Model for Tabular Data and Metadata on the Web

W3C Proposed Recommendation 17 November 2015

This version:
http://www.w3.org/TR/2015/PR-tabular-data-model-20151117/
Latest published version:
http://www.w3.org/TR/tabular-data-model/
Latest editor's draft:
http://w3c.github.io/csvw/syntax/
Test suite:
http://www.w3.org/2013/csvw/tests/
Implementation report:
http://www.w3.org/2013/csvw/implementation_report.html
Previous version:
http://www.w3.org/TR/2015/CR-tabular-data-model-20150716/
Editors:
Jeni Tennison, Open Data Institute
Gregg Kellogg, Kellogg Associates
Authors:
Jeni Tennison, Open Data Institute
Gregg Kellogg, Kellogg Associates
Ivan Herman, W3C
Repository:
We are on GitHub
File a bug
Changes:
Diff to previous version
Commit history

Abstract

Tabular data is routinely transferred on the web in a variety of formats, including variants on CSV, tab-delimited files, fixed field formats, spreadsheets, HTML tables, and SQL dumps. This document outlines a data model, or infoset, for tabular data and metadata about that tabular data that can be used as a basis for validation, display, or creating other formats. It also contains some non-normative guidance for publishing tabular data as CSV and how that maps into the tabular data model.

An annotated model of tabular data can be supplemented by separate metadata about the table. This specification defines how implementations should locate that metadata, given a file containing tabular data. The standard syntax for that metadata is defined in [tabular-metadata]. Note, however, that applications may have other means to create annotated tables, e.g., through some application specific API-s; this model does not depend on the specificities described in [tabular-metadata].

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

The CSV on the Web Working Group was chartered to produce a recommendation "Access methods for CSV Metadata" as well as recommendations for "Metadata vocabulary for CSV data" and "Mapping mechanism to transforming CSV into various formats (e.g., RDF, JSON, or XML)". This document aims to primarily satisfy the "Access methods for CSV Metadata" recommendation (see section 5. Locating Metadata), though it also specifies an underlying model for tabular data and is therefore a basis for the other chartered Recommendations.

This definition of CSV used in this document is based on IETF's [RFC4180] which is an Informational RFC. The working group's expectation is that future suggestions to refine RFC 4180 will be relayed to the IETF (e.g. around encoding and line endings) and contribute to its discussions about moving CSV to the Standards track.

Many files containing tabular data embed metadata, for example in lines before the header row of an otherwise standard CSV document. This specification does not define any formats for embedding metadata within CSV files, aside from the titles of columns in the header row which is defined in CSV. We would encourage groups that define tabular data formats to also define a mapping into the annotated tabular data model defined in this document.

This document was published by the CSV on the Web Working Group as a Proposed Recommendation. This document is intended to become a W3C Recommendation. The W3C Membership and other interested parties are invited to review the document and send comments to public-csv-wg@w3.org (subscribe, archives) through 15 December 2015. Advisory Committee Representatives should consult their WBS questionnaires. Note that substantive technical comments were expected during the Last Call review period that ended 30 October 2015.

Please see the Working Group's implementation report.

Publication as a Proposed Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 September 2015 W3C Process Document.

Table of Contents

1. Introduction

Tabular data is data that is structured into rows, each of which contains information about some thing. 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.

Tabular data is routinely transferred on the web in a textual format called CSV, but the definition of CSV in practice is very loose. Some people use the term to mean any delimited text file. Others stick more closely to the most standard definition of CSV that there is, [RFC4180]. Appendix A describes the various ways in which CSV is defined. This specification refers to such files, as well as tab-delimited files, fixed field formats, spreadsheets, HTML tables, and SQL dumps as ***tabular data files***.

In section 4. Tabular Data Models, this document defines a model for tabular data that abstracts away from the varying syntaxes that are used for when exchanging tabular data. The model includes annotations, or metadata, about collections of individual tables, rows, columns, and cells. These annotations are typically supplied through separate metadata files; section 5. Locating Metadata defines how these metadata files can be located, while [tabular-metadata] defines what they contain.

Once an annotated table has been created, it can be processed in various ways, such as display, validation, or conversion into other formats. This processing is described in section 6. Processing Tables.

This specification does not normatively define a format for exchanging tabular data. However, it does provide some best practice guidelines for publishing tabular data as CSV, in section section 7. Best Practice CSV, and for parsing both this syntax and those similar to it, in section 8. Parsing Tabular Data.

2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words MAY, MUST, MUST NOT, SHOULD, and SHOULD NOT are to be interpreted as described in [RFC2119].

This specification makes use of the ***compact IRI Syntax***; please refer to the Compact IRIs from [JSON-LD].

This specification makes use of the following namespaces:

csvw:
    `http://www.w3.org/ns/csvw#`
dc:
    `http://purl.org/dc/terms/`
rdf:
    `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs:
    `http://www.w3.org/2000/01/rdf-schema#`
schema:
    `http://schema.org/`

xsd:
    `http://www.w3.org/2001/XMLSchema#`

3. Typographical conventions

The following typographic conventions are used in this specification:

`markup`
    Markup (elements, attributes, properties), machine processable values (string, characters, media types), property name, or a file name is in red-orange monospace font.

variable
    A variable in pseudo-code or in an algorithm description is in italics.

definition
    A definition of a term, to be used elsewhere in this or other specifications, is in bold and italics.

definition reference
    A reference to a definition *in this document* is underlined and is also an active link to the definition itself.

`markup definition reference`
    A references to a definition *in this document*, when the reference itself is also a markup, is underlined, red-orange monospace font, and is also an active link to the definition itself.

external definition reference
    A reference to a definition *in another document* is underlined, in italics, and is also an active link to the definition itself.

`markup external definition reference`
    A reference to a definition *in another document*, when the reference itself is also a markup, is underlined, in italics red-orange monospace font, and is also an active link to the definition itself.

hyperlink
    A hyperlink is underlined and in blue.

[reference]
    A document reference (normative or informative) is enclosed in square brackets and links to the references section.

NOTE

Notes are in light green boxes with a green left border and with a "Note" header in green. Notes are normative or informative depending on the whether they are in a normative or informative section, respectively.

EXAMPLE 1

```
Examples are in light khaki boxes, with khaki left border, and with a
numbered "Example" header in khaki. Examples are always informative.
The content of the example is in monospace font and may be syntax colored.
```

4. Tabular Data Models

This section defines an ***annotated tabular data model***: a model for tables that are annotated with metadata. ***Annotations*** provide information about the cells, rows, columns, tables, and groups of tables with which they are associated. The ***values*** of these annotations may be lists, structured objects, or atomic values. ***Core annotations*** are those that affect the behavior of processors defined in this specification, but other annotations may also be present on any of the components of the model.

Annotations may be described directly in [tabular-metadata], be embedded in a tabular data file, or created during the process of generating an annotated table.

String values within the tabular data model (such as column titles or cell string values) MUST contain only Unicode characters.

NOTE

In this document, the term annotation refers to any metadata associated with an object in the annotated tabular data model. These are not necessarily web annotations in the sense of [annotation-model].

4.1 Table groups

A ***group of tables*** comprises a set of annotated tables and a set of annotations that relate to that group of tables. The core annotations of a group of tables are:

- ***id*** — an identifier for this group of tables, or `null` if this is undefined.
- ***notes*** — any number of additional annotations on the group of tables. This annotation may be empty.
- ***tables*** — the list of tables in the group of tables. A group of tables MUST have one or more tables.

Groups of tables MAY in addition have any number of annotations which provide information about the group of tables. Annotations on a group of tables may include:

- titles or descriptions of the group of tables.
- information about the source or provenance of the group of tables.
- links to other groups of tables (e.g. to those that provide similar data from a different time period).

When originating from [tabular-metadata], these annotations arise from *common properties* defined on *table group descriptions* within metadata documents.

4.2 Tables

An ***annotated table*** is a ***table*** that is annotated with additional metadata. The core annotations of a table are:

- ***columns*** — the list of columns in the table. A table MUST have one or more columns and the order of the columns within the list is significant and MUST be preserved by applications.
- ***table direction*** — the direction in which the columns in the table should be displayed, as described in section 6.5.1 Bidirectional Tables; the value of this annotation may also become the value of the text direction annotation on columns and cells within the table, if the `textDirection` property is set to `inherit` (the default).
- ***foreign keys*** — a list of foreign keys on the table, as defined in [tabular-metadata], which may be an empty list.
- ***id*** — an identifier for this table, or `null` if this is undefined.
- ***notes*** — any number of additional annotations on the table. This annotation may be empty.
- ***rows*** — the list of rows in the table. A table MUST have one or more rows and the order of the rows within the list is significant and MUST be preserved by applications.

