FRICTIONLESS STANDARDS

Table Schema

A simple format to declare a schema for tabular data. The schema is designed to be expressible in JSON.

Author(s)	Paul Walsh, Rufus Pollock
Created	12 November 2012
Updated	5 October 2021
JSON Schema	table-schema.json
Version	1

Language

The key words `MUST`, `MUST NOT`, `REQUIRED`, `SHALL`, `SHALL NOT`, `SHOULD`, `SHOULD NOT`, `RECOMMENDED`, `MAY`, and `OPTIONAL` in this document are to be interpreted as described in RFC 2119

Introduction

Table Schema is a simple language- and implementation-agnostic way to declare a schema for tabular data. Table Schema is well suited for use cases around handling and validating tabular data in text formats such as CSV, but its utility extends well beyond this core usage, towards a range of applications where data benefits from a portable schema format.

Concepts

Tabular data

Tabular data consists of a set of rows. Each row has a set of fields (columns). We usually expect that each row has the same set of fields and thus we can talk about *the* fields for row, giving the names of the fields. By contrast, in other situations, e.g. tables in SQL databases, the field names are explicitly designated.

To illustrate, here's a classic spreadsheet table:

```
   field     field
     |         |
     |         |
     V         V

    A     |    B    |    C    |    D      <--- Row (Header)
    ------------------------------------
    valA  |   valB  |  valC   |   valD    <--- Row
    ...
```

In JSON, a table would be:

```
[
  { "A": value, "B": value, ... },
  { "A": value, "B": value, ... },
  ...
]
```

Physical and logical representation

In order to talk about the representation and processing of tabular data from text-based sources, it is useful to introduce the concepts of the *physical* and the *logical* representation of data.

The *physical representation* of data refers to the representation of data as text on disk, for example, in a CSV or JSON file. This representation may have some *type* information (JSON, where the primitive types that JSON supports can be used) or not (CSV, where all data is represented in string form).

The *logical representation* of data refers to the "ideal" representation of the data in terms of primitive types, data structures, and relations, all as defined by the specification. We could say that the specification is about the logical representation of data, as well as about ways

FRICTIONLESS STANDARDS

places where it prevents ambiguity for those engaging with the specification, especially implementors.

For example, `constraints` should be tested on the logical representation of data, whereas a property like `missingValues` applies to the physical representation of the data.

Descriptor

A Table Schema is represented by a descriptor. The descriptor `MUST` be a JSON `object` (JSON is defined in RFC 4627).

It `MUST` contain a property `fields` . `fields` `MUST` be an array where each entry in the array is a field descriptor (as defined below). The order of elements in `fields` array `SHOULD` be the order of fields in the CSV file. The number of elements in `fields` array `SHOULD` be the same as the number of fields in the CSV file.

The descriptor `MAY` have the additional properties set out below and `MAY` contain any number of other properties (not defined in this specification).

The following is an illustration of this structure:

```js
{
  // fields is an ordered list of field descriptors
  // one for each field (column) in the table
  "fields": [
    // a field-descriptor
    {
      "name": "name of field (e.g. column name)",
      "title": "A nicer human readable label or title for the field",
      "type": "A string specifying the type",
      "format": "A string specifying a format",
      "example": "An example value for the field",
      "description": "A description for the field"
      ...
    },
    ... more field descriptors
  ],
  // (optional) specification of missing values
  "missingValues": [ ... ],
```

FRICTIONLESS STANDARDS

```
  "foreignKeys": ...
}
```

Field Descriptors

A field descriptor `MUST` be a JSON `object` that describes a single field. The descriptor provides additional human-readable documentation for a field, as well as additional information that may be used to validate the field or create a user interface for data entry.

Here is an illustration:

```js
{
  "name": "name of field (e.g. column name)",
  "title": "A nicer human readable label or title for the field",
  "type": "A string specifying the type",
  "format": "A string specifying a format",
  "example": "An example value for the field",
  "description": "A description for the field",
  "constraints": {
      // a constraints-descriptor
  }
}
```

The field descriptor `object` `MAY` contain any number of other properties. Some specific properties are defined below. Of these, only the `name` property is `REQUIRED` .

`name`

The field descriptor MUST contain a `name` property. This property `SHOULD` correspond to the name of field/column in the data file (if it has a name). As such it `SHOULD` be unique (though it is possible, but very bad practice, for the data file to have multiple columns with the same name). `name` `SHOULD NOT` be considered case sensitive in determining uniqueness. However, since it should correspond to the name of the field in the data file it may be important to preserve case.

