PESC Compliant JSON

Version 1.0.0

02/04/2019

A publication of the
Technical Advisory Board
Postsecondary Electronic Standards Council (PESC)

Executive Summary

Business Problem

Currently, PESC provides standards for the electronic exchange of Transcripts, Applications for Admission, Electronic Portfolios, Test Scores, Common Credentials, and other standards. These standards provide exchange partners with a basis for creating exchange software and interpreting the data sent to them. These standards currently support Electronic Data Exchange (EDI) and eXtensible Markup Language (XML) formats. With the recent increased use of Javascript Object Notation (JSON) as an exchange medium for web services and other data exchanges, users of PESC standards have expressed the desire to use JSON as an exchange medium.

Solution

PESC has embarked on a phased approach to provide PESC Compliant exchanges of PESC standards using JSON. The first phase is to provide rules for interpreting XML schema standards in the generation and parsing of JSON. This might appear to be a manual process. However, the EdExchange project, using Java technology, has demonstrated that the XML schema can be used to automate the creation of programming language objects. In turn, these enforce the constraints of the schema on the generation of JSON, as well as determine the validity of an incoming JSON instance. This document provides detailed rules and examples that will assist the PESC community in generating and consuming PESC compliant JSON. Our experience with EdExchange is that most of these rules are implicit in tools such as JAXB, and those that are not implemented by default can be implemented via configuration options.

The second phase of this project will explore the application of PESC standards through JSON schema language, JSON-LD, and/or OpenAPI specifications. In addition, PESC will continue to search for the holy grail of a modeling language that will act as Chomsky's "deep structure" [3] for standards. This would allow one specification to encapsulate the constraints on any type of serialization and provide for the translation between them. This second phase is not in the scope of this document.

Table of Contents

1 Introduction

1.1 Overview

This document describes a set of rules for the creation of JSON exchanges that must be followed if an exchange is to be considered PESC compliant. PESC uses XML Schema Language to specify the data model for its various standards (e.g., High School Transcript). The rules in this document instruct the implementer how to interpret the XML Schema as a data model for JSON exchanges. In addition, the document summarizes guidelines for simplifying XML Schema to promote consistency between XML and JSON serializations.

1.2 Purpose

The purpose of this document is to establish JSON as a viable format for PESC data exchanges without sacrificing standardization. There have been many attempts to define translation rules between XML and JSON. Most of these rely on direct syntactical transformations without reference to an underlying data model, thus resulting in difficult interpretations and excessive type checking on the part of the receiving application. For example, a repeatable element in an XML schema that exists as a single element in an XML instance document would be rendered in JSON as a name-value pair (e.g., {"A": 3}, but if the element was repeated, it would be serialized as a JSON array (e.g., {"A": [3, 4]}). The receiving application must do type checking and process the two cases differently. In the data-model-aware situation, the type would always be an array and type checking would not be needed.

This document is the first step in establishing JSON as a standard of data exchange for PESC. The next step is exploring alternative expression of data and validation models that would complement or replace XML Schema Language. Some of the alternatives that PESC will explore include JSON Schema, JSON-LD, and the Content Assembly Mechanism (CAM).

1.3 Scope

This document applies to the exchange of JSON-formatted content for any PESC standard.

1.4 Intended Audience

The audience for this document consists of managers and programmers wishing to exchange JSON content compliant with the PESC data model for any of its standards.

1.5 Assumptions

The reader should have knowledge of XML, XML Schema, and JSON. For a review of these topics, the following sites have easy to read tutorials:

- XML: https://www.w3schools.com/xml/
- XML Schema: https://www.w3schools.com/xml/schema_intro.asp
- JSON: http://www.json.org/

2 XML Schema Simplification

In order to align with technology trends in information exchange while still supporting PESC's existing standards, the standards for both XML and JSON should promote expression of comparable semantics, simplicity of translation, and ease of implementation. Thus, the education community can have a choice of exchange formats without sacrificing interoperability. To accomplish this objective, the following requirements should be followed when creating new XML schemas for PESC standards:

- Do not define mixed elements with complex content.
- Limit the use of attributes.
- Use a single namespace if possible so that name conflicts will not occur in JSON.
- Do not define global elements in XML schemas as this will require namespace qualification of elements in instance documents.
- An element name should not be used twice in a sequence. However, an element can be repeatable.