- ***schema*** — a URL referencing a schema applied to this table, or `null`.
- ***suppress output*** — a boolean that indicates whether or not this table should be suppressed in any output generated from converting the group of tables, that this table belongs to, into another format, as described in section 6.7 Converting Tables.
- ***transformations*** — a (possibly empty) list of specifications for converting this table into other formats, as defined in [tabular-metadata].
- ***url*** — the URL of the source of the data in the table, or `null` if this is undefined.

The table MAY in addition have any number of other annotations. Annotations on a table may include:

- titles or descriptions of the table,
- information about the source or provenance of the data in the table, or
- links to other tables (e.g. to indicate tables that include related information).

When originating from [tabular-metadata], these annotations arise from *common properties* defined on *table descriptions* within metadata documents.

4.3 Columns

A ***column*** represents a vertical arrangement of cells within a table. The core annotations of a column are:

- ***about URL*** — the about URL *URI template* used to create a URL identifier for each value of cell in this column relative to the row in which it is contained, as defined in [tabular-metadata].
- ***cells*** — the list of cells in the column. A column MUST contain one cell from each row in the table. The order of the cells in the list MUST match the order of the rows in which they appear within the rows for the associated table.
- ***datatype*** — the expected datatype for the values of cells in this column, as defined in [tabular-metadata].
- ***default*** — the default value for cells whose string value is an empty string.
- ***lang*** — the code for the expected language for the values of cells in this column, expressed in the format defined by [BCP47].
- ***name*** — the name of the column.
- ***null*** — the string or strings which cause the value of cells having string value matching any of these values to be `null`.
- ***number*** — the position of the column amongst the columns for the associated table, starting from 1.
- ***ordered*** — a boolean that indicates whether the order of values of a cell should be preserved or not.
- ***property URL*** — the expected property URL *URI template* used to create a URL identifier for the property of each value of cell in this column relative to the row in which it is contained, as defined in [tabular-metadata].
- ***required*** — a boolean that indicates that values of cells in this column MUST NOT be empty.
- ***separator*** — a string value used to create multiple values of cells in this column by splitting the string value on the separator.
- ***source number*** — the position of the column in the file at the url of the table, starting from 1, or `null`.
- ***suppress output*** — a boolean that indicates whether or not this column should be suppressed in any output generated from converting the table, as described in section 6.7 Converting Tables.
- ***table*** — the table in which the column appears.
- ***text direction*** — the indicator of the text direction values of cells in this column, as described in section 6.5.1 Bidirectional Tables; the value of this annotation may be derived from the table direction annotation on the table, if the `textDirection` property is set to `inherit` (the default).
- ***titles*** — any number of human-readable titles for the column, each of which MAY have an associated language code as defined by [BCP47].

- ***value URL*** — the expected value URL *URI template* used to create the URL identifier for the value of each cell in this, as defined in [tabular-metadata].
- ***virtual*** — a boolean that indicates whether the column is a virtual column. ***Virtual columns*** are used to extend the source data with additional empty columns to support more advanced conversions; when this annotation is `false`, the column is a ***real column***, which exists in the source data for the table.

NOTE

Several of these annotations arise from *inherited properties* that may be defined within metadata on *table group*, *table* or individual *column* descriptions.

Columns MAY in addition have any number of other annotations, such as a description. When originating from [tabular-metadata], these annotations arise from *common properties* defined on *column descriptions* within metadata documents.

4.4 Rows

A ***row*** represents a horizontal arrangement of cells within a table. The core annotations of a row are:

- ***cells*** — the list of cells in the row. A row MUST contain one cell from each column in the table. The order of the cells in the list MUST match the order of the columns in which they appear within the table columns for the row's table.
- ***number*** — the position of the row amongst the rows for the table, starting from 1.
- ***primary key*** — a possibly empty list of cells whose values together provide a unique identifier for this row. This is similar to the name of a column.
- ***titles*** — any number of human-readable titles for the row, each of which MAY have an associated language code as defined by [BCP47].
- ***referenced rows*** — a possibly empty list of pairs of a foreign key and a row in a table within the same group of tables (which may be another row in the table in which this row appears).
- ***source number*** — the position of the row in the original url of the table, starting from 1, or `null`.
- ***table*** — the table in which the row appears.

Rows MAY have any number of additional annotations. The annotations on a row provide additional metadata about the information held in the row, such as:

- the certainty of the information in that row.
- information about the source or provenance of the data in that row.

Neither this specification nor [tabular-metadata] defines a method to specify such annotations. Implementations MAY define a method for adding annotations to rows by interpreting notes on the table.

4.5 Cells

A ***cell*** represents a cell at the intersection of a row and a column within a table. The core annotations of a cell are:

- ***about URL*** — an absolute URL for the entity about which this cell provides information, or `null`.
- ***column*** — the column in which the cell appears; the cell MUST be in the cells for that column.
- ***errors*** — a (possibly empty) list of validation errors generated while parsing the value of the cell.

- ***ordered*** — a boolean that, if the value of this cell is a list, indicates whether the order of that list should be preserved or not.
- ***property URL*** — an absolute URL for the property associated with this cell, or `null`.
- ***row*** — the row in which the cell appears; the cell MUST be in the cells for that row.
- ***string value*** — a string that is the original syntactic representation of the value of the cell, e.g. how the cell appears within a CSV file; this may be an empty string.
- ***table*** — the table in which the cell appears.
- ***text direction*** — which direction the text within the cell should be displayed, as described in section 6.5.1 Bidirectional Tables; the value of this annotation may be derived from the table direction annotation on the table, if the `textDirection` property is set to `inherit` (the default).
- ***value*** — the semantic value of the cell; this MAY be a ***list*** of values, each of which MAY have a ***datatype*** other than a string, MAY have a ***language*** and MAY be `null`. For example, annotations might enable a processor to understand the string value of the cell as representing a number or a date. By default, if the string value is an empty string, the semantic value of the cell is `null`.
- ***value URL*** — an absolute URL for this cell's value, or `null`.

NOTE

There presence or absence of quotes around a value within a CSV file is a syntactic detail that is not reflected in the tabular data model. In other words, there is no distinction in the model between the second value in `a,,z` and the second value in `a,"",z`.

NOTE

Several of these annotations arise from or are constructed based on *inherited properties* that may be defined within metadata on *table group*, *table* or *column* descriptions.

Cells MAY have any number of additional annotations. The annotations on a cell provide metadata about the value held in the cell, particularly when this overrides the information provided for the column and row that the cell falls within. Annotations on a cell might be:

- notes to aid the interpretation of the value.
- information about the source or provenance of the data in that cell.
- indication of the units of measure used within a cell.

Neither this specification nor [tabular-metadata] defines a method to specify such annotations. Implementations MAY define a method for adding annotations to cells by interpreting notes on the table.

NOTE

Units of measure are not a built-in part of the tabular data model. However, they can be captured through notes or included in the converted output of tabular data through defining datatypes with identifiers that indicate the unit of measure, using virtual columns to create nested data structures, or using common properties to specify Data Cube attributes as defined in [vocab-data-cube].

4.6 Datatypes

Columns and cell values within tables may be annotated with a ***datatype*** which indicates the type of the values obtained by parsing the string value of the cell.

Datatypes are based on a subset of those defined in [xmlschema11-2]. The annotated tabular data model limits cell values to have datatypes as shown on the diagram:

- the datatypes defined in [xmlschema11-2] as derived from and including `xsd:anyAtomicType`.
- the datatype `rdf:XMLLiteral`, a sub-type of `xsd:string`, which indicates the value is an XML fragment.
- the datatype `rdf:HTML`, a sub-type of `xsd:string`, which indicates the value is an HTML fragment.
- the datatype `csvw:JSON`, a sub-type of `xsd:string`, which indicates the value is serialized JSON.
- datatypes derived from any of these datatypes.