FRICTIONLESS STANDARDS

`title`

A human readable label or title for the field

`description`

A description for this field e.g. “The recipient of the funds”

`example`

An example value for the field

Types and Formats

`type` and `format` properties are used to give The type of the field (string, number etc) - see below for
more detail. If type is not provided a consumer should assume a type of
“string”.

A field’s `type` property is a string indicating the type of this field.

A field’s `format` property is a string, indicating a format for the field type.

Both `type` and `format` are optional: in a field descriptor, the absence of a `type` property indicates that the field is of the type “string”, and the absence of a `format` property indicates that the field’s type `format` is “default”.

Types are based on the type set of json-schema with some additions and minor modifications (cf other type lists include those in Elasticsearch types).

The type list with associated formats and other related properties is as follows.

string

The field contains strings, that is, sequences of characters.

`format` :

- **default**: any valid string.
- **email**: A valid email address.
- **uri**: A valid URI.
- **binary**: A base64 encoded string representing binary data.
- **uuid**: A string that is a uuid.

number

The field contains numbers of any kind including decimals.

The lexical formatting follows that of decimal in XMLSchema : a non-empty finite-length sequence of decimal digits separated by a period as a decimal indicator. An optional leading sign is allowed. If the sign is omitted, "+" is assumed. Leading and trailing zeroes are optional. If the fractional part is zero, the period and following zero(es) can be omitted. For example: '-1.23', '12678967.543233', '+100000.00', '210'.

The following special string values are permitted (case need not be respected):

- NaN: not a number
- INF: positive infinity
- -INF: negative infinity

A number MAY also have a trailing:

- exponent: this MUST consist of an E followed by an optional + or - sign followed by one or more decimal digits (0-9)

This lexical formatting may be modified using these additional properties:

- **decimalChar**: A string whose value is used to represent a decimal point within the number. The default value is ".".

- **bareNumber**: a boolean field with a default of `true` . If `true` the physical contents of this field must follow the formatting constraints already set out. If `false` the contents of this field may contain leading and/or trailing non-numeric characters (which implementors MUST therefore strip). The purpose of `bareNumber` is to allow publishers to publish numeric data that contains trailing characters such as percentages e.g. `95%` or leading characters such as currencies e.g. `€95` or `EUR 95` . Note that it is entirely up to implementors what, if anything, they do with stripped text.

`format` : no options (other than the default).

integer

The field contains integers - that is whole numbers.

Integer values are indicated in the standard way for any valid integer.

Additional properties:

- **bareNumber**: a boolean field with a default of `true` . If `true` the physical contents of this field must follow the formatting constraints already set out. If `false` the contents of this field may contain leading and/or trailing non-numeric characters (which implementors MUST therefore strip). The purpose of `bareNumber` is to allow publishers to publish numeric data that contains trailing characters such as percentages e.g. `95%` or leading characters such as currencies e.g. `€95` or `EUR 95` . Note that it is entirely up to implementors what, if anything, they do with stripped text.

`format` : no options (other than the default).

boolean

The field contains boolean (true/false) data.

In the physical representations of data where boolean values are represented with strings, the values set in `trueValues` and `falseValues` are to be cast to their logical representation as booleans. `trueValues` and `falseValues` are arrays which can be customised to user need. The default values for these are in the additional properties section below.

- **falseValues**: `[ "false", "False", "FALSE", "0" ]`

`format` : no options (other than the default).

object

The field contains data which is valid JSON.

`format` : no options (other than the default).

array

The field contains data that is a valid JSON format arrays.

`format` : no options (other than the default).

date

A date without a time.

`format` :

- **default**: An ISO8601 format string.
    - date: This MUST be in ISO8601 format YYYY-MM-DD
    - datetime: a date-time. This MUST be in ISO 8601 format of YYYY-MM-DDThh:mm:ssZ in UTC time
    - time: a time without a date
- **any**: Any parsable representation of the type. The implementing library can attempt to parse the datetime via a range of strategies. An example is `dateutil.parser.parse` from the `python-dateutils` library.
- **<PATTERN>**: date/time values in this field can be parsed according to `<PATTERN>` . `<PATTERN>` MUST follow the syntax of standard Python / C strptime . (That is, values in the this field should be parsable by Python / C standard `strptime` using `<PATTERN>`). Example for `"format": "%d/%m/%y"` which would correspond to dates like: `30/11/14`

time

A time without a date.

`format` :

- **default**: An ISO8601 time string e.g. `hh:mm:ss`
- **any**: as for `date`
- **<PATTERN>**: as for `date`

datetime

A date with a time.

`format` :

- **default**: An ISO8601 format string e.g. `YYYY-MM-DDThh:mm:ssZ` in UTC time
- **any**: as for `date`
- **<PATTERN>**: as for `date`

year

A calendar year as per XMLSchema `gYear` .

