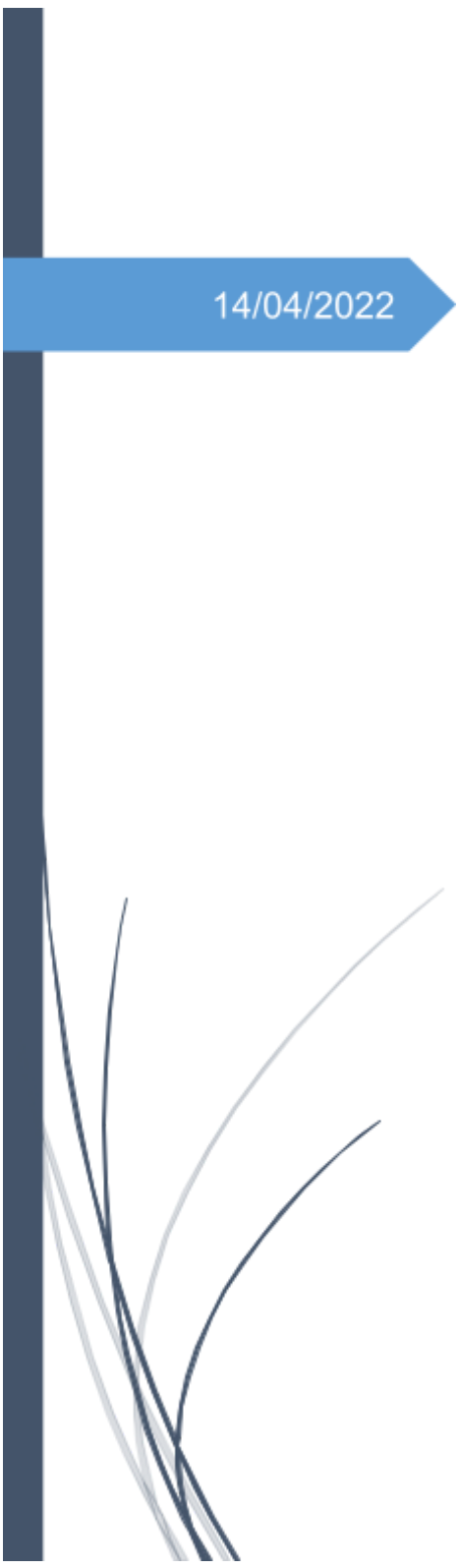
JSON-TAB Format

Presentation

14/04/2022



Environmental Sensing

CC BY SA

Table of Contents

1 Introduction

The JSON-TAB format is applicable to any tabular, indexed or multi-dimensional data.

This format makes it possible to:

- integrate data of a high semantic level,
- interoperability with all JSON parsers,
- avoid data duplication,
- reduce the size of data,
- integrate meta-data

This format is an application of the JSON-NTV format.

A binary version is also defined (Appendix) with CBOR format (RFC 8949)

1.1 Conventions used

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The grammatical rules in this document are to be interpreted as described in [RFC5234].

1.2 Terminology

The terms Json-Text, Json-Value (Value), Object, Member, Element, Array, Number, String, False, Null, True are defined in the JSON grammar.

The terms Geometry-type, Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon, GeometryCollection, GeoJSON-Types are defined in GeoJSON grammar.

Timestamp is defined in Date and Time format.

1.3 Rules

Values in Array are ordered and independent from the other Values.

Members in Objects are not ordered.

2 OBJECTIVES

2.1 WHY A NEW TEXTUAL STANDARD FOR TABULAR DATA ?

The main operational standard used to exchange textual tabular data is CSV format (RFC 4180).

Unfortunately CSV format is obsolete (last revision in 2005) and current CSV tools do not comply with the standard.

2.2 KEY DESIGN FEATURES

The format's focus is on simplicity, lightness and web usage.

