Java JSON Tools

version 1.3

Bruno Ranschaert

Your feedback is very important to me. You are welcome to send me your remarks or suggestion so that I can improve the library.
mailto://nospam@sdi-consulting.com

Table of Contents

1. Introduction

1.1. Introduction

JSON (JavaScript Object Notation) is a file format to represent data. It is similar to XML but has different characteristics. It is suited to represent configuration information, implement communication protocols and so on. XML is more suited to represent annotated documents. JSON parsing is very fast, the parser can be kept lean and mean. It is easy for humans to read and write. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language. The format is specified on http://www.json.org/, for the details please visit this site.

JSON is a very simple format. As a result, the parsing and rendering is fast and easy, you can concentrate on the content of the file in stead of the format. In XML it is often difficult to fully understand all features (e.g. name spaces, validation, ...). As a result, XML tends to become part of the problem i.s.o. the solution. In JSON everything is well defined, all aspects of the representation are clear, you can concentrate on how you are going to represent your application concepts.

The following example comes from the JSON example page http://www.json.org/example.html:

```
{
   "widget" :
      {
         "debug" : "on",
         "text" :
            {
               "onMouseUp" : "sun1.opacity = (sun1.opacity / 100) * 90;",
               "hOffset" : 250,
               "data" : "Click Here",
               "alignment" : "center",
               "style" : "bold",
               "size" : 36,
               "name" : "text1",
               "vOffset" : 100
            },
         "image" :
            {
               "hOffset" : 250,
               "alignment" : "center",
               "src" : "Images/Sun.png",
               "name" : "sun1",
               "vOffset" : 250
            },
         "window" :
            {
               "width" : 500,
```

```
               "height" : 500,
               "title" : "Sample Konfabulator Widget",
               "name" : "main_window"
            }
      }
}
```

This project wants to provide the tools to manipulate and use the format in a Java application.

1.2. Dependencies

The parser uses ANTLR 2.7.6, so the ANTLR runtime is needed for this. It might work with other versions, I simply did not test it. The project is based on the maven2 build system.

The JSON Tools libraries are written using the new language features from JDK 1.5. Enumerations and generices are used because these make the code nicer to read. There are no dependencies to the new libraries. If you want to use the libraries for an earlier version of the JDK, the retrotranslator tool might be an option (http://retrotranslator.sourceforge.net).

1.3. License

The library is released under the LGPL. You are free to use it for commercial or non-commercial applications as long as you leave the copyright intact and add a reference to the project. Let me know what you like and what you don't like about the library so that I can improve it.

```
    JSONTOOLS - Java JSON Tools
    Copyright (C) 2006 S.D.I.-Consulting BVBA
    http://www.sdi-consulting.com
    mailto://nospam@sdi-consulting.com

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
```

1.4. About S.D.I-Consulting

Visit my website http://www.sdi-consulting.com.
Visit the project website: http://jsontools.sdicons.com, the project is hosted on the Berlios service at http://jsontools.berlios.de.

1.5. Extensions to the JSON format

Comments. I added line comments which start with '#'.It is easier for the examples to be able to put comments in the file. The comments are not retained, they are skipped and ignored.

2. The Core Tools

2.1. Parsing - Reading JSON

The most important tool in the tool set is the parser, it enables you to convert a JSON file into a Java model. All JSON objects remember the position in the file (line, column), so if you are doing post processing of the data you can always refer to the position in the original file.

Invoking the parser is very simple:

```
JSONParser lParser = new JSONParser(JSONTest.class.getResourceAsStream("/config.json"));
JSONValue lValue = lParser.nextValue();
```

The JSON model is a hierarchy of types, the hierarchy looks like this:

```
JSONValue
   JSONComplex
      JSONObject
      JSONArray
   JSONSimple
      JSONNull
      JSONBoolean
      JSONString
      JSONNumber
         JSONInteger
         JSonDecimal
```

2.2. Rendering - Writing JSON

The classes in the JSON model can render themselves to a String. You can choose to render to a pretty form, nicely indented and easily readable by humans, or you can render to a compact form, no spaces or indentations are provided. This is suited to use on a communications channel when you are implementing a communication protocol.

