*Balisage:* ***The Markup Conference***

# JXON: an Architecture for Schema and Annotation Driven JSON/XML Bidirectional Transformations

## David Lee

Senior Principal Software Engineer
Epocrates, Inc.
<dlee@epocrates.com>

### Abstract

JSON and XML are seen by some as competing markup formats for content and data. JSON has become predominant in the mobile and browser domains while XML dominates the Server, Enterprise and Document domains. Where these domains meet and need to exchange information there is pressure for one domain to impose on the other their markup format. JXON is an architecture that addresses this problem by providing for high quality bidirectional transformations between XML and JSON. Previous approaches provide for only a single mapping intended to cover all cases, but generally cover few cases well. JXON uses Schema and annotations to allow highly customizable transformations that can be tuned for individual schemas, elements, attributes and types yet still be easily configured.

### Table of Contents

## Introduction - JXON

JXON is both a design architecture and a reference implementation of a tool for JSON[1]/XML [2] transformations. Unlike other XML/JSON transformation architectures and tools, JXON attempts to fulfill the needs of both XML and JSON authors and developers equally.

JXON provides the ability to easily describe both unidirectional and bidirectional XML/JSON transformations that produce markup which authors of that particular format would like to use.

## Diving In

To get a quick idea of the problem that JXON is attempting to solve consider the following pair of documents representing the same information in *XML* and JSON.

```
<BOOKS>
  <BOOK id="1">
    <TITLE>My Favorite Book</TITLE>
    <PRICE>1.23</PRICE>
  </BOOK>
  <BOOK id="1a">
    <TITLE>XML for Dummies</TITLE>
    <PRICE>5.25</PRICE>
  </BOOK>
  <BOOK id="3">
    <TITLE>JSON for Dummies</TITLE>
    <PRICE>200.95</PRICE>
  </BOOK>
</BOOKS>
```

```
{ "BOOKS" : [
  { "id" : "1"  , "title": "My Favorite Book" , "price" : 1.23 },
  { "id" : "1a" , "title": "XML for Dummies"  , "price" : 5.25},
  { "id" : "3"  , "title": "JSON for Dummies" , "price" : 200.95 }
]}
```

The XML and JSON are both in styles which a native author of that markup type may wish to use. It appears that there is an obvious and simple mapping between the formats, and in fact there should be a simple reversible lossless transformation so that given a document in either XML or JSON it could be transformed to the other format and back and end up identical. This seems like such a simple problem not even worth discussing. But in reality there are no existing tools which can actually do this generically without hand coding a transformation, both directions, in one or more programming languages. The resulting code in say *XSLT*[3] or JAVA[4] could be very large and tedious to write. Furthermore such code may not easily be reused for a new document schema.

There are many subtleties to this problem. Just a few to think about that highlight the issues:

- How would a translation know to consistently use a string value for "id" and "title" but a numeric value for "price"?
- Where does the "BOOK" element come from when translating to XML?
- How does the JSON to XML transformation code know to make "id" into an attribute in XML but not price or title?
- How does the XML to JSON translation know to construct an array in JSON?
- Where does the name conversion rule for "TITLE" vs "title" and "PRICE" vs "price" occur?

As a comparison the default XML to JSON transformation from json.org produces the following JSON

```
{
 "childNodes": [
  {
   "childNodes": [
    {
     "childNodes": ["My Favorite Book"],
     "tagName": "TITLE"
    },
    {
     "childNodes": [1.23],
     "tagName": "PRICE"
    }
   ],
   "id": 1,
   "tagName": "BOOK"
  },
  {
   "childNodes": [
    {
     "childNodes": ["XML for Dummies"],
     "tagName": "TITLE"
    },
    {
     "childNodes": [5.25],
     "tagName": "PRICE"
    }
   ],
   "id": "1a",
   "tagName": "BOOK"
  },
  {
   "childNodes": [
    {
     "childNodes": ["JSON for Dummies"],
     "tagName": "TITLE"
    },
    {
     "childNodes": [200.95],
     "tagName": "PRICE"
    }
   ],
   "id": 3,
   "tagName": "BOOK"
  }
 ],
 "tagName": "BOOKS"
}
```

Note some non-ideal artifacts of this transformation (which are common among existing tools)

- Inconsistent typing of the "id" value
- Very verbose and complex JSON representation
- JSON arrays where simple values should be used
- Pairs of 'name/value' members where native JSON member names should be used.
- Unnecessary distinctions between how attributes and child elements are serialized.

Other existing tools attempt to solve this problem various ways, optimizing for either XML or JSON and often making simplifying assumptions which are valid for only some kinds of documents. The results can be good for some special cases but overall there is no existing solution that provides good transformations that work equally well in both directions for a wide variety of documents and produce markup close to what a human author would want to create or use.

### *JXON attempts to solve this problem*

As an example, the following is an RNG[5] Compact Notation schema which fully describes both the schema and rules for the bidirectional lossless transformation for the example above. This is all that is needed for JXON to produce both XML to JSON and JSON to XML transformations.

```
default namespace = ""
#<jxon:pattern name="simple"/>
grammar {
start =  Books

#<jxon:children wrap="array"/>
Books = element BOOKS { Book+ }

#<jxon:json_name omit="true"/>
Book = element BOOK {
      attribute id { xsd:NMTOKEN },
      Title,
      Price
}
#<jxon:json_name name="title"/>
Title =    element TITLE { text }

#<jxon:json_name name="price"/>
Price = element PRICE { xsd:decimal }
}
```

And the equivalent schema in XSD using Annotations instead of comments.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"  xmlns:jxon="http://www.xmlsh.org/jxon">
  <xs:annotation> <xs:appinfo>
      <jxon:pattern name="simple"/>
    </xs:appinfo>
   </xs:annotation>

  <xs:element name="BOOKS">
    <xs:annotation > <xs:appinfo>
        <jxon:children wrap="array"/>
      </xs:appinfo>
     </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="BOOK"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="BOOK">
    <xs:annotation> <xs:appinfo>
        <jxon:json_name omit="true"/>
      </xs:appinfo></xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="TITLE"/>
        <xs:element ref="PRICE"/>
      </xs:sequence>
      <xs:attribute name="id" use="required" type="xs:NMTOKEN"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="TITLE" type="xs:string">
    <xs:annotation> <xs:appinfo>
        <jxon:json_name name="title"/>
      </xs:appinfo></xs:annotation>
  </xs:element>
  <xs:element name="PRICE" type="xs:decimal">
    <xs:annotation> <xs:appinfo>
        <jxon:json_name name="price"/>
      </xs:appinfo>
   </xs:annotation>
  </xs:element>
</xs:schema>
```

## JSON/XML what's the problem?

JSON has become a very popular and useful markup language among some developers and use cases, particularly in browser and mobile applications. These developers require JSON to be delivered from and to services which often use XML as their back end data model. This support requires clean and robust transformations of XML to and from JSON. Because of subtle but significant differences in both the data model and the serialization format this transformation is either done by hand coding, or by sacrificing the clarity of the markup on either the JSON or the XML.

JSON and XML are conceptually similar. They are both text based markup languages designed to represent data in a format which is human readable, interchangeable across environments and parseable by most programming languages. On the surface they seem quite similar, with the major apparent difference being a slightly terser notation for JSON for simple things (lack of end tags). There are many mappings between JSON and XML, and programs that implement those mappings. This, in theory, allows one to choose the markup and tool-set of your choice and transform it to the other at will. Servers that store XML data can produce JSON data for clients that need it, and visa-versa.

Reality, unfortunately, is not so simple. JSON and XML are fundamentally (and often subtly) incompatible in their abstract data models. They are both sufficiently rich that they can describe each other's data model, so creating a mapping from arbitrary XML to JSON and visa-versa is *possible* but the result can be very complex. This has led people to design mappings which accommodate only a subset of XML or tailored to particular schemas.

### *Anonymous values*

JSON values are anonymous. They only acquire names by being referenced in an object (map). For example, the following is a valid JSON document.

```
        "Hello World"
```

The analogous XML document would need a root element with child text content or perhaps an attribute.

```
        <root>Hello World</root>
or
        <root value="Hello World"/>
```

In the degenerate case this means entire JSON documents can be composed of single (or arrays of) anonymous values which have no direct mapping to the XML data model.

When mapping JSON to XML this additional markup cannot generally be derived from the JSON data itself, but rather needs external rules to decide on the approach. When mapping XML to JSON the presence of an element with text children or attribute value may not necessarily imply creation of an anonymous JSON value, but rather the element and attribute names are commonly used to create named members of JSON objects such as

```
        { "root" : "Hello World" }
                                  or
        {"root" : { "value" : "Hello World" }}
```

### *Arrays*

JSON has a native representation of arrays which does not exist as native types in XML. XSD Schema has support for values which can be interpreted as tokenized lists, but there is no direct markup for an array of objects. Arrays can be represented in XML in several ways ; repeated elements, repeated elements grouped by a common parent, usefully named attributes, usefully named elements and tokenized strings.

For example the following JSON array

```
        [ 1 , "String" ]
```

Could be represented in XML as any of the following (and many other ways)