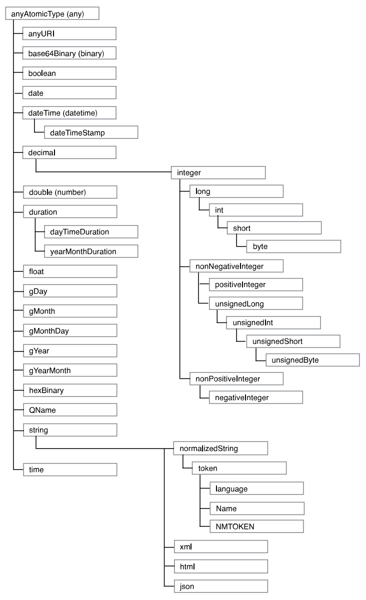

Fig. 1 Diagram showing the built-in datatypes, based on [xmlschema11-2]; names in parentheses denote aliases to the [xmlschema11-2] terms (see the diagram in SVG or PNG formats)

The core annotations of a datatype are:

- ***id*** — the absolute URL that identifies the datatype, or `null` if undefined; **converters** SHOULD use this URL when serializing values of this datatype. Processors MAY use this annotation to perform additional validation on column values using this datatype.
- ***base*** — the absolute URL that identifies the datatype from which this datatype is derived. This MUST be the URL of a built-in datatype as listed above, or `null` if the datatype is `xsd:anyAtomicType`. All values of the datatype MUST be valid values of the base datatype.
- ***format*** — a string or object that defines the format of a value of this type, used when parsing a cell string value as described in 6.4 Parsing Cells.
- ***length*** — a number that the exact length of a cell value as described in section 4.6.1 Length Constraints.
- ***minimum length*** — a number that the minimum length of a cell value as described in section 4.6.1 Length Constraints.
- ***maximum length*** — a number that the maximum length of a cell value as described in section 4.6.1 Length Constraints.
- ***minimum*** — a number that the minimum valid value (inclusive) of a cell value as described in section 4.6.2 Value Constraints.

- ***maximum*** — a number that the maximum valid value (inclusive) of a cell value as described in section 4.6.2 Value Constraints.
- ***minimum exclusive*** — a number that the minimum valid value (exclusive) of a cell value as described in section 4.6.2 Value Constraints.
- ***maximum exclusive*** — a number that the maximum valid value (exclusive) of a cell value as described in section 4.6.2 Value Constraints.

If the id of a datatype is that of a built-in datatype, the values of the other core annotations listed above MUST be consistent with the values defined in [xmlschema11-2] or above. For example, if the id is `xsd:integer` then the base must be `xsd:decimal`.

Datatypes MAY have any number of additional annotations. The annotations on a datatype provide metadata about the datatype such as title or description. These arise from *common properties* defined on datatype descriptions within metadata documents, as defined in [tabular-metadata].

NOTE

The id annotation may reference an XSD, OWL or other datatype definition, which is not used by this specification for validating column values, but may be useful for further processing.

4.6.1 Length Constraints

The length, minimum length and maximum length annotations indicate the exact, minimum and maximum lengths for cell values.

The length of a value is determined as defined in [xmlschema11-2], namely as follows:

- if the value is `null`, its length is zero.
- if the value is a string or one of its subtypes, its length is the number of characters (ie [UNICODE] *code points*) in the value.
- if the value is of a binary type, its length is the number of bytes in the binary value.

If the value is a list, the constraint applies to each element of the list.

4.6.2 Value Constraints

The minimum, maximum, minimum exclusive, and maximum exclusive annotations indicate limits on cell values. These apply to numeric, date/time, and duration types.

Validation of cell values against these datatypes is as defined in [xmlschema11-2]. If the value is a list, the constraint applies to each element of the list.

5. Locating Metadata

As described in section 4. Tabular Data Models, tabular data may have a number of annotations associated with it. Here we describe the different methods that can be used to locate metadata that provides those annotations.

In the methods of locating metadata described here, metadata is provided within a single document. The syntax of such documents is defined in [tabular-metadata]. Metadata is located using a specific order of precedence:

1. metadata supplied by the user of the implementation that is processing the tabular data, see section 5.1 Overriding Metadata.

2. metadata in a document linked to using a `Link` header associated with the tabular data file, see section 5.2 Link Header.
3. metadata located through default paths which may be overridden by a site-wide location configuration, see section 5.3 Default Locations and Site-wide Location Configuration.
4. metadata embedded within the tabular data file itself, see section 5.4 Embedded Metadata.

Processors MUST use the first metadata found for processing a tabular data file by using overriding metadata, if provided. Otherwise processors MUST attempt to locate the first metadata document from the `Link` header or the metadata located through site-wide configuration. If no metadata is supplied or found, processors MUST use embedded metadata. If the metadata does not originate from the embedded metadata, validators MUST verify that the *table group description* within that metadata is *compatible* with that in the embedded metadata, as defined in [tabular-metadata].

NOTE

When feasible, processors should start from a metadata file and publishers should link to metadata files directly, rather than depend on mechanisms outlined in this section for locating metadata from a tabular data file. Otherwise, if possible, publishers should provide a `Link` header on the tabular data file as described in section 5.2 Link Header.

NOTE

If there is no site-wide location configuration, section 5.3 Default Locations and Site-wide Location Configuration specifies default URI patterns or paths to be used to locate metadata.

5.1 Overriding Metadata

Processors SHOULD provide users with the facility to provide their own metadata for tabular data files that they process. This might be provided:

- through processor options, such as command-line options for a command-line implementation or checkboxes in a GUI.
- by enabling the user to select an existing metadata file, which may be local or remote.
- by enabling the user to specify a series of metadata files, which are merged by the processor and handled as if they were a single file.

For example, a processor might be invoked with:

EXAMPLE 2: Command-line CSV processing with column types

```
$ csvlint data.csv --datatypes:string,float,string,string
```

to enable the testing of the types of values in the columns of a CSV file, or with:

EXAMPLE 3: Command-line CSV processing with a schema

```
$ csvlint data.csv --schema:schema.json
```

to supply a schema that describes the contents of the file, against which it can be validated.

Metadata supplied in this way is called overriding, or user-supplied, metadata. Implementations SHOULD define how any options they define are mapped into the vocabulary defined in [tabular-metadata]. If the user selects existing metadata files, implementations MUST NOT use metadata located through the Link header (as described in section 5.2 Link Header) or site-wide location configuration (as described in section 5.3 Default Locations and Site-wide Location Configuration).

> NOTE
>
> Users should ensure that any metadata from those locations that they wish to use is explicitly incorporated into the overriding metadata that they use to process tabular data. Processors may provide facilities to make this easier by automatically merging metadata files from different locations, but this specification does not define how such merging is carried out.

5.2 Link Header

If the user has not supplied a metadata file as overriding metadata, described in section 5.1 Overriding Metadata, then when retrieving a tabular data file via HTTP, processors MUST retrieve the metadata file referenced by any `Link` header with:

- `rel="describedby"`, and
- `type="application/csvm+json"`, `type="application/ld+json"` or `type="application/json"`.

so long as this referenced metadata file describes the retrieved tabular data file (ie, contains a *table description* whose `url` matches the request URL).

If there is more than one valid metadata file linked to through multiple `Link` headers, then implementations MUST use the metadata file referenced by the last `Link` header.

For example, when the response to requesting a tab-separated file looks like:

EXAMPLE 4: HTTP response including Link headers

```
HTTP/1.1 200 OK
Content-Type: text/tab-separated-values
...
Link: <metadata.json>; rel="describedBy"; type="application/csvm+json"
```

an implementation must use the referenced `metadata.json` to supply metadata for processing the file.

If the metadata file found at this location does not explicitly include a reference to the requested tabular data file then it MUST be ignored. URLs MUST be normalized as described in section 6.3 URL Normalization.

> NOTE
>
> The `Link` header of the *metadata file* MAY include references to the CSV files it describes, using the `describes` relationship. For example, in the countries' metadata example, the server might return the following headers:

```
Link: <http://example.org/countries.csv>; rel="describes"; type="text/csv"
Link: <http://example.org/country_slice.csv>; rel="describes"; type="text/csv
```

However, locating the metadata SHOULD NOT depend on this mechanism.

5.3 Default Locations and Site-wide Location Configuration

If the user has not supplied a metadata file as overriding metadata, described in section 5.1 Overriding Metadata, and no applicable metadata file has been discovered through a `Link` header, described in section 5.2 Link Header, processors MUST attempt to locate a metadata documents through site-wide configuration.

In this case, processors MUST retrieve the file from the well-known URI `/.well-known/csvm`. (Well-known URIs are defined by [RFC5785].) If no such file is located (i.e. the response results in a client error `4xx` status code or a server error `5xx` status code), processors MUST proceed as if this file were found with the following content which defines default locations:

```
{+url}-metadata.json
csv-metadata.json
```

The response to retrieving `/.well-known/csvm` MAY be cached, subject to cache control directives. This includes caching an unsuccessful response such as a `404 Not Found`.

