Second draft

Configuration Properties

Steve Powell

April 30, 2009

We describe what we can put in a Configuration dictionary, and propose a mapping from existing JSON-style formats to these.

Contents

1 Introduction

The OSGi specification [1] in section 104 *Configuration Admin Service Specification* Version 1.2, describes the *Configuration Admin* service. This service allows the creation, dissemination, and persistence of configurations for objects (which may be bundles, devices, or other resources) in an OSGi framework environment.

Among the miraculous 'essential' requirements that implementations of this service should meet are (104.1.1 in [1]):

- *Immediate Response* – Changes in configuration should be reflected immediately.
- *Communications* – The Configuration Admin service should not assume "always-on" connectivity, so the API is also applicable for mobile applications in cars, phones, or boats.
- *Remote versus Local Management* – The Configuration Admin service must allow for a remotely managed OSGi Service Platform, and must not assume that configuration information is stored locally. Nor should it assume that the Configuration Admin service is always done remotely. Both implementation approaches should be viable.
- *Availability* – The OSGi environment is a dynamic environment that must run continuously (24/7/365). Configuration updates must happen dynamically and should not require restarting of the system or bundles.

Putting aside the difficulties inherent in these 'requirements', we concentrate on the nature of a Configuration, in particular on the nature of the configuration *Dictionary*, which captures the configuration properties we wish to describe.

Our descriptions are based upon the specification ([1]) and upon the feL$\iota\chi$ implementation in the jar `org.apache.felix.configadmin-1.0.10.jar` from `org.apache.felix`.

We use reasonably informal grammar production syntax, for the sake of brevity.

References

[1] OSGi Service Platform Service Compendium v4.1, April 2007 `http://www.OSGi.org`

2 The dictionary

Section 104.4.2 *Configuration Properties* in [1], describes the types of property values in a configuration dictionary object as follows (some minor alterations for readability):

```
type      ::= simple | vector | array
simple    ::= String | Integer | Long | Float | Double
            | Byte | Short | Character | Boolean
primitive ::= long | int | short | char | byte | double
            | float | boolean
array     ::= primitive‘[]’ | simple‘[]’
vector    ::= Vector<simple>
```

It goes on to describe what the keys may look like:

> The name or key of a property must always be a `String` object, and is not case-sensitive during look up, but must preserve the original case. The format of a property name should be:
>
> ```
> property-name ::= symbolic-name // See 1.3.2
> ```

Section 1.3.2 in [1] describes the dot-separated tokens that can form the string key—based upon the bundle-symbolic-name rules:

> Ensures the key complies with the symbolic-name production of the OSGi core specification (1.3.2):
>
> ```
> symbolic-name :: = token(‘.’token)*
> digit     ::= [0..9]
> alpha     ::= [a..zA..Z]
> alphanum  ::= alpha | digit
> token     ::= ( alphanum | ‘_’ | ‘-’ )+
> ```

The dictionary is then a simple `Map<property-name, property-type>` (normally implemented by means of a `Hashtable`).

In the feL$\iota\chi$ jar there is an implementation of this form of dictionary, called a `CaseInsensitiveDictionary`, which includes two (`static` package-private) methods that check for these rules of formation. They are used in the constructors, which can take general `Dictionary` properties and convert them.

```
static void checkKey( Object keyObject )
{ ... }
```

and

```
static Object checkValue( Object value )
{ ... }
```

These are simple-minded pieces of code that enforce the above rules. The `CaseInsensitiveDictionary` itself keeps the original case of the key around in parallel with the *LowerCaseKey* → *Value* map that stores the properties.

These rules do *not* permit arbitrarily nested values. Arrays of `simple` or `primitive` types, or Vectors of `simple` types, are the only 'structures' that are permitted.

In order to accommodate the sort of JSON-style configuration properties we currently enjoy, there has to be a mapping from JSON-like structures to these (simple-minded) keyed values.

We exploit the dot-delimited form of tokens to allow *un*-dotted tokens as keys in a structured hierarchy.

In the next section we see why this cannot be done in general.

