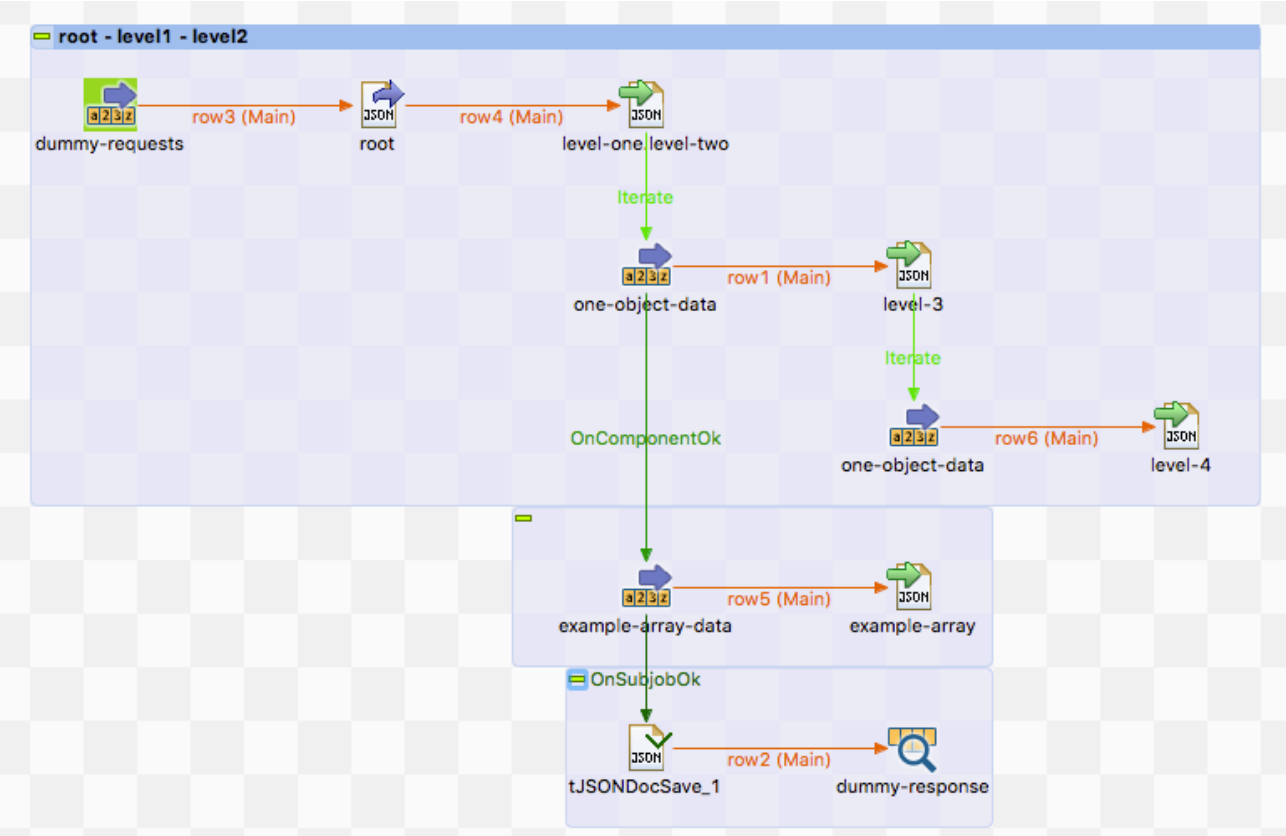
The tJSONDocOutput of the first (parent) flow memorizes all objects in a Map with a key (in case of there is one column marked as key column) and the current node as value.

The fetched keys will be provided as list (also SQL formatted if needed) to support the appropriated selection of the child objects for the read parent objects.

The referencing tJSONDocOutput for the children needs to read a foreign key column pointing to the root object (e.g. an order-number from an order-item). For every incoming record the child component takes the foreign key and ask the referenced parent component for an object with this key. The children will be added to the matching parents. You can take care there is always a parent (should be, but a cae of inconsistent data it could be the case a child point to a not existing parent). Simply let the component die if a child cannot be assigned to a parent.

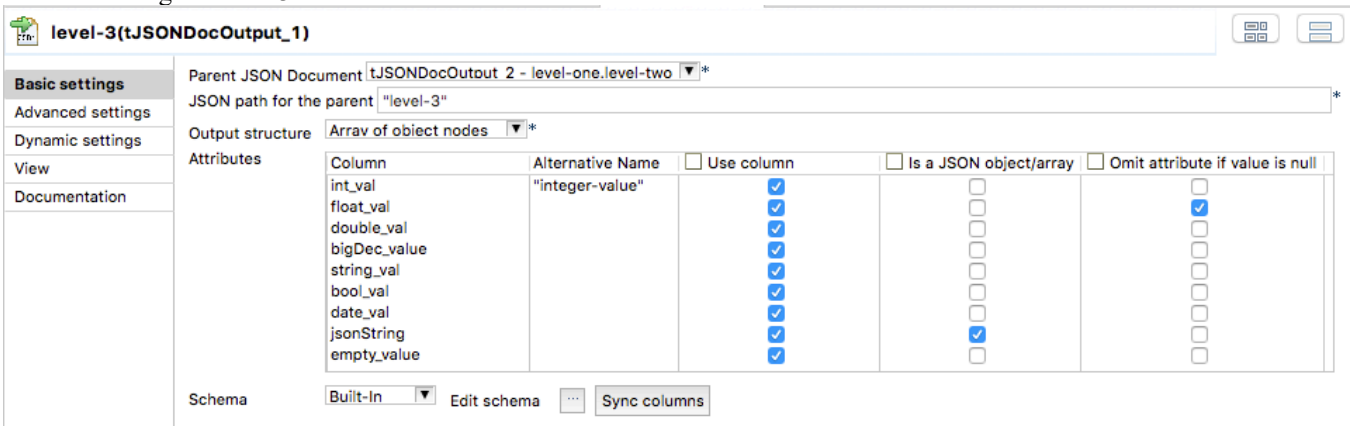
Scenario 1: Write a complex json document

This scenario shows how to build a multi-level json document like the example in tJSONDocOpen.



