# Guidelines on FpML to JSON Conversion

*FpML Architecture Working Group – April 2019*

## Contents

# Introduction

Recently, we have seen an increased use of JavaScript Object Notation (JSON) encoding based on FpML models. The purpose of this document is to provide mapping guidelines for the conversion of FpML messages to JSON. The paper considers JSON versus JSON schema, the practicality of a round trip conversion, and provides an initial set of guidelines. The FpML Architecture Working Group (AWG) is developing a JSON reference implementation based on these guidelines.

The AWG encourages continued feedback on the paper and the principles laid out. Comments can be sent to: archwgchair@fpml.org

## What Is JSON?

JavaScript Object Notation or JSON is an open-standard file format, commonly used for asynchronous browser–server communication. JSON is a language-independent data format. It was originally derived from JavaScript, and is syntactically equivalent to a subset of JavaScript, but is now supported by many programming languages. It provides an acceptable performance for single human to server interaction and it reduces development efforts as it is simpler and consumes fewer resources to parse than XML.

JSON is self-describing. Elements have readable names in a JSON document. An external schema or definitions in a data dictionary are not required to interpret JSON encoded data.

JSON format can be parsed and encoded in virtually every programming language. The list of supported languages can be found at www.json.org. JSON implementations operate within JavaScript. JSON interacts well with loosely typed languages such as Python, Ruby and JavaScript, which have data structures similar to JSON objects.

## JSON Schema

There is not yet an international standard defined for JSON Schema. JSON lacks a truly standard schema language and support for a diverse set of data types. Even if a third party JSON schema validator is used to check message content, additional coding is needed in every implementation to provide the same robustness as is provided by XML Schema validation.

# Types of Conversion

## Conversion at Schema Level

Although there is not yet an agreed JSON schema standard, one potential alternative to consider is the conversion between XML Schema and JSON Schema. However this alternative is quite complex for a number of reasons:

- The constraints that are defined by the JSON Schema are not as complex as those defined in an XML Schema. As a result, a one-to-one conversion is not possible and the conversion process needs to include a simplification process. For example, the multiplicity of sequences or choices in XML Schema needs to be translated into the repetition of the actual contained elements in JSON. This is not trivial in all cases.

- As mentioned in the previous section, there is no standard JSON Schema so the target for the conversion is not standard, it may change depending on the JSON Schema specification.
- Bidirectional transformation is a challenge as the JSON schema is much simpler than the XML schema.

### Conversion at Instance Document Level

With the previous comments in mind, the goal of this document is to encourage consistency in the mapping from FpML to JSON at the instance document level, as there does not currently exist a standard for XML to JSON conversion. Different converters available in the public domain present different outcomes for the same XML input file. It is therefore needed to establish the transformation convention for FpML to JSON, based on the methods used by the different JSON converters. The transformation conventions need to cater for the special features used by FpML.

The main advantage of mapping at the instance level rather than schema conversion is simplicity. In the Implementation section we describe a number of XML to JSON conversion rules that take into account FpML-specific features.

### Converting from JSON back to the FpML

The conversion from JSON back to the FpML is not seen as a primary requirement by the AWG at this point in time, for the reasons detailed below:

- The main use case is the conversion from FpML to JSON since JSON will be used as end point format for web services calls.
- The representation does not allow easy round tripping just based on the JSON as described below. Schema information should be processed in order to avoid losing information during the reverse conversion.

Even though the conversion from JSON to the XML is not a current goal, it is possible to implement such conversion. In order to ”reverse convert” at the level of the instance document, the rules from the guidelines described below in the Implementation section must be followed but in the opposite direction.

Problems will arise, however, when an XML schema has been converted into JSON schema and the JSON schema rules are subsequently followed to do a reverse conversion from JSON to XML. The reason for the difficulties stems from the fact that XML Schema is richer than JSON Schema, e.g. when defining constraints such as the multiplicity of the elements, patterns, and sequences and choices.

## Implementation

In this section we describe how the different XML structures FpML is using, are transformed to JSON. The conventions will be applied to components such as attributes and elements parts as names and values, element repetitions, optional/multiple/mandatory choices/sequences and patterns. Note: Even if the data order cannot be maintained in the JSON message, the conversion tries to maintain the order of the FpML to facilitate a reverse conversion.

