Read This First – Addendum 25th September 2012

Andy Heath 25th September 2012.

Several different approaches to a json binding have been explored in the construction of this binding. These notes describe the approaches that were explored and present a rationale for the approach chosen and show the structure of instances that conform to this binding.. For this draft specification the definitive json binding is version 4.0 as provided – the earlier versions that were explored are not provided but the notes for them are provided here to explain the rationale for the definitive version.

For each of pnp and drd three json schemas are provided

- Core – only core properties and vocabulary values
- Full-strict – full model properties and vocabulary values with no extensions permitted
- Full-relaxed – full model properties and vocabulary values that permits extensions in the form documented

I general a matching pair of schemas should be used, though instances that satisfy a Core schema will also satisfy the corresponding Full-strict and instances that satisfy a Full-strict will also satisfy the corresponding full-relaxed.

Note that current javascript libraries do not enforce uniqueness of fields within an object though the result of allowing such in an instance may not be well-defined. The json schema provided provided with this binding does not enforce that unqueness either. A validator that does is under construction.

A validator that might be used to determine whether an instance is valid to this binding is provided for convenience at

http://afa30.axelafa.com/

Examples of valid and non-valid instances are given in the accompanying examples.

Notes on approaches to JSON bindings for Access for All 3.0

Andy Heath, Mon 13 Aug 2012 22:43 BST

A number of technical issues arise and experimental “bindings” of IMS AfA 3.0 have been constructed to explore these.

First there is the question of what validation technologies to use for instances and what to validate them to. Two technologies are of particular value here:

1. There have been several drafts of an IETF JSON Schema “specification” all of which are out of date. The latest is Draft 3 http://tools.ietf.org/html/draft-zyp-json-schema-03 which expired in May

2011 but is widely used and several libraries implement validation to (parts of) this specification to different degrees of completeness. There are some areas where technical interpretation of this draft differs between different tools, one notably being the support for Hyperschema and the precise semantics of external schema extension. Also, in the light of changes between the different drafts much documentation out there is less than clear and its difficult to find a single authoratitive clearly-written source on interpretation of matters in the spec. Nevertheless IETF Draft 3 seems to be the most widely-implemented JSON specification. It is not without technical issues and problems. There is some discussion on the internet of a Draft 4 but no obvious development work on this presents itself.

2. LD-JSON is a specification initiative with the aim of supporting Linked Data. Its main addition to JSON is lightweight mechanisms for identifying nodes and contexts that taken together provide similar mechanisms to that of namespacing in XML. In addition the specification is providing additional programming language support, for example with markup to specify sets and lists. Its stated aim is almost backwards compatibility with traditional json. Features extra to traditional JSON are specified with additional largely unobtrusive markup. However, the specification also introduces some important changes to the semantics of JSON structures (objects and arrays). These changes have impact for the way the AfA JSON bindings are modelled, as discussed below. LD-JSON specifications are currently in draft in W3C, useful resources being http://json-ld.org/spec/latest/json-ld-syntax/ and http://json-ld.org/

Neither IETF Draft 3 nor LD-JSON alone are without technical challenges for binding JSON representing AfA 3.0 instances. These are discussed in the light of the bindings constructed below. They are surmountable but there are areas where there is no immediately obvious answer as to the best technical direction to go. All of the bindings constructed are “conformant” to IETF draft 3 and provide a Schema for validation of instances to that draft.

Bindings

This is discussed in terms of pnp instances but the arguments apply equally well to drd instances.

jsonSchemas3.3

This is modelled to have equivalent structure, names and values as the XML binding. An example pnp instance might be:

```
{ "pnp":
[
    { "accessModeRequired" : {
      "existingAccessMode" : "visual",
      "adaptationRequest" : "auditory"
                            }
    },
    { "accessModeRequired" : {
      "existingAccessMode" : "visual",
      "adaptationRequest" : "textual"
                            }
    },
    { "hazardAvoidance" : "flashing" },
    { "hazardAvoidance" : "sound"}
  ]
}
```

One deviation from the XML spec is that the root element in the XML is currently “AccessForAllUser” whereas in the json draft 3.3 binding it is “pnp”. It has been modelled here as part of the data instance in order to facilitate mapping between the json instances and *any* technology, though for use with AJAX xmlHttpRequest it may be more sensible to not have “pnp” as part of the data payload (it’s part of the message/protocol). jsonSchemas3.4 provides a Schema for exactly that scenario but which is otherwise identical with this one.

