"Making a difference by processing JSON as XML"

Robin La Fontaine
DeltaXML

DELTAXML

Today's story is about…

Viewing JSON as XML in order to access XML comparison

Using transformation to make life easier

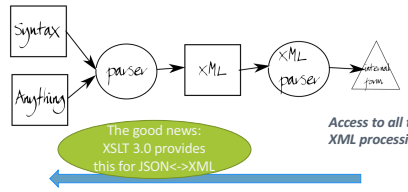
Representing changes in a way that makes sense

And a challenge for XMLT 4.0 …

DELTAXML

View all structured information as XML

This graphic borrowed from Steven Pemberton

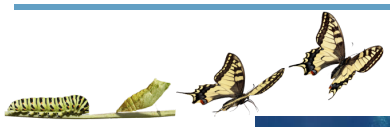


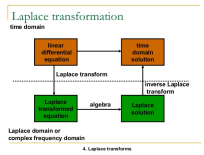
Access to all the XML processing stack

Transform back again, without loss

DELTAXML

If life is difficult, transform it





DELTAXML

JSON for XMLers

Structure
- Name/value pairs (a bit like attributes but value can be a structure)
- Text but also numbers, boolean and null
- { Objects } have name : value pairs
- [Arrays] have one or more values

Text
- Unicode
- Escape used for " \ / backspace form-feed line-feed carriage-return tab
- No pretty-printing for text, only for structure

DELTAXML

JSON for XMLers

Good stuff
- Simpler
- No difficult whitespace processing
- No mixed content!!
- Clear distinction between name/value pairs and arrays

Bad (well, not so good) stuff
- No attributes (they do come in useful)
- No structure within blocks of text (XML/HTML gets embedded)
- Parsing ambiguities

DELTAXML

Transforming JSON into XML

This is what you get with standard XSLT 3.0

```
{"Image":
  { "Width": 800,
  "Height": 600,
  "Title": "View from 15th Floor",

  "Thumbnail":
     { "Url":
     "http://www.example.com/image/481989943",

     "Height": 125,
     "Width": 100 },
  "Animated": false,
  "IDs":
     [ 116,
       943,
       234,
       38793 ] }}
```

```
<j:map
xmlns:j="http://www.w3.org/2013/XSL/json">
<j:map key="Image">
 <j:number key="Width">800</j:number>
 <j:number key="Height">600</j:number>
 <j:string key="Title">View from 15th
Floor</j:string>
 <j:map key="Thumbnail">
   <j:string
key="Url">http://www.example.com/image/481989
943</j:string>
   <j:number key="Height">125</j:number>
   <j:number key="Width">100</j:number></j:map>
<j:boolean key="Animated">false</j:boolean>
<j:array key="IDs">
   <j:number>116</j:number>
   <j:number>943</j:number>
   <j:number>234</j:number>
   <j:number>38793</j:number> </j:array> </j:map>
</j:map>
```

DELTAXML

Default XML is not ideal

"Width": 800,

Consider a typical change to: "Width":
{ "value": 800,
"units": "metre" },

When we look at the XML representation:

<j:number key="Width">800</j:number>

Does not align well with:

```
<j:map key="Width">
  <j:number key="value">800</j:number>
  <j:string key="unit">"metre"</j:string>
</j:map>
```

DELTAXML

Transforming XML into 'better' XML

<j:number key="Width">800</j:number>

Does not align well with:

```
<j:map key="Width">
  <j:number key="value">800</j:number>
  <j:string key="unit">"metre"</j:string>
</j:map>
```

```
<member key="Width">
   <number>800</number>
</member>
```

Aligns better with:

```
<member key="Width">
  <object>
    <member key="value">
       <number>800</number></member>
    <member key="unit">
       <string>metre</string></member>
  </object>
</member>
```

DELTAXML

Representing changes – patch

A	B
"hobbies": ["playing guitar badly", "reading", "Cinema"]	"hobbies": ["Badminton", "guitar", "reading"]

JSON standard patch:

["playing guitar badly", "reading", "Cinema"]

```
{"op": "remove",
 "path": "/hobbies/2" },
```
→ ["playing guitar badly", "reading"]

```
{ "op": "add",
  "path": "/hobbies/1",
  "value": "guitar" },
```
→ ["playing guitar badly", "guitar", " reading "]

```
{ "op": "replace",
  "path": "/hobbies/0",
  "value": "Badminton" }
```
→ ["Badminton", "reading", "Cinema"]

DELTAXML

Representing changes – patch

A	B
"hobbies": ["playing guitar badly", "reading", "Cinema"]	"hobbies": ["Badminton", "guitar", "reading"]

DECLARATIVE delta:

```
"hobbies": [
 {"dx_delta":
   { "A": "playing guitar badly",
     "B": "Badminton" }},
 {"dx_delta": {"B": "guitar"}},
 "reading",
 {"dx_delta": {"A": "Cinema"}} ]
```

DELTAXML

Representing changes – patch

A	B
"hobbies": ["playing guitar badly", "reading", "Cinema"]	"hobbies": ["Badminton", "guitar", "reading"]

DECLARATIVE delta:

Highlight A

```
"hobbies": [
 {"dx_delta":
   { "A": "playing guitar badly",
     "B": "Badminton" }},
 {"dx_delta": {"B": "guitar"}},
 "reading",
 {"dx_delta": {"A": "Cinema"}} ]
```