3 JSON Generation and Translation Rules

3.1 Requirements for Rules

- JSON exchange data shall comply with RFC8259, "The JavaScript Object Notation (JSON) Data Interchange Format" [2].
- The name "value" will be used to designate XML element values and thus may conflict with attributes of the same name. The name conflict rule below shall be used to resolve this conflict. JSON exchanges shall follow the data models expressed in XML schemas and interpreted by the rules below.
- Although there are no specific required reserved words, the intent is to allow implementers to utilize JSON-LD; thus no "@" sign plus key words should be present unless following JSON-LD syntax. The link to these key words can be found at https://json-ld.org/spec/FCGS/json-ld/20180607/#syntax-tokens-and-keywords.
- Any information needed for translating from JSON back to XML shall not be contained in the JSON itself. For example, "@" or "_" will not appear before an attribute name to denote that name was associated with an XML attribute. This will allow programmers to view PESC JSON as they would any application natively using JSON. However, translation tools may use metadata gathered by the tool from the XML schema for translating JSON back to XML. For example, the JAXB framework keeps information about attributes as Java annotatons. This allows JAXB to use these annotations for creating attributes in XML from JSON, just by matching the property name.
- If a name appears in JSON, the value should always be of the same type (string, number, boolean, object, or array), or the value may be null under defined circumstances (see 3.3.8 Nillable Elements).
- If an element is optional, it may be omitted from the JSON. This may require existence checking by a receiving application.

3.2 General Approach

The basic strategy is that XML elements are generally represented as a name-value pair. The XML tag name becomes the JSON property name. The value, whether simple or complex, becomes the JSON value. When XML attributes are present, even for simple content, the JSON value part will always be an object. Attributes are translated as a property representing simple name-value pairs.When an XML object contains attributes and simple content, a property named 'value' will have the value of the tag in the XML.

Examples:
<TAGNAME>TAGVALUE</TAGNAME> becomes
{"TAGNAME": "TAGVALUE"}

<TAGNAME someAttr="atrValue">TAGVALUE</TAGNAME> becomes
{"TAGNAME": {"value": "TAGVALUE", "someAttr": "attrValue"}}

To meet our goal of having JSON take the form of typical/customary JSON, some transformations may not work as someone coming from a pure XML world might expect. Consider numbers. If one sends a float of 1.00 and is processed somewhere in the middle as a JSON number, what is received should be 1. Now in the vast number of use cases this is not a problem. From a Computer Science perspective , 1.00 equals 1 as both should be parsed into a numeric type before being compared. However, if you are conveying a science question where significant digits matter, a directional heading where leading zeros should be maintained, or a similar case, you should ensure your data schema uses a string based type and not a numeric one. The astute reader will realize that this problem is nothing new; however, if one chooses to convert XML into JSON, it is much more likely to occur.

Similarly, JSON has no requirement to maintain the order of elements while XML does. This means that if you convert JSON into XML you either need to produce elements in the expected order AND use JSON tools that maintain that order, or correct the order for the XML representation. In order to accomplish this reordering the proper location of every element must be known. As you might imagine this tends to be a resource intensive process. As such, any JSON enabled software should clearly state whether it maintains document order of elements or not.

3.3 Rules

As with XML, it is understood that exchange partners may decide that certain rules, as specified below, do not fit their business models or tools. The JSON produced by violating these rules would not be considered PESC-compliant and may not work in an exchange expecting such compliance. However, PESC still encourages that exchanges use standards as guidelines, even if not compliant. PESC would also appreciate feedback as to the reasons for the deviations so that standards may be improved.

The examples below assume element A, which is part of a complex type, is being defined by a type definition.

```
<xs:complexType name="top">
   <xs:sequence>
```

```
        <xs:element name="A" type="AType" minOccurs="0" nillable="true"/>
    </xs:sequence>
</xs:complexType>
```

3.3.1 Name Collisions

There may be rare cases where a schema element defines both an attribute and a child element with the same name or an attribute on a simple content element with the name "value", which is reserved for specific purposes. This will cause a name conflict, which is not allowed in JSON objects. To resolve this conflict, the property name ih JSON should be preceded by an underscore (i.e., "_").

3.3.2 Optional Values, Arrays, or Objects

If the following rules would result in empty JSON values ("", [], {}), the name-value pair for that element may be omitted from the JSON if the element in XML is not required.

