**Jayway JsonPath**

**A Java DSL for reading JSON documents.**

JsonPath is available at the Central Maven Repository. Maven users add this to your POM.

<dependency>

<groupId>com.jayway.jsonpath</groupId>

<artifactId>json-path</artifactId>

<version>2.4.0</version>

</dependency>

JsonPath expressions always refer to a JSON structure in the same way as XPath expression are used in combination with an XML document. The "root member object" in JsonPath is always referred to as $ regardless if it is an object or array.

JsonPath expressions can use the dot–notation or the bracket–notation

$.store.book[0].title or $['store']['book'][0]['title']

**Operators**

| **Operator** | **Description** |
| --- | --- |
| $ | The root element to query. This starts all path expressions. |
| @ | The current node being processed by a filter predicate. |
| \* | Wildcard. Available anywhere a name or numeric are required. |
| .. | Deep scan. Available anywhere a name is required. |
| .<name> | Dot-notated child |
| ['<name>' (, '<name>')] | Bracket-notated child or children |
| [<number> (, <number>)] | Array index or indexes |
| [start:end] | Array slice operator |
| [?(<expression>)] | Filter expression. Expression must evaluate to a boolean value. |

**Functions**

Functions can be invoked at the tail end of a path - the input to a function is the output of the path expression. The function output is dictated by the function itself.

| **Function** | **Description** | **Output** |
| --- | --- | --- |
| min() | Provides the min value of an array of numbers | Double |
| max() | Provides the max value of an array of numbers | Double |
| avg() | Provides the average value of an array of numbers | Double |
| stddev() | Provides the standard deviation value of an array of numbers | Double |
| length() | Provides the length of an array | Integer |
| sum() | Provides the sum value of an array of numbers | Double |

**Filter Operators**

Filters are logical expressions used to filter arrays. A typical filter would be [?(@.age > 18)] where @ represents the current item being processed. More complex filters can be created with logical operators && and ||. String literals must be enclosed by single or double quotes ([?(@.color == 'blue')] or [?(@.color == "blue")]).

| **Operator** | **Description** |
| --- | --- |
| == | left is equal to right (note that 1 is not equal to '1') |
| != | left is not equal to right |
| < | left is less than right |
| <= | left is less or equal to right |
| > | left is greater than right |
| >= | left is greater than or equal to right |
| =~ | left matches regular expression [?(@.name =~ /foo.\*?/i)] |
| in | left exists in right [?(@.size in ['S', 'M'])] |
| nin | left does not exists in right |
| subsetof | left is a subset of right [?(@.sizes subsetof ['S', 'M', 'L'])] |
| anyof | left has an intersection with right [?(@.sizes anyof ['M', 'L'])] |
| noneof | left has no intersection with right [?(@.sizes noneof ['M', 'L'])] |
| size | size of left (array or string) should match right |
| empty | left (array or string) should be empty |

**Path Examples**

Given the json

{

"store": {

"book": [

{

"category": "reference",

"author": "Nigel Rees",

"title": "Sayings of the Century",

"price": 8.95

},

{

"category": "fiction",

"author": "Evelyn Waugh",

"title": "Sword of Honour",

"price": 12.99

},

{

"category": "fiction",

"author": "Herman Melville",

"title": "Moby Dick",

"isbn": "0-553-21311-3",

"price": 8.99

},

{

"category": "fiction",

"author": "J. R. R. Tolkien",

"title": "The Lord of the Rings",

"isbn": "0-395-19395-8",

"price": 22.99

}

],

"bicycle": {

"color": "red",

"price": 19.95

}

},

"expensive": 10

}

| **JsonPath (click link to try)** | **Result** |
| --- | --- |
| [$.store.book[\*].author](http://jsonpath.herokuapp.com/?path=$.store.book%5b*%5d.author) | The authors of all books |
| [$..author](http://jsonpath.herokuapp.com/?path=$..author) | All authors |
| [$.store.\*](http://jsonpath.herokuapp.com/?path=$.store.*) | All things, both books and bicycles |
| [$.store..price](http://jsonpath.herokuapp.com/?path=$.store..price) | The price of everything |
| [$..book[2]](http://jsonpath.herokuapp.com/?path=$..book%5b2%5d) | The third book |
| [$..book[-2]](http://jsonpath.herokuapp.com/?path=$..book%5b2%5d) | The second to last book |
| [$..book[0,1]](http://jsonpath.herokuapp.com/?path=$..book%5b0,1%5d) | The first two books |
| [$..book[:2]](http://jsonpath.herokuapp.com/?path=$..book%5b:2%5d) | All books from index 0 (inclusive) until index 2 (exclusive) |
| [$..book[1:2]](http://jsonpath.herokuapp.com/?path=$..book%5b1:2%5d) | All books from index 1 (inclusive) until index 2 (exclusive) |
| [$..book[-2:]](http://jsonpath.herokuapp.com/?path=$..book%5b-2:%5d) | Last two books |
| [$..book[2:]](http://jsonpath.herokuapp.com/?path=$..book%5b2:%5d) | Book number two from tail |
| [$..book[?(@.isbn)]](http://jsonpath.herokuapp.com/?path=$..book%5b?(@.isbn)%5d) | All books with an ISBN number |
| [$.store.book[?(@.price < 10)]](http://jsonpath.herokuapp.com/?path=$.store.book%5b?(@.price%20%3C%2010)%5d) | All books in store cheaper than 10 |
| [$..book[?(@.price <= $['expensive'])]](http://jsonpath.herokuapp.com/?path=$..book%5b?(@.price%20%3C=%20$%5b%27expensive%27%5d)%5d) | All books in store that are not "expensive" |
| [$..book[?(@.author =~ /.\*REES/i)]](http://jsonpath.herokuapp.com/?path=$..book%5b?(@.author%20=~%20/.*REES/i)%5d) | All books matching regex (ignore case) |
| [$..\*](http://jsonpath.herokuapp.com/?path=$..*) | Give me everything |
| [$..book.length()](http://jsonpath.herokuapp.com/?path=$..book.length()) | The number of books |