In the introduction we already saw a pretty rendering of some widget data. The same structure can be rendered without pretty printing in order to reduce whitespace. This can be an interesting feature when space optimization is very important, e.g. communication protocols.

```
{"widget":{"debug":"on","text":{"onMouseUp":"sun1.opacity = (sun1.opacity / 100) *
90;","hOffset":250,"data":"Click
Here","alignment":"center","style":"bold","size":36,"name":"text1","vOffset":100},"image"
:{"hOffset":250,"alignment":"center","src":"Images/Sun.png","name":"sun1","vOffset":250},
"window":{"width":500,"height":500,"title":"Sample Konfabulator
Widget","name":"main_window"}}}
```

2.3. Java Serialization

This tool enables you to render POJO's to a JSON file. It is similar to the XML serialization in Java or the XML Stream library, but it uses the JSON format. The result is a very fast text serialization, you can customize it if you want. The code is based on the SISE project, it was adjusted to make use of and benefit from the JSON format.

Marshalling (converting from Java to JSON) as well as unmarshalling is very straightforward:

```
myTestObject = ...
Marshall marshall = new MarshallImpl();
JSONObject result = marshall.marshall(myTestObject);
```

And the other way around:

```
JSONObject myJSONObject = ...
MarshallValue lResult = marshall.unmarshall(myJSONObject);
... = lResult.getReference()
```

You might wonder what the MarshallValue is all about, why is unmarshalling giving an extra object back? The answer is that we went to great lengths to provide marshalling/unmarshalling for both Java reference types as Java basic types. A basic type needs to be fetched using specific methods (there is no other way). In order to provide these specific methods we need an extra class.

2.3.1. Primitive types

Primitive types are represented like this.

```
{
   ">" : "P",
   "=" : "1",
   "t" : "int"
}
```

The **">"** attribute with value **"P"** indicates a primitive type. The **"="** attribute contains the representation of the value and the **"t"** attribute contains the original java type.

2.3.2. Reference types

An array is defined recursively like this. We can see the **">"** attribute this time with the **"A"** value, indicating that the object represents an array. The **"C"** attribute contains the type representation for arrays as it is defined in java. The "=" attribute contains a list of the values.

```
{
   ">" : "A",
   "c" : "I",
   "=" :
      [
         {
            ">" : "P",
            "=" : "0",
            "t" : "int"
         },
         {
            ">" : "P",
            "=" : "1",
            "t" : "int"
         },
         {
            ">" : "P",
            "=" : "2",
            "t" : "int"
         },
         {
            ">" : "P",
            "=" : "3",
            "t" : "int"
         },
         {
            ">" : "P",
            "=" : "4",
            "t" : "int"
         },
         {
            ">" : "P",
            "=" : "5",
            "t" : "int"
         }
      ]
}
```

An object is represented like this.

```
{
   ">" : "O",
   "c" : "com.sdicons.json.serializer.MyBean",
   "&" : "id0",
   "=" :
      {
         "int2" :
            {
               ">" : "null"
            },
         "ptr" :
            {
               ">" : "R",
```

```
            "*" : "id0"
         },
      "name" :
         {
            ">" : "O",
            "c" : "java.lang.String",
            "&" : "id2",
            "=" : "This is a test..."
         },
      "int1" :
         {
            ">" : "null"
         },
      "id" :
         {
            ">" : "O",
            "c" : "java.lang.Integer",
            "&" : "id1",
            "=" : "1003"
         }
   }
}
```

The **">"** marker contains **"O"** for object this time. The "**C"** attribute contains a fully qualified class name. The **"&"** contains a unique id, it can be used to refer to the object so that we are able to represent recursive data structures. The **"="** attribute contains a JSON object having a property for each Java Bean property. The property value is recursively a representation of a Java object.

Note that there is a special notation to represent java null values.

```
{
   ">" : "null"
}
```

Also note that you can refer to other objects with the reference object which looks like this:

```
{
   ">" : "R",
   "*" : "id0"
}
```

2.4. Validation

This tool enables you to validate your JSON files. You can specify which content you expect, the validator can check these constraints for you. The system is straightforward to use and extend. You can add your own rules if you have specific needs. The validation definition is in JSON - as you would expect.

Built-in rules:

```
{
   "name" : "Some rule name",
   “type” : “<built-in-type>”
```

```
   ---
}
```

A validation document consists of a validation rule. This rule will be applied to the JSONValue that has to be validated. The validation rules can be nested, so it is possible to create complex rules out of simpler ones. The **"type"** attribute is obligatory. The **"name"** is optional, it will be used during error reporting and for re-use. The predefined rules are listed below. The name can come in handy while debugging. The name of the failing validation will be available in the exception. If you give each rule its own name or number, you can quickly find out on which predicate the validation fails.