We have a json document with 4 levels and an additional value array.
Please note the order in which the rows and iterates will be processed.
At first one row will processed and right after this one row the iteration takes places also once a time.
One iteration per one output flow record. This is important because in the deeper levels you can build json objects as children of the current written object of the addressed parent object.

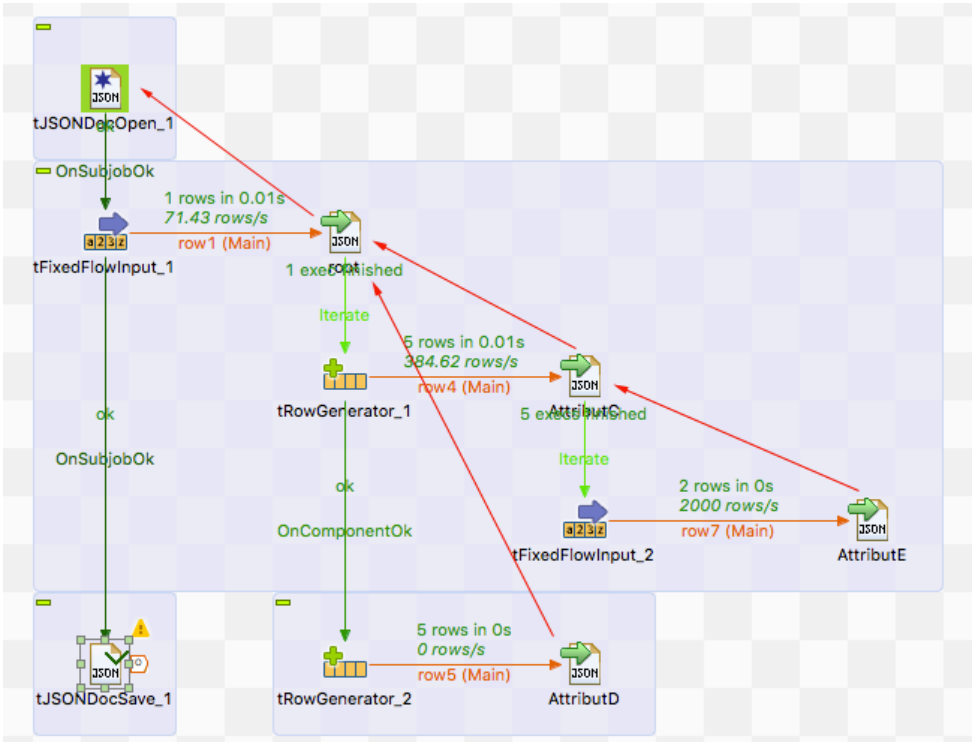
Here the settings of level-3:



As you can see it is based on the level-one.level-two node.

Scenario 2: Example of multi-level document creation

This example shows a bit deeper how the concept of referencing to a parent component works.
This example does not use the efficent foreign key method.