The key features of this format are the following:

- JSON as the base format
  - JSON is simple and readable as simple text
  - JSON supports rich structure including nesting and basic types
  - JSON is web-native and very widely used and supported
  - JSON format has binary representation (i.e. CBOR format)
- column-oriented structure (instead of row-oriented)
  - columns carry the semantics of the data
  - columns can be optimized based on their inter-relationships
- several representations available
  - four levels, from the simplest to the most optimized, are available
  - meta-data (header or schema) can be integrate
- high semantic level of data
  - all JSON representations can be included following JSON-NTV format

3 TABULAR DATA REPRESENTATION

3.1 TABULAR DATA

Tabular data *is data that is structured into rows, each of which contains information about some things. Each row contains the same number of cells (although some of these cells may be empty), which provide values of properties of the thing described by the row. In tabular data, cells within the same column provide values for the same property of the things described by each row. This is what differentiates tabular data from other line-oriented formats. (Model for Tabular Data and Metadata on the Web - W3C - Recommendation 17 december 2015)*

In tabular data, column and row are not equivalent, the columns (or fields) represent the 'semantics' of the data and the rows represent the objects arranged according to the structure defined by the columns.

If we now observe how tabular data are used, we can identify four main uses:

- **association**: this consists of coupling each value of a field to a value of another field (“coupled” relationship between two fields),
- **classification**: This involves grouping the data by category in order to be able to make a statistical use of it, for example (“derived” relationship between two fields),
- **crossing**: This consists of representing all the combinations between several fields, such as in matrix representations (“crossed” relationship between several fields),
- **characterization**: It corresponds to the documentation of defined properties (no specific relationship).

Example :

id	produit	aliment	contenant	quantité	prix	validité	disponibilité
11	pomme	fruit	sachet	1 kg	1	du 1/7/2022 au 31/12/2022	oui
12	pomme	fruit	carton	10 kg	9	du 1/7/2022 au 31/12/2022	oui
13	orange	fruit	sachet	1 kg	2	du 1/7/2022 au 31/12/2022	fin 2022
14	orange	fruit	carton	10 kg	18	du 1/7/2022 au 31/12/2022	fin 2022
15	piment	légume	sachet	1 kg	1.5	du 1/7/2022 au 31/12/2022	fin 2022
16	piment	légume	carton	10 kg	13	du 1/7/2022 au 31/12/2022	fin 2022
17	banane	fruit	sachet	1 kg	0.5	du 1/7/2022 au 31/12/2022	oui
18	banane	fruit	carton	10 kg	4	du 1/7/2022 au 31/12/2022	oui

This is a price list of different foods based on packaging for the year 2022.

We find here:

- *association: between "contenant" and "quantité",*
- *classification: between "produit" and "aliment",*
- *crossing: between "produit" and "quantité",*
- *characterization: between "produit" and "disponibilité"*

3.2 Representation of fields

A tabular object can be represented by a simple list of fields where a field represents a column. Each field has the same number of values (the length of the tabular object).

A field can be represented by several formats:

Full format :

A field is associated in the JSON standard with an Array entity.

The simplest JSON representation of a field is therefore an Array

Example (field "prix") :

[1, 9, 2, 18, 1.5, 13, 0.5, 4]

Complete format :

The "Full format" representation has the disadvantage of being bulky when data is duplicated in a field.

The second format is to represent a field by two arrays:

- codec: list of different values,
- keys: list of indexes of list values

The field is reconstituted by replacing the integers in the "keys" list with the corresponding values from the "codec" list.

Example ("produit" field):

Codec : ["orange" , "piment" , "pomme" , "banane"]

Keys : [2, 2, 0, 0, 1, 1, 3, 3]

Unique format :

This representation corresponds to a field composed of a single duplicate value.

The "unique format" representation consists in representing only this unique value.

It is therefore a "complete format" representation in which the "keys" list is implicit.

Example ("validité" field):

Codec : ["du 1/7/2022 au 31/12/2022"]

Keys : implicit ([0, 0, 0, 0, 0, 0, 0, 0])

Note:

This format also makes it possible to represent tabular metadata

Implicit format :

This representation is associated with "coupled" fields. These fields have a one-to-one correspondence.

The representation consists of the "codec" list and the reference to the associated ("parent") field.

Example ("quantité" field is associated with "contenant" field) :

Codec : ["1 kg" , "10 kg"]

Parent : 3 (field n° 3 : "contenant")

Keys : implicit ("keys" list of the "contenant" field)

Relative format :

This representation is associated with "derived" fields.