## Elements

The element names will be kept when the message is converted to JSON. They form part of the name of the JSON object. Depending on the configuration these elements have inside an XML message, the conversion might differ as follows:

### Element with multiple element content children

In this instance the elements' children will be converted into JSON as an array constituted of the pair of the element name. At this point it is important to distinguish between the possible content of the element children for the construction of the array.

#### *Repeating Elements*

If the same name element is repeated under the same structure in the same position multiple times, this will be converted to a JSON array using square brackets. The name of the pair will be the name of the element and the value of the pair will be the array in square brackets, separating every object inside by a comma. An example is the <partyTradeIdentifier> element that may be repeated twice under <tradeHeader> element. It would be translated into a <partyTradeIdentifier> array with two objects:

```
  <tradeHeader>
    <partyTradeIdentifier>
      <partyReference href="party1" />
      <tradeId tradeIdScheme="http://www.partyA.com/swaps/trade-id">TW9235</tradeId>
    </partyTradeIdentifier>
    <partyTradeIdentifier>
      <partyReference href="party2" />
      <tradeId tradeIdScheme="http://www.barclays.com/swaps/trade-id">SW2000</tradeId>
    </partyTradeIdentifier>
...
```

```
"tradeHeader": {
                "partyTradeIdentifier": [
                  {
                    "partyReference": {"href": "party1"},
                    "tradeId": {
                      "tradeIdScheme": "http://www.partyA.com/swaps/trade-id",
                      "value": "TW9235"
                    }
                  },
                  {
                    "partyReference": {"href": "party2"},
                    "tradeId": {
                      "tradeIdScheme": "http://www.barclays.com/swaps/trade-id",
                      "value": "SW2000"
                    }
                  }
                ],
```

### *Non-Repeating Elements*

If the elements are non-repeating, an enclosing with brackets is used for the children, maintaining the same-level children inside these brackets, and the children of the children in a new object between brackets. As an example, see the below representation of <effectiveDate>, with children elements <unadjustedDate> and <dateAdjustments> at the same level. <dateAdjustments> further contains <businessDayConvention>:

```
<effectiveDate>
 <unadjustedDate>1994-12-23</unadjustedDate>
 <dateAdjustments>
  <businessDayConvention>FOLLOWING</businessDayConvention>
 </dateAdjustments>
</effectiveDate>
```

```
"effectiveDate": {
                "unadjustedDate": "1994-12-23",
                "dateAdjustments": {"businessDayConvention": "FOLLOWING"}
            },
```

## Attributes

When converting attributes found in the XML elements, the attribute name has to be kept when the message is converted to JSON. There are different configurations depending on the number of attributes and on whether the element contains other elements. The following principles apply for the conversion.

### Element with multiple attributes and element content

If an element contains multiple attributes plus other elements inside, the attributes are inside the object of the element in JSON, forming part of the sublist of the "element" name, at the same level as its child element. For example, the FpML root element would be converted in the following way:

```
<dataDocument xmlns="http://www.fpml.org/FpML-5/confirmation"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" fpmlVersion="5-10"
xsi:schemaLocation="http://www.fpml.org/FpML-5/confirmation ../../fpml-main-5-10.xsd
http://www.w3.org/2000/09/xmldsig# ../../xmldsig-core-schema.xsd">
  <trade>
...
```

```
{"dataDocument": {
    "xmlns": "http://www.fpml.org/FpML-5/confirmation",
    "xmlns:xsi": "http://www.w3.org/2001/XMLSchema-instance",
    "fpmlVersion": "5-10",
    "xsi:schemaLocation": "http://www.fpml.org/FpML-5/confirmation ../../fpml-main-5-10.xsd
http://www.w3.org/2000/09/xmldsig# ../../xmldsig-core-schema.xsd",
    "trade": {
```

...