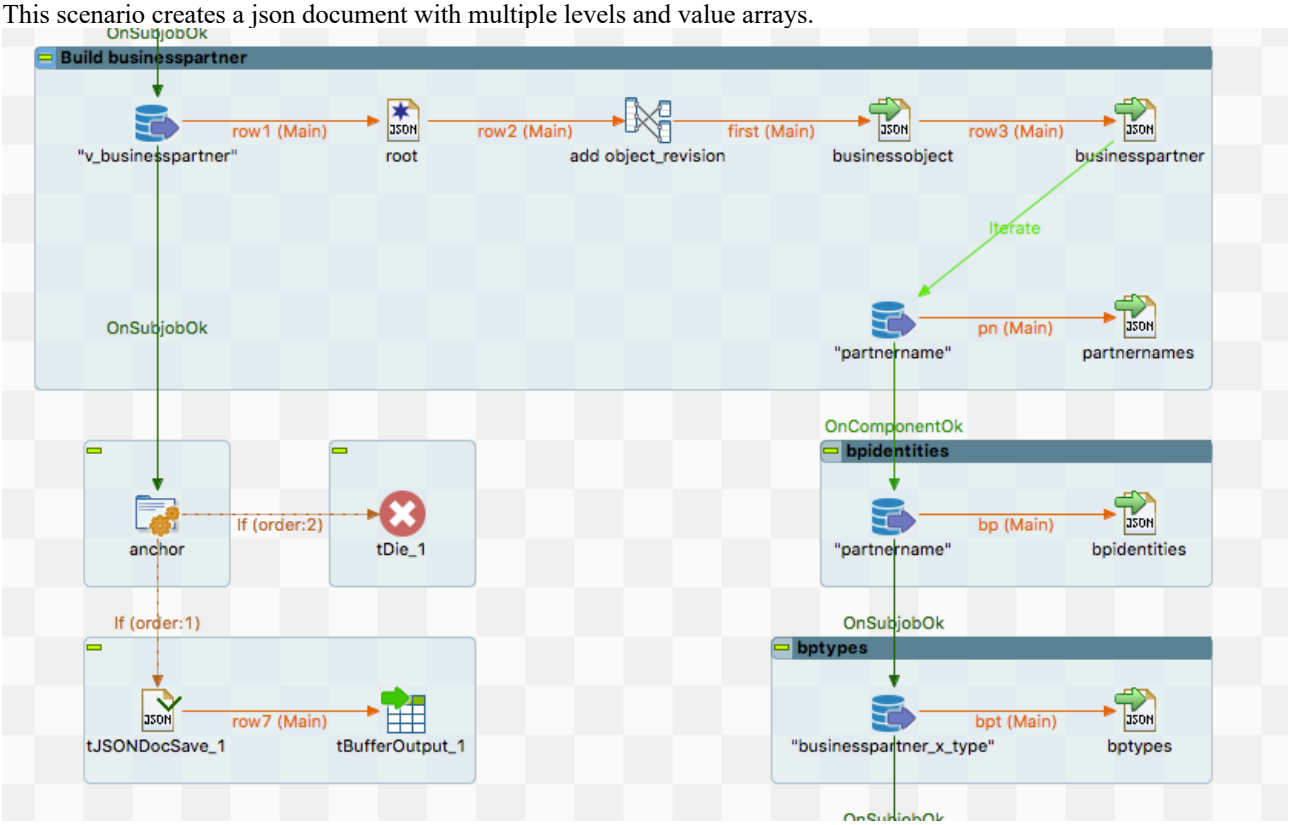


The red arrows show which parent component all components references.
Here the result:

```
{
  "AttributA" : "AAA",
  "AttributB" : "BBB",
  "AttributC" : [ {
    "AttributC_ID" : 1,
    "AttributE" : [ "EEE1", "EEE2" ]
  }, {
    "AttributC_ID" : 2,
    "AttributE" : [ "EEE1", "EEE2" ]
  }, {
    "AttributC_ID" : 3,
    "AttributE" : [ "EEE1", "EEE2" ]
  }, {
    "AttributC_ID" : 4,
    "AttributE" : [ "EEE1", "EEE2" ]
  }, {
    "AttributC_ID" : 5,
    "AttributE" : [ "EEE1", "EEE2" ]
  } ],
  "AttributD" : [ {
    "id" : 6
  }, {
    "id" : 7
  }, {
    "id" : 8
  }, {
    "id" : 9
  }, {
    "id" : 10
  } ]
}
```

Scenario 3: A real live scenario to create a complex json document

This scenario creates a json document with multiple levels and value arrays.

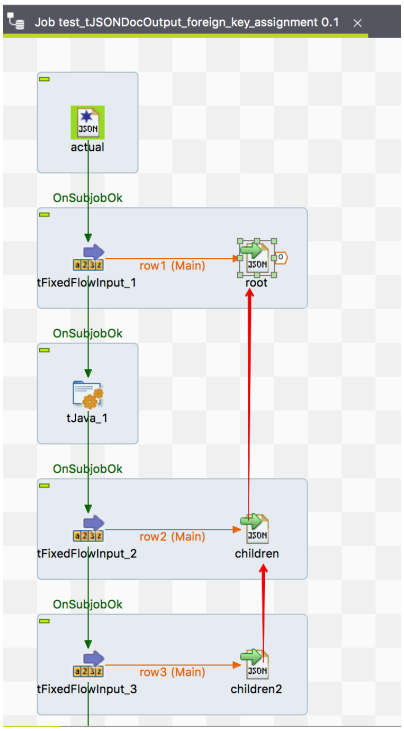


Per record from the database we create a json document and add in the first place the values from the first database input and continue per "businesspartner" with adding more objects like "partnernames".

At the end we write the content with the tJSONDocSave component (see next chapter) in tBufferOutput (we use this job within other jobs and tBufferOutput is a great way to provide content to the parent job.

Scenario 4: Foreign key assignment of child objects to parent objects

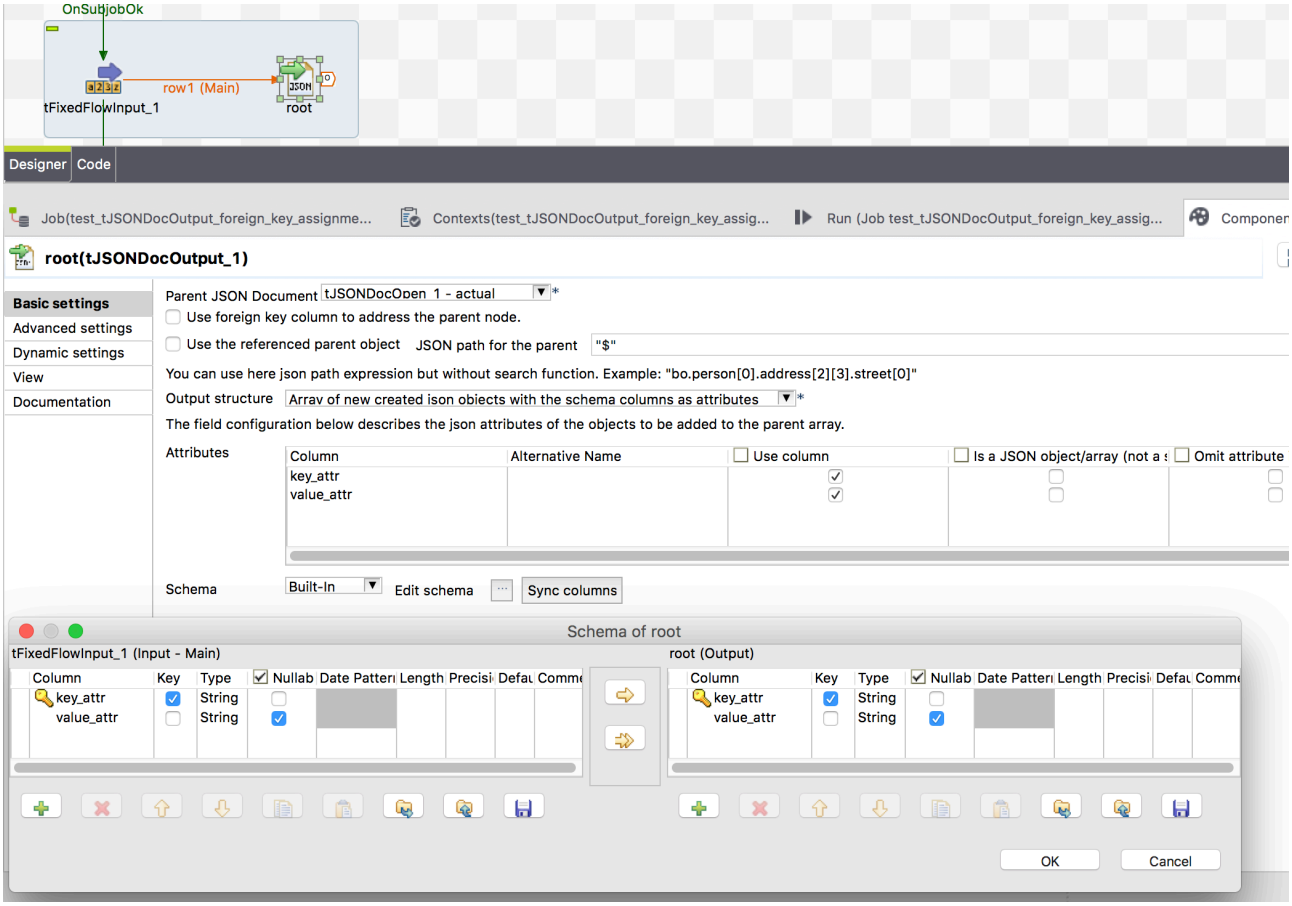
In this scenario we use the foreign key assignment.