This file MUST contain a URI template, as defined by [URI-TEMPLATE], on each line. Starting with the first such URI template, processors MUST:

1. Expand the URI template, with the variable `url` being set to the URL of the requested tabular data file (with any fragment component of that URL removed).
2. Resolve the resulting URL against the URL of the requested tabular data file.
3. Attempt to retrieve a metadata document at that URL.
4. If no metadata document is found at that location, or if the metadata file found at the location does not explicitly include a reference to the relevant tabular data file, perform these same steps on the next URI template, otherwise use that metadata document.

For example, if the tabular data file is at `http://example.org/south-west/devon.csv` then processors must attempt to locate a well-known file at `http://example.org/.well-known/csvm`. If that file contains:

EXAMPLE 5

```
{+url}.json
csvm.json
/csvm?file={url}
```

the processor will first look for `http://example.org/south-west/devon.csv.json`. If there is no metadata file in that location, it will then look for `http://example.org/south-west/csvm.json`. Finally, if that also fails, it will look for `http://example.org/csvm?file=http://example.org/south-west/devon.csv.json`.

If no file were found at `http://example.org/.well-known/csvm`, the processor will use the default locations and try to retrieve metadata from `http://example.org/south-west/`

devon.csv-metadata.json and, if unsuccessful, http://example.org/south-west/csv-metadata.json.

5.4 Embedded Metadata

Most syntaxes for tabular data provide a facility for ***embedding metadata*** within the tabular data file itself. The definition of a syntax for tabular data SHOULD include a description of how the syntax maps to an annotated data model, and in particular how any embedded metadata is mapped into the vocabulary defined in [tabular-metadata]. Parsing based on the default dialect for CSV, as described in 8. Parsing Tabular Data, will extract column titles from the first row of a CSV file.

EXAMPLE 6: http://example.org/tree-ops.csv

```
GID,On Street,Species,Trim Cycle,Inventory Date
1,ADDISON AV,Celtis australis,Large Tree Routine Prune,10/18/2010
2,EMERSON ST,Liquidambar styraciflua,Large Tree Routine Prune,6/2/2010
```

The results of this can be found in section 8.2.1 Simple Example.

For another example, the following tab-delimited file contains embedded metadata where it is assumed that comments may be added using a #, and that the column types may be indicated using a #datatype annotation:

EXAMPLE 7: Tab-separated file containing embedded metadata

```
# publisher City of Palo Alto
# updated 12/31/2010
#name GID on_street species trim_cycle  inventory_date
#datatype string  string  string  string  date:M/D/YYYY
  GID On Street Species Trim Cycle  Inventory Date
  1 ADDISON AV  Celtis australis  Large Tree Routine Prune  10/18/2010
  2 EMERSON ST  Liquidambar styraciflua Large Tree Routine Prune  6/2/2010
```

A processor that recognises this format may be able to extract and make sense of this embedded metadata.

6. Processing Tables

This section describes how particular types of applications should process tabular data and metadata files.

In many cases, an application will start processing from a metadata file. In that case, the initial metadata file is treated as overriding metadata and the application MUST NOT continue to retrieve other available metadata about each of the tabular data files referenced by that initial metadata file other than embedded metadata.

In other cases, applications will start from a tabular data file, such as a CSV file, and locate metadata from that file. This metadata will be used to process the file as if the processor were starting from that metadata file.

For example, if a validator is passed a locally authored metadata file spending.json, which contains:

EXAMPLE 8: Metadata file referencing multiple tabular data files sharing a schema

```
{
  "tableSchema": "government-spending.csv",
  "tables": [{
    "url": "http://example.org/east-sussex-2015-03.csv",
  }, {
    "url": "http://example.org/east-sussex-2015-02.csv"
  }, ...
  ]
}
```

the validator would validate all the listed tables, using the locally defined schema at `government-spending.csv`. It would also use the metadata embedded in the referenced CSV files; for example, when processing `http://example.org/east-sussex-2015-03.csv`, it would use embedded metadata within that file to verify that the CSV is *compatible* with the metadata.

If a validator is passed a tabular data file `http://example.org/east-sussex-2015-03.csv`, the validator would use the metadata located from the CSV file: the first metadata file found through the `Link` headers found when retrieving that file, or located through a site-wide location configuration.

NOTE

Starting with a metadata file can remove the need to perform additional requests to locate linked metadata, or metadata retrieved through site-wide location configuration

6.1 Creating Annotated Tables

After locating metadata, metadata is *normalized* and coerced into a single *table group description*. When starting with a metadata file, this involves normalizing the provided metadata file and verifying that the embedded metadata for each tabular data file referenced from the metadata is *compatible* with the metadata. When starting with a tabular data file, this involves locating the first metadata file as described in section 5. Locating Metadata and normalizing into a single descriptor.

If processing starts with a tabular data file, implementations:

1 Retrieve the tabular data file.

2 Retrieve the first metadata file (`FM`) as described in section 5. Locating Metadata:

> **2.1** metadata supplied by the user (see section 5.1 Overriding Metadata).
>
> **2.2** metadata referenced from a *Link Header* that may be returned when retrieving the tabular data file (see section 5.2 Link Header).
>
> **2.3** metadata retrieved through a site-wide location configuration (see section 5.3 Default Locations and Site-wide Location Configuration).
>
> **2.4** embedded metadata as defined in section 5.4 Embedded Metadata with a single `tables` entry where the `url` property is set from that of the tabular data file.

3 Proceed as if the process starts with `FM`.

If the process starts with a metadata file:

1 Retrieve the metadata file yielding the metadata `UM` (which is treated as overriding metadata, see section 5.1 Overriding Metadata).

2 Normalize `UM` using the process defined in *Normalization* in [tabular-metadata], coercing `UM` into a *table group description*, if necessary.

3 For each table (`TM`) in `UM` in order, create one or more annotated tables:

3.1 Extract the dialect description (`DD`) from `UM` for the table associated with the tabular data file. If there is no such dialect description, extract the first available dialect description from a group of tables in which the tabular data file is described. Otherwise use the default dialect description.

3.2 If using the default dialect description, override default values in `DD` based on HTTP headers found when retrieving the tabular data file:

- If the media type from the `Content-Type` header is `text/tab-separated-values`, set *delimiter* to `TAB` in `DD`.
- If the `Content-Type` header includes the `header` parameter with a value of `absent`, set *header* to `false` in `DD`.
- If the `Content-Type` header includes the `charset` parameter, set *encoding* to this value in `DD`.

3.3 Parse the tabular data file, using `DD` as a guide, to create a basic tabular data model (`T`) and extract embedded metadata (`EM`), for example from the header line.

NOTE

This specification provides a non-normative definition for parsing CSV-based files, including the extraction of embedded metadata, in section 8. Parsing Tabular Data. This specification does not define any syntax for embedded metadata beyond this; whatever syntax is used, it's assumed that metadata can be mapped to the vocabulary defined in [tabular-metadata].

3.4 If a `Content-Language` HTTP header was found when retrieving the tabular data file, and the value provides a single language, set the *lang* inherited property to this value in `TM`, unless `TM` already has a *lang* inherited property.

3.5 Verify that `TM` is compatible with `EM` using the procedure defined in *Table Description Compatibility* in [tabular-metadata]; if `TM` is not compatible with `EM` validators MUST raise an error, other processors MUST generate a warning and continue processing.

3.6 Use the metadata `TM` to add annotations to the tabular data model `T` as described in Section 2 Annotating Tables in [tabular-metadata].

6.2 Metadata Compatibility

When processing a tabular data file using metadata as discovered using section 5. Locating Metadata, processors MUST ensure that the metadata and tabular data file are compatible, this is typically done by extracting embedded metadata from the tabular data file and determining that the provided or discovered metadata is compatible with the embedded metadata using the procedure defined in *Table Compatibility* in [tabular-metadata].

6.3 URL Normalization

Metadata Discovery and Compatibility involve comparing URLs. When comparing URLs, processors MUST use *Syntax-Based Normalization* as defined in [RFC3968]. Processors MUST perform *Scheme-Based Normalization* for HTTP (`80`) and HTTPS (`443`) and SHOULD perform *Scheme-Based Normalization* for other well-known schemes.

6.4 Parsing Cells

Unlike many other data formats, tabular data is designed to be read by humans. For that reason, it's common for data to be represented within tabular data in a human-readable way. The datatype, default, lang, null, required, and separator annotations provide the information needed to parse the string value of a cell into its (semantic) value annotation. This is used:

- by **validators** to check that the data in the table is in the expected format,
- by **converters** to parse the values before mapping them into values in the target of the conversion,
- when **displaying** data, to map it into formats that are meaningful for those viewing the data (as opposed to those publishing it), and
- when **inputting** data, to turn entered values into representations in a consistent format.