3.3.3 Complex Content with Attribute

Attributes on a complex element with complex content will be treated as another name-value pair in the object's properties.

Schema:

```
<xs:complexType name="top">
    <xs:sequence>
        <xs:element name="A" type="AType" minOccurs="0" nillable="true"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="AType">
    <xs:sequence>
        <xs:element name="B" type="xs:string" minOccurs="0">
    </xs:sequence>
    <xs:attribute name="attr" type="xs:string" use="optional"/>
</xs:complexType>
```

Translation:

<A attr="text"><B>text2</B></A> becomes "A":{"attr": "text", "B": "text2"}
<A attr="text"></A> becomes "A":{"attr": "text"}
<A><B>text2</B></A> becomes "A":{"B": "text2"}
<A/> becomes "A": {} or A is omitted

Generation:

attr="text" and B="text2" becomes "A":{"attr": "text", "B": "text2"}
attr="text" and B=no data becomes "A":{"attr": "text"}
attr=no data and B=no data becomes "A":{} or A is omitted.

3.3.4 Simple Content with Attribute

Simple content with an attribute will be converted into a JSON object named for the simple element. If the attribute is optional according to the schema, the attribute will be generated only if it has a value. However, even if the attribute is not present, the JSON serialization will always be an object with a "value" property.

Schema:

```
<xs:complexType name="top">
   <xs:sequence>
      <xs:element name="A" type="AType" minOccurs="0" nillable="true"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="AType">
   <xs:simpleContent>
      <xs:extension base="xs:string">
         <xs:attribute name="attr" use="optional"/>
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>
```

Translation:

<A attr="text">text2</A> becomes "A":{"attr":"text","value":"text2"}
<A>text2</A> becomes "A": {"value": "text2"}
<A/> becomes "A": {"value": ""} (since an empty tag is meaningful)

Generation:

A="text" and attr=no data→"A": {"value": "text"}
A="text" and attr="text2"→"A": {attr="text2", "value": "text"}
A=empty string and attr=no data→"A": {"value": ""} if A is an optional child
A=no data and attr=no data→omit A

3.3.5 XML Types to JSON Types

The schema type determines the type of a JSON value.

xs:string, xs:token, etc.
<A>3.3</A> becomes "A": "3.3"

xs:integer, xs:decimal, etc.
<A>3.3</A> becomes "A": 3.3

xs:boolean
<A>true</A> becomes "A": true

xs:date, xs:time, xs:dateTime to JSON String using ISO 8601 string format
<A/>1990-09-02T03:03:00-0500</A> becomes "A":"1990-09-02T03:03:00-0500"

3.3.6 Repeatable Element

The values of a repeatable element are translated to a JSON array even if the element only has one instance.

Schema:

```
<xs:complexType name="top">
   <xs:sequence>
      <xs:element name="A" type="AType" minOccurs="0" nillable="true"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="AType">
   <xs:sequence>
      <xs:element name="B" type="xs:string" minOccurs="0" maxOccurs="unbounded">
      <xs:element name="C" type="xs:string" minOccurs="0">
   </xs:sequence>
</xs:complexType>
```

Translation:

```
<A>
   <B>text1</B>
   <B>text2</B>
   <C>text3</C>
</A>
```

becomes

```
"A":{"B": ["text1", "text2"], "C": "text3"}
```

```
<A>
   <B>text1</B>
   <C>text3</C>
</A>
```

becomes

```
"A":{"B": ["text1"], "C": "text3"}
```

Generation:

B="text1" only and C="text3" becomes "A":{"B": ["text1"], "C": "text3"}
B=no data and C="text3" becomes "A":{"C": "text3"}
B=no data and C=no data becomes "A":{} or omitted

3.3.7 XML List Type

If the schema specifies a list then the space separated list is specified as an array.

Schema:

```
<xs:element name="A" type="AType" minOccurs="0" nillable="true"/>
<xs:simpleType name="AType">
   <xs:list itemType="xs:integer"/>
</xs:simpleType>
```

Translation:
<A>1 2 3</A> becomes "A": [1, 2, 3]

Generation:
A= a list of "C", "CD", and "E" becomes ["C", "CD", "E"]
A= no data becomes "A": [] or omitted

3.3.8 Nillable Elements

Elements defined with the xs:nillable="true" (by default xs:nillable is false) may carry the xsi:nil attribute in the instance documents. These elements will be assigned the value of null in JSON. The xsi:nil will not be treated as an attribute for translation puposes.

