Built-ins for JSON Rules

Emilian Pascalau[1]

Hasso Plattner Institute, Germany
emilian.pascalau@hpi.uni-potsdam.de

Abstract. JSON Rules is a declarative rule language for the World Wide Web. It has been created to satisfy at least the following list of requirements: (1) create and execute rules in browser; (2) support for ECA and PR rules; (3) the Working Memory contains event-facts. Here we extend the language with the concept of built-ins (predicates and actions). We focus on the relation with RIF-DTB, however with a strong emphasis on the environment where the rules are going to be executed: the web browser. As such we introduce here an initial set of built-ins, as well as the architectural aspects that should be taken into consideration for an engine implementing the JSON Rules language.

1 Introduction

There is a multitude of rule languages out there (i.e. Drools [1], F-Logic [2]) each of them with their specificities. However interoperability in a world that moves at high speeds and where business are interconnected is a must. RIF is the W3C Rule Interchange Format[1] and also the name of the W3C group that it is in charge of this standard. Although the main goal of RIF is to provide a rule interchange language, it is more than that. RIF provides different versions, called dialects in order to tackle the serious trade-offs in nowadays rule languages. Almost a mandatory request for any rule language it is to provide at least the guidelines for translating to and from RIF.

JSON Rules introduced initially in [3] and later extended in [4] is a rule language designed to be run in a web browser and to model web based scenarios. The main goals of the language are: to move the reasoning process to the client-side; to offer support for intelligent user interfaces; to handle business workflows; to allow rule-based reasoning with semantic data inside HTML pages (reasoning with RDFa [5]); to create intelligent mashups - rule based mashups; to enable users to collaborate and share information on the WWW (rule sharing and interchange). JSON Rules fulfills the following requirements: rules run in the browser; uses Event-Condition-Action rules; rules are defined on top of DOM structure; uses DOM events plus user defined events; actions are defined by users and can be any JavaScript function calls. JSON Rules language uses a special type of Working Memory (WM) – the Document Object Model (DOM) [6] of the page. As in any forward chaining engine, the main effect of JSON

[1] http://www.w3.org/2005/rules/

rules execution is the update of the WM i.e. the DOM of the page. Therefore, the language design is tailored to the environment where rules are going to be executed.

With respect to its condition language, JSON Rules was influenced in its syntax by other rule languages such as Drools [1]. Any valid JavaScript function call is allowed as actions. This covers also the OMG Production Rule Representation ([7]) and RIF Production Rule Dialect [8].

Any JSON Rule is identified by an `id`. A rule may have a `priority` - if this attribute is missing then its implicit value is `1`. The rule engine executes the rules in a down counting order (from greater priority to lower priority). However, the execution order for rules with the same priority is irrelevant. One significant property of a JSON Rule is the `appliesTo` property. This property stores an array of URLs to which this rule refers to i.e. a list of URLs defining the context in which this rule can be applied. Based on this property, rules are grouped in rule sets.

An `EventExpression` is an expression capturing any DOM Event [9]. Therefore, it has a `type`, `target`, `timeStamp` and `phase` properties, each of them allowing a `JSONTerm` as a value. Inside the rule engine, `EventExpression` is matched with DOM Events at their occurrence (we allow only atomic events with no duration i.e. the events are consumed immediately when they occur) time. Consuming events yields into event-facts creation in the WM and such facts are immediately consumed. The engine itself is event-based i.e. all internal processes and activities are event-driven. This is imposed by the environment where the engine runs - the Web browser. Generally, the engine consumes events, through its Working Memory, check conditions and execute actions.

A JSON Rule may contain a list of conditions, logically interpreted as a conjunction of atoms. The language provides the following types of atoms:
`Description`, `JavaScriptBooleanExpression`, `XPathCondition`, `NodeEquality` and `Negation`. The `Description` conditional is similar to Drools [1] pattern. With such conditional property restrictions can be defined as well as property bindings, in a Drools like fashion. `JavaScriptBooleanCondition` stands for JavaScript boolean expression; any JavaScript boolean expression can be used.
The `XPathCondition` is used to test that a DOM Node is found in the list of nodes returned by evaluating an XPath expression. The last two `NodeEquality` and `Negation` are pretty much obvious. They are used to test equality between two terms, respectively to negate a conditional atom.