This is the expected json document:

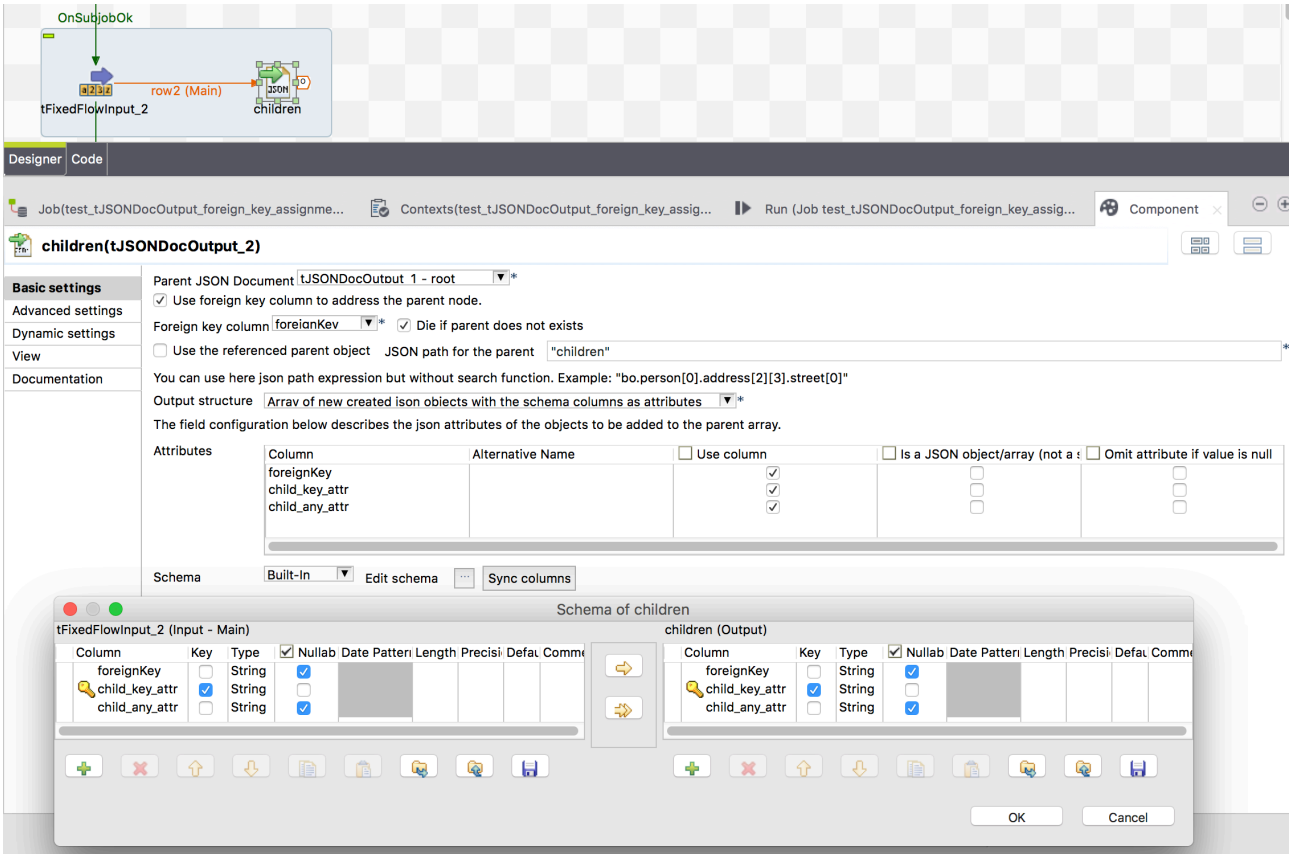
```
[ {
  "key_attr" : "a",
  "value_attr" : "x",
  "children" : [ {
    "foreignKey" : "a",
    "child_key_attr" : "a1",
    "child_any_attr" : "0",
    "children2" : [ {
      "key2" : "k1"
    } ]
  }, {
    "foreignKey" : "a",
    "child_key_attr" : "a2",
    "child_any_attr" : "1",
    "children2" : [ {
      "key2" : "k2"
    }, {
      "key2" : "k3"
    } ]
  } ]
}, {
  "key_attr" : "b",
  "value_attr" : "y",
  "children" : [ {
    "foreignKey" : "b",
    "child_key_attr" : "b1",
    "child_any_attr" : "2",
    "children2" : [ {
      "key2" : "k4"
    } ]
  }, {
    "foreignKey" : "b",
    "child_key_attr" : "b2",
    "child_any_attr" : "3",
    "children2" : [ {
      "key2" : "k5"
    } ]
  } ]
}, {
  "key_attr" : "c",
  "value_attr" : "z",
  "children" : [ {
    "foreignKey" : "c",
    "child_key_attr" : "c1",
    "child_any_attr" : "4"
  } ]
} ]
```