**Reading a Document**

The simplest most straight forward way to use JsonPath is via the static read API.

String json = "...";

List<String> authors = JsonPath.read(json, "$.store.book[\*].author");

If you only want to read once this is OK. In case you need to read an- other path as well this is not the way to go since the document will be parsed every time you call JsonPath.read(...). To avoid the problem you can parse the json first.

String json = "...";

Object document = Configuration.defaultConfiguration().jsonProvider().parse(json);

String author0 = JsonPath.read(document, "$.store.book[0].author");

String author1 = JsonPath.read(document, "$.store.book[1].author");

JsonPath also provides a fluent API. This is also the most flexible one.

String json = "...";

ReadContext ctx = JsonPath.parse(json);

List<String> authorsOfBooksWithISBN = ctx.read("$.store.book[?(@.isbn)].author");

List<Map<String, Object>> expensiveBooks = JsonPath

.using(configuration)

.parse(json)

.read("$.store.book[?(@.price > 10)]", List.class);

**What is Returned When?**

When using JsonPath in java its important to know what type you expect in your result. JsonPath will automatically try to cast the result to the type expected by the invoker.

//Will throw an java.lang.ClassCastException

List<String> list = JsonPath.parse(json).read("$.store.book[0].author")

//Works fine

String author = JsonPath.parse(json).read("$.store.book[0].author")

When evaluating a path you need to understand the concept of when a path is definite. A path is indefinite if it contains:

* .. - a deep scan operator
* ?(<expression>) - an expression
* [<number>, <number> (, <number>)] - multiple array indexes

Indefinite paths always returns a list (as represented by current JsonProvider).

By default a simple object mapper is provided by the MappingProvider SPI. This allows you to specify the return type you want and the MappingProvider will try to perform the mapping. In the example below mapping between Long and Date is demonstrated.

String json = "{\"date\_as\_long\" : 1411455611975}";

Date date = JsonPath.parse(json).read("$['date\_as\_long']", Date.class);

If you configure JsonPath to use JacksonMappingProvider or GsonMappingProvider you can even map your JsonPath output directly into POJO's.

Book book = JsonPath.parse(json).read("$.store.book[0]", Book.class);

To obtainin full generics type information, use TypeRef.

TypeRef<List<String>> typeRef = new TypeRef<List<String>>() {};

List<String> titles = JsonPath.parse(JSON\_DOCUMENT).read("$.store.book[\*].title", typeRef);

**Predicates**

There are three different ways to create filter predicates in JsonPath.

**Inline Predicates**

Inline predicates are the ones defined in the path.

List<Map<String, Object>> books = JsonPath.parse(json).read("$.store.book[?(@.price < 10)]");

You can use && and || to combine multiple predicates [?(@.price < 10 && @.category == 'fiction')] , [?(@.category == 'reference' || @.price > 10)].

You can use ! to negate a predicate [?(!(@.price < 10 && @.category == 'fiction'))].

**Filter Predicates**

Predicates can be built using the Filter API as shown below:

import static com.jayway.jsonpath.JsonPath.parse;

import static com.jayway.jsonpath.Criteria.where;

import static com.jayway.jsonpath.Filter.filter;

...

...

Filter cheapFictionFilter = filter(

where("category").is("fiction").and("price").lte(10D)

);

List<Map<String, Object>> books =

parse(json).read("$.store.book[?]", cheapFictionFilter);

Notice the placeholder ? for the filter in the path. When multiple filters are provided they are applied in order where the number of placeholders must match the number of provided filters. You can specify multiple predicate placeholders in one filter operation [?, ?], both predicates must match.

