Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 6, 2016

P. Cordell
Codalogic
A. Newton
ARIN

March 5, 2016

Co-Constraints for JSON Content Rules
draft-cordell-jcr-co-constraints-00

Abstract

JSON Content Rules (JCR) provides a powerful, intuitive and concise method for defining the structure of JSON [RFC7159] messages. However, modern JSON usage patterns occasionally mean that JCR alone is not able to capture the required constraints in a satisfactory way. The document describes JCR Co-Constraints (JCRCC) which defines additional JCR directives and annotations that can be added to a JCR ruleset in order to define more detailed constraints on JSON messages.

Status of This Memo

Copyright Notice

1. Introduction

JSON Content Rules [JCR] provides a powerful, intuitive and concise method for defining the structure of JSON [RFC7159] messages. In addition to describing the overall structure of JSON messages, JCR aims capture the constraints that are imposed on individual items within a message. However, modern JSON usage occasionally requires constraints that can’t be expressed by JCR alone. JCR Co-Constraints (JCRCC) defines additional JCR directives and annotations that can be added to a JCR ruleset in order to define more detailed constraints on items within a JSON message, and also supports specifying constraints that depend on the relationship of multiple JSON items.

JCRCC constraints represent an additional layer of validation on top of the validation offered by JCR alone. JCRCC constraints may indicate that a JSON instance that was determined to be valid by the rules of a JCR ruleset, is in fact invalid. However, if the JCR ruleset indicates that the JSON instance is invalid, JCRCC constraints can not override that and declare the instance valid. A JCR processor may ignore the JCRCC annotations and directives, perhaps only issuing a warning for encountering an unknown annotation or directive.

JCRCC uses the annotations @{id}, @{when} and @{assert} along with the directive #{constraint}. The @{id} annotation is used to indicate an item in a JSON message that contributes to the assessment of a JSON instances validity. The other three each include a ’condition’ expression that yields a Boolean true or false result. The validity of the JSON instance is dependent on the results of the various condition expressions. Condition expressions are made up of identifiers, comparators, combiners and functions. Processing of the condition expressions is triggered according to a ’conceptual processing model’. Each of these aspects is described in more detail below.

2. Annotations and Directives

JCRCC uses the annotations @{id}, @{when} and @{assert} plus the directive #{constraint}.

2.1. The @{id} Annotation

The @{id} annotation creates an identifier for a rule in a JCR ruleset. A maximum of one @{id} annotation is permitted per rule. It has the form:

```
@{id name}
```

where 'name' corresponds to the 'name' production in the JCR ABNF.

The @{id} annotation associates an identifier with the rule on which it is placed. The identifier can then be used in condition expressions to reference the corresponding item in JSON instance items that are mapped to the JCR rule during validation.

For example, with a JCR rule of:

```
"type" @{id t} : string
```

might associate the identifier 't' with a JSON instance item such as:

```
"type" : "shutdown"
```

2.2. The @{when} Annotation

The @{when} annotation has two similar roles. If a JCR rule indicates that a JSON instance item is optional, then it can be used to describe the conditions when the item is present or absent. Similarly, if a JCR rule indicates that an item has a group or type choice as it's type, then the @{when} annotation can be used to indicate which of the possible sub-rules is applicable in the current validation instance. Only one @{when} annotation per rule is permitted.

The @{when} annotation includes a single 'condition'. In the case of using the @{when} annotation with an optional instance, if the condition yields a 'true' result, then the item associated with the JCR rule MUST be present, otherwise it MUST be absent.

When the @{when} annotation is used to select the applicable member/ type rule within a group or type choice, the condition of each @{when} annotation is evaluated in turn (from left to right as shown in the JCR rule) and the member/type rule that corresponds to the first @{when} condition that yields a 'true' result is selected. If none of the @{when} annotations on a group or type choice yields true, this indicates an invalid instance. When a member/type rule within a group or type choice that has @{when} annotations on other members/types, but does not itself have an @{when} annotation, this indicates the default case. In essence, if a rule has @{when} annotations, then an absent @{when} annotation on a member/type rule is equivalent to @{when true}.

As an example, a @{when} annotation on an optional item may look as follows:

```
        ? @{when $t == "shutdown"} "uptime" : integer
```

This indicates that the "uptime" member should be present if the JSON instance item associated with a JCR rule with an @{id t} annotation has the value "shutdown".

A @{when} annotation on a group may look as follows:

```
        details ( @{when $t == "boot"} boot-details |
                @{when $t == "shutdown"} shutdown-details |
                default-details )
```

This indicates that the JCR rule named ’boot-details’ is applicable when the JSON instance item associated with an @{id t} annotation has the value "boot", the rule ’shutdown-details’ is applicable when the value of the $t item is "shutdown", otherwise the rule ’default-details’ is applicable. (The rules identified by ’boot-details’, ’shutdown-details’ and ’default-details’ might be groups that act as mixins for the rule in which the ’details’ rule is used.)