### Element with only a single attribute

If an element contains only one attribute, the element "name" contains the object formed by the 'name:value' pair of the attribute. For example, FpML party references illustrated below would be converted in the following way:

```
<payerPartyReference href="party1" />
```

```
"partyReference": {"href": "party1"},
```

### Element with an attribute and a value

If an element contains an attribute and a value, the element "name" contains a new object formed by the 'name:value' pair of the attribute and another pair which has "value" as the object name and the value of the element as its pair. For example, FpML identification elements with a scheme attribute, such as the trade ID that follows the below syntax would be converted to:

```
<tradeId tradeIdScheme="http://www.partyA.com/swaps/trade-id">TW9235</tradeId>
```

```
"tradeId": {
                "tradeIdScheme": "http://www.partyA.com/swaps/trade-id",
                "value": "TW9235"
            }
```

### *Exception: Optional Coding Schemes Attributes*

In the case an element has in its definition an optional coding scheme attribute, the conversion must show the value of the element, not as a pair value of the element tag, but as a new object with a pair with “value” as tag name and the value of the element as its pair. One example of this case would be the element <businessCenter> of complexType “BusinessCenter” which has an optional attribute “businessCenterScheme”, that would be converted like:

```
<businessCenter businessCenterScheme="http://www.fpml.org/coding-scheme/business-center">USNY</businessCenter>
```

```
{
  "businessCenter": {
    "businessCenterScheme": "http://www.fpml.org/coding-scheme/business-center",
    "value": "USNY"
  }
}
```

```
<businessCenter>USNY</businessCenter>
```

```
{
  "businessCenter": {
```

```
      "value": "USNY"
    }
}
```

### Element with an attribute and element content

If an element contains an attribute plus elements inside, the attribute would be inside the object of the element in JSON, forming part of the sublist of the "element" name, at the same level as its element children. For example, the <calculationPeriodDates> element would be converted in the following way:

```
<calculationPeriodDates id="floatingCalcPeriodDates">
        <effectiveDate>
```

```
            "calculationPeriodDates": {
               "id": "floatingCalcPeriodDates",
               "effectiveDate": {
```

### Default attributes

The W3C XML Schema allows the possibility of defining a default value for each attribute. FpML uses this feature within the definition of the coding scheme types. When FpML defines a standard URI for a coding scheme, then its scheme attribute has a default value that may be overridden in the instance document.

However, FpML producers may decide not to populate optional scheme attributes in the instance documents since this information may be defined in a specification outside the xml. This decision has the objective, most of the times, to simplify and reduce the size of the instance documents.

Regarding the conversion to JSON, the AWG decided that default attributes should not appear in the resulting JSON if they don't appear in the original XML. The JSON should not be enriched in the conversion process and it should contain only the attributes that appear in the original XML.

## FpML Implications

There are certain characteristics of arrays that need to be taken into account when converting from FpML. Specifically, in cases where elements can either contain arrays of objects under them or zero or one-element arrays. An example is the <swapStream> within the <swap> structure where <swapStream> can be repeated one or more times. If <swapStream> appears more than once, following previous arrays guidelines, then the conversion will be fine, with the object pair of name <swapStream> and value an array of the different <swapStream> children structures:

```
  <swap>
    <swapStream>
      <payerPartyReference href="party1" />
...
```

```
    </swapStream>
    <swapStream id="swap_fixed">
      <payerPartyReference href="party2" />
...

    </swapStream>
  </swap>
```

```
"swap": {"swapStream": [
        {
          "payerPartyReference": {"href": "party1"},
...
        },
        {
          "id": "swap_fixed",
          "payerPartyReference": {"href": "party2"},

...
        }
    ]}
```

But, as commented before, if it is found that the <swap> structure in the XML only contains one <swapStream> element, then the array structure should be maintained, even with only one element in the array.

```
  <swap>
    <swapStream>
      <payerPartyReference href="party1" />
...

    </swapStream>
  </swap>
```

```
"swap": {"swapStream": [
        {
          "payerPartyReference": {"href": "party1"},
...
        }
    ]}
```

Existing conversion tools take the only element in the array as a sub-object of the main object.

It is not possible to know from a single XML instance document how many times an element can appear within another element, hence the knowledge from a XSD schema is required. If an element can appear more than once according to the XSD schema, this element will be converted as an array, even when it only appears once in the instance document.

The structure of the resulting JSON message will be the same irrespective of whether an element appears once or more. Example showed, <swapStream> in the context of <swap> would always appear as an array object in the JSON message, irrespective of whether there is more than one object <swapStream>.