3 JSON structures

Here is a simple (sanitised) description of what a JSON (JavaScript Object Notation) property structure looks like:

```
object ::= ‘{’ pair*(‘,’) ‘}’
pair ::= string ‘:’ value
array ::= ‘[’ value*(‘,’) ‘]’
value ::= string | number | object | array | ‘true’ | ‘false’ | ‘null’
string ::= ‘"’ char* ‘"’
char ::= (any-Unicode-character-except-"-or-\-or-control-character)
        | ‘\"’ | ‘\\’ | ‘\/’ | ‘\b’ | ‘\f’ | ‘\n’ | ‘\r’ | ‘\t’
        | ‘\u’ (four-hex-digits)
number ::= int {frac} {exp}
int ::= {‘-’} digits
frac ::= ‘.’ digits
exp ::= e digits
digits ::= [0-9]+
e ::= (‘e’|‘E’) {‘+’|‘-’}
```

where it is permitted to have 'whitespace' around any components of `object` and between components of `pair` and `array`. (We do not mention comments but we assume that they are semantically equivalent to whitespace, wherever they occur. The syntax can be decided later.)

Syntax note We have used an extension to the (arbitrary number of) repetitions notation (`*`). The term `a*(b)` denotes arbitrary numbers of terms `a`, separated by terms `b`, if there are more than one of them. Thus `pair*(',')` denotes zero or more `pair`s, interleaved with commas, if there are two or more—a comma-separated list of `pair`s. Similarly, `a+(b)` insists that there is at least one `a` but is otherwise identical.

The `object` production is the major recursive feature of this specification, allowing arbitrarily deep structures to be described.

It is clear that this is too rich a structure for us to map (easily) into a configuration dictionary. For one thing, the keys we would want to use are arbitrary strings in JSON files (`string` in the `pair` production above) but need to be merged into the key strings of configuration dictionaries.

We have to compromise.

By limiting the string part of the pair production (and a few other minor restrictions) we can eliminate this problem, and introduce a reasonably well-structured mechanism for recording configuration information, and an ability to map it onto existing configuration dictionaries without loss.

4 Reduct of JSON syntax

We propose a modification of the JSON syntax to restrict the values used in the `string` part of the `pair` production. In addition, we lift the (rather messy) requirement for quotation marks around these strings – we will call them 'name's.

We need to eliminate the `null` value (as the rules do not permit the null value as either a key or a value).

Our new syntax for JSON-like structures (we could call them JS2N structures?) is:

```
object ::= pair+(',')
pair ::= name ':' value
array ::= '[' value*(',') ']'
value ::= string | number | '{' object '}' | array | 'true' | 'false'
name ::= alpha (alphanum|'_'|'-')*
```

```
alpha ::= [a-zA-Z]
alphanum ::= alpha | digit
string ::= '"' char* '"'
char ::= (any-Unicode-character-except-"-or-\-or-control-character)
        | '\"' | '\\' | '\/' | '\b' | '\f' | '\n' | '\r' | '\t'
        | '\u' (four-hex-digits)
number ::= int {frac} {exp}
int ::= {'-'} digits
frac ::= '.' digits
exp ::= e digits
digits ::= digit+
digit ::= [0-9]
e ::= ('e'|'E') {'+'|'-'}
```

The key alterations here are:

in `pair` we use an *unquoted* term to name the `value` it precedes;

in `object` we no longer require outermost braces (but in `value` we need them around embedded objects);

also in `object` we require at least one `pair`;

in `value` we remove `'null'`; and

in `name` we introduce the severe restrictions upon the form of names.

We again do not mention 'whitespace' or comments, for brevity.

These `name`s *must* begin with a letter (of either case), and can only contain letters, digits, underscores and hyphens. This is (just) enough for us to form 'compound' names that can go into a dictionary. The general idea of this is to 'flatten' the JS2N syntax into dictionary form as described below.

5 JS2N and configuration dictionary mapping example

The basic idea can be illustrated by an example.

Here is a JS2N structure (adapted from the `ConfigPoint` tests in `kernel.config`):

```
a:{ a:"aString",
    c:4.2,
```

```
      m:[ { x:{ a: "aString",
                c:42,
                d:1,
              },
            y:{ a: "aStringWithOddCharacters.\/\\*",
                f:1,
              },
            v:"2.5.6"
          },
          1953
        ],
      aa:"foo/bar"
    },
  b: { b:"[1,2]" }
```

which contains a smattering of interesting structures (by the way, there is no reason to suppose that the values in arrays must all have the same form).