Here is an example of how you can create a validator.

```
// First we create a parser to read the validator specification which is
// defined using the (what did you think) JSON format.
// The validator definition is located in the "my-validator.json"  resource in the
// classpath.
JSONParser lParser = new JSONParser(MyClass.class.getResourceAsStream("my-validator.json"));

// We parse the validator spec and convert it into a Java representation.
JSONObject lValidatorObject = (JSONObject) lParser.nextValue();

// Finally we can convert our validator using the Java model.
Validator lValidator = new JSONValidator(lValidatorObject);
```

And now that you have the validator, you can start validating your data.

```
// First we create a parser to read the data.
JSONParser lParser = new JSONParser(MyClass.class.getResourceAsStream("data.json"));

// We parse the datafile and convert it into a Java representation.
JSONValue lMyData = lParser.nextValue();

// Now we can use the validtor to check on our data. We can test if the data has the
// correct format or not.
lValidator.validate(lMyData);
```

2.4.1. Basic rules

These rules are the basic rules in boolean logic.

2.4.1.1. "type" : "true"

Parameters: -
Description: This rule always succeeds.
Example: A validator that wil succeed on all JSON data structures.

```
{
   "name" :"This validator validates *everything*",
   "type" :"true"
}
```

2.4.1.2. "type":"false"

Parameters: -
Description: This rule always fails.
Example: A validator that rejects all data structures.

```
{
   "name" :"This validator rejects all",
   "type" :"false"
}
```

2.4.1.3. "type":"and"

Parameters:

- "rules" : array of nested rules.

Description: All nested rules have to hold for the and rule to succeed.
Example: A validator that succeeds if the object under scrutiny is both a list and has content consisting of integers.

```
{
   "name" :"List of integers",
   "type" :"and",
   "rules" : [{"type":"array"},{"type":"content","rule":{"type":"int"}}]
}
```

2.4.1.4. "type":"or"

Parameters:

- "rules" : array of nested rules.

Description: One of the nested rules has to succeed for this rule to succeed.
Example: A validator that validates null or integers.

```
{
   "name" :"Null or int",
   "type" :"or",
   "rules" : [{"type":"int"},{"type":"bool"}]
}
```

2.4.1.5. "type":"not"

Parameters:

- "rule": A single nested rule.

Description: The rule succeeds if the nested rule fails and vice versa.

2.4.2. Type rules

These rules are predefined rules which allows you to specify type restrictions on the JSON data elements. The meaning of these predicates is obvious, they will not be discussed. See the examples for more information.

2.4.2.1. "type":"complex", "type":"array", "type":"object", "type":"simple", "type":"null", "type":"bool", "type":"string", "type":"number", "type":"int", "type":"decimal"

2.4.3. Attribute rules

These rules check for attributes of certain types.

2.4.3.1. "type" : "length"

Parameters:

- "min": (optional) The minimal length of the array.
- "max": (optional) The maximal length of the array.

Description: Applicable to complex objects and string objects. The rule will fail if the object under investigation has another type. For array objects the number of elements is counted, for objects the number of properties and for strings, the length of its value in Java (not the JSON representation; "\n" in the file counts as a single character).

Example: A validator that only wants arrays of length 5.

```
{
   "name"  :"Array of length 5",
   "type"  :"and",
   "rules" : [{"type":"array"}, {"type":"length","min":5,"max":5}]
}
```

2.4.3.2. "type" : "range"

Parameters:

- "min": (optional) The minimal value.
- "max": (optional) The maximal value.

Description: Applicable to JSONNumbers, i.e. JSONInteger and JSONDecimal.

Example: Allow numbers between 50 and 100.

```
{
   "name" :"Range validator",
   "type" :"range",
   "min" : 50,
   "max" : 100
}
```

2.4.3.3. "type":"enum"

Parameters:

- ˝values˝ : An array of JSON values.

Description: The value has to occur in the provided list. The list can contain simple types as well as complex nested types.

Example: An enum validator.

```
{
   "name" :"Enum validator",
   "type" :"enum",
   "values" : [13, 17, "JSON", 123.12, [1, 2, 3], {"key":"value"}]
}
```

2.4.3.4. "type":"regexp"

Parameters:

- ˝pattern˝: A regexp pattern.