## Data types

In JSON, the types used for data are numbers, Booleans, and strings. The types are easily identifiable in JSON. Differentiation can also be made between empty objects, empty strings and nulls. This is contrary to XML where data types are not distinguished by just looking at the instance document and the XML Schema Definitions (XSDs) are needed to determine them. As a result, the XML Schema (XSDs) is needed as input for a correct conversion process from XML data to JSON language.

### Numbers

Numbers are typed without quotes ("") as a pair value in JSON. Because in XML numbers are not distinguished from strings, we need to be aware of the type definition in XML Schema before converting to JSON. One example of number field in FpML is the field <periodMultiplier>, which will be always converted to JSON number data type:

<periodMultiplier>6</periodMultiplier>

"periodMultiplier": 6

Current implementations of JSON parsing libraries limit the precision of the JSON numeric type. The JSON numeric is more restrictive than the XML decimal type. That means that for very big numbers or numbers with lots of decimal digits, the conversion from XML to JSON could lose precision. However though the AWG is aware of this issue, decided that it is important to keep as much as possible the type of the original data source to avoid losing type information in the conversion process. As consequence, FpML decimal elements should be converted to JSON numbers. The AWG welcomes feedback on this approach from the FpML user community in order to evaluate its impact.

### Strings

Most of the data types used in XML will be typed in JSON as a string (date, token, ID, string). These are typed in JSON within quotes (""). Awareness of the XML data type, defined in the schema, is also needed for a correct conversion of strings.

An example of string field is 'Period' which contains token XML field:

<period>M</period>

"period": "M"

### Booleans

Booleans are typed as 'true' or 'false' without quotes in JSON. Awareness is needed when XML is converted to avoid putting them between quotes, which would be a string. Also sometimes in XML

the Boolean type can be filled up with numbers, so awareness is needed for a correct conversion to JSON.

<initialExchange>true</initialExchange>

"initialExchange": true

### Empty Entities in XML

In JSON it is possible to have empty strings, empty objects and nulls. However they will look exactly the same in XML, with nothing between the tags. It is important to be aware of this fact, since in FpML there are a number of empty elements in the schema. In most cases, in FpML, if the element does not exist, it does not appear in the message. An example of an empty element in XML and its possible conversions to JSON instances, depending on what is the represented content:

XML:
<EmptyObjectOrEmptyStringOrNull></EmptyObjectOrEmptyStringOrNull>

JSON:
"EmptyObject": {},
"EmptyString": "",
"Null": null

The conversion from an FpML empty element WITH attributes will not generate empty quotes for the content of the element. For example:

XML:
<partyReference href=”party1”/>

JSON:
"partyReference":{"href":"party1"},

On the other hand, there are different use cases for the conversion from an empty FpML element WITHOUT attributes to JSON:

- An element which can hold a string value but which is currently empty should map to an empty string ""
  - XML:
    - <partyName/>
  - JSON:
    - "partyName": {
      "value": ""
      }

- An element which can hold a number or boolean value but which is currently empty should map to a JSON null
    - XML:
        - <amount/>
    - JSON:
        - "amount": null
- An element of a type that cannot hold any content (e.g. derived from FpML's Empty) should map to an empty object {}
    - XML:
        - <nonReliance/>
    - JSON:
        - "nonReliance": {},

### xsd:any

Even though the use of xsd:any in FpML is limited to the definition of the math component within the interest rate and equity product streams, it constitutes an issue in terms of JSON conversion since there is no schema definition to support its content. For this particular situation, the AWG decided that the FpML to JSON conversion should implement a "best of possible" conversion approach. This means a generic XML to JSON conversion without taking any schema information.

## Conclusion

With the implementation issues described above, the main conclusion is that to make a proper conversion from FpML to JSON, the FpML Schema has to be considered. The FpML message does not contain sufficient information to define the proper types in JSON in all cases. In addition, the examples have shown that special attention is needed in certain cases to avoid a selection of incorrect JSON data types.

To transpose the maximum amount of information to the JSON message, the data types must be converted as advised in the previous guidelines. As we have seen with array elements, it is not in all cases possible to convert data types correctly from the XML message without considering XML Schema information.