In traditional json objects are denoted by {} containment and arrays by [] containment. An object is an unordered set of attribute:value pairs, an array is an ordered list of values. Attribute values (but not attribute names) can be arrays and array elements can be objects. Array elements do not have to be unique (a consequence of arrays being ordered) but IET Draft 3 says that object attribute names within an object SHOULD be unique within that object. This is important to AfA modelling.

AfA 3.0 specifies that some properties may occur multiple times in an instance (note that some may not). Both accessModeRequired and hazardAvoidance are examples in the pnp instance above of properties that occur multiple times in the instance. To facilitate this, the value of a “pnp” has been modelled as an array of objects, each of which is a “property” of AfA. By modelling this as

an array we permit multiple instances of each property. An attribute-value pair cannot be an array element as it is, so each is wrapped as an object to permit it to be an array element. In the case of accessModeRequired (and other properties not shown here) the value of the property is **also** structured as an object "to keep the sub-fields together".

However, this is not the end of the story. Modelled in this way (and as will be clear later in these notes, modelled in **any** way) its very difficult or impossible to model a multiplicity of 0..1. We can require a property be present, we can require it not be present but in this style we cannot say "its there zero or once" or even "its there exactly once".

To see the problems this raises the following is an instance that is also valid by the V3.3 pnp schemas

```
{ "pnp":
[
  { "atInteroperable" : true },
  { "atInteroperable" : false },
  { "accessModeRequired" : {
          "existingAccessMode" : "visual",
          "adaptationRequest" : "auditory",
          "existingAccessMode" : "textual",
                          }
  }
]
}
```

Note the problems that atInteroperable is present twice and with conflicting values and existingAccessMode is present twice for the given accessModeRequired. One solution to this problem would be additional validation with another technology than JSON Schema to eliminate additional properties where they should only occur once, such as atInteroperable and existingAccessMode within accessModeRequired. AfA requires atInteroperable have a multiplicity of 0..1 and within each accessModeRequired both existingAccessMode and adaptationRequest require a multiplicity of 1. Software to do this extra validation, whilst not completely trivial as it requires (for this binding) a primitive symbol table for counting occurences with local scope (e.g. inside each accessModeRequired is a local scope) is perfectly feasible and is under construction.

Other ways to model this and technical approaches being developed in LD-JSON go some way to alleviating this difficulty but neither completely solves it. These issues are explored further in the section on the schemas V4.0 binding.

jsonSchemas3.4

These schemas adopt an identical approach to the 3.3 schemas but without incorporating the “pnp” in the data payload. Everything is moved one branch up the tree. An instance that validates to the 3.3 schemas, when the outer object and pnp attribute name is removed, will validate to the 3.4 schemas. An example instance matching the one given earlier is

```
[
    { "accessModeRequired" : {
      "existingAccessMode" : "visual",
      "adaptationRequest" : "auditory”
                            }
    },
    { "accessModeRequired" : {
      "existingAccessMode" : "visual",
      "adaptationRequest" : "textual"
                            }
    },
    { "hazardAvoidance" : "flashing" },
    { "hazardAvoidance" : "sound"}
]
```

jsonSchemas4.0

In this binding we adopt mechanisms closer to habitual practices of JSON programmers and best suited to migration to LD-JSON, should that become the binding of choice. At the same time, the binding works now, with traditional json.