Usual lexical representation is `YYYY` . There are no format options.

yearmonth

A specific month in a specific year as per XMLSchema `gYearMonth` .

Usual lexical representation is: `YYYY-MM` . There are no format options.

duration

A duration of time.

We follow the definition of XML Schema duration datatype directly

FRICTIONLESS STANDARDS

extended format PnYnMnDTnHnMnS, where nY represents the number of years, nM the number of months, nD the number of days, 'T' is the date/time separator, nH the number of hours, nM the number of minutes and nS the number of seconds. The number of seconds can include decimal digits to arbitrary precision. Date and time elements including their designator may be omitted if their value is zero, and lower order elements may also be omitted for reduced precision.

`format` : no options (other than the default).

geopoint

The field contains data describing a geographic point.

`format` :

- **default**: A string of the pattern "lon, lat", where `lon` is the longitude and `lat` is the latitude (note the space is optional after the `,`). E.g. `"90, 45"` .
- **array**: A JSON array, or a string parsable as a JSON array, of exactly two items, where each item is a number, and the first item is `lon` and the second item is `lat` e.g. `[90, 45]`
- **object**: A JSON object with exactly two keys, `lat` and `lon` and each value is a number e.g. `{"lon": 90, "lat": 45}`

geojson

The field contains a JSON object according to GeoJSON or TopoJSON spec.

`format` :

- **default**: A geojson object as per the GeoJSON spec .
- **topojson**: A topojson object as per the TopoJSON spec

any

Any `type` or `format` is accepted. When converting from physical to logical representation, the behaviour should be similar to String field type.

Rich Types

A richer, “semantic”, description of the “type” of data in a given column MAY be provided using a `rdfType` property on a field descriptor.

The value of the `rdfType` property MUST be the URI of a RDF Class, that is an instance or subclass of RDF Schema Class object

Here is an example using the Schema.org RDF Class `http://schema.org/Country` :

```
| Country | Year Date | Value |
| ------- | --------- | ----- |
| US      | 2010      | ...   |
```

The corresponding Table Schema is:

```js
{
  fields: [
    {
      "name": "Country",
      "type": "string",
      "rdfType": "http://schema.org/Country"
    }
    ...
  }
}
```

Constraints

The `constraints` property on Table Schema Fields can be used by consumers to list constraints for validating field values. For example, validating the data in a Tabular Data Resource against its Table Schema; or as a means to validate data being collected or updated via a data entry interface.

All constraints `MUST` be tested against the logical representation of data, and the physical representation of constraint values `MAY` be primitive types as possible in JSON, or

FRICTIONLESS STANDARDS

following properties.

Property	Type	Applies to	Description
`required`	boolean	All	Indicates whether this field cannot be `null`. If required is `false` (the default), then `null` is allowed. See the section on `missingValues` for how, in the physical representation of the data, strings can represent `null` values.
`unique`	boolean	All	If `true`, then all values for that field MUST be unique within the data file in which it is found.
`minLength`	integer	collections (string, array, object)	An integer that specifies the minimum length of a value.
`maxLength`	integer	collections (string, array, object)	An integer that specifies the maximum length of a value.
`minimum`	integer, number, date, time and datetime, year, yearmonth	`integer`, `number`, `date`, `time`, `datetime`, `year`, `yearmonth`	Specifies a minimum value for a field. This is different to `minLength` which checks the number of items in the value. A `minimum` value constraint checks whether a field value is greater than or equal to the specified value. The range checking depends on the `type` of the field. E.g. an integer field may have a minimum value of 100; a date field might have a minimum date. If a `minimum` value constraint is specified then the field descriptor `MUST` contain a `type` key.
`maximum`	integer, number, date, time and datetime,	`integer`, `number`, `date`, `time` and `datetime`,	As for `minimum`, but specifies a maximum value for a field.

`pattern`	string	`string`	used to test field values. If the regular expression matches then the value is valid. The values of this field `MUST` conform to the standard XML Schema regular expression syntax.
`enum`	array	All	The value of the field must exactly match a value in the `enum` array.

Implementors:

- Implementations `SHOULD` report an error if an attempt is made to evaluate a value against an unsupported constraint.
- A constraints descriptor may contain multiple constraints, in which case implementations `MUST` apply all the constraints when determining if a field value is valid.
- Constraints `MUST` be applied on the logical representation of field values and constraint values.

Other Properties

In additional to field descriptors, there are the following “table level” properties.