Description: For strings, requires a predefined format according to the regular expression.

Example: A validator that validates strings containing a sequence of a's , b's and c's.

```
{
   "name" :"A-B-C validator",
   "type" :"regexp",
   "pattern" : "a*b*c*"
}
```

2.4.3.5. "type":"content"

Parameters:

- ˝rule˝: The rule that specifies how the content of a complex structure – an array or the property values of an object - should behave.

Description: Note that in contrast with the ˝properties˝ rule (for objects), you can specify in a single rule what all property values of an object should look like.

Example: See ˝type˝:˝and˝.

2.4.3.6. "type":"properties"

Parameters:

- ˝pairs˝: A list of key/value pair descriptions. Each description contains three properties: **"key"** the key string; **"optional"** a boolean indicating whether this property is optional or not and **"rule"** a validation rule that should be applied to the properties value. Note that in contrast wit the ˝content˝ rule above you can specify a rule per attribute.

Description: This predicate is only applicable (and only has meaning) on object data structures. It will fail on any other type.

Example:

```
{
   "name" :"Contact spec.",
   "type" :"properties",
   "pairs" : [{"key":"name", "optional":false, "rule":{"type":"string"}},
              {"key":"country", "optional":false, "rule":{"type":"string"}},
              {"key":"salary", "optional":true, "rule":{"type":"decimal"}}
             ]
}
```

It will validate objects looking like this:

```
{"name":"Bruno Ranschaert", "country":"Belgium", "salary":100.0 }
```

2.4.4. Structural rules

2.4.4.1. "type":"ref"

Parameters:

- "*": The name of the rule to invoke.

Description: This rule lets you specify recursive rules. Be careful not to create infinite validations which is quite possible using this rule. The containing rule will be fetched just before validation, there will be no error message during construction when the containing rule is not found. The rule will fail in this case. If there are several rules with the same name, only the last one with that name is remembered and the last one will be used.

Example: A validator that validates nested lists of integers. A ref is needed to enable recursion in the validator.

```
{
   "name" :"Nested list of integers",
   "type" :"and",
   "rules" :
      [
         {"type":"array"},
         {"type":"content",
          "rule":
             {"type" : "or",
              "rules":
                [{"type":"int"},
                 {"type":"ref", "*" : "Nested list of integers"}]
             }}
       ]
}
```

2.4.4.2. "type" : "let"

Parameters:

- "rules" : A list of rules.
- "*" : The name of the rule that should be used.

Description: Lets you specify a number of named rules in advance. It is a

convenience rule that lets you specify a list of global shared validation rules in advance before using these later on. It becomes possible to first define a number of recurring types and then give the starting point. It is a utility rule that lets you tackle more complex validations. Note that it makes no sense to define anonymous rules inside the list, it is impossible to refer to these later on.
Example:

```
{
   "name" :"Let test -  a's or b's",
   "type" :"let",
   "*"    : "start",
   "rules" :
      [{"name":"start", "type":"or", "rules":[{"type":"ref", "*":"a"},
                                              {"type":"ref", "*":"b"}]},
       {"name":"a", "type":"regexp", "pattern":"a*"},
       {"name":"b", "type":"regexp", "pattern":"b*"}
      ]
}
```

1.1.1.1. "type":"custom"

Parameters:

- "class" : The fully qualified class name of the validator.

Description: An instance of this validator will be created and will be given a hash map of validations. A custom validator should be derived from "com.sdicons.json.validator.impl.predicates.CustomValidator".
Example:

```
{
   "name" :"Custom test",
   "type" :"custom",
   "class" : "com.sdicons.json.validator.MyValidator"
}
```

The validator class looks like this:

```
public class MyValidator
extends CustomValidator
{
    public MyValidator(
       String aName, JSONObject aRule,HashMap<String, Validator> aRuleset)
    {
        super(aName, aRule, aRuleset);
    }

    public void validate(JSONValue aValue) throws ValidationException
    {
        // Do whatever you need to do on aValue ...
        // If validation is ok, simply return.
        // If validation fails, you can use:
        // fail(JSONValue aValue) or fail(String aReason, JSONValue aValue)
        // to throw the Validation exception for you.
    }
}
```

1.1.1.2. "type":"switch"

Parameters:

- "key": The key name of the object that will act as the discriminator.
- "case": A list of objects containing the parameters **"values"** and **"rule"**. The first one is a list of values the second one a validator rule.