```
        <array>1 String</array>

        <array><entry>1</entry><entry>String</entry><array>

        <array entry1="1" entry2="String"/>

        <entry>1</entry><entry>String</entry>
```

Mapping XML to JSON arrays is possible but since there is no single mapping, and since XML semantics by itself doesnt indicate array representation, one must know via external information to treat specific XML markup as a JSON array. Given any of the above examples, its not obvious that they should be mapped to a JSON array as opposed to some other JSON structure (such as a simple value or object).

### *Implicit Types (e.g. "Duck Typing")*

JSON values make use of implicit types which can be inferred by the serialization format. For example `1` is a Numeric type, while `"1"` is String type. This is common in programming languages, but not as common in markup languages. In XML, without a schema, the token `1` is xs:anyType and it's up the application to infer a type (possibly using a schema, or up to application logic).

This leads to problems mapping XML to JSON. When a plain character string value from XML is mapped, what JSON type should be used? It's tempting to use the value instance to deduce the type, but this could be very dangerous. An application may be expecting string types and if one instance happens to translate `1` to a numeric type because it looks like one, and another translates "one" to a string this could break the application.

Translating atomic values from JSON to XML is not as problematic as generally typing can simply be ignored because the serialization format for string and numbers is the same in XML, although other atomic types like boolean, dates etc can be more problematic.

### *Identifiers*

JSON allows any string as a identifier (Object member name). XML is much more restrictive for identifiers. Translation of XML identifiers (element and attribute names) to JSON member names causes few problems, but the reverse can lead to invalid XML. For example "a value" is a valid member name in JSON but not a valid attribute or element name in XML

### *Namespaces*

XML supports namespaces, while JSON does not. Mapping QNames in XML with namespaces to member names in JSON can lead to ambiguous, complex or duplicate names. Translating member names in JSON to XML QName may lose namespaces.

### *Processing Instructions*

XML supports Processing Instructions, which are absent from the JSON data model.

### *Attributes*

JSON has no concept or representation for Attributes (as distinct from other values).

When mapping XML to JSON, attributes are often translated to named object members (along with child elements). When mapping JSON to XML it requires external knowledge to determine when to map members to attributes.

### *Character Set*

JSON supports a broader set of text characters (Unicode code points) then XML. For example JSON supports the NUL character ('\0') whereas NUL is invalid in XML even if encoded as an entity (&#0;). This implies that JSON character data cannot be mapped to XML character data losslessly without application specific encodings.

### *Comments*

XML directly supports comments while JSON does not. Comments were originally allowed in JSON but Douglas Crockford removed them [6]

Comments could be encoded in JSON using special object member names (such as "_comment") requiring application support to understand these were comments and not data.

A common case is to simply strip comments from XML when mapping to JSON, and to not insert comments when mapping JSON to XML.

### *Document Node*

JSON does not have a Document Node, where XML does. This is generally not problem in practice as the XML Document Node is usually implicitly created in the model, and has no textual representation.

### *Serialization*

The text Serialization format for JSON and XML differ significantly. Often the serialization format is the main focus of JSON vs. XML mappings. I assert that the serialization format is completely irrelevant for purposes of mapping and transformations; the formats are well defined and there exist a large number of implementations which can parse and serialize XML and JSON. If one focuses instead on the abstract data model, the issue of serialization (both parsing and generation) becomes a trivial implementation issue.

### *Encodings*

Related to the Serialization format, encodings for JSON and XML can differ. JSON is defined to support only the UTF-8 encoding while XML can use many different encodings. Like serialization, I assert that encoding is irrelevant. If one works with the JSON and XML abstract data Models instead of their serialization and encoding formats these issues disappear.

## Existing Designs and Implementations

There are many existing models, designs, and implementations for translating XML to JSON, JSON to XML or both. The advantages and limitations were considered in the design of JXON. These include (but not limited to) the following

- The JSON to XML converter from json.org JSONORG
- JSONx jsonx
- Badgerfish badgerfish
- Rabbitfish rabbitfish
- JSON Markup Language (JsonML) JSONML
- XSLTJSON, XML to JSON using XSLT XSLTJSON
- XML to JSON jQuery Plugin JQUERY
- Boomerang - A bidirectional programming language for ad-hoc, textual data. BOOMERANG
- XSugar - Dual Syntax for XML Languages XSUGAR
- OGF Standards: Data Format Description Language (DFDL) DFDL
- MLJSON - An XML Facade over JSONMLJSON

### *Limitations of existing implementations*

Existing implementations and designs share many common limitations which make them less then ideal for many use cases. The existing models tend to fall into these use cases.

- Represent arbitrary JSON in an XML format
- Represent arbitrary XML in a JSON format
- Convert XML to JSON in a JSON friendly format
- Convert JSON to XML in a XML friendly format

In cases where round tripping is a goal inevitably either the JSON or the XML end up being very "non friendly". Conversely mappings to "friendly" formats tend to lose too much information to support round tripping (or only in a small subset of documents).

All investigated designs have fixed rules that apply either globally, or if configurable, to an entire document. None of the designs can adjust the mapping rules localized to portions of a document.

The major limitations which JXON attempts to address include the following.

- Single model for all documents and parts of documents

  Most implementations impose a single model for transformation for all documents and parts of documents. They attempt to fit a single pattern for all uses. This "one size fits all" approach ends up being "one size fits nothing", as every mapping imposes some compromises, whether its information loss, usability or applicability to a particular use. Since different use cases may accept different sets of compromises, trying to fit a single transformation mapping for a large set of use cases ends up with a poor fit for most of them.
- JSON or XML Centric

  Some designs focus on producing JSON from arbitrary XML, others on producing XML from arbitrary JSON. The result is that one side or the other is "stuck" with the artifacts of the mapping and do not end up with a document in a format which would be natural for markup developers in that language. JXON attempts to produce bidirectional transformations to "friendly" formats in each directly equally.

## Rationale and Use Cases

JXON is based on the assumption that there are real world needs for a JSON/XML transformation. Why is there such a need? Many existing systems work just fine using XML or JSON by itself with no need of an alternative representation for data. The problems which JXON is intended to address arise when data crosses domains. Due to technical and as well social reasons domains have evolved which embrace either XML or JSON almost exclusively. XML has largely been adopted by the enterprise and content storage domains, as well as enterprise messaging. JSON has evolved into the browser and mobile space which started using XML but have changed over the last decade to be almost entirely JSON based. The problem is when these domains need to exchange data. One solution is to simply produce data in the format required at the point of request. This is a feasible solution when the data is dynamically generated (computed data, simple messaging, data residing in-memory in native language structures), or exists in an entirely separate format to begin with (such as relational data). In many cases, however, data already exists in XML or JSON in one domain and needs to be exchanged with another domain. For example a document may exist in XML form on a server and needs to be deployed to a browser or mobile device in JSON format. One approach to this problem is to try to "push" one's domain format into the other's domain. For example, if clients require JSON documents then simply store the JSON format on the server. This can work when there is really no compelling reason to have XML on the server in the first place, and has spawned the growth of JSON databases. Alternatively one can try to force clients to accept XML because that is the format on the server.

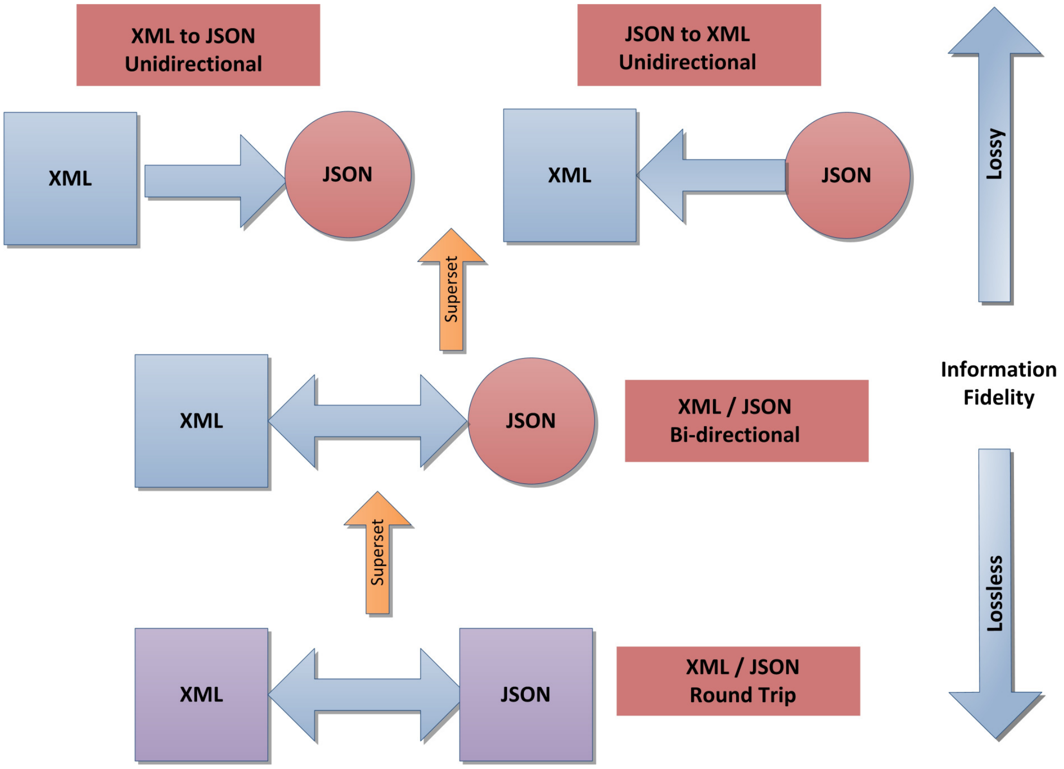
I argue that it is preferable in many cases to accept that there are compelling reasons for each domain to want data in its own format, and to provide for a transformation from one format to another closest to the point of the domain boundary. This allows the tools already in use in the particular domain to work with their 'native' format and to minimize the impact of format differences to domain boundary layer. This way neither domain needs to make compromises in design and implementation to satisfy issues of a different domain.

JXON is designed to make that boundary crossing as easy as possible and with as few compromises as possible.

### *Categories of Transformations*

There are several major categories of JSON/XML transformation problems which are really subsets of functionality. Not all use cases require all levels of functionality but by supporting the most stringent requirements implicitly creates support for the rest. In general there is a correlation between information fidelity and the category of transformation. For example a unidirectional XML to JSON transformation may lose significant amounts of information (and yet still be useful). While a full round trippable transformation, by its very nature, must maintain a maximum amount of information in either the instance documents or the transformation logic. Note that these transformation categories are not specific to XML and JSON but apply equally well to any kind of document formats.

**Figure 1: Types JSON/XML transformations**



Types of JSON/XML transformations

### Unidirectional Transformations

Some transformations are *unidirectional*; information goes only one way. Unidirectional transformations may have a wide range of information fidelity, but are often useful with significant information loss. Some unidirectional transformations are generic, that is the rules are static and do not depend on the source or target schemas. Other transformations may be tailored for specific source and target document schemas.

### Bi-directional Transformations

Bi-directional support transforming in both directions (from XML to JSON or JSON to XML).

### Round trip transformations

Round trip transformations are a special kind of bi-directional transformations that attempt to produce semantically equivalent documents after transforming from one format to another then back.

## *Use Cases for XML/JSON transformations*

There are several use cases that require or benefit by one of the categories of XML/JSON transformations. The more useful cases are where the data in a domain not only is desired in a specific format, but also persist or may be manipulated by tools specialized for that format and that the data needs to be exchanged with another domain using a different format. Examples of good use cases for transformations include the following.