The process of parsing a cell creates a cell with annotations based on the original string value, parsed value and other column annotations and adds the cell to the list of cells in a row and cells in a column:

- The raw string value becomes the string value annotation on the cell.
- The ordered annotation on the column becomes the ordered annotation on the cell.
- The text direction annotation on the column becomes the text direction annotation on the cell.
- The row becomes the row annotation on the cell.
- The column becomes the column annotation on the cell.

After parsing, the cell value can be:

- `null`,
- a single value with an associated optional datatype or language, or
- a sequence of such values.

The process of parsing the string value into a single value or a list of values is as follows:

1. unless the datatype base is `string`, `json`, `xml`, `html` or `anyAtomicType`, replace all carriage return (`#xD`), line feed (`#xA`), and tab (`#x9`) characters with space characters.

2. unless the datatype base is `string`, `json`, `xml`, `html`, `anyAtomicType`, or `normalizedString`, strip leading and trailing whitespace from the string value and replace all instances of two or more whitespace characters with a single space character.

3. if the normalized string is an empty string, apply the remaining steps to the string given by the column default annotation.

4. if the column separator annotation is not `null` and the normalized string is an empty string, the cell value is an empty list. If the column required annotation is `true`, add an error to the list of errors for the cell.

5. if the column separator annotation is not `null`, the cell value is a list of values; set the list annotation on the cell to `true`, and create the cell value created by:

   5.1 if the normalized string is the same as any one of the values of the column null annotation, then the resulting value is `null`.

   5.2 split the normalized string at the character specified by the column separator annotation.

   5.3 unless the datatype base is `string` or `anyAtomicType`, strip leading and trailing whitespace from these strings.

   5.4 applying the remaining steps to each of the strings in turn.

6 if the string is an empty string, apply the remaining steps to the string given by the column default annotation.

7 if the string is the same as any one of the values of the column null annotation, then the resulting value is `null`. If the column separator annotation is `null` and the column required annotation is `true`, add an error to the list of errors for the cell.

8 parse the string using the datatype format if one is specified, as described below to give a value with an associated datatype. If the datatype base is `string`, or there is no `datatype`, the value has an associated language from the column lang annotation. If there are any errors, add them to the list of errors for the cell; in this case the value has a datatype of `string`; if the datatype base is `string`, or there is no `datatype`, the value has an associated language from the column lang annotation.

9 validate the value based on the length constraints described in section 4.6.1 Length Constraints, the value constraints described in section 4.6.2 Value Constraints and the datatype format annotation if one is specified, as described below. If there are any errors, add them to the list of errors for the cell.

The final value (or values) become the value annotation on the cell.

If there is a about URL annotation on the column, it becomes the about URL annotation on the cell, after being transformed into an absolute URL as described in *URI Template Properties* of [tabular-metadata].

If there is a property URL annotation on the column, it becomes the property URL annotation on the cell, after being transformed into an absolute URL as described in *URI Template Properties* of [tabular-metadata].

If there is a value URL annotation on the column, it becomes the value URL annotation on the cell, after being transformed into an absolute URL as described in *URI Template Properties* of [tabular-metadata]. The value URL annotation is `null` if the cell value is `null` and the column virtual annotation is `false`.

6.4.1 Parsing examples

This section is non-normative.

When datatype annotation is available, the value of a cell is the same as its string value. For example, a cell with a string value of `"99"` would similarly have the (semantic) value `"99"`.

If a datatype base is provided for the cell, that is used to create a (semantic) value for the cell. For example, if the metadata contains:

EXAMPLE 9

```
"datatype": "integer"
```

for the cell with the string value `"99"` then the value of that cell will be the integer `99`. A cell whose string value was not a valid integer (such as `"one"` or `"1.0"`) would be assigned that string value as its (semantic) value annotation, but also have a validation error listed in its errors annotation.

Sometimes data uses special codes to indicate unknown or null values. For example, a particular column might contain a number that is expected to be between `1` and `10`, with the string `99` used in the original tabular data file to indicate a null value. The metadata for such a column would include:

EXAMPLE 10

```
"datatype": {
  "base": "integer",
  "minimum": 1,
  "maximum": 10
},
"null": "99"
```

In this case, a cell with a string value of `"5"` would have the (semantic) value of the integer `5`; a cell with a string value of `"99"` would have the value `null`.

Similarly, a cell may be assigned a default value if the string value for the cell is empty. A configuration such as:

EXAMPLE 11

```
"datatype": {
  "base": "integer",
  "minimum": 1,
  "maximum": 10
},
"default": "5"
```

In this case, a cell whose string value is `""` would be assigned the value of the integer `5`. A cell whose string value contains whitespace, such as a single tab character, would also be assigned the value of the integer `5`: when the datatype is something other than `string` or `anyAtomicType`, leading and trailing whitespace is stripped from string values before the remainder of the processing is carried out.

Cells can contain sequences of values. For example, a cell might have the string value `"1 5 7.0"`. In this case, the separator is a space character. The appropriate configuration would be:

EXAMPLE 12

```
"datatype": {
  "base": "integer",
  "minimum": 1,
  "maximum": 10
},
"default": "5",
"separator": " "
```

and this would mean that the cell's value would be an array containing two integers and a string: `[1, 5, "7.0"]`. The final value of the array is a string because it is not a valid integer; the cell's errors annotation will also contain a validation error.

Also, with this configuration, if the string value of the cell were `""` (i.e. it was an empty cell) the value of the cell would be an empty list.

A cell value can be inserted into a URL created using a *URI template property* such as `valueUrl`. For example, if a cell with the string value `"1 5 7.0"` were in a column named `values`, defined with:

EXAMPLE 13

```
    "datatype": "decimal",
    "separator": " ",
    "valueUrl": "{?values}"
```

then after expansion of the URI template, the resulting `valueUrl` would be `?values=1.0,5.0,7.0`. The canonical representations of the decimal values are used within the URL.

6.4.2 Formats for numeric types

By default, numeric values must be in the formats defined in [xmlschema11-2]. It is not uncommon for numbers within tabular data to be formatted for human consumption, which may involve using commas for decimal points, grouping digits in the number using commas, or adding percent signs to the number.

If the datatype base is a numeric type, the datatype format annotation indicates the expected format for that number. Its value MUST be either a single string or an object with one or more of the properties:

decimalChar
: A string whose value is used to represent a decimal point within the number. The default value is `"."`. If the supplied value is not a string, implementations MUST issue a warning and proceed as if the property had not been specified.

groupChar
: A string whose value is used to group digits within the number. The default value is `null`. If the supplied value is not a string, implementations MUST issue a warning and proceed as if the property had not been specified.

pattern
: A number format pattern as defined in [UAX35]. Implementations MUST recognise number format patterns containing the symbols `0`, `#`, the specified decimalChar (or `"."` if unspecified), the specified groupChar (or `","` if unspecified), `E`, `+`, `%` and `‰`. Implementations MAY additionally recognise number format patterns containing other special pattern characters defined in [UAX35]. If the supplied value is not a string, or if it contains an invalid number format pattern or uses special pattern characters that the implementation does not recognise, implementations MUST issue a warning and proceed as if the property had not been specified.

If the datatype format annotation is a single string, this is interpreted in the same way as if it were an object with a pattern property whose value is that string.

If the groupChar is specified, but no pattern is supplied, when parsing the string value of a cell against this format specification, implementations MUST recognise and parse numbers that consist of:

1. an optional `+` or `-` sign,
2. followed by a decimal digit (`0-9`),
3. followed by any number of decimal digits (`0-9`) and the string specified as the groupChar,
4. followed by an optional decimalChar followed by one or more decimal digits (`0-9`),
5. followed by an optional exponent, consisting of an `E` followed by an optional `+` or `-` sign followed by one or more decimal digits (`0-9`), or
6. followed by an optional percent (`%`) or per-mille (`‰`) sign.

or that are one of the special values:

1. `NaN`,
2. `INF`, or
3. `-INF`.

Implementations MAY also recognise numeric values that are in any of the standard-decimal, standard-percent or standard-scientific formats listed in the Unicode Common Locale Data Repository.

Implementations MUST add a validation error to the errors annotation for the cell, and set the cell value to a string rather than a number if the string being parsed:

- is not in the format specified in the pattern, if one is defined
- otherwise, if the string
  - does not meet the numeric format defined above,
  - contains two consecutive groupChar strings,
- contains the decimalChar, if the datatype base is `integer` or one of its sub-types,
- contains an exponent, if the datatype base is `decimal` or one of its sub-types, or
- is one of the special values `NaN`, `INF`, or `-INF`, if the datatype base is `decimal` or one of its sub-types.