As usual, a rule has associated a list of actions that have to be performed, if the rule conditions hold. An action is a call to any available JavaScript function. Actions are executed sequentially. The reader may notice that such kind of actions can determine also side effects i.e. more than simple updates on the Working Memory (communication with a server that has no effect on the WM).

As stated above since it is almost mandatory for any rule language to provide means of translation to and from RIF this paper makes the first steps towards

translation from RIF to JSON Rules and vice versa by tackling the problem of built-ins.

2 Built-ins

This section deals with built-ins for the JSON Rules language, along with the architectural and technical aspects imposed by the context (web browsers) and the programming language (JavaScript, ECMAScript [10]) in which the JSON Rules [3] engine is running and has been implemented.

Built-in means constructed as a non-detachable part of a larger structure. In case of rule systems built-ins encapsulate commonly used functionality, provided as predicates or functions. Built-ins also help in maintaining the clean declarative design of rules.

As described in the Conclusion of RIF-UCR [11] *"the goal of the RIF working group is to provide representational interchange formats for processes based on the use of rules and rule-based systems. These formats act as "interlingua" to interchange rules and integrate with other languages, in particular (Semantic) Web mark-up languages."*

JSON Rules foresees compliance with Rule Interchange Format (RIF)[2]. As such the JSON Rules aims to provide the built-ins defined in RIF-DTB [12], beside others that refer to the mashup context.

2.1 RIF built-ins short intro

RIF-DTB [12] is the reference document concerning built-in datatypes, predicates, functions supposed to be supported by RIF dialects such as: RIF Core Dialect [13], RIF Basic Logic Dialect [14] and RIF Production Rules Dialect [8]. According to RIF-DTB document each dialect takes use of a superset or a subset of datatypes, predicates or functions defined in RIF-DTB. The datatypes taken into account by the RIF-DTB [12] are imported either from *W3C XML Schema Definition Language (XSD)* [15] or from *rdf:PlainLiteral: A Datatype for RDF Plain Literals* [16]. Predicates or functions have been ported from *XQuery 1.0 and XPath 2.0 Functions and Operators* [17] or from *rdf:PlainLiteral: A Datatype for RDF Plain Literals* [16].

The list of predicates and function built-ins taken into consideration by RIF-DTB comprises among others: predicates for `literal` comparison, `numeric` functions and predicates, function and predicates on `boolean` values, on `string`, on date/time and duration, on `XMLLiterals`, on `rdf:PlainLiteral`.

2.2 JSON Rules context prerequisites

The general technical guide line that governs the architectural aspects and any other aspects is the compliance and conformity with the ECMAScript standard [10].

[2] `http://www.w3.org/2005/rules/`

Beside the built-ins (predicates and functions) classification introduced in RIF-DTB [12] that span all the RIF dialects, we introduce here a set of built-in actions that serve a different purpose, mainly actions needed in dealing with mashups. As introduced in [4] one of the main purposes of JSON Rules is to be an execution language for mashups.

Although we are not going to address here the problem of translating RIF constants' names, symbols, or namespaces, minimal introduction of ECMAScript concepts are necessary.

One of the most important concept is the *closure* concept. The ECMAScript [10] standard explains a *closure* as a *"function with some arguments already bound to values"*. Others define a closure[3] as *"an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression)"*.

As stated in [18] *"JavaScript's extreme dynamism equips it with tremendous flexibility, and one of the most interesting yet least understood facets of its dynamism involves context"*.

On top of these two concepts (*closure*, *context*) the concept of *namespace* is defined. We understand by the notion of namespace hierarchies of nested objects as defined also in [18].

Another characteristic of JavaScript is that it is weakly typed.

2.3 Guidelines for defining JSON Rules built-ins

There are a couple of general guidelines that should be taken into account when defining functions and predicates, either user defined or built-in:(1) in order to avoid name clashing in JavaScript global context it is recommended to use proper namespaces; (2) predicates should always return a default value, `undefined` value should be avoided.

To simplify the process of creating namespaces JavaScript libraries such as Dojo[4] could be used.