### Message passing

Many enterprise systems operate on XML based messages. If these are to be sent to a domain needing JSON then the messages must be transformed from XML to JSON. Responses may come back in JSON and need to be transformed back to XML. In this case unidirectional transformations may be sufficient (one for requests and one for responses), although if the request and response correspond to the same schema then a bidirectional transformation may be better suited.

### XML Content Databases

If data is stored in an XML Content database and JSON is required by a JSON consumer then transformation from XML to JSON is needed. If JSON producer requires data to be stored in an XML Content database then it needs to transformed to XML. The transformations may or may not require round trip (lossless) transformations depending on the needs of the application.

### HTML/XHTML

If data is in HTML format there are tools available to transform it to XHTML format. If that data is required in JSON then a XML to JSON transformation is required. The reverse is also possible, generating HTML from JSON.

### XML Schemas

A schema language for JSON is being developed JSON schema but implementations at the time of this writing are sparse and the technology is immature. If a high quality JSON to XML transformation is available then XML Schema validation could be used to validate JSON documents. This would leverage the mature technologies for XML Schemas without having to reinvent them specific to JSON.

### Using XML technologies with JSON data

Many XML technologies are very mature and powerful. Equivalent JSON technologies are only starting to be developed. There is simply no equivalent to tools like XSLT, XQuery[7] , XProc[8] , Schematron[9] for JSON.

Use of a high quality XML/JSON transformation could allow use of XML technologies on JSON data without having to reinvent the entire XML ecosystem for JSON. This may require bidirectional transformations but not necessary round trip (lossless) transformations depending on the needs of the application. In the case where the source and target data are different schemas, a pair of unidirectional transformations is needed instead of a bidirectional transformation.

### Preexisting data

There exists 'in the wild' a huge amount of preexisting content in XML format. This content could be made available to JSON oriented domains by the use of XML to JSON transformations instead of having to re-author the content directly in JSON. In cases where there is existing JSON content, or processes that produce JSON content only, that content can be made available to XML oriented domains by the use of a JSON to XML transformation instead of re-authoring the content.

### XML Standard Schemas

There is a large body of XML standard schemas which have evolved over the last decade. They span a huge section of industry, research, academia and publishing. These schemas took significant effort to develop. By using a XML to JSON transformation this body of knowledge can be leveraged into a new markup technology without having to start from scratch.

### Developer Familiarity

One of the big reasons for the different domains which have adopted JSON or XML is developer familiarity. Web and browser developers, having experience with JavaScript are more comfortable with JSON even on platforms where XML support is equal or even better. Similarly server and data processing developers are often more comfortable with XML even in use cases where JSON is well supported. If there is a technology which bridges XML and JSON easily and accurately then developers don't need to leave their area of familiarity.

# Design

## *Goals*

The goals of this design are to address many of the limitations of existing implementations and to provide a simple method for customizing the transformation. At the same time it's recognized that not all limitations can be addressed, nor necessarily need be addressed, and that this design may not meet the objectives of everyone. The use cases *See* section “Use Cases for XML/JSON transformations” were of particular use in focusing the design goals.

### Primary Objectives

The following are the primary design objectives. Not all objectives can necessarily be met with the same transformation.

- Unidirectional and bidirectional transformations from a instance of a JSON document to an instance of an XML document.
- The option of lossless round trip transformations.
- The XML and JSON instances should be in a form which is desirable by XML and JSON developers respectively.
- A simple method of modifying the transformation rules for parts of a document as well as the whole document for a class of documents.
- A pattern based rule system to allow specifying common patterns for mappings instead of having to code explicit details.
- Implicit mappings of JSON and XSD atomic types and common structures by leveraging XML Schema information.

## *Design Overview*

**Figure 2: JXON Processor**

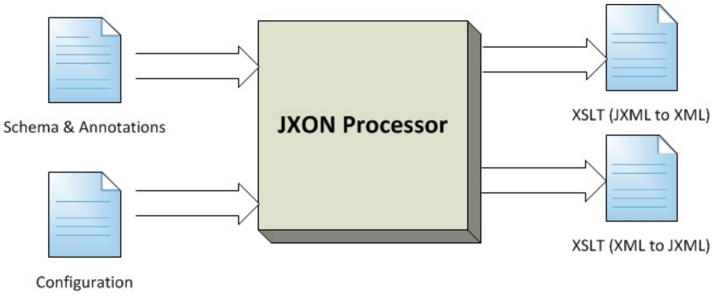
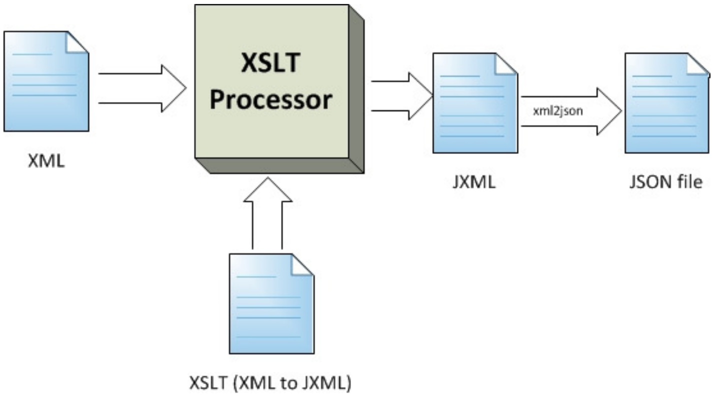


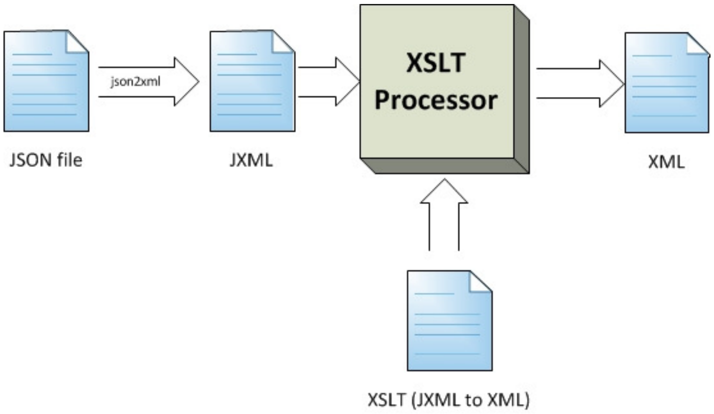
Figure 1. JXON Processor data flow

**Figure 3: XML to JSON**



Transforming XML to JSON data flow

**Figure 4: JSON to XML**



Transforming JSON to XML data flow

### Transformation of the Object Model

As mentioned previously, the text format of XML vs. JSON tends to get a great deal of focus, while I assert it is actually a trivial issue. This design performs all transformation logic at the Object Model level not at the text format level. To accomplish this a very simple XML schema is used which represents the JSON Object Model precisely. This implementation uses the JXML schema (http://xml.calldei.com/JsonXML) to represent the JSON object model and the conversion tools supplied by xmlsh (http://www.xmlsh.org). In particular the json2xml (http://www.xmlsh.org/CommandXml2json) and xml2json (http://www.xmlsh.org/CommandJson2xml ) commands. Other schemas and tools could just as easily be used as long as they provide a precise representation of the JSON object model in XML (such as JSONx jsonx ).

Transformation at the Object Model level provides the following benefits:

- Reduces the problem to an XML-to-XML transformation problem instead of a Markup-to-Markup problem.

- Decouples transformation logic from serialization logic
- Allows all processing to be implemented with XML tools which are mature and feature full.
- Allows focus on the transformation logic instead of serialization logic.
- Provides for some schema validation of the generated JSON to check for JSON well-formedness prior to serializing as JSON.

### Patterns and Rule Properties

A major problem with XML/JSON transformation is that there are many ways to do it. These can be classified into 'patterns' which define the rules for bidirectional transformation of a given set of terms.

Patterns are defined in a global configuration file which defines specific named patterns which can be referenced by the annotations, and configuration properties of each pattern which can be selectively overwritten in a specific annotation. The properties of a pattern define basic building blocks of transformation such as element and attribute wrapping, name translations, handling of namespaces and typing of atomic values. The code to implement a pattern can be very complex, especially if reverse transformation is desired. By having the framework predefine patterns, a developer can simply tag an element, attribute, or type with a pattern name and the rules will be applied to that element (and optionally its derived and contained types).

This is intended to be extensible so that new patterns can be added to the code to enrich the range of transformation options, and need only be identified by name to be applied. At present there are two patterns supported "full" and "simple". These are intended to demonstrate extremes between transformation categories. More patterns are in development.

Patterns are applied to items (documents, elements, attributes, types) by using the annotation *<jxon:pattern name="pattern name"/>* This applies all the rule properties associated with that pattern as applicable to the item type. It also applies the properties to the derived and contained types. For example applying a pattern at the schema root applies the pattern to all items. Applying to an element affects the element, its attributes, and all local element declarations. Applying a pattern to a type affects the type and all derived types.

*See* section “Patterns”

### Markup Agnostic but XML Based

While a primary goal of the design is to provide a transformation which is markup agnostic ( treats XML and JSON as equals). The reality is that the tools for processing markup are much more mature and feature full in the XML space than the JSON space. This practical issue lead to a design which is "XML Based" but also "markup agnostic". The technologies used are primarily XML based, and the design itself is based on XML Schema, Annotations in schema, XQuery and XSLT processing. This does force a compromise in design which is definitely XML oriented, but still providing the end result (input and output of the transformations) markup agnostic.

### Schema and annotation based transformations

Analysis of the existing implementation limitations and the primary objectives lead to the conclusion that a Schema and associated annotations could provide much of the external information needed guide the transformations in both directions.

For example on generating JSON, Schema type information can inform the choice of which JSON types to generate. On generating XML, Schema can provide the missing pieces of structure which may have been omitted in the JSON, for example the correct element names, wrapping elements, namespaces etc.

Schema Annotations allow specifying ancillary information which is to apply only to a specific element, attribute or type without having to add addressing to the rule-set. The addressing is implicit by the location of the annotation. Since types can be annotated as well as specific elements and attributes, patterns can applied to all elements/attributes of that type in one location. Similarly type hierarchies can be annotated by placing the annotations at the appropriate location in the type hierarchy.

Individual properties of patterns can be overridden at any location, which acts similarly to object derivation. For example the "simple" pattern can be used for the document as a whole, while on an individual element its JSON name can be modified via annotation while inheriting all other derived properties of the pattern.

#### *XSD Schema*

The initial implementation directly supports XSD Schema only. This is an implementation artifact derived by use of the Apace Schema API, not a design restriction. XSD Annotations may inserted into any schema item (document, element, type etc) which will override either the current in-effect pattern or any of the pattern rules. JXON annotation must occur within the "appinfo" XSD annotation element. Only elements within the jxon namespace are processed, the rest are ignored.

The following at the beginning of a schema indicates that the entire document defaults to the "full" pattern

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:jxon="http://www.xmlsh.org/jxon">
        <xs:annotation>
                <xs:appinfo>
                        <jxon:pattern  name="full"/>
                </xs:appinfo>
        </xs:annotation>
        ...
```

The following indicates that the ITEM element and all its child elements use the "simple" pattern and that the ITEM element itself will be represented by the JSON Name "BOOK".