Implementations MUST use the sign, exponent, percent, and per-mille signs when parsing the string value of a cell to provide the value of the cell. For example, the string value `"-25%"` must be interpreted as `-0.25` and the string value `"1E6"` as `1000000`.

6.4.3 Formats for booleans

Boolean values may be represented in many ways aside from the standard `1` and `0` or `true` and `false`.

If the datatype base for a cell is `boolean`, the datatype format annotation provides the true value followed by the false value, separated by `|`. For example if `format` is `Y|N` then cells must hold either `Y` or `N` with `Y` meaning `true` and `N` meaning `false`. If the format does not follow this syntax, implementations MUST issue a warning and proceed as if no format had been provided.

The resulting cell value will be one or more boolean `true` or `false` values.

6.4.4 Formats for dates and times

By default, dates and times are assumed to be in the format defined in [xmlschema11-2]. However dates and times are commonly represented in tabular data in other formats.

If the datatype base is a date or time type, the datatype format annotation indicates the expected format for that date or time.

The supported date and time format patterns listed here are expressed in terms of the date field symbols defined in [UAX35]. These formats MUST be recognised by implementations and MUST be interpreted as defined in that specification. Implementations MAY additionally recognise other date format patterns. Implementations MUST issue a warning if the date format pattern is invalid or not recognised and proceed as if no date format pattern had been provided.

NOTE

For interoperability, authors of metadata documents SHOULD use only the formats listed in this section.

The following date format patterns MUST be recognized by implementations:

- `yyyy-MM-dd` e.g., `2015-03-22`
- `yyyyMMdd` e.g., `20150322`
- `dd-MM-yyyy` e.g., `22-03-2015`
- `d-M-yyyy` e.g., `22-3-2015`
- `MM-dd-yyyy` e.g., `03-22-2015`
- `M-d-yyyy` e.g., `3-22-2015`
- `dd/MM/yyyy` e.g., `22/03/2015`
- `d/M/yyyy` e.g., `22/3/2015`
- `MM/dd/yyyy` e.g., `03/22/2015`
- `M/d/yyyy` e.g., `3/22/2015`
- `dd.MM.yyyy` e.g., `22.03.2015`
- `d.M.yyyy` e.g., `22.3.2015`
- `MM.dd.yyyy` e.g., `03.22.2015`
- `M.d.yyyy` e.g., `3.22.2015`

The following time format patterns MUST be recognized by implementations:

- `HH:mm:ss.S` with one or more trailing `S` characters indicating the maximum number of fractional seconds e.g., `HH:mm:ss.SSS` for `15:02:37.143`
- `HH:mm:ss` e.g., `15:02:37`
- `HHmmss` e.g., `150237`
- `HH:mm` e.g., `15:02`
- `HHmm` e.g., `1502`

The following date/time format patterns MUST be recognized by implementations:

- `yyyy-MM-ddTHH:mm:ss.S` with one or more trailing `S` characters indicating the maximum number of fractional seconds e.g., `yyyy-MM-ddTHH:mm:ss.SSS` for `2015-03-15T15:02:37.143`
- `yyyy-MM-ddTHH:mm:ss` e.g., `2015-03-15T15:02:37`
- `yyyy-MM-ddTHH:mm` e.g., `2015-03-15T15:02`

- any of the date formats above, followed by a single space, followed by any of the time formats above, e.g., `M/d/yyyy HH:mm` for `3/22/2015 15:02` or `dd.MM.yyyy HH:mm:ss` for `22.03.2015 15:02:37`

Implementations MUST also recognise date, time, and date/time format patterns that end with timezone markers consisting of between one and three `x` or `X` characters, possibly after a single space. These MUST be interpreted as follows:

- `X` e.g., `-08`, `+0530`, or `Z` (minutes are optional)
- `XX` e.g., `-0800`, `+0530`, or `Z`
- `XXX` e.g., `-08:00`, `+05:30`, or `Z`
- `x` e.g., `-08` or `+0530` (`Z` is not permitted)
- `xx` e.g., `-0800` or `+0530` (`Z` is not permitted)
- `xxx` e.g., `-08:00` or `+05:30` (`Z` is not permitted)

For example, date format patterns could include `yyyy-MM-ddTHH:mm:ssXXX` for `2015-03-15T15:02:37Z` or `2015-03-15T15:02:37-05:00`, or `HH:mm x` for `15:02 -05`.

The cell value will one or more dates/time values extracted using the `format`.

NOTE

For simplicity, this version of this standard does not support abbreviated or full month or day names, or double digit years. Future versions of this standard may support other date and time formats, or general purpose date/time pattern strings. Authors of schemas SHOULD use appropriate regular expressions, along with the `string` datatype, for dates and times that use a format other than that specified here.

6.4.5 Formats for durations

Durations MUST be formatted and interpreted as defined in [xmlschema11-2], using the [ISO8601] format `-?PnYnMnDTnHnMnS`. For example, the duration `P1Y1D` is used for a year and a day; the duration `PT2H30M` for 2 hours and 30 minutes.

If the datatype base is a duration type, the datatype format annotation provides a regular expression for the string values, with syntax and processing defined by [ECMASCRIPT]. If the supplied value is not a valid regular expression, implementations MUST issue a warning and proceed as if no format had been provided.

NOTE

Authors are encouraged to be conservative in the regular expressions that they use, sticking to the basic features of regular expressions that are likely to be supported across implementations.

The cell value will be one or more durations extracted using the `format`.

6.4.6 Formats for other types

If the datatype base is not numeric, `boolean`, a date/time type, or a duration type, the datatype format annotation provides a regular expression for the string values, with syntax and processing defined by [ECMASCRIPT]. If the supplied value is not a valid regular expression, implementations MUST issue a warning and proceed as if no format had been provided.

NOTE

Authors are encouraged to be conservative in the regular expressions that they use, sticking to the basic features of regular expressions that are likely to be supported across implementations.

Values that are labelled as `html`, `xml`, or `json` SHOULD NOT be validated against those formats.

NOTE

Metadata creators who wish to check the syntax of HTML, XML, or JSON within tabular data should use the datatype format annotation to specify a regular expression against which such values will be tested.

6.5 Presenting Tables

This section is non-normative.

When presenting tables, implementations should:

- use the table direction annotation on each table, and the text direction annotation on each cell, to determine the ordering of columns and characters within cells, as described in section 6.5.1 Bidirectional Tables
- use the titles annotation on each column to provide a header for the column, selecting the first title in a language based on the user's locale and preferences, as described in section 6.5.2 Column and row labelling
- add links to headers based on the property URLs of the cells in the first row of the table
- present cell values, particularly boolean, numeric and date/time values, in a lexical form based on the user's locale and preferences
- add links to the presentation of rows and cells based on the about URL and value URL annotations on cells
- highlight or otherwise indicate cells with errors
- provide a way of viewing non-core annotations on table groups, tables, columns, rows and cells
- provide links to download the raw tabular data file that is being displayed

6.5.1 Bidirectional Tables

There are two levels of bidirectionality to consider when displaying tables: the directionality of the table (i.e., whether the columns should be arranged left-to-right or right-to-left) and the directionality of the content of individual cells.

The table direction annotation on the table provides information about the desired display of the columns in the table. If table direction is `ltr` then the first column should be displayed on the left and the last column on the right. If table direction is `rtl` then the first column should be displayed on the right and the last column on the left.

If table direction is `auto` then tables should be displayed with attention to the bidirectionality of the content of the cells in the table. Specifically, the values of the cells in the table should be scanned breadth first: from the first cell in the first column through to the last cell in the first row, down to the last cell in the last column. If the first character in the table with a strong type as defined in [BIDI] indicates a RTL directionality, the table should be displayed with the first column on the right and the last column on the left. Otherwise, the table should be displayed with the first column on the left and the last column on the right. Characters such as whitespace, quotes, commas, and numbers do not have a strong type, and therefore are skipped when identifying the character that determines the directionality of the table.

Implementations should enable user preferences to override the indicated metadata about the directionality of the table.

Once the directionality of the table has been determined, each cell within the table should be considered as a separate paragraph, as defined by the Unicode Bidirectional Algorithm (UBA) in [BIDI]. The directionality for the cell is determined by looking at the text direction annotation for the cell, as follows:

1. If the text direction is `ltr` then the base direction for the cell content should be set to left-to-right.
2. If the text direction is `rtl` then the base direction for the cell content should be set to right-to-left.
3. If the text direction is `auto` then the base direction for the cell content should be set to the direction determined by the first character in the cell with a strong type as defined in [BIDI].