Because the `function` word is reserved in JavaScript its usage should be avoided in a different context other than defining a JavaScript function.

Another important issue that must be taken into account is that the JavaScript code for a page runs in a common context, and as such any redefinition of the same object will override the initial implementation.

2.4 From RIF-DTB to JSON Rules built-ins

This section explains how RIF built-in datatypes, predicates and functions are translated into JSON Rules, respectively JavaScript.

In order to preserve the semantics type mapping is necessary. Table 1 defines the mapping between RIF built-in datatypes and JavaScript datatypes. E4X [19] datatype is also taken into account. Since JavaScript is weakly typed in most

[3] `http://www.jibbering.com/faq/faq_notes/closures.html`
[4] `http://www.dojotoolkit.org/`

of the cases a custom type must be built. In Table 1 this is emphasized by the `cust. type` notation. The same table specifies for each custom type, the base JavaScript type or types on which the custom type must be based.

Table 1. Mapping of RIF built-in Datatypes to JavaScript Datatypes

RIF datatype	**JavaScript datatype**	**RIF datatype**	**JavaScript datatype**
xs:string	String	xs:nonNegativeInteger	Number, cust. type
xs:normalizedString	String, cust. type	xs:unsignedLong	Number, cust. type
xs:token	String, cust. type	xs:unsignedInt	Number, cust. type
xs:language	String, cust. type	xs:unsignedShort	Number, cust. type
xs:Name	String, cust. type	xs:unsignedByte	Number, cust. type
xs:NCName	String, cust. type	xs:positiveInteger	Number, cust. type
xs:ID	String, cust. type	xs:decimal	Number, cust. type
xs:IDREF	String, cust. type	xs:boolean	Boolean
xs:NMTOKEN	String, cust. type	xs:dateTime	Date
xs:ENTITY	String, cust. type	xs:date	Date, cust. type
xs:NOTATION	String, cust. type	xs:time	Date, cust. type
xs:anyURI	String, cust. type	xs:gYearMonth	Date, cust. type
xs:hexBinary	Array, cust. type	xs:gMonthDay	Date, cust. type
xs:base64Binary	Array, cust. type	xs:gYear	Date, cust. type
xs:float	Number, cust. type	xs:gDay	Date, cust. type
xs:double	Number	xs:gMonth	Date, cust. type
xs:duration	String+Date, cust. type	xs:NMTOKENS	String+Array, cust. type
xs:integer	Number, cust. type	xs:IDREFS	String+Array, cust. type
xs:nonPositiveInteger	Number, cust. type	xs:ENTITIES	String+Array, cust. type
xs:negativeInteger	Number, cust. type	xs:QName	String, cust. type
xs:long	Number, cust. type	xs:anyType	E4X XML object
xs:int	Number, cust. type	rdf:PlainLiteral	String
xs:short	Number, cust. type	rdf:XMLLiteral	E4X XML object
xs:byte	Number, cust. type		

As previously stated datatypes as well as functions and predicates must be grouped by means of proper namespaces. In the RIF-DTB case datatypes are identified by the following namespace: `http://www.w3.org/2001/XMLSchema#`. As convention this gets translated into `org.w3c.xs`. For example the `xs:duration` datatype is identified with: `http://www.w3.org/2001/XMLSchema#duration`. This one gets translated into `org.w3c.xs.Duration`. The namespace for RDF datatypes is `org.w3c.rdf`.

In a similar way the namespaces for the RIF predicates and functions are: `org.w3c.rif.pred` and respectively `org.w3c.rif.func`.

The implementation for custom datatypes is inspired from J2EE implementation (i.e. `javax.xml.datatype.Duration`). The `Duration` custom type provides the following list of methods: `add(org.w3c.xs.Duration rhs)`; `addTo(Date date)`; `compare(org.w3c.xs.Duration duration)`; `getDays()`; `getHours()`; `getMinutes()`; `getMonths()`; `getSeconds()`; `getSign()`; `getYears()`; `isLongerThan(org.w3c.xs.Duration duration)`; `isShorterThan(org.w3c.xs.Duration duration)`; `negate()`.