The values of a "derived" field are inferred from the values of the "parent" field.

Example ("aliment" field is associated with "produit" field) :

Codec : ["fruit" , "légume"]

Parent : 1 (field n° 1 : "produit")

Keys : [0, 1, 0, 0] (the absolute "keys" list is obtained by replacing the values 0, 1, 2, 3 of the "keys" list of the "product" field by 0, 1, 0, 0 i.e.: [0, 0, 0, 0, 1 , 1, 0, 0])

Primary format :

This representation is associated with "crossed" fields. The values of a "crossed" field are calculated from the "codec" list and a "repetition coefficient".

Example "contenant" (this field is associated to "produit" field) :

Codec : ["carton", "sachet"]

Coefficient: 1 ("contenant" is the 2nd field of type "crossed")

Keys : implicit ([0, 1, 0, 1, 0, 1, 0, 1])

Example "produit" (this field is associated to "contenant" field) :

Codec : ["pomme" , "orange" , "piment" , "banane"]

Coefficient: 2 ("produit" is the 1st field of type "crossed")

Keys : implicit ([0, 0, 1, 1, 2, 2, 3, 3])

3.3 Level of representation

Three levels, from the simplest to the most optimized, are available to convert tabular data in JSON structure.

- Level 1 : "full"

  The fields are converted into "full format" or "unique format". A tabular object has a single full representation.

- Level 2 : "default"

  The fields are converted into "full format", "complete format" or "unique format"

  Nota : some fields can be converted into "full format", others into "complete format"

- Level 3 : "optimize"

  This level requires an analysis of the relationships between fields.

  "primary" fields ("crossed" fields that form a partition) are converted into "primary format". Other fields are converted into "full format", "unique format", "complete format", "implicit format" or "relative format" according to their position in the tree.

Level 1 is the usual representation of tabular data.

Level 2 avoids duplication of information by adding simple encoding.

Level 3 avoids duplication of information and minimizes encoding.

Several representations are available for a tabular object at level 2 or 3.

Fields structure :

level		Structure		Codec		parent		keys		
n°	mode	Type index	format	= len parent	< len parent	implicit parent	explicit parent	Absolute (= len)	Relative (< len)	implicit
all		unique	unique		x (1)	x (root)				x (list of 0)
1	full	Not unique	full	x		x (root)				x (range)
2	default	Full codec	full	x		x (root)				x (range)
		Reduce codec	complete		x	x (root)		x		
3	optimize	Root coupled	full	x		x (root)				x (range)
		Root derived	complete		x	x (root)		x		
		derived	relative		x		x		x	
		coupled	implicit	x			x			x (parent)
		primary	primary		x	x (root)			x (coef)	

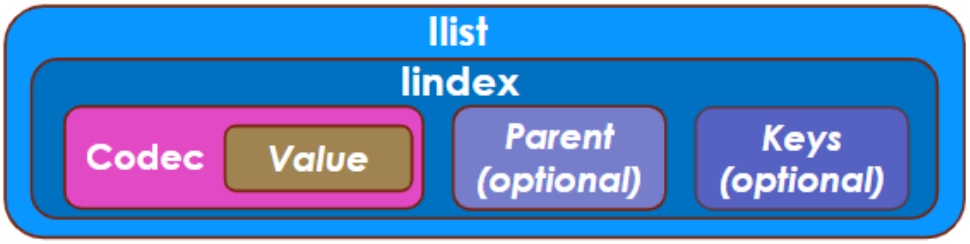
4 JSON-TAB REPRESENTATION

4.1 STRUCTURE

An Ilist object defines a tabular object. An Ilist object can have a name.

An Iindex object is associated with a field (list of Values) in Ilist Object. An Iindex object can have a name and a type (the type is a default type for the values included). A Value can have a name and a type. The Iindex length is equal to the Ilist length (len).

The figure below shows the structure of Ilist objects (including the several formats of fields).



At level 1 ('full'), Keys and Parent data is not used.

At level 2 ('default') Parent is not used.