> ***Note***
>
> Note that the "json_name" element is a sibling, not a child of "pattern". This indicates that the name only applies to this element, not to its descendant local elements.

```
<xs:element name="ITEM">
                <xs:annotation>
                        <xs:appinfo>
                                <jxon:pattern  name="simple"/>
                                <jxon:json_name name="BOOK"/>
                        </xs:appinfo>
                </xs:annotation>
...
```

#### *RNG Schema*

The design supports use of any XML schema which can be annotated. However the current implementation only directly supports XSD.

RelaxNG (both Compact and XML formats) is indirectly supported by the use of comments in RNG and translating the RNG into XSD and the comments into XSD annotations.

The following RNG compact syntax uses comments starting with *#<jxon:* to map a different JSON name to an element

```
#<jxon:json_name  name="TITLE"/>
element NAME { xs:string }
```

This is converted to the following XSD notation.

```
<xs:element name="NAME">
                <xs:annotation>
                        <xs:appinfo>
                                <jxon:json_name name="TITLE"/>
                        </xs:appinfo>
                </xs:annotation>
...
```

Similarly comments in the RNG XML syntax are supported in the same way.

### Rule Generation

A pair of transformation rules (XSLT files) for a class of documents is generated.

The XML Schema (for the XML document) is read along with the global configuration. The schema is traversed and a XSLT file is produced for each direction (JSON to XML and XML to JSON) based on each element, attribute, and type defined in the schema. Schema Annotations are used to choose which pattern applies to each term, and provides optional customization to the patterns for that item. Patterns can be applied recursively, or only to targeted items.

The result is a pair of XSLT files.

### Transformation

The XSLT files produced by the rule generation can be used without any ancillary information. Reference to the schema or configuration information is not needed.

Running a transformation from JSON to XML requires converting the JSON to JXML format then running XSLT. Schema validation can be applied after the transformation.

Running a transformation from XML to JSON requires running the XSLT then converting from the result JXML to JSON. Schema validation can be applied to the input prior to running the transformation, and to the JXML after transformation to validate JSON well-formedness.

### Bidirectional ("Round Tripping")

JXON is intended for bidirectional transformations ("Round Tripping"), although the same architecture and implementation could be used for unidirection transformations (XML to JSON or JSON to XML). A bidirectional transformation can be either Lossless or Lossy (or somewhere in between). This can depend on the patterns being used, the schema, and actual instances of documents.

The definition of "Lossless" and "Lossy" depends on what one considers "important information", although the terms do convey some meaning. Consider a scale measuring the results of a round trip transformation with one side exact byte for byte bidirectional transformation for every document instance transformed ("Lossless"), and the other side no correlation whatsoever between the two documents ("Unidirectional"), with "Lossy" being somewhere in between. Even this linear scale is misleading because some document instances corresponding to the same schema may have different translation characteristics then others. The design goals of JXON are to provide reasonably defined translation characteristics. Choosing a particular pattern or configuration attributes of a pattern will effect the translation, and it is up to the developer to determine the importance of the specific effects.

The JXON architecture does not attempt to implement "Purely Lossless" transformations at the text serialization (byte sequence) level.

#### *Acceptable Losses*

The JXON architecture does not attempt to preserve some characteristics of XML or JSON at all. These are considered "Acceptable Losses" for this project and for the purposes of definitions do not contribute to the meaning of "Lossless".

The following XML information is not attempted to be preserved.

- Serialization Format

  The Text Serialization format of XML is not considered. Transformations are done at the XDM abstraction level so characteristics such as encoding, ignorable whitespace, character entities, CDATA etc are not considered as a significant characteristic.
- External Entities

  External entities in XML are expanded as part of the XML parsing and are not represented in JSON as entities. They are not converted to external entities when generating XML from JSON.
- DTD and Schema

  DTD and Schema references in XML are not retained as part of the transformation. They may be reconstructed as part of the XML generation but no attempt is made at representing DTD or Schema information in the JSON output.
- Comments

  Comments are removed during XML processing. They exist as part of the XDM model but in the current implementation they are ignored.
- Processing Instructions

  Processing instructions are removed during XML processing. They exist as part of the XDM model but in the current implementation they are ignored.
- Application level XML markup

  XML markup which has meaning by application specific processors is not explicitly preserved, or may be partially preserved. For example XInclude processing (like External Entities) may be merged into the XDM model during parsing so is not transfered into the JSON markup explicitly. XLink and XPointer markup is not attempted to be processed specially.

  XML markup which has specific meaning to an application may not have any related meaning in JSON.. For example an XSLT file translated to JSON looses its meaning in that the resultant JSON document has no supporting processors for XSLT.

#### *Optional Losses*

Sometimes perfect Lossless bidirectional transformation is not important. JXON provides the ability to perform Lossy transformations which can still produce useful results. A simple example is value typing. If there is no schema or annotation to describe the type of a value then transformations can lose that type. A numeric type in JSON round tripped may become a String type. This change of type information may or may not matter to the application.

Another case is where attributes or elements are requested to be ignored (in JSON or XML). This obviously causes problems unidirectional as well as bidirectional transformations. Omitted elements and attributes could be regenerated on reverse transformation according to the schema (or annotations) but the values cannot be extracted from the document instance.

A more subtle type of losses can occur when assumptions are made about documents which is not explicit in the schema or the JXON annotation. For example the "simple" pattern translates XML elements and attributes to JSON members. This pattern can round trips losslessly as long as there are no repeated element children or duplicate element or attribute names. However given an XML document that doesn't conform to the assumptions can produce invalid JSON, or the round trip may be produce ambiguous results.

Future enhancements could detect and warn the developers of the sort of losses or errors which could be expected.

### Implicit Information

The JXON architecture makes significant use of "Implicit Information". Since an XSLT file is generated for both directions of mapping, not all information needed to build the documents during transformation need to be in the documents themselves. This allows many features not available with other JSON/XML designs .

A simple example is naming. Suppose a XML document has an element <my_element> and we wish to create a JSON document with an object member "The Element". This can be specified as a name attribute to the pattern for "my_element" (via annotation in the schema) and will be encoded into both XSLT files. The resulting JSON document has no explicit reference to the XML element name, yet via the XSLT will create the correct element name. The same concept solves a large number of issues usually considered deficient in many XML/JSON mapping tools such as:

- Namespaces

  XML Namespaces can be preserved round-tripped through JSON without encoding the namespace itself in the JSON file. JSON objects can produce elements and attributes with namespaces without having the namespace information in the JSON document itself.
- Element and Attribute distinction

  Some patterns map elements and attributes both to named JSON members. The distinction between element and attribute is not present in the JSON document. On the reverse transformation the elements and attributes can be reconstructed.
- JSON Arrays

  JSON arrays can be created from various XML structures. The resultant JSON may lose the original XML element or attribute names, but on the reverse translation these items are reconstructed. Starting from JSON, arrays can be wrapped in generated XML elements or into a tokenized list value.
- Value type Information

  When translating XML to JSON the value type can be lost ( e.g. `"value" : 1` vs. `<value>1</value>`)

  When translating back from XML to JSON the type information can be restored even though it is not present in the instance document.
- Constructed or deleted elements

  Some patterns construct or delete markup, yet are reversible. For example wrapping elements may be removed when XML is converted to JSON. On the reverse transformation they can be reconstructed. Anonymous JSON objects (such as the root document) can have elements synthesized.
- Composed Values

  XSD "list" items in XML are a single atomic value. When translated to JSON they can be split into Array notation (separate JSON objects). On the reverse transformation they can be reformed back into a single atomic value.

## JXON Transformations

The current JXON design and implementation supports the following types of transformations. Since JXON is still in development these will likely change as experience is gained and the implementation matures. A primary goal of choosing the types of transformations is to find the right level of control of the mapping logic which is both implementable and not too difficult for the user to control. The rules use information from the schema as well as allow configuration properties externally (via the configuration file and inline schema annotations). These transformation rules, and their properties, are then grouped into "patterns" which can be applied 'as is' or overridden with specific changes.

### *Atomic Types*

JXON uses the schema type declarations for simple types to attempt to determine the proper JSON type. The XSD schema type hierarchy is searched until a matching known type is found. If a matching type is found then the corresponding JSON type is used, otherwise the String type is used.

Any type derived from xs:double, xs:float or xs:decimal are mapped to a JSON Number.

Any type derived from xs:boolean is mapped to a JSON Boolean literal.

All other types are mapped to a JSON String.

Empty XML elements are mapped a JSON null.