NOTE

If the *textDirection* property in metadata has the value `"inherit"`, the text direction annotation for a cell inherits its value from the table direction annotation on the table.

When the titles of a column are displayed, these should be displayed in the direction determined by the first character in the title with a strong type as defined in [BIDI]. Titles for the same column in different languages may be displayed in different directions.

6.5.2 Column and row labelling

The labelling of columns and rows helps those who are attempting to understand the content of a table to grasp what a particular cell means. Implementations should present appropriate titles for columns, and ensure that the most important information in a row is kept apparent to the user, to aid their understanding. For example:

- a table presented on the screen might retain certain columns in view so that readers can easily glance at the identifying information in each row
- as the user moves focus into a cell, screen readers announce a label for the new column if the user has changed column, or for the new row if the user has changed row

When labelling a column, either on the screen or aurally, implementations should use the first available of:

1. the column's titles in the preferred language of the user, or with an undefined language if there is no title available in a preferred language; there may be multiple such titles in which case all should be announced
2. the column's name
3. the column's number

When labelling a row, either on the screen or aurally, implementations should use the first available of:

1. the row's titles in the preferred language of the user, or with an undefined language if there is no title available in a preferred language; there may be multiple such titles in which case all should be announced
2. the values of the cells in the row's primary key
3. the row's number

6.6 Validating Tables

Validators test whether given tabular data files adhere to the structure defined within a *schema*. Validators MUST raise errors (and halt processing) and issue warnings (and continue processing) as defined in [tabular-metadata]. In addition, validators MUST raise errors but MAY continue validating in the following situations:

- if the *table description* is not compatible with the embedded metadata extracted from the tabular data file, as defined in *Table Compatibility* in [tabular-metadata].
- if there is more than one row with the same primary key, that is where the cells listed for the primary key for the row have the same values as the cells listed for the primary key for another row,
- for each row that does not have a unique referenced row for each of the foreign keys on the table in which the row appears, or
- for each error on each cell.

6.7 Converting Tables

Conversions of tabular data to other formats operate over a annotated table constructed as defined in *Annotating Tables* in [tabular-metadata]. The mechanics of these conversions to other formats are defined in other specifications such as [csv2json] and [csv2rdf].

Conversion specifications MUST define a default mapping from an annotated table that lacks any annotations (i.e., that is equivalent to an un-annotated table).

Conversion specifications MUST use the *property value* of the `propertyUrl` of a column as the basis for naming machine-readable fields in the target format, such as the name of the equivalent element or attribute in XML, property in JSON or property URI in RDF.

Conversion specifications MAY use any of the annotations found on an annotated table group, table, column, row or cell, including non-core annotations, to adjust the mapping into another format.

Conversion specifications MAY define additional annotations, not defined in this specification, which are specifically used when converting to the target format of the conversion. For example, a conversion to XML might specify a `http://example.org/conversion/xml/element-or-attribute` property on columns that determines whether a particular column is represented through an element or an attribute in the data.

7. Best Practice CSV

This section is non-normative.

There is no standard for CSV, and there are many variants of CSV used on the web today. This section defines a method for expressing tabular data adhering to the annotated tabular data model in CSV. Authors are encouraged to adhere to the constraints described in this section as implementations should process such CSV files consistently.

NOTE

This syntax is not compliant with `text/csv` as defined in [RFC4180] in that it permits line endings other than `CRLF`. Supporting `LF` line endings is important for data formats that are used on non-Windows platforms. However, all files that adhere to [RFC4180]'s definition of CSV meet the constraints described in this section.

Developing a standard for CSV is outside the scope of the Working Group. The details here aim to help shape any future standard.

7.1 Content Type

The appropriate content type for a CSV file is `text/csv`. For example, when a CSV file is transmitted via HTTP, the HTTP response should include a `Content-Type` header with the value `text/csv`:

```
Content-Type: text/csv
```

7.2 Encoding

CSV files should be encoded using UTF-8, and should be in Unicode Normal Form C as defined in [UAX15]. If a CSV file is not encoded using UTF-8, the encoding should be specified through the `charset` parameter in the `Content-Type` header:

```
Content-Type: text/csv;charset=ISO-8859-1
```

7.3 Line Endings

The ends of rows in a CSV file should be `CRLF` (`U+000D U+000A`) but may be `LF` (`U+000A`). Line endings within escaped cells are not normalised.

7.4 Lines

Each line of a CSV file should contain the same number of comma-separated values.

Values that contain commas, line endings, or double quotes should be escaped by having the entire value wrapped in double quotes. There should not be whitespace before or after the double quotes. Within these escaped cells, any double quotes should be escaped with two double quotes (`""`).

7.4.1 Headers

The first line of a CSV file should contain a comma-separated list of names of columns. This is known as the ***header line*** and provides titles for the columns. There are no constraints on these titles.

If a CSV file does not include a header line, this should be specified using the `header` parameter of the media type:

```
Content-Type: text/csv;header=absent
```

7.5 Grammar

This grammar is a generalization of that defined in [RFC4180] and is included for reference only.

The EBNF used here is defined in XML 1.0 [EBNF-NOTATION].

```
[1] csv      ::=header record+
[2] header   ::=record
[3] record   ::=fields #x0D? #x0A
[4] fields   ::=field ("," fields)*
[5] field    ::=WS* rawfield WS*
[6] rawfield::='"' QCHAR* '"' |SCHAR*
[7] QCHAR    ::=[^"] |'""'
[8] SCHAR    ::=[^",#x0A#x0D]
[9] WS       ::=[#x20#x09]
```

8. Parsing Tabular Data

This section is non-normative.

As described in section 7. Best Practice CSV, there may be many formats which an application might interpret into the tabular data model described in section 4. Tabular Data Models, including using different separators or fixed format tables, multiple tables within a single file, or ones that have metadata lines before a table header.

> NOTE
>
> Standardizing the parsing of CSV is outside the chartered scope of the Working Group. This non-normative section is intended to help the creators of parsers handle the wide variety of CSV-based formats that they may encounter due to the current lack of standardization of the format.

This section describes an algorithm for parsing formats that do not adhere to the constraints described in section 7. Best Practice CSV, as well as those that do, and extracting embedded metadata. The parsing algorithm uses the following flags. These may be set by metadata properties found while Locating Metadata, including through user input (see Overriding Metadata), or through the inclusion of a *dialect description* within a metadata file:

comment prefix
: A string that, when it appears at the beginning of a row, indicates that the row is a comment that should be associated as a `rdfs:comment` annotation to the table. This is set by the `commentPrefix` property of a *dialect description*. The default is `null`, which means no rows are treated as comments. A value other than `null` may mean that the source numbers of rows are different from their numbers.

delimiter
: The separator between cells, set by the `delimiter` property of a *dialect description*. The default is `,`.

encoding
: The character encoding for the file, one of the encodings listed in [encoding], set by the `encoding` property of a *dialect description*. The default is `utf-8`.

escape character
: The string that is used to escape the quote character within escaped cells, or `null`, set by the `doubleQuote` property of a *dialect description*. The default is `"` (such that `""` is used to escape `"` within an escaped cell).

header row count
: The number of ***header rows*** (following the skipped rows) in the file, set by the `header` or `headerRowCount` property of a *dialect description*. The default is `1`. A value other than `0` will mean that the source numbers of rows will be different from their numbers.

line terminators
: The strings that can be used at the end of a row, set by the `lineTerminators` property of a *dialect description*. The default is `[CRLF, LF]`.

quote character
: The string that is used around escaped cells, or `null`, set by the `quoteChar` property of a *dialect description*. The default is `"`.

skip blank rows
: Indicates whether to ignore wholly empty rows (i.e. rows in which all the cells are empty), set by the `skipBlankRows` property of a *dialect description*. The default is `false`. A value other than `false` may mean that the source numbers of rows are different from their numbers.

skip columns
: The number of columns to skip at the beginning of each row, set by the `skipColumns` property of a *dialect description*. The default is `0`. A value other than `0` will mean that the source numbers of columns will be different from their numbers.

skip rows
: The number of rows to skip at the beginning of the file, before a header row or tabular data, set by the `skipRows` property of a *dialect description*. The default is `0`. A value greater than `0` will mean that the source numbers of rows will be different from their numbers.

trim
: Indicates whether to trim whitespace around cells; may be `true`, `false`, `start`, or `end`, set by the `skipInitialSpace` or `trim` property of a *dialect description*. The default is `true`.