The structure of Iindex depends on the representation of the fields defined in chapter 3.2 :

4.2 JSON FORMAT

With the NTV format, the objects are:

- Ilist: NV-list of Iindex
- Iindex: NTV-list or NTV-single
- Codec: TV-list of Values or one Value
- Value: NTV-entity
- Parent : number or string
- Keys: V-list of integer data

NTV fields structure :

Structure		parent		keys		
format	NTV	implicit	explicit	Absolute (= len)	Relative (< len)	implicit
full	NTV-list of values	x (root)				x (range)
unique	NTV-single	x (root)				x (list of 0)
complete	NV-list len=2	x (root)		x		
relative	NV-list len=3		x		x	
implicit	NV-list len=2		x			x (parent)
primary	NV-list len=2	x (root)			x (coef)	

Note:

If Codec is an NTV-single or an empty TV-list, Parent and Keys are not present.

If Parent and Keys are not present, Codec and Iindex are merged.

The JSON format of a tabular object is the JSON-NTV format of the Ilist object.

For better readability, the JSON-value for Iindex MAY be separated by a line Separator '\n'.

5 EXAMPLES

5.1 IINDEX EXAMPLE

The examples in chapter 3.2 have the following JSON format:

Format	Representation
Full	*{ 'prix': [1, 9, 2, 18, 1.5, 13, 0.5, 4] }*
Complete	*{ "produit": [["orange" , "piment" , "pomme" , "banane"], [2, 2, 0, 0, 1, 1, 3, 3]] }*
Unique	*{ "validité": "du 1/7/2022 au 31/12/2022" }*
Implicit	*{ "quantité": [["1 kg" , "10 kg"], 3] }*
Relative	*{ "aliment": [["fruit" , "légume"], 1, [0, 1, 0, 0]] }*
Primary	*[["carton", "sachet"], [1]]* *(without name)*

{ "age": 25 }	*Iindex with name and codec*
{ "measure": [2.4, 48.9] }	*Iindex with name and codec*
[[2.4, 48.9], [0, 0, 1, 1]]	*Iindex with codec and Keys*
{"measure": [[2.4, 48.9], [0, 0, 1, 1]] }	*Iindex with name, codec and Keys*
{"::geopoint": [[2.4, 48.9], [4.2, 84.9]] }	*Iindex with type and codec*
{ "city:geopoint": [2.4, 48.9] }	*Iindex with type, name and codec*
{ "city"::geopoint": [[[2.4, 48.9], [4.2, 84.9]],0 ,[0,0,1,1]] }	*Iindex with all data*

5.2 ILIST EXAMPLES

The examples below illustrate the JSON Ilist format with 'full level' and 'optimize level'.

Ilist	full level	optimize level
Matrix	[['a', 'a', 'b', 'b', 'c', 'c'], [10, 20, 10, 20, 10, 20], [1, 2, 3, 4, 5, 6]]	[[['a', 'b', 'c'], [2]], [[10, 20], [1]], [1, 2, 3, 4, 5, 6]]
Single	[[1, 2, 3, 4, 5, 6], ['a', 'a', 'a', 'a', 'a', 'a']]	[[1, 2, 3, 4, 5, 6], ['a']]
Complete	[1, 2, 3, 3, 5, 5]	[[1, 2, 3, 5], [0, 1, 2, 2, 3, 3]]
Coupled	[[1, 2, 3, 3, 5, 5], ['a', 'b', 'c', 'c', 'e', 'e']]	[[[1, 2, 3, 5], [0, 1, 2, 2, 3, 3]], [['a', 'b', 'c', 'e'], 0]]
Derived	[[1, 2, 3, 4, 5, 6], ['a', 'a', 'b', 'b', 'c', 'c'], [10, 10, 10, 10, 20, 20]]	[[1, 2, 3, 4, 5, 6], [['a', 'b', 'c'], [0, 0, 1, 1, 2, 2]], [[10, 20], 1, [0, 0, 1]]]
Matrix + coupled	[['a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'], [10, 20, 10, 20, 10, 20, 10, 20], ['t1', 't1', 't2', 't2', 't3', 't3', 't4', 't4'], [1, 2, 3, 4, 5, 6, 7, 8]]	[[['a', 'b', 'c', 'd'], [2]], [[10, 20], [1]], [['t1', 't2', 't3', 't4'], 0], [1, 2, 3, 4, 5, 6, 7, 8]]
Matrix + coupled + derived	[['a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'], [10, 20, 10, 20, 10, 20, 10, 20], ['t1', 't1', 't2', 't2', 't3', 't3', 't4', 't4'], [100, 100, 100, 100, 200, 200, 200, 200], [1, 2, 3, 4, 5, 6, 7, 8]]	[[['a', 'b', 'c', 'd'], [2]], [[10, 20], [1]], [['t1', 't2', 't3', 't4'], 0], [[100, 200], 0], [1, 2, 3, 4, 5, 6, 7, 8]]