Here is the same structure 'rendered' in a configuration properties object:

```
'a.a' -> "aString",
'a.c' -> 4.2,
'a.m.1.x.a' ->  "aString",
'a.m.1.x.c' -> 42,
'a.m.1.x.d' -> 1,
'a.m.1.y.a' ->  "aStringWithOddCharacters.\/\\*",
'a.m.1.y.f' -> 1,
'a.m.1.v' -> "2.5.6",
'a.m.2' -> 1953,
'a.aa' -> "foo/bar",
'b.b' -> "[1,2]"
```

where `->` is shorthand for 'maps to' in the properties map, and the `String` type and `long` and `double` primitive types are used where necessary.

Notice that there were two elements in the array (`'a.m'`), represented by the numeric components of the token used as the key, and that the proposed JS2N syntax prohibits a numeric name, so no confusion can occur with a `name`. Notice also that proper arrays (or vectors) are not used in the translation. This means that some information can be lost (but see below).

The properties object is a map with no particular ordering on the keys it contains, so when rendering it back into a JS2N form, there is no reason to suppose it will be in the same order. Is it possible that the structures

will be messed up? Answer, no, provided that the names of the pairs in an object production are all distinct. To generate the form it is merely necessary to lexicographically order the keys (using numeric ordering for the numeric components) and produce the appropriate syntax to re-render it.

What does 'distinct' mean for key strings in a configuration properties object? It means identical *except for case distinctions*. Case is preserved (the last updater wins) but is not significant for comparisons. In the feL$\iota\chi$ implementation the special `CaseInsensitiveDictionary` is used for this purpose.

We would make some special cases; for example, in the case of a top-level array of primitives (all of the same type), we might render this as an actual array in the dictionary. There doesn't appear to be a way of forcing the use of boxed primitive types (`Integer`, `Long`, etc.) easily, nor of enforcing a vector when desired.

Under this scheme, all configuration dictionaries can be rendered in JS2N format, though converting them back (without the special case) would render them in a different structural format, and boxed types and the smaller primitives would be expanded (`int`, `short`, `byte` are all replaced by `long`, `char` is replaced by `String`, and `float` is replaced by `double`).

5.1 Lost in translation

We have already mentioned the 'conversion' that would occur for simple types, but there are other subtleties that occur, for example, in arrays.

In JS2N arrays are essentially structures with unnamed values. Our scheme invents names for these and tries to avoid confusion by representing them with tokens that cannot arise from structure names. However, they could arise from configurations generated by other means.

We have to decide, for example, how to render a properties map like this into JS2N:

```
'a.01' -> "aha!",
'a.1' -> "oh dear",
'a.3' ->  "where's 2?",
'a.0D' -> "invalid token"
```

Here, the numeric token `01` is numerically equal to `1`, the second element of the 'array' appears to be missing, and is `a.0D` part of the array or not? The jury is out on these, but we probably need to throw a wobbly (sorry, exception) when attempting to render these in JS2N.

6 JS2N to configuration properties map

The function from JS2N forms to properties maps is designed to be 1-1 and total. It can be defined (recursively) by starting at the lowest levels of the structure to be constructed and working upwards.

All the `atom`s can be rendered directly as Java objects (`String`s, `double`s, `long`s or `boolean`s). Do this first. We will call these *basic* objects. We convert the JS2N form to a map from strings to basic objects. Such a map is called a *basic* map.

We use the following primitives. String is for string objects, Basic is for all the basic types, BasicMap is a primitive properties map (strings to basic values), POINT is the decimal point character in a string, str is a simple function for representing natural numbers as strings, mapObj turns a basic object into a map with one mapping in it – from the given string to the basic object, and mapMap prefixes a given string on to the front of all the keys in a given map:

$$[Char, Basic]$$
$$String == \mathrm{seq}\, Char$$
$$BasicMap == String \pfun Basic$$

$$\begin{array}{l} POINT : String \\ str : \mathbb{N} \fun String \\ mapObj : String \times Basic \fun BasicMap \\ mapMap : String \times BasicMap \fun BasicMap \\ \hline \forall\, s : String;\ b : Basic;\ m : BasicMap \bullet \\ \quad mapObj(s, b) = \{\, s \mapsto boj \,\} \land \\ \quad mapMap(s, m) = \{\, s \frown key \mapsto m\, key \,\} \end{array}$$

Starting at the lowest levels proceed as follows:

- at each point in the structure there is:
  - a basic object;
  - a basic map; or
  - a sequence of basic elements, each of which is either a basic object or a basic map.
- If we are at an array structure ($\divideontimes$) form the following basic map, depending on what we have:

  - a basic object *boj*: form the map $mapObj(str\,1, boj)$;
  - a basic map m: from the map $mapMap((str\,1) \frown POINT, m)$;
  - a sequence *lst* of basic elements, each of which is either a basic object or a basic map: for the *idx*'th member of the sequence–
    form the map $mapObj(str\,idx, lst\,idx)$ if it is a basic object,
    or the map $mapMap((str\,idx) \frown POINT, lst\,idx)$ if it is a basic map;
    then take the union (or map override, it shouldn't matter) of the maps produced to form a single basic map.

- If we are at a pair structure (`name:value`) there is only either a basic object or a basic map, representing the `value`,
  - so form the map $mapObj(\texttt{name}, boj)$ if `value` is a basic object *boj*,
  - or the map $mapMap(\texttt{name} \frown POINT, m)$ if `value` is a basic map m.

- If we are at a structure element we have a sequence of basic maps only (one for each `name:value` pair), simply take the union of all these maps to form a basic map.

If each union in the algorithm results in a well-formed map (no duplicate keys) then the resulting map will be basic and correctly represent the configuration information and its structure. (*Proof?*)

At the array formation stage ($\divideontimes$) we can decide to form the array `[]` values allowed by configuration properties; provided the elements are all basic *and of the same type*. The notion of a *BasicMap* would need to be extended to include these arrays in its range, a function would take us from appropriate sequences of *Basic* values to these arrays and all would be well.

7 Configuration properties to JS2N map

The reverse map is both easier and trickier.

The first step is to collate the keys in the properties map. These are tokens (see the `symbolic-name` definition in the section *The dictionary*) separated by periods. We treat each token (which is never empty) as either numeric (if it is formed entirely of digits) or a name.

At this stage we can check that the names all begin with a letter. If they do not, we have to adopt a policy to cope with names that do not conform to the JS2N forms – perhaps we revert to a JSON indexing form.

We can sort the keys lexicographically: this is left-to-right, without regard to case. When this is done, we have another conformance problem: when cases disagree, which of them do we take? For example in:

```
‘aname.1’ -> "lions",
‘aName.2’ -> "and tigers",
‘anaMe.3’ -> "and bears",
‘Aname.4’ -> "oh my!"
```

which top-level name for the (array) `aname` do we take? [Probably the first one (after collating them).]

At this point, given that we have some numeric tokens, do we modify them to get array subscripts? (The same examples above with qualifiers `01`, `1`, `3`, and `0D`, do not satisfy the rules—how do we cope with these?)

Given that the numbers are all consecutive, starting at one, and are formed correctly (no leading zeros, for example), we can form an array from them (if they are all of the same basic type).

Another problem can occur if there are two keys like `a.b` and `a` in the same properties file. In this case we find that there is an object and a basic value with the same name. This is not representable.

The algorithm can proceed from the top down, by attrition of the map keys, reversing the maps we used before. We use the following primitives to explain the process, with the same definition of *BasicMap*, *String*, *Basic* and so on, that we had before:

$$
\begin{array}{l}
unstr : String \pfun \mathbb{N} \\
prefixSelect : String \times BasicMap \pfun BasicMap \\
prefixReduce : String \times BasicMap \pfun BasicMap \\
intSelect : BasicMap \pfun BasicMap \\
\hline
unstr \circ str = \mathrm{id}\,\mathbb{N} \land \mathrm{dom}\, unstr = \mathrm{ran}\, str \\
\\
\forall\, s : String;\ m : BasicMap \bullet \\
\quad prefixSelect(s, m) = \{\, k : \mathrm{dom}\, m \mid s \ \mathsf{prefix}\ k \bullet squash(k \setminus s) \mapsto m\, k \,\} \land \\
\quad prefixReduce(s, m) = \{\, k : \mathrm{dom}\, m \mid s \ \mathsf{prefix}\ k \,\} \lhd m \land \\
\quad intSelect\, m = \{\, k : \mathrm{dom}\, m \mid \exists\, n : \mathbb{N} \mid str\, n \ \mathsf{prefix}\ k \,\} \lhd m
\end{array}
$$

str and *unstr* are pretty self-explanatory, and *prefixSelect* is the inverse of *mapMap* in much the same way. The rule is clumsy to express, but would look like:

$$
\begin{array}{l}
\quad s : String;\ m : BasicMap \\
\vdash \\
\quad prefixSelect(s, mapMap(s, m)) = m
\end{array}
$$