The first tJSONDocOuput is used to build the root level objects:

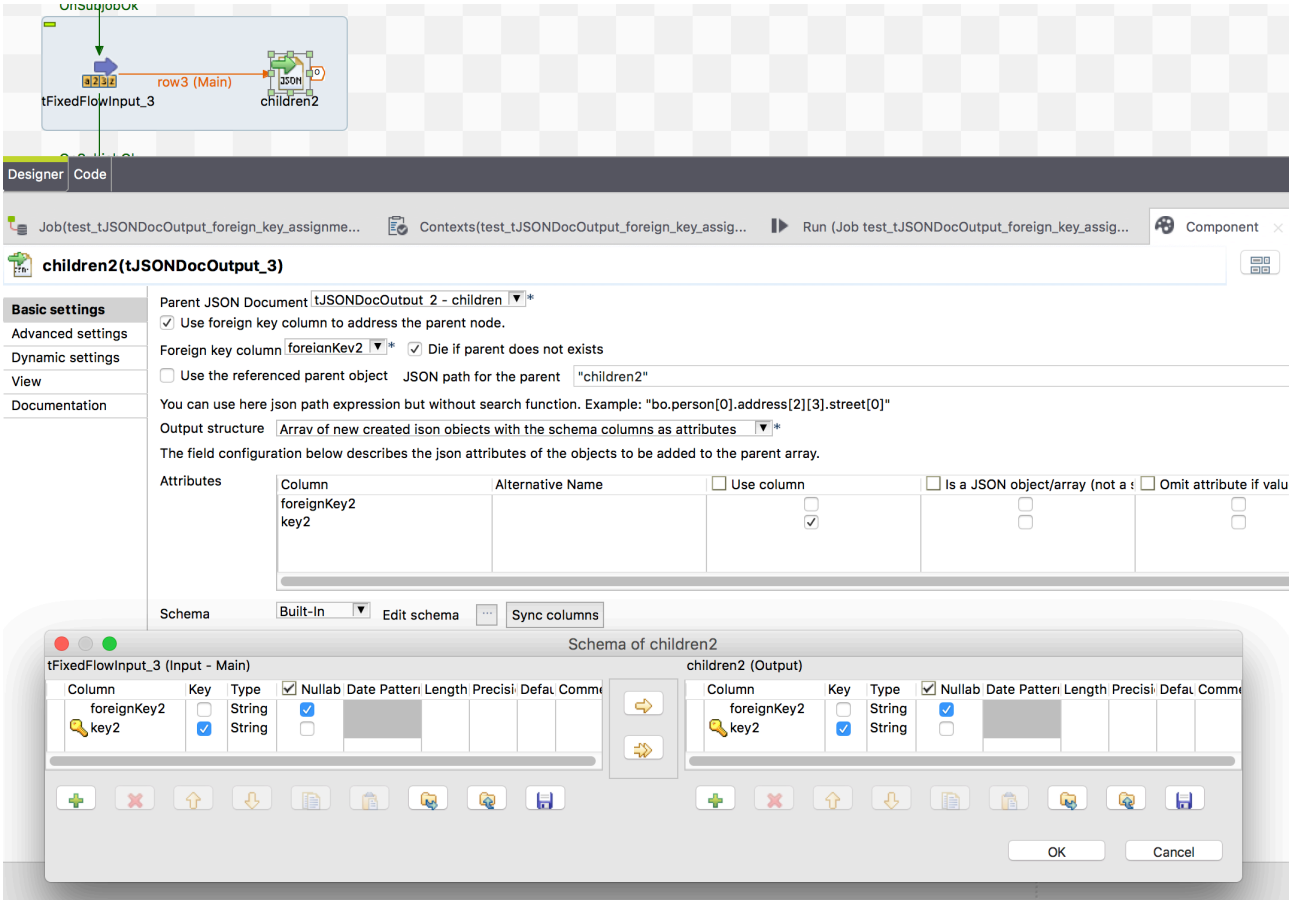


The component which is traget of a foreign key relation need one key field in the schema! All nodes build in this component will be kept in a Map with the key value as key.

The second tJSONDocOutput gets as incoming flow the child objects including a column carrying the foreign key (the actual reference to the parent). Typically, the column with the foreign key is often not part of the desired json object, it is not necessary to use its value within the target json document. Simply uncheck the “Use column” option for this column, this will not affect the foreign-key-mechanism.



The third tJSONDocOutput component build the last level “children2” objects:



Component tJSONDocSave

This component is dedicated to providing the JSON document as pretty formatted string to any kind of output.
It is not mandatory to use this component because it would simple be fine to use the return value CURRENT_NODE from tJSONDocOpen to have the content of the json document.

Basic settings

Property	Content
JSON Document root	Choose here thet JSONDocOpen component which current processing node should be used to render the document output.
Write content into a file	With this option and the file chooser setup the output file in which the conent will be written. The charset is UTF-8 and it uses UNIX style line breaks.
Output schema column	If there is an output flow, choose here the column in which the content has to set as value. As column type String is needed.
Pretty Print	The string output will be formatted in a human readable way. Otherwise everything is in a condensed one-line String.
Ignore empty values	Empty values and arrays will be filtered out
Write content to standard out	Does what is says. This is actually I kind of debug option to see the content on the standard output of the job.

Return values

Return value	Content
ERROR_MESSAGE	If something went wrong, e.g. we cannot write into the file, the error message goes here.
OUTPUT_FILE_PATH	The path to the output file if set. This is useful if the path is calculated and you need that in the further processing.
JSON_STRING	The String content of the json document.

Component tJSONDocInputStream

This component is dedicated to read very large JSON files and extract attributes with JSONPath expressions.
In the current state the capabilities of using JSONPath is limited:
Only the dot-notation is allowed
Search is not possible, only addressing of objects and arrays with the notation [*] is possible.