Description: The switch validator is a convenience one. It is a subset of the or validator, but the problem with the or validator is that it does a bad job for error reporting when things go wrong. The reason is that all rules fail and it is not always clear why, because the reason a rule fails might be some levels deeper. The switch validator selects a validator based on the value of a property encountered in the value being validated. The error produced will be the one of the selected validator. The first applicable validator is used, the following ones are ignored.
Example: The top level rule in the validator for validators contains a switch that could have been described by an or, but the switch gives better error messages.

2.4.5. Example: Validator for validators

This example validator is able to validate validators. The example is a bit contrived because the validators really don't need validation because it is built-in in the construction. It is interesting because it can serve as a definition of how to construct a validator.

```
{
   "name":"Validator validator",
   "type":"let",
   "*":"rule",
   "rules":
   [
      ########## START ##########
      {
         "name":"rule",
         "type":"switch",
         "key":"type",
         "case":
         [
            {"values":["true", "false", "null"], "rule":{"type":"ref","*":"atom-rule"}},
            {"values":["int", "complex", "array", "object", "simple",
                       "null", "bool", "string", "number", "decimal"],
                       "rule":{"type":"ref","*":"type-rule"}},
            {"values":["not", "content"], "rule":{"type":"ref","*":"rules-rule"}},
            {"values":["and", "or"], "rule":{"type":"ref","*":"ruleset-rule"}},
            {"values":["length", "range"], "rule":{"type":"ref","*":"minmax-rule"}},
            {"values":["ref"], "rule":{"type":"ref","*":"ref-rule"}},
            {"values":["custom"], "rule":{"type":"ref","*":"custom-rule"}},
            {"values":["enum"], "rule":{"type":"ref","*":"enum-rule"}},
            {"values":["let"], "rule":{"type":"ref","*":"let-rule"}},
            {"values":["regexp"], "rule":{"type":"ref","*":"regexp-rule"}},
            {"values":["properties"], "rule":{"type":"ref","*":"properties-rule"}},
            {"values":["switch"], "rule": {"type":"ref","*":"switch-rule"}}
         ]
      },
      ########## RULESET ##########
      {
         "name":"ruleset",
         "type":"and",
```

```
      "rules":[{"type":"array"},{"type":"content","rule":{"type":"ref","*":"rule"}}]
   },
   ########## PAIRS ##########
   {
      "name":"pairs",
      "type":"and",
      "rules":[{"type":"array"},{"type":"content","rule":{"type":"ref","*":"pair"}}]
   },
   ########## PAIR ##########
   {
      "name":"pair",
      "type":"properties",
      "pairs" :
       [{"key":"key",      "optional":false, "rule":{"type":"string"}},
        {"key":"optional", "optional":false, "rule":{"type":"bool"}},
        {"key":"rule",     "optional":false, "rule":{"type":"ref","*":"rule"}}
       ]
   },
   ########## CASES ##########
   {
      "name":"cases",
      "type":"and",
      "rules":[{"type":"array"},{"type":"content","rule":{"type":"ref","*":"case"}}]
   },
   ########## CASE ##########
   {
      "name":"case",
      "type":"properties",
      "pairs" :
       [{"key":"values",   "optional":false, "rule":{"type":"array"}},
        {"key":"rule",     "optional":false, "rule":{"type":"ref","*":"rule"}}
       ]
   },
   ########## ATOM ##########
   {
      "name":"atom-rule",
      "type":"properties",
      "pairs" :
       [{"key":"name", "optional":true, "rule":{"type":"string"}},
        {"key":"type", "optional":false, "rule":
                    {"type":"enum","values":["true", "false", "null"]}}
       ]
   },
   ########## RULESET-RULE ##########
   {
      "name":"ruleset-rule",
      "type":"properties",
      "pairs" :
       [{"key":"name",  "optional":true,  "rule":{"type":"string"}},
        {"key":"type",  "optional":false, "rule":{"type":"enum","values":["and", "or"]}},
        {"key":"rules", "optional":false, "rule":{"type":"ref","*":"ruleset"}}
       ]
   },
   ########## RULES-RULE ##########
   {
      "name":"rules-rule",
      "type":"properties",
      "pairs" :
       [{"key":"name", "optional":true,  "rule":{"type":"string"}},
        {"key":"type", "optional":false, "rule":{"type":"enum","values":["not", "content"]}},
