How to Compare Two Seemingly Equal Objects With Keys in Different Order

The Problem

In most cases, when creating objects in Java, JSON, etc., as developers we don’t care about the order of an object’s keys. For example, we consider { “b”: 2, “a”: 1} to be equivalent to { “a”: 1, “b”: 2}

There are times when we must care, because the code cares. Two such examples include:

- Writing dataweave: Dataweave returns **false** if you write {a: 1, b: 2} == {b: 2, a: 1}
- Writing Munit Tests: Using AssertThat with say payload={a: 1, b: 2} and the “is” clause set to MunitTools::equalTo({ “a”: 1, “b”: 2}) fails, causing the test to fail.

Possible Solutions

There are two possible solutions:

1. Create the compared-to object with keys in the same order as the code generates
2. Reorder the keys in the object produced by your code and the object to which you want to compare before doing the compare.

The solution is to ensure that both objects have their keys in the same order. The question is, how do we do that? The rest of this document discusses two techniques.

Make the Objects Key-order Match

If writing DataWeave, just ensure the two objects have their keys in the same order. If using MunitTools::euqlTo() (or similar), ensure that the object passed into this method has keys in the same order as the object your comparing against.

Pros

It’s relatively easy for short objects.

Cons

There are several drawbacks to this approach:

1. If the object has lots of keys, ensuring the keys are in the same order can be tedious. The difficulty tends to increase non-linearly as the number of keys increases.
2. If a code changes to have a new key, you not only have to add the key in the compared-to object, but also put it in the right place.
3. You cannot (easily) use this technique if the code can generate the keys in different order based on the flow of the code.

Sort the Keys in Both Objects Before Comparing Said Objects

Create a temporary version of each object with its keys sorted in alphabetic order and compare these temporary objects. In DataWeave, that means creating a function to return an input object with the keys sorted, which can then be used to compare with the result of the same function on the second object. That is, instead of writing:

```
{ a: 1, b: 2} == { b: 2, a: 1} // returns false
```

Write:

```
Reorder({ a: 1, b: 2}) == Reorder({ b: 2, a: 1}) // returns true
```

Pros
This works EVERY time.

Cons
It adds extra time to the unit test.

The DataWeave Code
The question then becomes, “How to write the code that will reorder the keys of an object”? Here is one simple solution:

```
fun sortObjectByKey(obj: Object)  = {((obj pluck (value, key) -> (key))
orderBy $
map { ($): obj[$]})}
```

Test this function with `({ b: 2, a: 1}` yields `({ a: 1, b: 2}.`

The only issue with this function is that it doesn’t handle nested objects in values of an object or elements in an array.

Here is a recursive version of the function that does handle nested objects in values. Of an object or elements of an array:

```
fun sortObjectByKeyNested(obj: Object)  = {((obj pluck (value, key) -> (key))
orderBy $
map { ($): obj[$]
    match {
        case o is Object -> sortObjectByKeyNested(o)
        case a is Array ->
            a map (item) -> item match {
                case o is Object -> sortObjectByKeyNested(o)
                else -> $
            }
        else -> $
    }})}
```

Test this function with this object:
```
{
   "attr3": "DECIMAL(12,4)",
 "attr1": "BIGINT",
  "attr2": "VARCHAR(1024)",
  "foo": {
      "b": 1,
      "a": 2
  },
  "baz": [ "a", "b",
        { "c": 1, "abc": 2}]
}
```

And it will generate this object:

```
{
  attr1: "BIGINT",
  attr2: "VARCHAR(1024)",
  attr3: "DECIMAL(12,4)",
  baz: [
    "a",
    "b",
    {
      abc: 2,
      c: 1
    }
  ],
  foo: {
    a: 2,
    b: 1
  }
}
```

Notice that all of the keys in the outer object are sorted, as are the keys in the value associated with "foo" and the third element of the array "baz."

Once you have these functions in a module, you can now write:

```
sortObjectByKeyNested(obj1) == sortObjectByKeyNested(obj2)
```

and it will return true provided obj1 and obj2's have the same set of keys and values, regardless of the keys' order.

Summary

When you want to compare two objects in the MuleSoft ecosystem, and they may not have their keys in the same order, you can use functions such as those shown in this document to compare them for logical equivalence.

The Computer Classroom, Inc. hopes you find these patterns useful.

If you're interested in having Mule Developers at The Computer Classroom, Inc. help you with your Mule application development, please email sales@compclass.com. We'd be happy to help!