In the RIF-DTB context built-in predicates and functions are defined with the following artifacts: (1) name of the built-in; (2) external schema of the built-in (the signature of the built-in); (3) for a RIF built-in function - how it maps its arguments into a result - in RIF terms this means the mapping of $I_{external}(\sigma)$ in the formal semantics of [20] and [14]; (4) for a RIF built-in predicate - how it gives the truth value when arguments are substituted with values from the domain - this corresponds to the mapping $I_{truth} \circ I_{external}(\sigma)$ in the formal semantics of [20] and [14]; (5) the domains for the built-ins' arguments.

RIF-DTB explains that *"typically, built-in functions and predicates are defined over the value spaces of appropriate datatypes, i.e. the domains of the arguments. When an argument falls outside of its domain, it is understood as an error."*

In RIF-DTB the predicate evaluating greater than is defined as follows:

name pred:numeric-greater-than
schema $(?arg_1 ?arg_2; pred : numeric-greater-than(?arg_1 ?arg_2))$
domains xs:integer, xs:double, xs:float, or xs:decimal for both arguments
mapping When both a_1 and a_2 belong to their domains
$I_{truth} \circ I_{external}(?arg_1 ?arg_2; pred : numeric-greater-than(?arg_1 ?arg_2))(a_1 a_2) =$ t if and only if $op : numeric-greater-than(a_1, a_2)$ returns true, as defined in [17], f otherwise. Also RIF-DTB specifies that *"if an argument value is outside of its domain, the truth value of the function is left unspecified."*

The numeric-greater-than predicate could be implemented in JSON Rules as depicted in Example 1.

Since the hyphen based notation used by RIF in the predicates and function names can not be used in JavaScript the camel hump notation should be used instead. As such the name `numeric-greater-than` gets translated into `numericGreaterThan`.

The mapping of the signature is obvious, with respect to the list of arguments.

Based on Table 1 `xs:integer`, `xs:double`, `xs:float`, or `xs:decimal` RIF-DTB datatypes are translated into custom types but based on the JavaScript `number` datatype.

With respect to RIF note that if the type of any of the arguments is outside of the specified domain then no result should be given. This situation can be dealt at least in three ways: (1) the value returned could be the JavaScript special value `undefined`; (2) JavaScript isNaN() function could be used and in case of true the `Number.NaN` should be returned; (3) an exception could be raised.

However since the RIF specification states that this predicate should behave as defined in [17], in case of a NaN value, false is returned. As such this approach can not be used here. In other cases any of the three approaches could be used.

Example 1 (Defining a built-in predicate).

```
org=new function(){};
org.w3c=new function(){};
org.w3c.rif=new function(){};
org.w3c.rif.pred=new function(){};
org.w3c.rif.pred.numericGreaterThan=function(arg1,arg2) {
    try{
        if (isNaN(arg1) || isNaN(arg2))
            return false;
        if (arg1>arg2)
            return true;
            else
                return false;
    }catch(e){
        //log error
        //console.log(e); i.e. if Firebug is used
    }
    return false;
}
```

This approach could be used in general to map all RIF built-in predicates and functions.

In addition the approach could be used to define libraries of actions oriented towards usage of well known services such Twitter[5], Facebook[6], Youtube[7] and so forth through their APIs.

2.5 JSON Rules built-ins for mashups

In mashups case there could be identified a set of operations that happen regularly, such as loading of services, memory cleaning etc. These type of functionality is provided under the `org.jsonrules.builtin.mashups.system` namespace. Functionality such as `load` or `memoryCleanUp` is provided as built-in functions the named namespace.

The existence of services is a prerequisite for the mashups, and as such they have to be available for the mashup, when this is instantiated.

Recall that we have in mind mashups modeled with JSON Rules that run in the browser [4]. All the involved services interact with each other in a common context which is the *choreographer* page. The choreographer is accessible to the user through a web browser.

[5] `http://twitter.com/`
[6] `http://www.facebook.com/`
[7] `http://www.youtube.com/`

However firstly they need to be made available to the choreographer. The browser loads the content of the choreographer and a DOM Basic Event `load` it is raised. These functions should be used in general in relation with a `load` event, because its main purpose is to make available for the choreographer the necessary services.