Basic settings

Property	Content
JSON file path	Set here the file path of the input file
JSON Path to loop	Address the JSON structure you want to loop over. E.g. address the attribute (referencing an array). The syntax here is not fully JSON Path complient and currently very limited. A path must start with $ and all attributes are separated with . (dot) Arrays must be declared in the path with [*] – currently there is not possibility to access one particular element of an array.
Attributes	**Columns:** Schema column **JSON Path:** it the path starts without $ it will extend the loop path to address the attribute **Use column:** switch off the column if you do not want to set this attribute
Schema	The schema editor. Configure here the schema for reading.
Die on error	Let the component die if something went wrong.
Reject schema	The schema of the reject flow. The reject flow will only be used if you switch off the Die on error option.

Advanced settings

Property	Content
Provide JSON loop object	If you want to use the json object returned in every iteration of the loop for some other tJSONDocInput components, switch on this option. Because of the json object must be created and this could have performance impacts, this is an optional feature.

Return values

Return value	Content
ERROR_MESSAGE	If something went wrong - the error message goes here.
NB_LINES	Number rows
NB_LINES_REJECTED	Number lines rejected
CURRENT_NODE	This is the current JsonNode referenced by the loop path. This JsonNode is only prsent if you have switched on the advanced option " Provide JSON loop object". To build such an object reduces the performance typically for about 2-5% - depending of the complexity of the loop element.

Scenario: Reading a json with nested arrays

This describes the reading of a small document to illustrate the parsing feature.

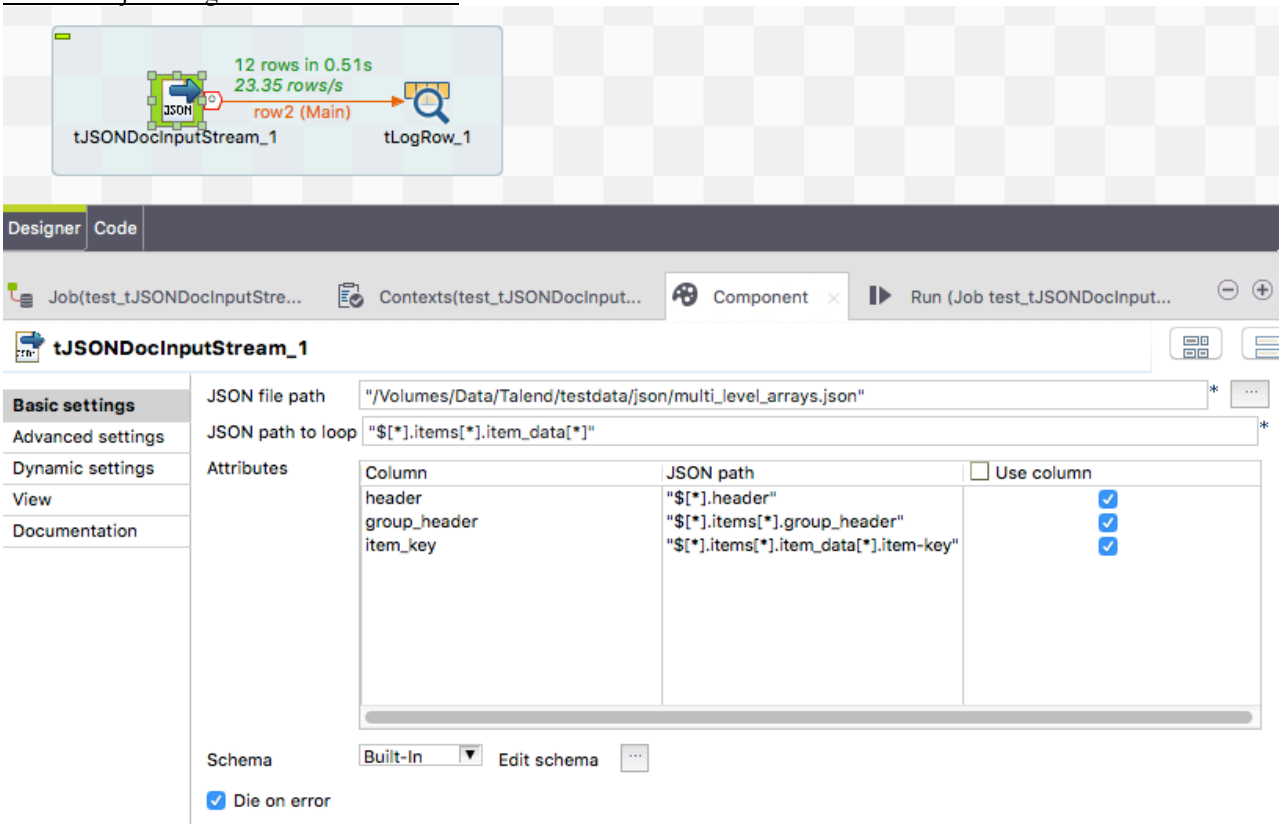
This is the json input:

```
[
  {
    "header": "global header1",
    "items": [
      {
        "group_header": "group_header11",
        "item_data": [
           {"item-key": 111, "item-value" : {"a1" : "b1"}},
           {"item-key": 112},
           {"item-key": 113, "item-value" : {"a3" : "b3"}}
        ]
      },
      {
        "group_header": "group_header12",
        "item_data": [
          {"item-key": 121},
          {"item-key": 122},
          {"item-key": 123}
        ]
      }
    ]
  },
  {
    "header": "global header2",
    "items": [
      {
        "group_header": "group_header21",
        "item_data": [
          {"item-key": 211},
          {"item-key": 212, "item-value" : {"a4" : "b4"}},
          {"item-key": 213}
        ]
      },
      {
        "group_header": "group_header22",
        "item_data": [
          {"item-key": 221},
          {"item-key": 222},
          {"item-key": 223, "item-value" : {"a5" : "b5"}}
        ]
      }
    ]
  }
]
```