Missing Values

Many datasets arrive with missing data values, either because a value was not collected or it never existed. Missing values may be indicated simply by the value being empty in other cases a special value may have been used e.g. `-` , `NaN` , `0` , `-9999` etc.

`missingValues` dictates which string values should be treated as `null` values. This conversion to `null` is done before any other attempted type-specific string conversion. The default value `[ "" ]` means that empty strings will be converted to null before any other processing takes place.
Providing the empty list `[]` means that no conversion to null will be done, on any value.

`missingValues` MUST be an `array` where each entry is a `string` .

FRICTIONLESS STANDARDS

Examples:

```js
"missingValues": [""]
"missingValues": ["-"]
"missingValues": ["NaN", "-"]
```

Primary Key

A primary key is a field or set of fields that uniquely identifies each row in the table. Per SQL standards, the fields cannot be `null`, so their use in the primary key is equivalent to adding `required: true` to their `constraints`.

The `primaryKey` entry in the schema `object` is optional. If present it specifies the primary key for this table.

The `primaryKey`, if present, MUST be:

- Either: an array of strings with each string corresponding to one of the field `name` values in the `fields` array (denoting that the primary key is made up of those fields). It is acceptable to have an array with a single value (indicating just one field in the primary key). Strictly, order of values in the array does not matter. However, it is RECOMMENDED that one follow the order the fields in the `fields` has as client applications may utilize the order of the primary key list (e.g. in concatenating values together).
- Or: a single string corresponding to one of the field `name` values in the `fields` array (indicating that this field is the primary key). Note that this version corresponds to the array form with a single value (and can be seen as simply a more convenient way of specifying a single field primary key).

Here's an example:

```
  "fields": [
```

FRICTIONLESS STANDARDS

```
    ...
  ],
  "primaryKey": "a"
```

Here's an example with an array primary key:

```
"schema": {
  "fields": [
    {
      "name": "a"
    },
    {
      "name": "b"
    },
    {
      "name": "c"
    },
    ...
  ],
  "primaryKey": ["a", "c"]
}
```

Foreign Keys

A foreign key is a reference where values in a field (or fields) on the table ('resource' in data package terminology) described by this Table Schema connect to values a field (or fields) on this or a separate table (resource). They are directly modelled on the concept of foreign keys in SQL.

The `foreignKeys` property, if present, `MUST` be an Array. Each entry in the array must be a `foreignKey`. A `foreignKey` `MUST` be a `object` and MUST have the following properties:

- `fields` - `fields` is a string or array specifying the field or fields on this resource that form the source part of the foreign key. The structure of the string or array is as per `primaryKey` above.
- `reference` - `reference` MUST be a `object`. The `object`

Schema is located). For self-referencing foreign keys, i.e. references between fields in this Table Schema, the value of `resource` `MUST` be `""` (i.e. the empty string).

  - `MUST` have a property `fields` which is a string if the outer `fields` is a string, else an array of the same length as the outer `fields`, describing the field (or fields) references on the destination resource. The structure of the string or array is as per `primaryKey` above.

Here's an example:

FRICTIONLESS STANDARDS

```
  {
    "name": "state-codes",
    "schema": {
      "fields": [
        {
          "name": "code"
        }
      ]
    }
  },
  {
    "name": "population-by-state"
    "schema": {
      "fields": [
        {
          "name": "state-code"
        }
        ...
      ],
      "foreignKeys": [
        {
          "fields": "state-code",
          "reference": {
            "resource": "state-codes",
            "fields": "code"
          }
        }
      ]
  ...
```

An example of a self-referencing foreign key:

```js
"resources": [
  {
    "name": "xxx",
    "schema": {
      "fields": [
        {
          "name": "parent"
        },
        {
```

FRICTIONLESS STANDARDS

```
          "foreignKeys": [
            {
              "fields": "parent"
              "reference": {
                "resource": "",
                "fields": "id"
              }
            }
          ]
        }
      }
    ]
```

Comment: Foreign Keys create links between one Table Schema and another Table Schema, and implicitly between the data tables described by those Table Schemas. If the foreign key is referring to another Table Schema how is that other Table Schema discovered? The answer is that a Table Schema will usually be embedded inside some larger descriptor for a dataset, in particular as the schema for a resource in the resources array of a Data Package . It is the use of Table Schema in this way that permits a meaningful use of a non-empty `resource` property on the foreign key.

Appendix: Related Work

Table Schema draws content and/or inspiration from, among others, the following specifications and implementations:

- XML Schema
- Google BigQuery
- JSON Schema
- DSPL
- HTML5 Forms
- Elasticsearch

Edit this page

Last Updated: 7/3/2023, 9:10:36 AM

FRICTIONLESS
STANDARDS