The examples below show how to represent Array with a single value.

[ ]	*Empty Ilist*
[25] or [[25]]	*Ilist with 1 Iindex with 1 codec value*
[2, 1] or [[2], [1]] or [2, [1]]	*Ilist with 2 Iindex with 1 codec value*
[[2, 1]]	*Ilist with 1 Iindex with 2 codec values*
[[2, 1], [4,3]]	*Ilist with 2 Iindex with 2 codec values*

6 PARSING A JSON-VALUE

A NTV parser generates an NTV entity from a JSON-value.

A TAB decoder generates an Ilist object from an NTV entity.

Several dataset can generate inconsistent data:

- [listdata, integer, listinteger]
- [listdata, string, listinteger]
- [listdata, integer]
- [listdata, string]
- [listdata, listinteger]

To avoid this inconsistency, the order of data can be changed (e.g. [integer, listdata] or a name can be added (e.g. { 'name': [listdata, listinteger] }) or a null value can be added (e.g. [listdata, integer, listinteger, "null"]).

APPENDIX : RESERVED VALUES

to complete

APPENDIX : CBOR FORMAT

The Concise Binary Object Representation (CBOR – RFC8949) is a data format whose design goals include the possibility of extremely small code size, small message size, and extensibility without the need for version negotiation.

CBOR is based on the JSON data model: numbers, strings, arrays, maps (called objects in JSON), and a few values such as false, true, and null.

The CBOR format can be used with different options to minimize length:

- The precision of float values is adjustable from half precision (two bytes) to double precision (eight bytes),
- The datetime can be described by a standard text string (RFC3339) or by a numerical value (Epoch-based: six bytes).
- The TypeValue can be represented with a code value instead of string value
- The coordinates value can be described with integer instead of float (val_int = round(val_float)*10**7 : four bytes).

Example (Json format):

{"type": "observation",

"datation":["2021-01-04T10:00:00",[["2021-01-05T08:00:00","2021-01-5T12:00:00"]]],

"location":[[2.4123456, 48.9123456], [[[2.4123456, 48.9123456], [4.8123456, 45.8123456], [5.4123456, 43.3123456], [2.4123456, 48.9123456]]],

"property": [{"prp": "PM10"}, {"prp": "Temp"}],

"result": [51.348, {"low": 2.457}, 20.88, "high"],

"coupled": {"datation": "location"}}

Example optimized (Cbor format):

{0 : [0,1,2],

1 : [[dt(2021, 1, 4, 10),[[dt(2021,1,5,8), dt(2021, 1, 5, 12]]],

[[2.4123456, 48.9123456], [[[2.4123456, 48.9123456], [4.8123456, 45.8123456], [5.4123456, 43.3123456], [2.4123456, 48.9123456]]],

[{"prp": "PM10"}, {"prp": "Temp"}]],

2 : [51.34375, {"low": 2.45703125}, 20.875, "high"],

3 : {0: 1} }

With :

- *Observation key codification:*

    - *0: "order"*
    - *1: "features"*
    - *2: "result"*
    - *3: "coupled"*
- *Order and coupled value codification:*
    - *0: "datation"*
    - *1: "location"*
    - *2: "property"*
- *Datation value: timestamp format*
- *Location value: integer representation (four bytes)*
- *Result value: half precision (two bytes)*

Length (bytes):

- JSON: 388
- CBOR: 298
- CBOR optimized: 133