Expected Output:

```
.--------------+--------------+--------.
|               tLogRow_1                |
|==============+==============+========|
|header        |group_header  |item_key|
|==============+==============+========|
|global header1|group_header11|111     |
|global header1|group_header11|112     |
|global header1|group_header11|113     |
|global header1|group_header12|121     |
|global header1|group_header12|122     |
|global header1|group_header12|123     |
|global header2|group_header21|211     |
|global header2|group_header21|212     |
|global header2|group_header21|213     |
|global header2|group_header22|221     |
|global header2|group_header22|222     |
|global header2|group_header22|223     |
'--------------+--------------+--------'
```

This is the job design to achive the results:



Component tJSONDocValidationInput

In case of json schema validation takes place, this component returns as flow all detected problems.
This is helpful in case of you have to save or process them.

The component references to a tJSONDocOpen or tJSONDocSave components because currently only these both are performing schema validation.
http://json-schema.org/learn/getting-started-step-by-step.html

Basic settings

Property	Content
Validating JSON component	Choose here the component which performs the json schema validation.

The component provides the problems detected within the validation as flow with a fixed schema.

Column	Type	Description
JSON_PATH	String	JSON path to the current problem.
VALIDATION_TYPE	String	Type of the problem. Examples: **type**: A value for an attribute has a different type than in the schema defined **unionType**: same as type but multiple types are allowed in the schema. Please keep in mind null ist for JSON also a type! **additionalProperties**: A property was found which is not defined in the schema and the affected node is not configured to allow additional properties. **required**: A attribute is missing but required in the object.
VALIDATION_MESSAGE	String	The descriptive message to the problem

Example job using both validating components

(tJSONDocOpen and tJSONDocSave)

The second validation output is triggered from a component which dies in case of validation errors. In such scenarios use the OnComponentError trigger and it is necessary to use a dedicated tDie component because the tJSONDocValidationInput_2 will work most likely sucessful and will therfore reset the error state of the job so far. To let the job intentially die (or fail with an exit code > 0) you have to do this for your self with a dedicated tDie component.

Component tJSONDocDiff

This component calculates the differences between 2 json documents.
In the first place designed to support test cases but also usable for any other use case depeneding in the knowlegde about the difference.

The component expects 2 different documents: the reference document and the test document.
The reference is the document how it should look like and the test document contains the test data.