        {"key":"rule", "optional":false, "rule":{"type":"ref","*":"rule"}}
       ]
   },
   ########## TYPE ##########
   {
```

```
        "name":"type-rule",
        "type":"properties",
        "pairs" :
         [{"key":"name", "optional":true, "rule":{"type":"string"}},
          {"key":"type", "optional":false, "rule":{"type":"enum",
                   "values":["int", "complex", "array", "object",
                   "simple", "null", "bool", "string", "number",
                   "decimal"]}}
         ]
     },
     ########## MINMAX ##########
     {
        "name":"minmax-rule",
        "type":"properties",
        "pairs" :
         [{"key":"name", "optional":true, "rule":{"type":"string"}},
          {"key":"type", "optional":false, "rule":{"type":"enum","values":["length", "range"]}},
          {"key":"min", "optional":true, "rule":{"type":"number"}},
          {"key":"max", "optional":true, "rule":{"type":"number"}}
         ]
     },
     ########## REF ##########
     {
        "name":"ref-rule",
        "type":"properties",
        "pairs" :
         [{"key":"name", "optional":true, "rule":{"type":"string"}},
          {"key":"type", "optional":false, "rule":{"type":"enum","values":["ref"]}},
          {"key":"*",    "optional":false, "rule":{"type":"string"}}
         ]
     },
     ########## CUSTOM ##########
     {
        "name":"custom-rule",
        "type":"properties",
        "pairs" :
         [{"key":"name", "optional":true, "rule":{"type":"string"}},
          {"key":"type", "optional":false, "rule":{"type":"enum","values":["custom"]}},
          {"key":"class", "optional":true, "rule":{"type":"string"}}
         ]
     },
     ########## ENUM ##########
     {
        "name":"enum-rule",
        "type":"properties",
        "pairs" :
         [{"key":"name", "optional":true, "rule":{"type":"string"}},
          {"key":"type", "optional":false, "rule":{"type":"enum","values":["enum"]}},
          {"key":"values", "optional":true, "rule":{"type":"array"}}
         ]
     },
     ########## LET ##########
     {
        "name":"let-rule",
        "type":"properties",
        "pairs" :
         [{"key":"name",  "optional":true,  "rule":{"type":"string"}},
          {"key":"type",  "optional":false, "rule":{"type":"enum","values":["let"]}},
          {"key":"rules", "optional":false, "rule":{"type":"ref","*":"ruleset"}},
          {"key":"*",     "optional":false, "rule":{"type":"string"}}
         ]
     },
     ########## REGEXP ##########
     {
        "name":"regexp-rule",
```

```
         "type":"properties",
         "pairs" :
          [{"key":"name", "optional":true, "rule":{"type":"string"}},
           {"key":"type", "optional":false, "rule":{"type":"enum","values":["regexp"]}},
           {"key":"pattern", "optional":false, "rule":{"type":"string"}}
          ]
      },
      ########## PROPERTIES ##########
      {
         "name":"properties-rule",
         "type":"properties",
         "pairs" :
          [{"key":"name",  "optional":true, "rule":{"type":"string"}},
           {"key":"type",  "optional":false, "rule":{"type":"enum","values":["properties"]}},
           {"key":"pairs", "optional":false, "rule":{"type":"ref","*":"pairs"}}
          ]
      },
      ########## SWITCH ##########
      {
         "name":"switch-rule",
         "type":"properties",
         "pairs" :
          [{"key":"name",  "optional":true, "rule":{"type":"string"}},
           {"key":"type",  "optional":false, "rule":{"type":"enum","values":["switch"]}},
           {"key":"key",   "optional":false, "rule":{"type":"string"}},
           {"key":"case",  "optional":false, "rule":{"type":"ref","*":"cases"}}
          ]
      }
   ]
}
```

3. Tool Extensions

3.1. Log4J

Is in the jsontools-log4j library. It contains a Log4J layout that will format the messages in JSON format. In combination with the existing appenders this makes it possible to create JSON output which will be easy to parse using the core tools.

4. Future extensions

There are many other applications which could benefit from the JSON format. It will depend on the feedback I receive and my own needs which features will be implemented first. Here are some candidates:

Spring configuration file.

Hibernate mapping files. In a substantial application the current XML files take ages to parse and to load. This process could be sped up by using the JSON format.