**Table I**

Example Atomic Type Conversion

| xs:type | XML | JSON |
|---|---|---|
| xs:integer | 1 | 1 |
| xs:boolean | true | true |
| xs:string | John Doe | "John Doe" |
| empty | <empty/> | null |

### value element

In the JXON schema the "value" element controls how atomic values are transformed. In the following example, the value element specifies that schema information should be used for both the value transformation as well as determine JSON array mapping

```
        <value wrap="schema" type="schema" />
```

The following overrides default behaviour and specifies that the JSON Number type should be used with no wrapping

```
        <value type="number" />
```

### Structured Types

JXON maps XSD "list" types (and derived types such as IDTOKENS) to JSON arrays and on reverse maps the associated arrays back to a space separated atomic value.

**Table II**

Example list type conversions

| xs:type | XML | JSON |
|---|---|---|
| xs:NMTOKENS | Hi There | [ "Hi","There"] |
| xs:IDREFS | id1 id2 id3 | [ "id2" , "id2" , "id3"] |
| <xs:list itemType="xs:integer"/> | 1 2 3 4 | [ 1 , 2 , 3 , 4] |

The value element can be used to force wrapping by a JSON array even if the schema does not identify a value as a list type. For example the following overrides the schema information and tells the JXON processor to wrap the value with a JSON array of strings.

```
        <value type="string" wrap="array"/>
```

### Naming and namespaces

JXON provides a flexible method for configuring name transformations. JSON and XML names can differ significantly in possible characters and practical use. The default configuration is to ignore namespaces and use the same name for XML elements and attributes as JSON members. Although namespaces are not present in the JSON document they are reconstructed when translating JSON to XML. Thie naming rule can be overridden on a type, element or attribute basis (and child items) by supplying a regular expression which translates the full XML name (in Clark notation[10]) to the JSON name. This allows the uri and localname portions of a QName to be used in generation of a JSON name. Since the document instance is not avaiable during rule generation, namespace prefixes are not known so are not accounted for in the naming rules. You do not have to supply the reverse regex for reverse mappings, in fact the mappings can be lossy while still being fully reversable because the naming information is stored in the XSLT files. The JSON member names have to be unique within a JSON object, however, in order to be able to match the corresponding XSLT rule (and XML name). It is interesting to note that even if namespace information is not mapped into the JSON output, that namespaces are not lost on reverse mapping. It is only in the case where multiple element or attributes with the same name but in different namespaces are used within a single schema and document instance that namespaces need be considered at all in the nameing mappings to provide for lossless round tripping.

This naming rule can also be used to get around the problem of same-named attributes and elements in XML by assigning them unique names in JSON.

**Table III**

| Name rule | Clark Name | JSON Name |
|---|---|---|
| <json_name search="\{([^}]*)\}?(.+)$" replace="$2"/> | {http://www.myorg.org}name | name |
| <json_name name="Json Name"/> | {http://www.myorg.org}name | Json Name |
| <json_name search="\{([^}]*)\}?(.+)$" replace="1_$2"/> | {http://www.myorg.org}name | 1_name |

### Elements and Attributes

XML Elements and Attributes are both mapped to JSON Object members. JSON Object members must be a direct child of a JSON Object. This implies that all XML Elements and Attributes must be placed as direct children of a JSON Object (must not be parent-less or a direct child of a JSON Array.). The Element and Attribute rules interact with the Wrapping Rules to enforce this constraint as well as provides variants on how elements and attributes are placed within the containing object.

#### children element

The children element in JXON specifies how to encode XML element children in JSON. The following declaration in the "full" pattern specifies that child elements should be wrapped in a JSON object as the "_children" member

```
        <children wrap="object" name="_children" />
```

The following declaration used in the "simple" pattern specifies that child elements are transformed into JSON members without wrapping

```
        <children wrap="none" />
```

Of course this will only produce valid JSON if there are no repeated elements or are duplicates of unwrapped attributes.

#### attributes element

The attributes element in JXON specifies how to encode XML attributes in JSON. The following declaration in the "full" pattern specifies that attributes are wrapped in a JSON object as the "_attributes" member.

```

        <attributes wrap="object" name="_attributes" />
```

The following declaration used in the "simple" pattern specifies that attributes are transformed into JSON members without wrapping

```
        <attributes wrap="none" />
```

### Text elements

Text elements can be treated as simple values or as objects. They can be wrapped along with child elements, or in their own member. The JXON text element determines the behaviour.

The following example from the "simple" pattern wraps all text into an object member named "_text".

```
        <text wrap="object" name="_text" />
```

The following example from the "full" pattern treats text the same as child elements (putting each text element as a separate value in the _children array)

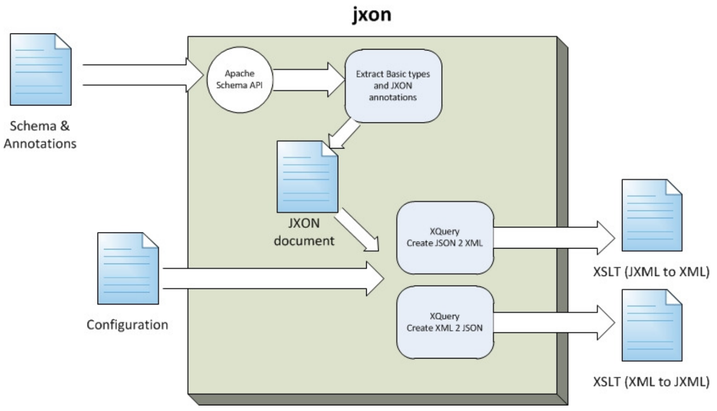
```
        <text wrap="none"/>
```

## Implementation

The current JXON implementation is an experimental prototype with the intent of exploring the design and usefulness of the architecture. The intent is to become a production quality tool, but at the time of this writing it is still in prototype phase; functional but not yet feature complete. The code is entirely open source, uses only open source libraries and is licensed with the "Simplified BSD License".[11] It is distributed as part of the JSON extension module for xmlsh http://www.xmlsh.org/ModuleJson. The code can run within xmlsh itself or as an independant command or embedded in a java application.

As described in the Design Overview, JXON produces a pair of XSLT files from a schema and a configuration file. These XSLT files can then be used for transformations without further reference to the schema or the JXON processor. Input and output to the XSLT files are XML. On the JSON side this is XML in JXML format which can be converted to and from JSON losslessly, or converted to other lossless formats such as JSONXjsonx.

**Figure 5: JXON Processor implementation**



JXON Processor implementation internals

The resulting XSLT is produced as XSLT 2.0 for a non-schema-aware processor so that the entire process may be executed entirely with open source software.

### *JXON Processor*

The JXON processor (Figure 2) is run once per schema to produce the pair of transformation XSLT files. The program (called "jsonxslt" Figure 5.) is written partially in Java and partially in XQuery. The Java portion parses command line arguments, reads the Schema (using the Apache implementation of the XML Schema API) and generates a much simpler form of the schema and annotations in XML form. The intermediary schema format corresponds to the same schema as the configuration format as well as the annotation (JXON). This allows the configuration data to be cleanly merged with the annotation and the resulting document and internal representations all having the same namespace and sharing the same meaning for items.

Java is used for this component because parsing, and particularly comprehending XSD schema is a challenging task suited well to reusing an existing API. The simplified schema and the global configuration file is then passed to the XQuery component, once each for conversion to XSLT for the JSON to XML translation and again for producing the XML to JSON. XQuery was chosen for this component because it is a good fit for XQuery. The process is largely that of querying and producing XML (XSLT) output from a complex set of algorithms. The two XQuery programs are tightly coupled (and share common modules). This is necessary because the specifics of the XSLT rules generated for one direction need to match exactly the rules generated for the reverse direction in order to achieve round trip transformations.

The result is 2 XSLT files which have encoded all the rules, augmented by the annotations, into a set of XSLT templates. These templates match every element and attribute of the input (Either XML or JXML), apply the encoded transformation and output the transformed XML. (*See* section “Example XSLT generation”

#### Parsing Schema

##### *RNG Support*

Ideally native support for RelaxNG will be built into the project but for now support is by means of the *trang* *TRANG* converter and a transformation which converts XSD comments into XSD Annotations. Comments are used instead of RNG annotations because trang does not sufficiently support translating RNG annotations to XSD annotations.

### *Converting between JSON and JXML*

As described in the design, the JXON processor consumes and produces JSON in JXML format. This could in theory be any other full fidelity JSON representation in XML.

This JXML format may be useful by itself as it can be converted to other JSON XML formats such as JSONX jsonxor used as an input to other processors that operate purely on the JSON Object Model as apposed to the JSON Serialized Text form.

To convert from JSON to JXML and reverse, two tools are used from xmlsh command library (json2xml, xml2json). These tools are very simple and can be run standalone (outside the xmlsh environment), or could be re-implemented easily. Source code is available for these tools as well as all of xmlsh, also licensed using the Simplified BSD License. The json2xml tool makes use of the JSON API available from json.org, with some minor enhancements, mainly to preserve the order of object members while parsing JSON. The xml2json tool does not use the JSON API because of severe limits to that library, rather the serialization was written from scratch.

Neither of these tools are particularly complicated and could be replaced by user written code. The techniques for parsing and serializing JSON are well known and many implementations are available. You can see from the sample below that the JXML format is extremely verbose, but its advantage is it directly models the JSON data model.

### *XSLT processing*

Once the JXON process is complete the 2 resulting XSLT files can be used at any time. One file is used to transform JSON as represented in JXML to the target XML *See* Figure 4.

The other file is used to transform XML to JXML *See* Figure 3.

The conversion between JXON and JSON is performed using the json2xml and xml2json tools *See* section "Converting between JSON and JXML"

The XSLT files are currently created to run with any XSLT 2.0 non-schema aware processor. The implementation uses Saxon[12]

## Retrospective; Limitations and Lessons

JXON began as an experiment to explore the feasibility of intelligent cross markup transformations. Key to this experiment is the recognition that Markup Schema contains significant information which can be used to aid cross markup transformation. Recognizing this I assert that in essence cross markup transformation is fundamentally a Schema oriented issue, not a document instance issue. Lack of JSON Schema places a severe limit in the amount of automated decision making possible leaving one side only with schema information. This limit did make many design decisions easier, although less ideal, in particular to focus on the XML schema as the source of transformation logic.