Filters can also be combined with 'OR' and 'AND'

Filter fooOrBar = filter(

where("foo").exists(true)).or(where("bar").exists(true)

);

Filter fooAndBar = filter(

where("foo").exists(true)).and(where("bar").exists(true)

);

**Roll Your Own**

Third option is to implement your own predicates

Predicate booksWithISBN = new Predicate() {

@Override

public boolean apply(PredicateContext ctx) {

return ctx.item(Map.class).containsKey("isbn");

}

};

List<Map<String, Object>> books =

reader.read("$.store.book[?].isbn", List.class, booksWithISBN);

**Path vs Value**

In the Goessner implementation a JsonPath can return either Path or Value. Value is the default and what all the examples above are returning. If you rather have the path of the elements our query is hitting this can be achieved with an option.

Configuration conf = Configuration.builder()

.options(Option.AS\_PATH\_LIST).build();

List<String> pathList = using(conf).parse(json).read("$..author");

assertThat(pathList).containsExactly(

"$['store']['book'][0]['author']",

"$['store']['book'][1]['author']",

"$['store']['book'][2]['author']",

"$['store']['book'][3]['author']");

**Tweaking Configuration**

**Options**

When creating your Configuration there are a few option flags that can alter the default behaviour.

**DEFAULT\_PATH\_LEAF\_TO\_NULL**  
This option makes JsonPath return null for missing leafs. Consider the following json

[

{

"name" : "john",

"gender" : "male"

},

{

"name" : "ben"

}

]

Configuration conf = Configuration.defaultConfiguration();

//Works fine

String gender0 = JsonPath.using(conf).parse(json).read("$[0]['gender']");

//PathNotFoundException thrown

String gender1 = JsonPath.using(conf).parse(json).read("$[1]['gender']");

Configuration conf2 = conf.addOptions(Option.DEFAULT\_PATH\_LEAF\_TO\_NULL);

//Works fine

String gender0 = JsonPath.using(conf2).parse(json).read("$[0]['gender']");

//Works fine (null is returned)

String gender1 = JsonPath.using(conf2).parse(json).read("$[1]['gender']");

**ALWAYS\_RETURN\_LIST**  
This option configures JsonPath to return a list even when the path is definite.

Configuration conf = Configuration.defaultConfiguration();

//Works fine

List<String> genders0 = JsonPath.using(conf).parse(json).read("$[0]['gender']");

//PathNotFoundException thrown

List<String> genders1 = JsonPath.using(conf).parse(json).read("$[1]['gender']");

**SUPPRESS\_EXCEPTIONS**  
This option makes sure no exceptions are propagated from path evaluation. It follows these simple rules:

* If option ALWAYS\_RETURN\_LIST is present an empty list will be returned
* If option ALWAYS\_RETURN\_LIST is **NOT** present null returned

**JsonProvider SPI**

JsonPath is shipped with five different JsonProviders:

* [JsonSmartJsonProvider](https://code.google.com/p/json-smart/) (default)
* [JacksonJsonProvider](https://github.com/FasterXML/jackson)
* [JacksonJsonNodeJsonProvider](https://github.com/FasterXML/jackson)
* [GsonJsonProvider](https://code.google.com/p/google-gson/)
* [JsonOrgJsonProvider](http://www.json.org/java/index.html)

Changing the configuration defaults as demonstrated should only be done when your application is being initialized. Changes during runtime is strongly discouraged, especially in multi threaded applications.

Configuration.setDefaults(new Configuration.Defaults() {

private final JsonProvider jsonProvider = new JacksonJsonProvider();

private final MappingProvider mappingProvider = new JacksonMappingProvider();

@Override

public JsonProvider jsonProvider() {

return jsonProvider;

}

@Override

public MappingProvider mappingProvider() {

return mappingProvider;

}

@Override

public Set<Option> options() {

return EnumSet.noneOf(Option.class);

}

});

Note that the JacksonJsonProvider requires com.fasterxml.jackson.core:jackson-databind:2.4.5 and the GsonJsonProvider requires com.google.code.gson:gson:2.3.1 on your classpath.

**Cache SPI**

In JsonPath 2.1.0 a new Cache SPI was introduced. This allows API consumers to configure path caching in a way that suits their needs. The cache must be configured before it is accesses for the first time or a JsonPathException is thrown. JsonPath ships with two cache implementations

* com.jayway.jsonpath.spi.cache.LRUCache (default, thread safe)
* com.jayway.jsonpath.spi.cache.NOOPCache (no cache)

If you want to implement your own cache the API is simple.

CacheProvider.setCache(new Cache() {

//Not thread safe simple cache

private Map<String, JsonPath> map = new HashMap<String, JsonPath>();

@Override

public JsonPath get(String key) {

return map.get(key);

}

@Override

public void put(String key, JsonPath jsonPath) {

map.put(key, jsonPath);

}

});