The term services here has a broader understanding and comprises at least the following: web page, RPC service, AJAX object.

The signature of `org.jsonrules.builtins.mashups.load` function is: `org.jsonrules.builtins.mashups.load($what,$where)`. In RIF-DTB terminology the *signature of a function* is the `schema of a function`.

The `$what` argument refers to what needs to be loaded. this could be an URI (Uniform Resource Identifier) or it could be a reference to an AJAX object.

$where argument refers to the place where the service or the response of an AJAX request will be stored. This could be for example an `iframe`, a `div`, both of them identified by an id, a reference to a JavaScript object.

3 Conclusions

This paper touched the problem of built-ins for JSON Rules in relation with RIF. This is a step towards translation JSON Rules to and from RIF. RIF built-in datatypes, predicates and functions have been taken into consideration, as well as other type of built-ins that in principal are useful in the mashups context. To maintain similar behavior functionality grouped, namespaces have been suggested for RIF-DTB and mashups environments. Beside all these technical aspects necessary for built-ins definition in the context of JSON Rules has been taken into account.

References

1. Proctor, M., Neale, M., Frandsen, M., Jr., S.G., Tirelli, E., Meyer, F., Verlaenen, K.: Drools 4.0.7. `http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html_single/index.html` (May 2008)
2. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM **42** (1995) 741–843
3. Giurca, A., Pascalau, E.: JSON Rules. In: Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE 2008. Volume 425., CEUR Workshop Proceedings (2008) 7–18
4. Pascalau, E., Giurca, A.: A Rule-Based Approach of Creating and Executing Mashups. In: Proceedings of the 9th IFIP Conference on e-Business, e-Services, and e-Society (I3E 2009). LNCS, Springer (2009) 82–95 forthcoming.
5. Adida, B., Birbeck, M.: RDFa Primer Bridging the Human and Data Webs. W3C Working Draft (October 2008) `http://www.w3.org/TR/xhtml-rdfa-primer/`.
6. Hors, A.L., Hegaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation (April 2004) `http://www.w3.org/TR/DOM-Level-3-Core/`.

7. OMG: Production Rule Representation (PRR), Beta 1. Technical report, OMG (November 2007)
8. de Sainte Marie, C., Paschke, A., Hallmark, G.: RIF Production Rule Dialect. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-prd/.
9. Hohrmann, B., Hegaret, P.L., Pixley, T.: Document Object Model (DOM) Level 3 Events. Technical report, W3C (December 2007)
10. ECMA: ECMAScript Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (December 1999)
11. Paschke, A., Hirtle, D., Ginsberg, A., Patranjan, P.L., McCabe, F.: RIF Use Cases and Requirements. W3C Working Draft (December 2008) http://www.w3.org/TR/rif-ucr/.
12. Polleres, A., Boley, H., Kifer, M.: RIF Datatypes and Built-Ins 1.0. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-dtb/.
13. Boley, H., Hallmark, G., Kifer, M., Paschke, A., Polleres, A., Reynolds, D.: RIF Core Dialect. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-core/.
14. Boley, H., Kifer, M.: RIF Basic Logic Dialect. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-bld/.
15. Peterson, D., Gao, S.S., Malhotra, A., Sperberg-McQueen, C.M., Thompson, H.S., Biron, P.V.: W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. W3C Candidate Recommendation (April 2009) http://www.w3.org/TR/2009/CR-xmlschema11-2-20090430/.
16. Bao, J., Hawke, S., Motik, B., Patel-Schneider, P.F., Polleres, A.: rdf:PlainLiteral: A Datatype for RDF Plain Literals. W3C Candidate Recommendation (June 2009) http://www.w3.org/TR/rdf-plain-literal/.
17. Malhotra, A., Melton, J., Walsh, N.: XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation (January 2007) http://www.w3.org/TR/xpath-functions/.
18. Russell, M.A.: Dojo The Definitive Guide. O'REILLY (2008)
19. ECMA: ECMAScript for XML (E4X) Specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf (December 2005)
20. Boley, H., Kifer, M.: RIF Framework for Logic Dialects. W3C Working Draft (July 2009) http://www.w3.org/TR/rif-fld/.