The @{when} annotation can reference identifiers in siblings, ancestors, and descendants. To avoid circular or ambiguous dependencies, the identifiers in descendants must not be part of arrays or descendants of itself or descendants of siblings that have @{when} annotations. The latter restriction avoids needing to know whether a secondary @{when} annotation yields ’true’ in order to determine if the @{when} annotation being assessed yields ’true’. When seeking identifiers, siblings are inspected first, followed by the nearest ancestor, followed by the nearest descendent. If it is desired to look for an identifier that is a descendent without first looking for an identifier that is an ancestor, then the ’descendent()’ method can be called on the name of the identifier. For example:

```
        ? @{when descendent($s) == "on"} "watts" : integer
```

2.3. The @{assert} Annotation

The @{assert} annotation is used to specify additional constraints on an item that can’t be expressed using JCR alone. The @{assert} annotation contains a single condition that must yield ’true’ for the

JSON instance containing the respective item to be considered valid. A maximum of one @{assert} annotations is permitted per rule.

@{assert} annotations are evaluated after all sibling @{when} annotations have been evaluated, and constraints specified by the underlying JCR rule have been assessed. An item may have both a @{when} annotation and a @{assert} annotation. If the condition in the @{when} annotation yields 'false', then the item it corresponds to should be absent in the JSON instance, so the @{assert} condition is not evaluated. If, for example, the JSON instance item does not have the type specified by the underlying JCR rule, then validation fails at that point and the @{assert} annotation is not assessed.

When seeking identifiers referenced in an @{assert} annotation, siblings are inspected first, followed by the nearest ancestor, followed by the nearest descendent. If it is desired to look for an identifier that is a descendent without first looking for an identifier that is an ancestor, then the 'descendent()' method can be called on the name of the identifier.

An example @{assert} annotation might be:

```
        "index" : @{assert $ % 2 == 0} integer ; Must be even
```

2.4. The #{constraint} Directive

The #{constraint} directive offers a way to express conditions external to @{when} and @{assert} annotations. #{constraint} directives can be viewed as a macro substitution mechanism. @{when}, @{assert} annotations and other #{constraint} directives can reference conditions defined by a #{constraint}. The format of a #{constraint} directive is as follows:

```
        #{constraint name condition}
```

where 'name' corresponds to the 'name' production in the JCR ABNF, and the 'condition' is the same as used in @{when} and @{assert} annotations and is as described below.

Conceptually at least, the condition in a #{constraint} directive is substituted into @{when}, @{assert} annotations and other #{constraint} directives wherever the constraint's name is referenced. (In practice, for the purposes of efficiency, the result of a #{constraint} directive may be cached or memoized, to avoid repeated computation of the sub-condition. However, such optimizations are beyond the scope of this documents.)

An example usage, equating to the earlier example, might be:

```
    #{constraint is_even $ % 2 == 0}
      "index" : @{assert @is_even} integer ; Must be even
```

3. Conditions

The @{when} annotation, @{assert} annotation and #{constraint} directive contain ’conditions’. These are made up of ’identifiers’, ’comparators’, ’combiners’ and ’functions’ as described below.

3.1. Identifiers

Identifiers are used to refer to items in the JCR, and #{constraint} directives. They have a few different forms.

’$’ on its own refers to the member / type expressed by the current JCR rule. For example:

```
      int-pairs @{assert count( $ ) % 2 == 0} [ : integer ]
```

The form ’$name’ and ’$alias.name’ refers to members / types identified by @{id} annotations. An ’alias’ is set up using the normal JCR #import directive and allows members / types outside the current ruleset to be identified. For example:

```
      "type" @{id t} : string
      ? "uptime" @{when $t == "shutdown"} : integer
```

The ’@name’ and ’@alias.name’ refers to a condition expressed in a #{constraint} directive. An ’alias’ is set up using the normal JCR #import directive and allows constraints outside the current ruleset to be identified. For example:

```
    #{constraint is_even $ % 2 == 0}
      "index" : @{assert @is_even} integer ; Must be even
```

3.2. Standalone Id Refernces

A reference to a member / type that is not part of a ’comparator’ sub-expression yields ’true’ if the referenced item is present in the JSON instance being validated, and ’false’ if not. For example, the following says that the ’dob’ member must be present if the ’name’ member is present:

```
    ? "name" @{id n} : string,
    ? "dob" @{when $n} : full-date
```

3.3. Operators

The values of members / types identified by identifiers, values yielded by other ’operators’ and values returned by ’functions’ can be subject to computations using ’operators’. The supported operators are ’+’, ’-’, ’*’, ’/’ and ’%’. They have their usual C-family programming language meaning.