### *Use of XSD Schema*

Recognizing that Schema was a core concept in the design, XSD Schema was chosen for the implementation. In concept any schema language should work, but XSD was chosen due to its prevalence and the availability of API's which can query the schema (XML Schema API). XSD Schema is complex. The fact that an API is needed to query it to get meaningful information (instead of say using XQuery directly on the schema), is an indication of how complex it is. Even the API itself is complex. This decision required the use of a Java component instead of relying entirely on XML tools.

Ideally any XML Schema language should be able to be able to be used. Since the source schema is only used early on in the work-flow (to generate a simpler intermediate document), supporting alternative schemas such as RNG natively should not be that difficult and may be introduced in a later version of the implementation. The current support for RNG via translation to XSD is a viable alternative.

#### Requires an XML Schema

Since the transformation generation is Schema based, this implies you must have an XML Schema to define a transformation. If you are starting with XML there are many tools to generate a schema based on sample XML. But if you are starting with JSON there are no such tools. This is an area which could use improvement. For now you are forced to create a schema which matches up with how the JSON would be generated using one of the existing patterns. Perhaps a schema-less transformation mode would be useful to bootstrap this process.

#### Recursive structures

JXON allows for recursive structures insomuch as XSD does. Since rules are associated with a specific element declaration you cannot target different rules for different levels of recursion.

#### Local Elements

JXON allows for local elements in XSD. Local elements of the same name as other local or global elements can be targeted individually. This allows samed-named XML elements to map to differently named JSON elements. However, as in recursion, it is only supported in the same fashion as XSD. If a document instance references the same element in different contexts they cannot be targeted differently unless it is a local element declaration.

### *Use of Schema Annotations*

The choice of using XML Schema (particularly XSD) then led to the decision to put the human augmented information into the XML schema document itself (as annotations). This decision is definitely debatable.

The advantage of annotating the schema itself instead of using an external "rules file" is that information can be associated directly to the specific item (type, element, attribute) without having to target that item externally with perhaps XPath or type references. The transformation logic then resides right with the item definitions in the schema and contains a minimal amount of extra information. On the other hand, this forces editing of the schema in order to modify the transformation logic. Different transformations for the same class of documents then requires separate schemas. This means that different schemas may need to be maintained that differ only in the JXON annotations. While JXON does not require that instance documents use the same schema during transformation as was used during the generation process, this can be a problem as changes to the authoritative schema require editing the schemas used for the JXON configurations.

Another issue of annotating schema directly is that it requires duplication of information if the same properties are to be applied to different items. This is mitigated by the use of "patterns" which group properties and allow you to specify only the pattern name. However if the rules were in an external file they could use a single definition for common properties and use instead multiple targets (e.g XPaths) to apply the properties to multiple items.

### *Intermediate file (JXON document) generation in Java*

Use of the Apache implementation of the XML Schema API requires use of a Java component for part of the generation process. To minimize the reliance on Java coding an intermediate XML file is generated with a simplified version of the schema along with the annotations. This worked out fairly well as it separated the logic for parsing the schema from the rest of the processing. One problem is that the format for the intermediate is a moving target. Exactly what constitutes a "minimal simplified representation of the schema" depends on the needs of downstream processing. Adding new features and making changes to the XSLT generation code often requires going back to the Java code to add or change the content of the intermediate file.

Once the initial stage is complete then the rest of the XSLT generation can be done using pure XML tools. This allowed for easier development and debugging.

Use of an intermediate representation for the schema also benefits from being schema language agnostic. A different input schema language (such as RelaxNG) could be used while generating the same JXON intermediate format and leaving the rest of the processing unchanged.

### *XQuery for XSLT generation*

XQuery is used to generate the XSLT output given the intermediate schema file and a configuration file. Choice of XQuery seems a good fit for the problem. The input and output is entirely XML and the processing logic is query and data oriented logic operations. This fits the XQuery use case quite well. Certainly XSLT or Java could have been used for this component, but the choice of XQuery worked well.

### *XSLT for transformation*

The output of the JXON processor is a pair of XSLT files. These XSLT files are used for the actual transformations. Choice of XSLT seems an obvious match. The transformation is a collection of matching templates along with generation rules. This fits the XSLT model very well.

#### Auto generation of XSLT

In practice the resultant XSLT can be both large and non-ideal. For example the XSLT for Docbook is over 10,000 templates. XSLT 2.0 without schema support was chosen for the initial implementation. The rationale is to provide an implementation which is entirely "open source" and can be run without licensing. However since the problem is fundamentally a schema problem, making use of a schema-aware XSLT may have significant value. For example when a rule applies to a type, templates are generated for every possible occurrence of that type. Using a schema aware XSLT processor could optimize this to matching on types instead of names (at least for the XML to JSON conversion).

Generation of the XSLT code is also quite tricky and difficult to debug which limits ease of adding transformation rules. In particular writing the code that generates XSLT which can transform bidirectionally is quite tedious. This is definitely an area that could use significantly improvement and optimization.

### *Naming and namespaces*

The mechanism for customizing name and namespace mappings is based on regular expressions. This is quite powerful but may not be powerful enough. The reason regular expressions were chosen is an artifact of the implementation; is that its the most flexible string manipulation feature in XQuery which doesn't require an "eval". This area could use enhancements.

The naming rules only have access to the localname and URI of the QName. This means that prefixes cannot be used as part of the name mappings. When generating XML from JSON prefixes are constructed and namespace declarations are added as needed. The resulting XML will most likely have a different lexical representation then a source XML after round tripping due to the difference in prefixes and locations of namespace declarations. Possibly making use of the prefix bindings in the XSD would be useful although there is no guarantee these same prefixes would be used in instance documents.

Using the default naming rules which ignore the namespace when generating JSON can lead to ambiguity problems if two QNames from different namespaces are allowed as attribute or child element names in a given element. This is resolved by overriding the naming rule for one of the ambiguous names.

Better control over prefix assignment and namespace declaration placement would be a useful enhancement.

### *Character Set*

JXON uses JXML's handling of non-XML representable characters such as \0 by keeping them in JSON escaped format when mapped to XML. While this is losses, it is non-ideal as it takes application logic to parse these values correctly. I know of no better solution to this problem as XML simply cannot encode the full Unicode code-set which is allowed in JSON.

## Appendix A. Appendix

### *Patterns*

The following is the JXON global configuration markup of the "full" and "simple" patterns.

```
<patterns xmlns="http://www.xmlsh.org/jxon" default="full">
        <pattern name="full">
                <element>
                        <!-- Wrap all child text and elements in an object -->
                        <children wrap="object" name="_children" />

                        <!-- Wrap all attributes in an object -->
                        <attributes wrap="object" name="_attributes" />

                        <!-- Do not wrap text by itself, it is in the _children -->
                        <text wrap="none"/>

                        <!-- Values are typed and wrapped according to schema -->
                        <value wrap="schema" type="schema" />
                        <json_name      search="\{([^}]*)\}?(.+)$" replace="$2"/>
                </element>

                <attribute>
                        <value wrap="schema" type="schema" />
                        <json_name      search="\{([^}]*)\}?(.+)$" replace="$2"/>
                </attribute>

        </pattern>

        <pattern name="simple">
                <element>
                        <children wrap="none" />
                        <attributes wrap="none" />
                        <text wrap="object" name="_text" />
                        <value wrap="schema" type="schema" />
                        <json_name      search="\{([^}]*)\}?(.+)$" replace="$2"/>
                        <override/>
                </element>
                <attribute>
                        <value wrap="schema" type="schema" />
                        <json_name      search="\{([^}]*)\}?(.+)$" replace="$2"/>
                </attribute>
        </pattern>

</patterns>
```

### *Example of JSON file converted into JXML*

JSON Sample

```
{
        "id": "0001",
        "type": "donut",
        "name": "Cake",
        "ppu": 0.55,

        "topping":
                [
                        { "id": "5001", "type": "None" },
                        { "id": "5002", "type": "Glazed" },
                        { "id": "5005", "type": "Sugar" },
                ]
}
```

JXML Sample

```
<object xmlns="http://www.xmlsh.org/jxml">
   <member name="id">
      <string>0001</string>
   </member>
   <member name="type">
      <string>donut</string>
   </member>
   <member name="name">
      <string>Cake</string>
   </member>
   <member name="ppu">
      <number>0.55</number>
   </member>
   <member name="topping">
      <array>
         <object>
            <member name="id">
               <string>5001</string>
            </member>
            <member name="type">
               <string>None</string>
            </member>
         </object>
         <object>
            <member name="id">
               <string>5002</string>
            </member>
            <member name="type">
               <string>Glazed</string>
            </member>
         </object>
         <object>
            <member name="id">
               <string>5005</string>
            </member>
            <member name="type">
               <string>Sugar</string>
            </member>
         </object>
      </array>
   </member>
</object>
```

### *Full example*

The following is a full example of a bidirectional round trip JSON/XML transformation using a variant of the "BOOKS" schema

### XSD Schema

The XSD schema marked up with JXON annotations

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:jxon="http://www.xmlsh.org/jxon">
        <!-- Default document pattern is "full" -->
        <xs:element name="BOOKLIST">
        <xs:annotation>
                <!--BOOKLIST and decendants follow "simple" pattern -->
                <xs:appinfo>
                        <jxon:pattern name="simple" />
                </xs:appinfo>
        </xs:annotation>
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="BOOKS"/>
                                <xs:element ref="CATEGORIES"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>

        <xs:element name="BOOKS">
                <xs:annotation>
                <xs:appinfo>
                        <!-- Wrap all child elements in a JSON Array -->
                        <jxon:children wrap="array" />
                </xs:appinfo>
        </xs:annotation>
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="ITEM" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>

        <xs:element name="CATEGORIES">


                <xs:complexType mixed="true">
                        <xs:sequence minOccurs="0" maxOccurs="unbounded">
                                <xs:element ref="CATEGORY"/>
                        </xs:sequence>
                        <xs:attribute name="DESC" type="xs:string" use="required"/>
                </xs:complexType>
        </xs:element>
        <xs:element name="CATEGORY">
                <xs:annotation>
                        <!-- CATEGORY and children use the simple pattern -->
                        <xs:appinfo>
                                <jxon:pattern  name="simple"/>
                        </xs:appinfo>
                </xs:annotation>
                <xs:complexType>
                        <xs:attribute name="CODE" type="xs:ID" use="required"/>
                        <xs:attribute name="DESC" type="xs:string" use="required">

