# KeyValue Feature

**Remark: This is currently in redesign.**

The KeyValue feature allows to read, process and write data as key-value pairs, which don't have an fixed schema like tuples.

To use the feature you have to install it and use the "KeyValueObject" data handler. The applicable operators will be automatically chosen. Until now selection and projection are supported. Also there are operators to transform key-value objects to tuples and the other way round. The feature also includes wrapper for handling of JSON data.



# Operators

The following new or modificated operators are provided in keyvalue feature.

## KeyValueToTuple (Deprecated, use ToTuple operator instead.)

Transforms key-value object to tuples. Use the SCHEMA attribute to define:

- the path of the input that should be part of the tuple. This could be a dot based path: e.g. root.id.value. This is only possible if the result is not a KeyValueObject! You can also use JSONPATH.
- optional the name that should be use in the schema of the tuple (remark special charactes from Jsonpath like "$", "*", etc. are replaced automatically
- the datatype of the element that is retrieved from the key value object. This could also be a key value object or List

Examples for schema could be:

```
        ['$','root','KeyValueObject'],
        ['monday', 'List(Integer)'],
        ['$..monday.length()','MondaySize','mondayCount','Integer'],
        ['$.uuid', 'String'],
        ['$.monday[0]',MondayFirst, 'Integer'],
```

WIth the type field you can set the type of the tuple.

```
#DEFINE SCHEMA [['timestamp.unixtimestamp', 'List(Integer)'],['timestamp.iso', 'String'],['track.name',
'String']]
tuple = KEYVALUETOTUPLE({schema=${SCHEMA}, TYPE = 'type'}, receiverJSON)
```

## TupleToKeyValue

Transforms tuples to key-value objects based on the tuples schema.

```
tupleToKeyValue = TUPLETOKEYVALUE(receiverTuple)
```

## Select

The select operator can be used for key value objects in the same way as for tuples.

```
selectJSON = SELECT({predicate='track.artist.name = "Evanescence"'}, receiverJSON)
selectJSONList = SELECT({predicate='timestamp.unixtimestamp[0] = 1162304033'}, receiverJSON)
```

## Project

For the key value projection the "paths" attribute has to be used. The "attributes" attribute only works for relational tuples.

```
 projectJSON = PROJECT({paths = [['timestamp.unixtimestamp', 'List(Integer)'],['timestamp.iso', 'String']]},
receiverJSON)
```

## Rename

To rename the attributes of key value objects the "pairs" parameter has to be set to true and the old and new attributenames have to be given in the "aliases" parameter.

```
renamed = RENAME({aliases = ['Zeit', 'ZeitNew', 'Band', 'BandNew'], pairs = 'true'}, mapped)
```

## Map

For the key value MAP operator the "kvexpressions" attribute has to be used. The "expressions" attribute only works for relational tuples.

```
output = MAP({
             kvexpressions = [
                                                              ['timestamp.unixtimestamp', 'Zeit'],
                                                              ['toLong(timestamp.unixtimestamp + 1)',
'Zeit2'],
                                ['track.artist.name', 'Band'],
                                ['track.name', 'Lied']
                                                                  ]
            }, input)
```

# MEP-Functions

There are some MEP-Functions that can be used in the MAP-Operator:

- KeyValueObject: asKeyValue(Object kv): Cast an object to a KeyValueObject (could be necessary if Odysseus can not determine the type of an object). KV must be a KeyValueObject, else there will be a ClassCastException
- KeyValueObject: toKeyValue(String): Creates a new KeyValueObject from a String JSON Expression
- Object: getElement(KeyValueObject kv, String path): Returns the one element identified by the path expression.
- Tuple: getElements(KeyValueObject kv, String path): Returns a tuple of elements identified by the path expression.
- Object: path(KeyValueObject kv, String path): Returns the one object that is identified by the JSONPATH epxression. Slower than getElement(s) but more expressive.

## Wrapper

The feature adds protocol handler for the common key-value data formats JSON and BSON (binary JSON).

## JSON

### Read JSON file

```
 json = ACCESS({
    source='json',
    wrapper='GenericPull',
    transport='File',
    protocol='JSON',
    dataHandler='KeyValueObject',
    options=[['filename','${WORKSPACEPROJECT}\scrobbles-2006-10_linebreak.json']]
/// schema=[['timestamp','STARTTIMESTAMP'], ['timestamp2','ENDTIMESTAMP']] /// only used for definition of
START- or ENDTIMESTAMP
})
```

## Write JSON file

```
SENDERjson = SENDER({
    transport='File',
    wrapper='GenericPush',
    protocol='JSON',
    dataHandler='KeyValueObject',
    SINK="SENDERjson",
    options=[['filename','${WORKSPACEPROJECT}\output\test2'],
                ['json.write.metadata','true'],
        ['json.write.starttimestamp','metadata.starttimestamp'],
        ['json.write.endtimestamp','metadata.endtimestamp'] ]
}, json)
```

## Transfer data with JSON

JSON protocol handler can not only be used to read and write files, but also to transfer data from one odysseus instance to another. The example shows the use of TCP as transport protocol, but RabbitMQ can be used the same way.

```
 SENDERjson = SENDER({
    transport='TCPServer',
    wrapper='GenericPush',
    protocol='JSON',
    dataHandler='KeyValueObject',
    SINK="SENDERjson",
    options=[
        ['port', '8080'],
        ['HOST','localhost'],
    ]}, json)
receiverJSON = RECEIVE({
    transport='TCPClient',
    source= 'ReceiverJSON',
    protocol='JSON',
    dataHandler='KeyValueObject',
    options=[['port', '8080']]
})
```