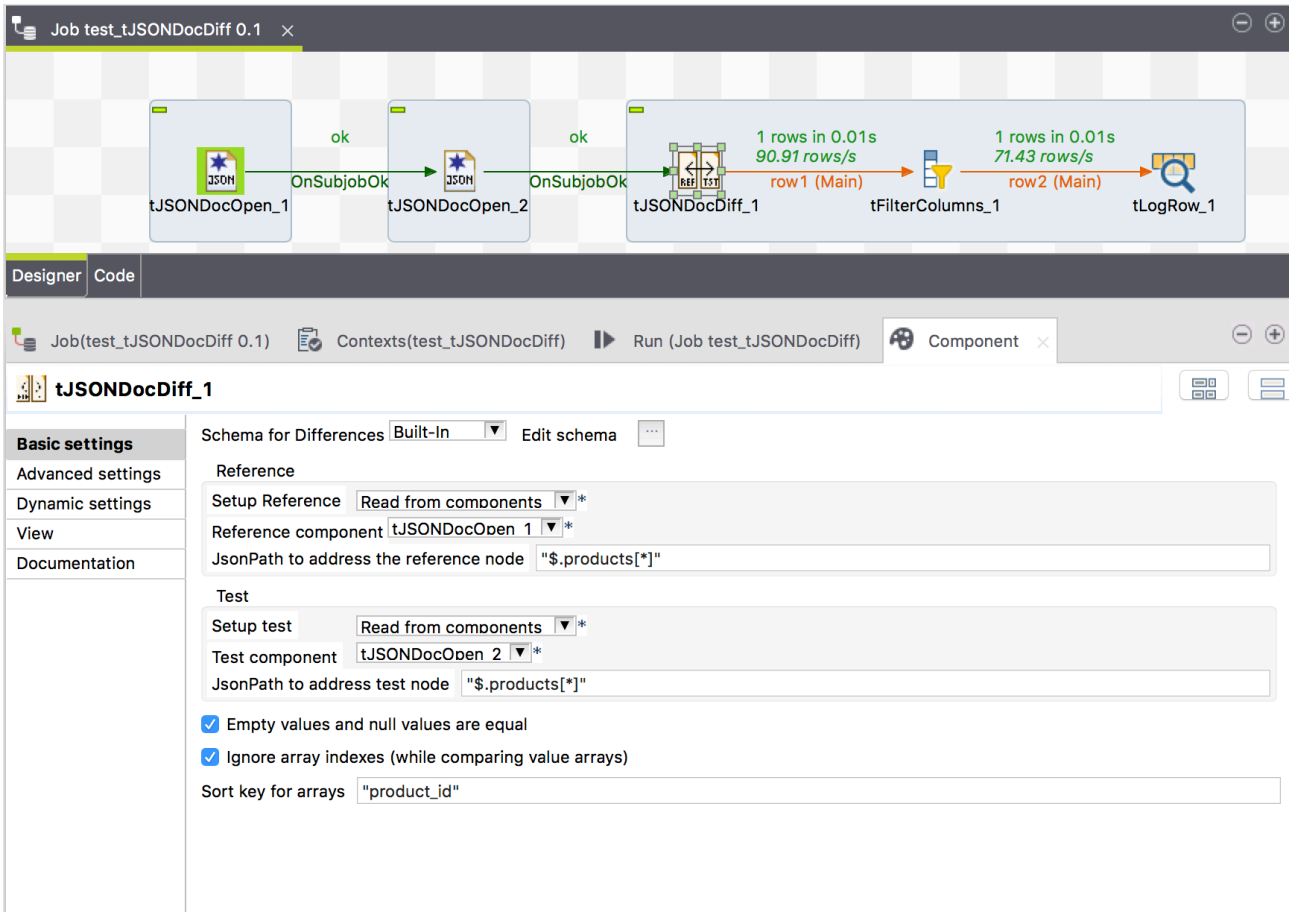
Basic settings

Property	Content
Schema	The outgoing flow with its columns. The schema is fixed designed an cannot be changed. (The columns are explained below).
Setup Reference	**Read from component**: The reference json document will be taken from a tJSONDoc component. This is the simplest way. **Read from input field**: The reference json document will be taken from the input field below. Setup the input in the setting “Reference input”
Reference component	This option is available if you choose “Read from component” above. Choose the tJSONDoc component which holds the reference document.
Reference input	This option is available if you choose “Read from input field” above. You can use context variables or globalMap variables here to setup the reference json document.
Reference json path	Both documents (reference and test) do not need be exactly equal in all objects. If you want to compare only some parts of the entire document, setup here the json path the object which should be the reference document.
Setup Test	**Read from component**: The test json document will be taken from a tJSONDoc component. This is the simplest way. **Read from input field**: The test json document will be taken from the input field below. Setup the input in the setting “Reference input”
Test component	This option is available if you choose "Read from component" above. Choose the tJSONDoc component which holds the test document.
Test input	This option is available if you choose “Read from input field” above. You can use context variables or globalMap variables here to setup the test json document.
Test json path	Both documents (reference and test) do not need be exactly equal in all objects. If you want to compare only some parts of the entire document, setup here the json path the object which should be the test document.
Empty values and null are equal	Means the component will ignore the difference between and empty value and null.
Ignore array indexes while comparing value arrays	Value arrays can have the same values but at different index. With this option, you ignore the exact index and only let the comparison check the existence of all values, regardless the position.
Sort key for arrays	If you have as root object arrays and this array contains objects you can sort the objects by an attribute (sort key) to let the comparison ignore the exact position of the objects in the root array.

Schema

Column	Type	Description
JSON_PATH	String	The path to the detected difference within the test object.
REF_VALUE_STRING	String	The reference value which is different to the test value.
REF_VALUE_NODE	Object	The reference value node which is different to the test value node. Please keep in mind, in JSON a value is actually also a node -> a ValueNode. The value here is of type JsonNode
TEST_VALUE_STRING	String	The tested value, which is different to the reference value.
TEST_VALUE_NODE	Object	The test value node which is different to the reference value node. Please keep in mind, in JSON a value is actually also a node -> a ValueNode. The value here is of type JsonNode
TYPE_MISMATCH	Boolean	If the difference is only the data type. "123456" looks like the same as 123456 but has a different data type: String instead of Integer. In such cases we get here a true.

Scenario: Comparing 2 json documents



This example shows a comparison in which we compare only parts of the json documents. These parts are addressed by the JsonPath expressions.

Component tJSONDocMerge

This component merges one document into another document.
It can merge only parts (addressed by JsonPath).
The identification of the parts rto merge into the target document works with key attributes.
The merge component can be triggert from OnSubjobOk or OnComponentOk. The component does not provide a flow, as input we use 2 different json documents (source and target) and the outcome is the changed target json document.
Source and target json document can be under the same root document, of corse in such cases the path to both must be different and a target node cannot by a part of the source node.

Basic settings

Property	Content
Source setup	**Read from components**: Read the json document which have to be merged into the traget document from a tJSONDoc component. **Read from input field**: Read the Read the json document which have to be merged into the traget document from a Java variable (globalMap or context).
Source component	If the option is active if the option above is set to "Read from components". Choose the tJSONDoc component which provides the source data.
Source input	If the option is active if the option above is set to "Read from input field". Use Java code like `(String) globalMap.get("my_source_json_var")` or `context.my_source_json` to provide the source json.
Loop path over nodes	Use a JsonPath expression here to address the parts of the source json document which should be merged into the target json document.
Key attribute to identifiy the source nodes	Setup here the attribute which is the key for the source nodes.
Target setup	**Read from components**: Take the json document which is the target document from a tJSONDoc component. **Read from input field**: Take the json document which is the traget document from a Java variable (globalMap or context).
Target component	If the option is active if the option above is set to "Read from components". Choose the tJSONDoc component which provides the target data.
Target input	If the option is active if the option above is set to "Read from input field". Use Java code like `(String) globalMap.get("my_target_json_var")` or `context.my_target_json` to provide the target json.
Loop path over nodes	Use a JsonPath expression here to address the parts of the target json document into the source parts should be merged.
Key attribute to identify the target nodes	Setup here the attribute which is the key for the source nodes. This attribute of type array. This allows to mount multiple particular source nodes into one target node.
Merge source node attributes into target nodes	This option allows to add the source attributes directly into the target node. Otherwise the source nodes will be mounted as separate objects into the target.
Target attribute to mount the source nodes	Setup here the attribute within the target node under which the matching source nodes have to be mounted. This option is available if you uncheck the option above.
Assign cloned nodes	This option allows to clone the source node before merging into the target. Especcially if you want to change the source nodes right after merging them but without affecting the merged nodes, your can use this option. Per default the source nodes will be added directly to the target which save memory.
Die if a source node could not be assigned	If a source node (identified by its key) cannot be assigned into a target node (also identified by a key or multiple keys) the component can fail with a meaningful error message.

Return values

Return value	Content
ERROR_MESSAGE	If something went wrong - the error message goes here.
CURRENT_NODE	The target node as JsonNode object.
COUNT_SOURCE_NODES	The number of source nodes.
COUNT_TARGET_NODES	The number of target nodes.
COUNT_MERGED_NODES	How many source nodes are merged
COUNT_MISSED_MERGES	How many source nodes could not be merged