                        </xs:attribute>
                        <xs:attribute name="NOTE" type="xs:string" use="optional"/>
                </xs:complexType>
        </xs:element>

        <xs:element name="ITEM">
        <xs:annotation>
                <xs:appinfo>
                        <!-- ITEM and children use the simple pattern -->
                        <jxon:pattern name="simple" />
                        <!-- Rename the XML ITEM element to BOOK in JSON -->
                        <jxon:json_name name="BOOK"/>

                </xs:appinfo>
        </xs:annotation>
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="TITLE" type="xs:string" minOccurs="1">
                                </xs:element>
                                <xs:element name="AUTHOR" type="xs:string"/>
                                <xs:element name="PUBLISHER" type="xs:string"/>
                                <xs:element name="PUB-DATE" type="xs:date"/>
                                <xs:element name="LANGUAGE" type="languageType"/>
                                <xs:element name="PRICE" type="moneyType">
                                </xs:element>
                                <xs:element name="QUANTITY" type="xs:integer"/>
                                <xs:element name="ISBN" type="ISBNType"/>
                                <xs:element name="PAGES" type="xs:integer">
                                <xs:annotation>
                                                <xs:appinfo>
                                                        <!-- Demonstrate we can override the default JSON type with string -->
                                                        <jxon:value  type="string"/>
                                                </xs:appinfo>
                                </xs:annotation>
                                </xs:element>
                                <xs:element name="DIMENSIONS" type="dimensionsType">
                                        <xs:annotation>
                                                <xs:appinfo>
                                                        <!-- Wrap dimension children in an object member named "value" -->
                                                        <jxon:text wrap="object" name="value" />
                                                </xs:appinfo>
                                        </xs:annotation>

                                </xs:element>
                                <xs:element name="WEIGHT" type="weightType">
                                        <xs:annotation>
                                        <xs:appinfo>
                                                <!-- wrap WEIGHT text in an object member named "amount" -->
                                                <jxon:text wrap="object" name="amount"/>
                                        </xs:appinfo>
                                </xs:annotation>
                                </xs:element>
                        </xs:sequence>
                        <xs:attribute name="CAT" type="xs:IDREF" use="required"/>
                        <xs:attribute name="TAX" type="xs:NMTOKEN" use="optional" default="V"/>
                </xs:complexType>
        </xs:element>

        <xs:simpleType name="languageType">
                <xs:restriction base="xs:string">
                        <xs:enumeration value="English"/>
                        <xs:enumeration value="French"/>
                        <xs:enumeration value="German"/>
                        <xs:enumeration value="Spanish"/>
                </xs:restriction>
        </xs:simpleType>

        <xs:complexType name="moneyType">
                <xs:simpleContent>
                        <xs:extension base="xs:decimal">
                                <xs:attribute name="currency" type="currencyType"/>
                        </xs:extension>
                </xs:simpleContent>
        </xs:complexType>

        <xs:simpleType name="currencyType">
                <xs:restriction base="xs:string">
                        <xs:enumeration value="USD"/>
                        <xs:enumeration value="GBP"/>
                        <xs:enumeration value="EUR"/>
                        <xs:enumeration value="CAD"/>
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="ISBNType">
                <xs:restriction base="xs:string">
                        <xs:pattern value="[0-9]{9}[0-9X]"/>
                </xs:restriction>
        </xs:simpleType>

        <xs:complexType name="dimensionsType">
```

```
                <xs:simpleContent>
                        <xs:extension base="dimensionsContentType">
                                <xs:attribute name="UNIT" type="lengthUnitType">
                                </xs:attribute>
                        </xs:extension>
                </xs:simpleContent>
        </xs:complexType>

        <xs:simpleType name="dimensionsContentType">
                <xs:restriction>
                        <xs:simpleType>
                                <xs:list itemType="dimensionType"/>
                        </xs:simpleType>
                        <xs:length value="3"/>
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="lengthUnitType">
                <xs:restriction base="xs:string">
                        <xs:enumeration value="in"/>
                        <xs:enumeration value="cm"/>
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="dimensionType">
                <xs:restriction base="xs:decimal">
                        <xs:minExclusive value="0.00"/>
                </xs:restriction>
        </xs:simpleType>

        <xs:complexType name="weightType">
                <xs:simpleContent>

                        <xs:extension base="xs:decimal">
                                <xs:attribute name="UNIT" type="weightUnitType"/>

                        </xs:extension>
                </xs:simpleContent>
        </xs:complexType>

        <xs:simpleType name="weightUnitType">
                <xs:restriction base="xs:string">
                        <xs:enumeration value="oz"/>
                        <xs:enumeration value="g"/>
                </xs:restriction>
        </xs:simpleType>
</xs:schema>
```

### BOOKS XML

The BOOKS xml file

```
<BOOKLIST>
<BOOKS>
        <ITEM CAT="MMP">
        <TITLE>Pride and Prejudice</TITLE>
        <AUTHOR>Jane Austen</AUTHOR>
        <PUBLISHER>Modern Library</PUBLISHER>
        <PUB-DATE>2002-12-31</PUB-DATE>
        <LANGUAGE>English</LANGUAGE>
        <PRICE>4.95</PRICE>
        <QUANTITY>187</QUANTITY>
        <ISBN>0679601686</ISBN>
        <PAGES>352</PAGES>
        <DIMENSIONS UNIT="in">8.3 5.7 1.1</DIMENSIONS>
        <WEIGHT UNIT="oz">6.1</WEIGHT>
        </ITEM>
        <ITEM CAT="P">
                <TITLE>Wuthering Heights</TITLE>
                <AUTHOR>Charlotte Bront&#xeb;</AUTHOR>
                <PUBLISHER>Penguin Classics</PUBLISHER>
                <PUB-DATE>2002-12-31</PUB-DATE>
                <LANGUAGE>English</LANGUAGE>
                <PRICE>6.58</PRICE>
                <QUANTITY>113</QUANTITY>
                <ISBN>0141439556</ISBN>
                <PAGES>430</PAGES>
    <DIMENSIONS UNIT="in">1 5.2 7.8</DIMENSIONS>
                <WEIGHT UNIT="oz">11.2</WEIGHT>
        </ITEM>
        <ITEM CAT="P">
                <TITLE>Tess of the d'Urbervilles</TITLE>
                <AUTHOR>Thomas Hardy</AUTHOR>
                <PUBLISHER>Bantam Classics</PUBLISHER>
                <PUB-DATE>1984-05-01</PUB-DATE>
                <LANGUAGE>English</LANGUAGE>
                <PRICE>4.95</PRICE>
                <QUANTITY>85</QUANTITY>
                <ISBN>0553211684</ISBN>
                <PAGES>480</PAGES>
    <DIMENSIONS UNIT="in">6.8 4.2 0.8</DIMENSIONS>
                <WEIGHT UNIT="oz">7.7</WEIGHT>
        </ITEM>
        <ITEM CAT="P">
                <TITLE>Jude the Obscure</TITLE>
                <AUTHOR>Thomas Hardy</AUTHOR>
                <PUBLISHER>Penguin Classics</PUBLISHER>
                <PUB-DATE>1998-09-01</PUB-DATE>
                <LANGUAGE>English</LANGUAGE>
                <PRICE>4.95</PRICE>
                <QUANTITY>129</QUANTITY>
                <ISBN>0140435387</ISBN>
                <PAGES>528</PAGES>
    <DIMENSIONS UNIT="in">7.8 5.2 0.9</DIMENSIONS>
```

```
                <WEIGHT UNIT="oz">10.9</WEIGHT>
        </ITEM>
        <ITEM CAT="H">
                <TITLE>The Big Over Easy</TITLE>
                <AUTHOR>Jasper Fforde</AUTHOR>
                <PUBLISHER>Hodder &amp; Stoughton</PUBLISHER>
                <PUB-DATE>2005-07-11</PUB-DATE>
                <LANGUAGE>English</LANGUAGE>
                <PRICE>16.47</PRICE>
                <QUANTITY>129</QUANTITY>
                <ISBN>0340835672</ISBN>
                <PAGES>346</PAGES>
    <DIMENSIONS UNIT="cm">22.5 18 3.5</DIMENSIONS>
                <WEIGHT UNIT="g">390</WEIGHT>
        </ITEM>
        <ITEM CAT="P">
                <TITLE>The Eyre Affair</TITLE>
                <AUTHOR>Jasper Fforde</AUTHOR>
                <PUBLISHER>Penguin</PUBLISHER>
                <PUB-DATE>2003-02-25</PUB-DATE>
                <LANGUAGE>English</LANGUAGE>
                <PRICE>16.47</PRICE>
                <QUANTITY>129</QUANTITY>
                <ISBN>0142001805</ISBN>
                <PAGES>384</PAGES>
    <DIMENSIONS UNIT="in">7.8 5 0.9</DIMENSIONS>
                <WEIGHT UNIT="oz">9</WEIGHT>
        </ITEM>

</BOOKS>
<CATEGORIES DESC="Miscellaneous categories">
    <CATEGORY CODE="P" DESC="Paperback"/>
    <CATEGORY CODE="MMP" DESC="Mass-market Paperback"/>
    <CATEGORY CODE="H" DESC="Hard Cover"/>
</CATEGORIES>
</BOOKLIST>
```

### BOOKS JSON

The BOOKS JSON document