3.4. Comparators

The values of members / types identified by identifiers, values yielded by ’operators’ and values returned by ’functions’ can be compared using ’comparators’. The comparators are the usual ’<’, ’<=’, ’==’, ’!=’, ’>=’ and ’>’, and have their usual C-family language meaning. Comparators yield a ’true’ or ’false’ result.

When an identifier referenced by a comparator is absent, then the comparison returns ’false’. For example:

```
        $t == "boot"
```

is equivalent to:

```
        ( $t && $t == "boot" )
```

Similarly:

```
        $t == "boot" || $other == "close"
```

is equivalent to:

```
        ( $t && $t == "boot" ) || ( $other && $other == "close" )
```

And:

```
        length( $first ) > length( $second )
```

is equivalent to:

```
        ( $first && $second && length( $first ) > length( $second ) )
```

3.5. Combiners

Multiple results of ’comparators’ or standalone identifiers can be combined using ’combiners’. The supported combiners are ’&&’ and ’||’. They have their usual C-family programming language meaning.

3.6. Functions

JCRCC supports a number of functions that can be used to yield specific information about a JSON instance item referenced by an identifier. Some functions can operate on multiple types of arguments, such as identifiers and strings. In the function descriptions below, arguments that can take multiple different types have each type listed, separated by the pipe symbol (|). For example, an argument description of "identifier | string" indicates that the function can take an identifier or a string as an argument.

The functions are as follows:

name(identifier) -
: Returns the member name of the JSON instance item associated with the identifier as a string.

length(identifier | string) -
: If the argument is an identifier, the value of the JSON instance item associated it MUST be a string. The function returns the length of the string in Unicode code points. To return the length of a JSON instances member name, do "length(name($t))".

count(identifier) -
: The JSON instance item associated with the identifier MUST be an array. The function return the number of items in the array.

capture(identifier | string, regex) -
: The regex in the capture function MUST include a capture expression (i.e. a suitable set of brackets). The regex is applied to the input string, or the string value of the JSON instance item associated with the identifier, and the sub-string captured by the regex capture expression is returned.

descendent(identifier) -
: The normal order of identifier look up is, siblings, followed by ancestors, followed by descendents. This function will cause the lookup to be in the order siblings followed by descendents. It returns a reference to a JSON instance item that can be used in place of an identifier. For example, "length(name(descendent($t)))".

error(q_string) -
: This function can be used for reporting error messages. The text in the q_string may be subject to value interpolation and internationalization. It always returns false.

is_integer(identifier), is_float(identifier) etc. -
This set of functions return true if the JSON instance item associated with the identifier has the corresponding type, and false otherwise, and false otherwise.

3.7. If-Then-Else

4. ABNF

The ABNF is 'work in progress'. It currently looks as below. This does not capture where spaces are permitted.

```
condition = relational ( * ( "&&" relational ) / * ( "||" relational ) )

relational = ["!"] value / value comparator value / ["!"] condition-group /
ternary

value = identifier / constant / function / "@" [ alias "." ] name

identifier = "$" / "$" [ alias "." ] name

constant = "null" / "true" / "false" / integer / float / q_string / regex

comparator = "==" / "!=" / "<" / "<=" / ">=" / ">"

condition-group = "(" condition ")"

ternary = "if" "(" condition ")" "then" "(" condition ")" "else" "(" conditi
on ")"

function = "name" "(" identifier ")" /
           "length" "(" identifier ")" /
           "count" "(" identifier ")" /
           "capture" "(" regex "," identifier ")" /
           "descendent" "(" identifier ")" / ; Starts looking up in the vari
able stack
           "error" "(" q_string ")" / ; string may be subject to interpolati
on and locatization
           "is_integer" "(" identifier ")" /
           "is_float" "(" identifier ")" /
           etc...
```

5. References

5.1. Normative References

[JCR] Newton, A., "A Language for Rules Describing JSON Content", October 2015, <https://www.ietf.org/id/draft-newton-json-content-rules-05.txt>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

5.2. Infomative References

[ARIN_JCR_VALIDATOR]
American Registry for Internet Numbers, "JSON Content Rules Validator (Work In Progress)", <https://github.com/arineng/jcrvalidator>.

[CODALOGIC_JCR_VALIDATOR]
Codalogic, "cl-jcr-parser (Work In Progress)", <https://github.com/codalogic/cl-jcr-parser>.

Appendix A. JCR Implementations

The following implementations, [ARIN_JCR_VALIDATOR] and [CODALOGIC_JCR_VALIDATOR] have influenced the development of this document.

Authors’ Addresses

Pete Cordell
Codalogic
PO Box 30
Ipswitch IPX 2WY
UK

Email: pete.cordell@codalogic.com
URI: http://www.codalogic.com

Andrew Lee Newton
American Registry for Internet Numbers
3635 Concorde Parkway
Chantilly, VA 20151
US

Email: andy@arin.net
URI: http://www.arin.net