Schema:
<xs:element name="A" type="AType" minOccurs="0" nillable="true"/>
<xs:simpleType name="AType" type="xs:integer"/>

Translation:
<A xsi:nil="true"/> becomes "A":null
<A xsi:nil="false"/> or <A/> is not valid XML for this integer simple type so it cannot appear in valid XML. Translation will not be necessary.

Generation:
A=no data is omitted
A=null value to be transmitted becomes "A": null

3.3.9 Required Empty Simple Element

If an element is required (minOccurs > 0) and the element is not nillable or xsi:nil is false, the empty tag (e.g., <A/> or <A></A>) will be translated into the empty string if the empty string is allowed by the type definition (e.g., xs:string with minLength="0"). If the XML instance document being translated is valid, the empty tag cannot occur for any type that does not include the empty string, and thus there will be no need for translation.

Translation:
<A/> becomes "A": "" if a string with minLength="0"
<A/> becomes "A": [""] if repeatable and a string with minLength="0"
<A/> cannot exist in a valid XML instance document if its type does not include the empty string

3.3.10 Required Empty Complex Content Element

A complex element with excluded children that must be present (i.e., minOccurs > 0) shall be represented as an empty object in JSON ("A": {}).

3.3.11 Sequence and Choice

XML schemas can specify that child elements be presented in a particular order through the xs:sequence and xs:choice constructs. JSON objects do not have an explicit order to their properties. Indeed, some JSON tools will alphabetize the property names for display. As a result, the order of JSON properties are not required to be in the same order as specified in the XML Schema xs:choice or xs:sequence. If translation from JSON to XML is required, the XML Schema may be used to reorder the property names for an XML instance document.

3.3.12 Union Types

The xs:union schema element allows for the defined element to be one of several types. For translation, this requires that the value be interpreted by determining the most specific constraint of the XML element value. For example, an integer is more constrained than a string. Processing of the union type requires type checking when parsing the JSON string so it should be discouraged in XML schemas.

Schema:

```
<xs:complexType name="top">
   <xs:sequence>
      <xs:element name="A" type="AType" minOccurs="0" nillable="true"/>
   </xs:sequence>
</xs:complexType>

<xs:simpleType name="AType">
  <xs:union memberTypes="xs:string xs:integer" />
 </xs:simpleType>
```

Translation:
<A>3</A> becomes "A": 3
<A>450-3</A> becomes "A": "450-3"

Generation:
A=number 34 becomes "A": 34
A=string String becomes "A": "String"
If the generator wants a number interpreted as a string, then the following translation could be created:A=string 345 becomes "A": "345"

3.3.13 Facets

Facets in XML schemas are used to further constrain the value of a simple type. These constraints should be used in generating JSON content. For example, if the maxLength in the schema for an element is 80, the value for that corresponding JSON property should not be greater than 80 characters.

String Facets:
xs:length, xs:minLength, xs:maxLength, xs:enumeration, xs:pattern, xs:whitespace

Number Facets:
xs:totalDigits, xs:fractionDigits, xs:minInclusive, xs:maxInclusive

Schema:

```
<xs:complexType name="top">
   <xs:sequence>
      <xs:element name="A" type="AType" minOccurs="0" nillable="true"/>
   </xs:sequence>
</xs:complexType>

<xs:simpleType name="Atype">
   <xs:restriction base="xs:decimal">
      <xs:totalDigits value="9"/>
      <xs:fractionDigits value="3"/>
   <xs:restriction>
</xs:simpleType>
```

Valid: {"A": 3.45}, {"A": 123456.123}
Invalid:
{"A": 0.12345}, {"A": 123456789.123}, {"A": "Three point five"}

Note: fractionDigits is the maximum number of digits to the right of the decimal, not the required number of digits.

3.3.14 Namespaces

Namespace definitions will be treated like any other attribute and added as properties to the JSON object. Namespace prefixes in XML will be part of the name used for JSON properties. Namespace definitions with prefixes not used in the XML instance document may be excluded from the JSON instance.