```
{
  "BOOKLIST":
   {
    "BOOKS":
     [
      {
       "BOOK":
        {
         "CAT":"MMP",
         "TITLE":"Pride and Prejudice",
         "AUTHOR":"Jane Austen",
         "PUBLISHER":"Modern Library",
         "PUB-DATE":"2002-12-31",
         "LANGUAGE":"English",
         "PRICE":4.95,
         "QUANTITY":187,
         "ISBN":"0679601686",
         "PAGES":"352",
         "DIMENSIONS":
          {
           "UNIT":"in",
           "value":
            [8.3,5.7,1.1]
          },
         "WEIGHT":
          {
           "UNIT":"oz",
           "amount":6.1
          }
        }
      },
      {
       "BOOK":
        {
         "CAT":"P",
         "TITLE":"Wuthering Heights",
         "AUTHOR":"Charlotte BrontÃ«",
         "PUBLISHER":"Penguin Classics",
         "PUB-DATE":"2002-12-31",
         "LANGUAGE":"English",
         "PRICE":6.58,
         "QUANTITY":113,
         "ISBN":"0141439556",
         "PAGES":"430",
         "DIMENSIONS":
          {
           "UNIT":"in",
           "value":
            [1,5.2,7.8]
          },
         "WEIGHT":
          {
           "UNIT":"oz",
           "amount":11.2
          }
        }
      },
      {
       "BOOK":
        {
         "CAT":"P",
         "TITLE":"Tess of the d'Urbervilles",
         "AUTHOR":"Thomas Hardy",
         "PUBLISHER":"Bantam Classics",
         "PUB-DATE":"1984-05-01",
         "LANGUAGE":"English",
         "PRICE":4.95,
         "QUANTITY":85,
         "ISBN":"0553211684",
         "PAGES":"480",
         "DIMENSIONS":
          {
           "UNIT":"in",
           "value":
            [6.8,4.2,0.8]
          },
         "WEIGHT":
          {
           "UNIT":"oz",
           "amount":7.7
          }
        }
      },
      {
       "BOOK":
        {
         "CAT":"P",
         "TITLE":"Jude the Obscure",
         "AUTHOR":"Thomas Hardy",
         "PUBLISHER":"Penguin Classics",
         "PUB-DATE":"1998-09-01",
         "LANGUAGE":"English",
         "PRICE":4.95,
         "QUANTITY":129,
         "ISBN":"0140435387",
         "PAGES":"528",
         "DIMENSIONS":
          {
           "UNIT":"in",
           "value":
            [7.8,5.2,0.9]
          },
         "WEIGHT":
          {
           "UNIT":"oz",
           "amount":10.9
          }
        }
      },
      {
       "BOOK":
        {
         "CAT":"H",
         "TITLE":"The Big Over Easy",
         "AUTHOR":"Jasper Fforde",
         "PUBLISHER":"Hodder & Stoughton",
         "PUB-DATE":"2005-07-11",
         "LANGUAGE":"English",
         "PRICE":16.47,
         "QUANTITY":129,
         "ISBN":"0340835672",
         "PAGES":"346",
         "DIMENSIONS":
          {
           "UNIT":"cm",
           "value":
            [22.5,18,3.5]
          },
         "WEIGHT":
          {
           "UNIT":"g",
           "amount":390
          }
        }
      },
      {
       "BOOK":
        {
         "CAT":"P",
         "TITLE":"The Eyre Affair",
         "AUTHOR":"Jasper Fforde",
         "PUBLISHER":"Penguin",
         "PUB-DATE":"2003-02-25",
         "LANGUAGE":"English",
         "PRICE":16.47,
         "QUANTITY":129,
         "ISBN":"0142001805",
         "PAGES":"384",
         "DIMENSIONS":
          {
           "UNIT":"in",
           "value":
            [7.8,5,0.9]
          },
         "WEIGHT":
          {
           "UNIT":"oz",
           "amount":9
          }
        }
      }],
    "CATEGORIES":
     {
      "_attributes":
       {
        "DESC":"Miscellaneous categories"
       },
      "_children":
       ["\n    ",
        {
         "CATEGORY":
          {
           "CODE":"P",
           "DESC":"Paperback"
          }
        },"\n    ",
        {
         "CATEGORY":
          {
           "CODE":"MMP",
           "DESC":"Mass-market Paperback"
          }
        },"\n    ",
        {
         "CATEGORY":
          {
           "CODE":"H",
           "DESC":"Hard Cover"
          }
        },"\n"]
     }
   }
 }
```

### *Example XSLT generation*

The markup for ITEM/DIMENSIONS in the above XSD produces these sets of XSLT patterns. Note that this code generation is in progress and is a prime target for change and optimization.

XSLT to convert XML to JSON

```
<xsl:template priority="1" match="ITEM/DIMENSIONS">
      <member name="DIMENSIONS">
         <xsl:choose>
         <!-- No attributes or child elements - jump to text  -->
           <xsl:when test="empty(@*|*)">
               <array>
                  <xsl:for-each select="tokenize(.,' ')">
                     <number>
                        <xsl:value-of select="."/>
                     </number>
                  </xsl:for-each>
               </array>
              </xsl:when>
              <!-- Otherwise need to make an object out of this -->
              <xsl:otherwise>
               <object>
                                        <!-- For each element and attribute make a member -->
                                  <xsl:for-each select="@*|*">
                     <xsl:apply-templates select="."/>
                  </xsl:for-each>
                  <!-- Wrap text in a _text node only for simple types -->
                                  <xsl:if test="string(.)">
                     <member name="value">
                        <array>
                           <xsl:for-each select="tokenize(.,' ')">
                              <number>
                                 <xsl:value-of select="."/>
                              </number>
                           </xsl:for-each>
                        </array>
                     </member>
                  </xsl:if>
               </object>
            </xsl:otherwise>
         </xsl:choose>
      </member>
   </xsl:template>
<xsl:template priority="1" match="ITEM/DIMENSIONS" mode="wrap">
      <object>
         <xsl:apply-templates select="."/>
      </object>
</xsl:template>
<xsl:template priority="1" match="ITEM/DIMENSIONS/text()" mode="#all">
      <array>
         <xsl:for-each select="tokenize(.,' ')">
            <number>
               <xsl:value-of select="."/>
            </number>
         </xsl:for-each>
      </array>
 </xsl:template>
<xsl:template priority="2" match="ITEM/DIMENSIONS/@UNIT" mode="#all">
      <member name="UNIT">
         <string>
            <xsl:value-of select="."/>
         </string>
      </member>
 </xsl:template>
```

XSLT to convert JSON to XML

```
<xsl:template priority="1"
                 match="member[@name='BOOK']/object/member[@name='DIMENSIONS']/object">
      <xsl:apply-templates select="*"/>
   </xsl:template>
   <xsl:template priority="1"
```

```
                 match="member[@name='BOOK']/object/member[@name='DIMENSIONS']/string | member[@name='BOOK']/object/member[@name='DIMENSIONS']/number">
      <xsl:value-of select="string()"/>
</xsl:template>
   <xsl:template priority="1" match="member[@name='BOOK']/object/member[@name='DIMENSIONS']">
      <xsl:element name="DIMENSIONS" namespace="">
         <xsl:apply-templates select="*"/>
      </xsl:element>
   </xsl:template>
   <xsl:template priority="1"
                 match="member[@name='BOOK']/object/member[@name='DIMENSIONS']/object/member[@name='value']">
      <xsl:copy-of select="string-join( array/(number|string) , ' ')"/>
 </xsl:template>

<xsl:template priority="2"
                 match="member[@name='BOOK']/object/member[@name='DIMENSIONS']/object/member[@name='UNIT']/string | member[@name='BOOK']/object/member[@nam
      <xsl:value-of select="string()"/>
</xsl:template>
<xsl:template priority="2"
                 match="member[@name='BOOK']/object/member[@name='DIMENSIONS']/object/member[@name='UNIT']">
      <xsl:attribute name="UNIT" namespace="">
         <xsl:apply-templates select="*"/>
      </xsl:attribute>
 </xsl:template>
```

## References

[JSONORG] The JSON to XML converter from json.org http://www.json.org/javadoc/org/json/XML.html

[jsonx] JSONx is an IBM® standard format to represent JSON as XML http://publib.boulder.ibm.com/infocenter/wsdatap/v3r8m1/index.jsp?topic=/xs40/convertingbetweenjsonandjsonx05.htm

[badgerfish] The Badgerfish notation for XML in JSON. The normative site http://badgerfish.ning.com/ has vanished off the web but many references still remain including XSLTJSON

[rabbitfish] The Rabbitfish notation for XML in JSON. References are made to rabbitfish notation in XSLTJSON

[XSLTJSON] XSLTJSON, XML to JSON using XSLT http://www.bramstein.com/projects/xsltjson/

[JSONML] JSON Markup Language (JsonML)http://jsonml.org/

[JQUERY] jQuery XML to JSON Plugin http://www.fyneworks.com/jquery/xml-to-json/

[BOOMERANG] Boomerang - A bidirectional programming language for ad-hoc, textual data. http://www.seas.upenn.edu/~harmony/

[XSUGAR] XSugar - Dual Syntax for XML Languages http://www.brics.dk/xsugar/

[JSONSCHEMA] JSON Schema http://json-schema.org//

[DFDL] OGF Standards: Data Format Description Language (DFDL) http://www.ogf.org/dfdl/

[APACHESCHEMA] Apace Schema API implementation of the XML Schema API http://www.w3.org/Submission/2004/SUBM-xmlschema-api-20040309/

[TRANG] Trang - Multi-format schema converter based on RELAX NG http://www.thaiopensource.com/relaxng/trang.html

[MLJSON] MLJSON provides a facade on top of MarkLogic for treating MarkLogic as a store for JSON documents and data. http://developer.marklogic.com/code/mljson

[1] JSON JavaScript Object Notation http://www.json.org

[2] XML - Extensible Markup Language (XML) http://www.w3.org/XML/

[3] XSLT - XSL Transformations (XSLT) Version 2.0 http://www.w3.org/TR/xslt20/

[4] JAVA http://www.java.com

[5] Relax NG - http://relaxng.org

[6] http://developer.yahoo.com/yui/theater/video.php?v=crockford-json

The main reason I took comments out was that I saw people who were trying to control what the parser would do based on what was in the comments, and that totally broke interoperability. There's no way I could control the way they were using comments, so the most effective fix was to take the comments out.

[7] XQuery 1.0: An XML Query Language http://www.w3.org/TR/xquery/

[8] XProc: An XML Pipeline Language http://www.w3.org/TR/xproc/

[9] Schematron http://www.schematron.com/

[10] "Clark Notation" is an informal name given to the proposal by James Clark for a simple represtntation of QNames http://www.jclark.com/xml/xmlns.htm

[11] http://www.freebsd.org/copyright/freebsd-license.html "Simplified BSD License" also known as the "2 clause BSD License" and the "FreeBSD License"

[12] SAXON - The XSLT and XQuery Processorhttp://saxon.sourceforge.net/