A mechanism often used by programmers working with JSON where there are repeated attributes is to wrap the values in an array and eliminate the repetition. For example, instead of

```
{ "hazardAvoidance" : "flashing",
  "hazardAvoidance" : "sound" }
```

we could have

```
{ "hazardAvoidance" : [ "flashing", “sound” ] }
```

We adopt this practice for AfA 3 properties that may occur more than once in an instance*.*

In addition, for a property like this that could have several values but having only one value in some particular instance might be coded by a json programmer without the array at all, for example

```
{ "hazardAvoidance" : "flashing" }
```

So we might have the position that in some instances `"hazardAvoidance"` is an array of values, in others it is a string. Dealing with this requires unions of types, which complicates implementations so we will here adopt the practice advocated in a recent draft of LD-JSON[1] that ***where AfA has a property that may occur multiple times we represent its value as an array, even if the property has only one value***.

In addition, in this binding we remove all extraneous levels and model an instance as a simple JSON object.

This ***does not*** solve the problem of repeated properties. The reasons are as follows:

IETF Draft 3 (and other sources, such as RFC 4627, as referenced by the group working on LD-JSON - D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)* July 2006. Internet RFC 4627. URL: http://www.ietf.org/rfc/rfc4627.txt) require that attribute names (keys) in a JSON object SHOULD be unique. However, as implemented by common toolkits such as jQuery and JSV, repeated attribute names are accepted. The Schemas constructed here (used to validate AfA instances using JSV) do not require that attribute names are unique and there is no easy way to do so. So whilst toolkits accept repeated attributes the value on accessing such an attribute is uncertain. Since attributes in an object are not ordered there is no way to know which value of a repeated property will be returned.

Arrays **do** (at least in traditional json[2]) provide ordering and they also provide a way that Schemas can specify the minimum and maximum number of items in an array contained in an instance. By very careful use of structure and arrays it might be possible to construct an artificial structure for this data that **can** be validated to reject multiple occurrences. This would have to specify every attribute that had only a single value (e.g. existingAccessMode inside accessModeRequired) as an array having a name property and a value property and with careful use of maxitems and minitems …. the reality is that such a structure would be aritificially convoluted just to achieve the validation and that would not be in keeping with the spirit of JSON object use. The better approach is a separate layer of validation to eliminate multiple occurrences of properties. By adopting the practices embodied in this binding that task is at least made substantially easier because now, we can require that **every** property is unique. This extra stage **is still needed.** Whether LD-JSON processors can validate to reject instances with non-unique attribute keys remains to be seen.

---

[1] (http://www.w3.org/TR/2012/WD-json-ld-syntax-20120712/, section 4.8, "Values of terms associated with a `@set` or `@list` container are always represented in the form of an array - even if there is just a single value that would otherwise be optimized to a non-array form .."

[2] the situation in LD-JSON is different – arrays are (at current draft) unordered but a global attribute in the @context section can decree they are all lists, which are ordered, thus maintaining the traditional json interpretation

To restate the practices we adopt in this binding:

- Every attribute name (key) is unique within the object that it occurs. Validation for this is done externally to the JSON Schema processing

- Where an AfA property can occur multiple times then the values of each occurrence are “wrapped” in an array, even where the property occurs only once

An example pnp instance might be:

```
{
   "accessModeRequired" :
      [ { "existingAccessMode" : "visual",
          "adaptationRequest" : "auditory"
        },
        { "existingAccessMode" : "visual",
          "adaptationRequest" : "textual"
        }
      ],
    "hazardAvoidance" : [ "flashing", "sound" ],
    "educationalComplexityOfAdaptation" : "simplified",
    "atInteroperable" : true,
    "languageOfAdaptation" : [ "eng" ]
}
```

Notice the following:

- accessModeRequired occurs once but has multiple values in an array, each is an object having multiple properties

- hazardAvoidance has multiple values (because in AfA 3.0 it can occur multiple times) but educationalComplexityOfAdaptation can occur only once and its value is a string

- languageOfAdaptation has one value which is an array element (because it **could** occur in an instance with multiple values).