And *prefixReduce* simply gives what's left of the map when *prefixSelect* has finished its evisceration. If the maps keys are well-formed it should be possible to say that:

$$\begin{array}{l} \quad s : \mathit{String};\ m : \mathit{BasicMap} \\ \vdash \\ \quad \mathit{mapMap}(s, \mathit{prefixSelect}(s, m)) \cup \mathit{prefixReduce}(s, m) = m \end{array}$$

Which is to say that extracting the prefix selected part of the map m, and reprefixing it, and adding this to the part that wasn't selected, should reconstruct the entire map.

The algorithm begins by taking a *key* (any key, it turns out) from the map m which is the properties map, and examining it for its form:

- If *key* has no periods in it (*POINT*s) in it, and it is a `name`, then extract the value of that key ($m\,key$) and form the pair: `key:Value`, where `Value` is the canonical form for the basic value found there (or an array, if this was an array of basic values). These are `String`s, `long`s, and so on. Remove the *key* form the map, and if this is empty, terminate (this recursive level). If not empty, output a comma and iterate.

- If *key* has no periods in it (*POINT*s) in it, and it is an `int`, then extract the map $\mathit{intSelect}(m)$. This map needs to be examined. If it is a simple map from a sequence of integer strings (starting at 1) to values, all of the same type, then we can construct the array of these values as output. The $\mathit{intSelect}(m)$ mappings may be removed from m before proceeding. If the resulting map is empty then terminate (this recursive level). If not empty, output a comma and iterate.

  If it is not such a simple map, then we can still construct the array, but more work (and recursion) is required–we construct the array by taking the mappings of $m' == \mathit{intSelect}(m)$ in sequence (index i), starting at $i = 1$ by first outputting `'['`. For the case i, if the $key \in \mathrm{dom}\ m'$ is exactly $\mathit{str}\ i$, then output the value $m'(\mathit{str}\ i)$ and remove the mapping for this key from m'. If the *key* has prefix $(\mathit{str}\ i)^\frown \mathit{POINT}$ we recurse (on the entire algorithm) with the map $\mathit{prefixSelect}((\mathit{str}\ i)^\frown \mathit{POINT}, m')$ and then update m' to $\mathit{prefixReduce}(s, m')$. If the map m' is not empty, output a comma, and proceed to the next i. If the map m' is empty, output `']'` and terminate the iteration.

- If *key* has a first term (say $(term \frown POINT)$ prefix *key*) then extract the map $prefixSelect(term \frown POINT, m)$ and output *term*, followed by a colon and '{'. Then recurse on the entire algorithm with the map $prefixSelect(term \frown POINT, m)$, and then output '}'. Reset the map to $prefixReduce(term \frown POINT, m)$ and if empty, terminate (this recursive level), otherwise output a comma and iterate (this level).

OK so this is impenetrable. But I think it'll work.

8 Implications

When looking in the `Configuration` object at the properties, a bundle will not necessarily be able to 'parse' the structure without help. What facilities will we provide? We can supply `util` classes to get array elements and traverse the configuration analogous to the `ConfigTree` stuff we already have, indeed we might be able to retro-fit the `ConfigTree` interface to 'read' configuration properties that derive from JS2N structures.

If we could do the latter, converting to the new configuration mechanisms might prove simpler—we could leave clients of `ConfigTree` *et al'* unchanged, until we get to them.

9 Discussion

Are we happy with the compromises inherent in here? If so, we can code up the algorithms pretty easily, and use them to populate/extract the configurations users want to use and/or export. [*Note:* this is not the form of a persisted configuration, though it might be.]

Initialisation of the ConfigurationAdmin service ought to occur very early, so that any bundle (subsystem) that needs configuration can get it from ConfigurationAdmin. Presumably, we would have a standard place to put configurations to 'start up with'. Is this the same as the persistent state? If this is a warm start we should use the persistent state, and not the 'initial' area; though on a cold start we should start again from the initial configuration.

How (and when) does one cause the current persistent configuration to be 'saved' to the initial configuration? This seems to me to be a crucial piece of the configuration design.