3.3.15 Schema Information

Attributes related to XML Schemas (e.g., xmlns:xsi namespace and xsi:SchemaLocation) may be excluded from the JSON instance.

3.3.16 Root Element

The XML root element name shall be included as a property of the top level JSON object.

3.3.17 XPath Expressions

Some PESC standards use XPath expressions to identify a particular element in an XML instance document. While there could be an interpretation of XPath for JSON, JSON tools are using other expressions to identify elements in a more straight-forward manner. JSONPath appears to be implemented in most programming languages. Since XPath expressions appear to be just strings in XML, it may require schema-specific code to identify and translate XPath to JSONPath.

/AcademicEPortfolio/Competencies[CompetencyID="Competency1"] becomes
$.AcademicEPortfolio.Competencies[?(@.CompencyID == "Comptency1")] or
$["AcademicEPortfolio"]['Competencies'][?(@.CompetencyID =="Comptency1")]

The JSONPath specification is found here:
http://goessner.net/articles/JsonPath/

This translation table was extracted from Goessners's JSONPath specification (above):

XPath	JSONPath	Description
/	$	the root object/element
.	@	the current object/element
/	. or []	child operator
..	n/a	parent operator
//	..	recursive descent. JSONPath borrows this syntax from E4X.
*	*	wildcard. All objects/elements regardless their names.
@	n/a	attribute access. JSON structures don't have attributes.
[]	[]	subscript operator. XPath uses it to iterate over element collections and for predicates. In Javascript and JSON it is the native array operator.
|	[,]	Union operator in XPath results in a combination of node sets. JSONPath allows alternate names or array indices as a set.
n/a	[start:end:step]	array slice operator borrowed from ES4.
[]	?()	applies a filter (script) expression.
n/a	()	script expression, using the underlying script engine.
()	n/a	grouping in Xpath

3.3.18 XML Features Not Translated

XML has several notations that do not have a corresponding construct in JSON. Therefore, to meet the "no special names" requirement, the following XML notations will not be translated from XML to JSON.

- Processing instructions
- Comments
- xsi attributes: xsi:lang, xsi:type, xsi:schemaLocation

4 Tools Support

To assist with the creation of data model aware JSON, various software tools may be used to encode the XML schema rules into language objects that can then be serialized into JSON, XML, or other language. Our experience with these tools indicates they may not enforce all constraints in their objects and that some additional code or post processing may be needed to meet this specification.

4.1 Java JAXB

Currently, a combination of JAXB (Java object model creation from XML Schema) and MoXY (JSON serialization) have been successfully used to create data-model-aware JSON. The PESC EdExchange program uses this tool to create JSON for transcript exchanges.

4.2 Python

The xmlschema package for Python has been used to translate between XML instance documents and JSON using the XML schema to drive the translation. This solution appears to implement most of the rules above. It has the advantage that XML is translated into Python dictionaries where additional transformations can be applied before converting to JSON. Unfortunately, some XML Schema Language constructs such as xs:union are not supported.

4.3 A4L Tool Set

The Access 4 Learning Community is excited enough about this JSON representation that they have already developed a set of reference tools for it. These tools start with their internal schema representation and produce both a flat file with all the needed information to properly serialize JSON from one of their standards plus code and transforms capable of doing the work. While these serve as great examples between starting with an internal format and the performance concerns of the resulting code, the expectation is that people will create native solutions using the flat file to process things correctly. This exercise can be found at GitHub here: https://github.com/nsip/sifxml2pescjson.

4.4 CAMV Editor

The CAMV editor uses OASIS-defined templates to provide a data model from XML Schema (and other sources) that can be used to translate between various data exchange respresentions including XML and JSON. This software is freely available as an open source project on SourceForge. The JSON Task Force plans to examine this software for compatibility with this PESC standard.

Appendix A: Revision History

DATE	SECTION/ PAGE	DESCRIPTION	Version	MADE BY
02/04/2019	Whole Document	JSON Task Force approved document	1.0.0	Michael Morris Steve Margenau Doug Holmes Jerald Bracken Alex Jackl

Appendix B: References

[1] PSEC, PESC Standards Forum for Education, PESC Policies and Procedures, 2015
[2] IETF, RFC8259, The JavaScript Object Notation (JSON) Data Interchange Format, 2018
[3] Chomsky, Noam, Current Issues in Linguistic Theory, 1964