DELTAXML

Declarative delta provides new opportunities

A	B
"hobbies": ["playing guitar badly", "reading", "Cinema"]	"hobbies": ["Badminton", "guitar", "reading"]

DECLARATIVE Full delta:

```
"hobbies": [
 {"dx_delta":
   { "A": "playing guitar badly",
     "B": "Badminton" }},
 {"dx_delta": {"B": "guitar"}},
 "reading",
 {"dx_delta": {"A": "Cinema"}} ]
```

DECLARATIVE changes-only delta:

```
"hobbies": [
 {"dx_delta":
   { "A": "playing guitar badly",
     "B": "Badminton" }},
 {"dx_delta": {"B": "guitar"}},
 null,
 {"dx_delta": {"A": "Cinema"}} ]
```

This is "reading" which is unchanged

Full context – all data

Changes only – omit unchanged data

DELTAXML

Representing changes – patch

A	B
"hobbies": ["playing guitar badly", "reading", "Cinema"]	"hobbies": ["Badminton", "guitar", "reading"]

OPERATIONAL patch:

```
{"op": "remove",
 "path": "/hobbies/2" },
{ "op": "add",
  "path": "/hobbies/1",
  "value": "guitar" },
{ "op": "replace",
  "path": "/hobbies/0",
  "value": "Badminton" }
```

Unidirectional

DECLARATIVE delta:

```
"hobbies": [
 {"dx_delta":
   { "A": "playing guitar badly",
     "B": "Badminton" }},
 {"dx_delta": {"B": "guitar"}},
 null,
 {"dx_delta": {"A": "Cinema"}} ]
```

Bidirectional/reversible

DELTAXML

Taking patch one step further – 'smart patch'

	OPERATIONAL Patch	DECLARATIVE Changes-only delta	DECLARATIVE Full context delta
'A' file to 'B' file	Yes	Yes	Both included
'B' file to 'A' file	No	Yes	Both included
What about 3 or more files?	No	Yes	Yes
Apply changes to variant of 'A' file (Target for changes)	No	No	Yes – so a 'smart patch' (is there a better name for this??)

DELTAXML

Confidence in the 'smart patch' change

'Apply the changes between A and B to Target'

(this is a simplified table, does not take account of non-leaf nodes)

Relationship between A and Target	Smart patch: Add (in B not A)	Smart patch: Delete (in A not B)	Smart patch: Change (in B and A)
In Target only (no changes can happen)	Good	n/a	n/a
Not in Target (no changes can be applied because it is not in target)	Good	Good	Good
Target equal to A	Good	Good	Good
Target not equal to A	Good	OK – may be an issue	Fair – could be a conflict

DELTAXML

Representing text changes in JSON

A	B
{"description": "This is a good example of Word by Word processing"}	{"description": "This is a great example of Word by Word processing"}

A simple representation using a string:

```
{"description": {"dx_delta": {
  "A": "This is a good example of Word by Word processing",
  "B": "This is a great example of Word by Word processing" }}}
```

A word-by-word representation using an array of strings:

```
{"description": {"dx_delta_string":
  [ "This is a ",
   {"dx_delta": { "A": "good", "B": "great" }},
   " example of Word by Word processing" ]}}
```

DELTAXML

Representing text changes in XML

A	B
This is a good example of Word by Word processing	This is a great example of Word by Word processing

A simple representation using a string:

```
<p dx:deltaV2="A!=B"><dx:textGroup dx:deltaV2="A!=B">
  <dx:text dx:deltaV2="A">This is a good example of Word by Word processing</dx:text>
  <dx:text dx:deltaV2="B">This is a great example of Word by Word processing</dx:text>
 </dx:textGroup>
</p>
```

A word-by-word representation using an array of strings:

```
<p dx:deltaV2="A!=B">This is a
 <dx:textGroup dx:deltaV2="A!=B">
   <dx:text dx:deltaV2="A">good</dx:text>
   <dx:text dx:deltaV2="B">great</dx:text>
</dx:textGroup>
example of Word by Word processing</p>
```

DELTAXML

Lessons on change representation

Similar representation(s) possible in JSON and XML

OPERATIONAL patch has limited applications

DECLARATIVE delta has more applications:

- Bi-directional for two files
- Changes only or full context
- Can represent changes between more than 2 files
- Can be used to propagate changes to multiple variant Targets

DELTAXML

Lessons on transformation

Transformation much easier in XML than JSON

- No equivalent of XPath or XSLT in JSON

XSLT 3.0 provides good bi-directional transformation JSON<->XML

If the XML structure is not easy to process, transform it!

- Not always easy to design loss-less bi-directional transformations
- Not always trivial to write reverse transformations

How about reversible transformations in XSLT 4.0?

DELTAXML

Reversible transformation

1. Specify two XML patterns – source and target
2. Specify an XPath to specify where to trigger the transformation
3. XSLT compiles this and guarantees reversibility
   It would need to generate the XPath for the reverse trigger point
4. Reversibility means:
   For any general XML source having one or more trigger points, the transformed source can be reverse-transformed back to XML that is equal to the source

DELTAXML

And coming back to planet earth again…

JSON users can enjoy the benefits of XML

XML users need no longer fear JSON
Just turn it into XML using XSLT 3.0

Difficult XSLT code can be turned into easy code

Using loss-less bi-directional transformation

The End
Thank you!

DELTAXML