The algorithm for using these flags to parse a document containing tabular data to create a basic annotated tabular data model and to extract embedded metadata is as follows:

1 Create a new table *T* with the annotations:
- columns set to an empty list
- rows set to an empty list
- id set to `null`
- url set to the location of the file, if known, or `null`
- table direction set to `auto`
- suppress output set to `false`
- notes set to `false`
- foreign keys set to an empty list
- transformations set to an empty list

2 Create a metadata document structure *M* that looks like:

```
{
  "@context": "http://www.w3.org/ns/csvw",
  "rdfs:comment": []
  "tableSchema": {
    "columns": []
  }
}
```

3 If the URL of the tabular data file being parsed is known, set the `url` property on *M* to that URL.

4 Set ***source row number*** to `1`.

5 Read the file using the encoding, as specified in [encoding], using the *replacement error mode*. If the encoding is not a Unicode encoding, use a normalizing transcoder to normalize into Unicode Normal Form C as defined in [UAX15].

NOTE

The *replacement error mode* ensures that any non-Unicode characters within the CSV file are replaced by U+FFFD, ensuring that strings within the tabular data model such as column titles and cell string values only contain valid Unicode characters.

6 Repeat the following the number of times indicated by skip rows:

- **6.1** Read a row to provide the row content.
- **6.2** If the comment prefix is not `null` and the row content begins with the comment prefix, strip that prefix from the row content, and add the resulting string to the `M.rdfs:comment` array.
- **6.3** Otherwise, if the row content is not an empty string, add the row content to the `M.rdfs:comment` array.
- **6.4** Add `1` to the source row number.

7 Repeat the following the number of times indicated by header row count:

- **7.1** Read a row to provide the row content.
- **7.2** If the comment prefix is not `null` and the row content begins with the comment prefix, strip that prefix from the row content, and add the resulting string to the `M.rdfs:comment` array.
- **7.3** Otherwise, parse the row to provide a list of cell values, and:
  - **7.3.1** Remove the first skip columns number of values from the list of cell values.
  - **7.3.2** For each of the remaining values at index *i* in the list of cell values:
    - **7.3.2.1** If the value at index *i* in the list of cell values is an empty string or consists only of whitespace, do nothing.
    - **7.3.2.2** Otherwise, if there is no column description object at index *i* in `M.tableSchema.columns`, create a new one with a `title` property whose value is an array containing a single value that is the value at index *i* in the list of cell values.
    - **7.3.2.3** Otherwise, add the value at index *i* in the list of cell values to the array at `M.tableSchema.columns[i].title`.
- **7.4** Add `1` to the source row number.

8 If header row count is zero, create an empty column description object in `M.tableSchema.columns` for each column in the current row after skip columns.

9 Set ***row number*** to `1`.

10 While it is possible to read another row, do the following:

10.1 Set the ***source column number*** to `1`.

10.2 Read a row to provide the row content.

10.3 If the comment prefix is not `null` and the row content begins with the comment prefix, strip that prefix from the row content, and add the resulting string to the *M*`.rdfs:comment` array.

10.4 Otherwise, parse the row to provide a list of cell values, and:

10.4.1 If all of the values in the list of cell values are empty strings, and skip blank rows is `true`, add `1` to the source row number and move on to process the next row.

10.4.2 Otherwise, create a new row *R*, with:

- table set to *T*
- number set to row number
- source number set to source row number
- primary key set to an empty list
- referenced rows set to an empty list
- cells set to an empty list

10.4.3 Append *R* to the rows of table *T*.

10.4.4 Remove the first skip columns number of values from the list of cell values and add that number to the source column number.

10.4.5 For each of the remaining values at index *i* in the list of cell values (where *i* starts at `1`):

10.4.5.1 Identify the column *C* at index *i* within the columns of table *T*. If there is no such column:

10.4.5.1.1 Create a new column *C* with:

- table set to *T*
- number set to *i*
- source number set to source column number
- name set to `null`
- titles set to an empty list
- virtual set to `false`
- suppress output set to `false`
- datatype set to `string`
- default set to an empty string
- lang set to `und`
- null set to an empty string
- ordered set to `false`
- required set to `false`
- separator set to `null`
- text direction set to `auto`
- about URL set to `null`
- property URL set to `null`
- value URL set to `null`
- cells set to an empty list

10.4.5.1.2 Append *C* to the columns of table *T* (at index *i*).

10.4.5.2 Create a new cell *D*, with:

- table set to *T*

- column set to *C*
- row set to *R*
- string value set to the value at index *i* in the list of cell values
- value set to the value at index *i* in the list of cell values
- errors set to an empty list
- text direction set to `auto`
- ordered set to `false`
- about URL set to `null`
- property URL set to `null`
- value URL set to `null`

10.4.5.3 Append cell *D* to the cells of column *C*.

10.4.5.4 Append cell *D* to the cells of row *R* (at index *i*).

10.4.5.5 Add `1` to the source column number.

10.5 Add `1` to the source row number.

11 If `M.rdfs:comment` is an empty array, remove the `rdfs:comment` property from *M*.

12 Return the table *T* and the embedded metadata *M*.

To ***read a row*** to provide ***row content***, perform the following steps:

1 Set the row content to an empty string.

2 Read initial characters and process as follows:

2.1 If the string starts with the escape character followed by the quote character, append both strings to the row content, and move on to process the string following the quote character.

2.2 Otherwise, if the string starts with the escape character and the escape character is not the same as the quote character, append the escape character and the single character following it to the row content and move on to process the string following that character.

2.3 Otherwise, if the string starts with the quote character, append the quoted value obtained by reading a quoted value to the row content and move on to process the string following the quoted value.

2.4 Otherwise, if the string starts with one of the line terminators, return the row content.

2.5 Otherwise, append the first character to the row content and move on to process the string following that character.

3 If there are no more characters to read, return the row content.

To ***read a quoted value*** to provide a ***quoted value***, perform the following steps:

1 Set the quoted value to an empty string.

2 Read the initial quote character and add a quote character to the quoted value.

3 Read initial characters and process as follows:

3.1 If the string starts with the escape character followed by the quote character, append both strings to the quoted value, and move on to process the string following the quote character.

3.2 Otherwise, if string starts with the escape character and the escape character is not the same as the quote character, append the escape character and the character following it to the quoted value and move on to process the string following that character.

3.3 Otherwise, if the string starts with the quote character, return the quoted value.

3.4 Otherwise, append the first character to the quoted value and move on to process the string following that character.

To ***parse a row*** to provide a ***list of cell values***, perform the following steps:

1 Set the list of cell values to an empty list and the ***current cell value*** to an empty string.

2 Set the ***quoted*** flag to `false`.

3 Read initial characters and process as follows:

3.1 If the string starts with the escape character followed by the quote character, append the quote character to the current cell value, and move on to process the string following the quote character.

3.2 Otherwise, if the string starts with the escape character and the escape character is not the same as the quote character, append the character following the escape character to the current cell value and move on to process the string following that character.

3.3 Otherwise, if the string starts with the quote character then:

3.3.1 If quoted is `false`, set the quoted flag to `true`, and move on to process the remaining string. If the current cell value is not an empty string, raise an error.

3.3.2 Otherwise, set quoted to `false`, and move on to process the remaining string. If the remaining string does not start with the delimiter, raise an error.

3.4 Otherwise, if the string starts with the delimiter, then:

3.4.1 If quoted is `true`, append the delimiter string to the current cell value and move on to process the remaining string.

3.4.2 Otherwise, conditionally trim the current cell value, add the resulting trimmed cell value to the list of cell values and move on to process the following string.

3.5 Otherwise, append the first character to the current cell value and move on to process the remaining string.

4 If there are no more characters to read, conditionally trim the current cell value, add the resulting trimmed cell value to the list of cell values and return the list of cell values.

To ***conditionally trim a cell value*** to provide a ***trimmed cell value***, perform the following steps:

1 Set the trimmed cell value to the provided cell value.

2 If trim is `true` or `start` then remove any leading whitespace from the start of the trimmed cell value and move on to the next step.

3 If trim is `true` or `end` then remove any trailing whitespace from the end of the trimmed cell value and move on to the next step.

4 Return the trimmed cell value.

NOTE

This parsing algorithm does not account for the possibility of there being more than one area of tabular data within a single CSV file.

8.1 Bidirectionality in CSV Files

This section is non-normative.

Bidirectional content does not alter the definition of rows or the assignment of cells to columns. Whether or not a CSV file contains right-to-left characters, the first column's content is the first cell of each row, which is the text prior to the first occurrence of a comma within that row.

For example, Egyptian Referendum results are available as a CSV file at https://egelections-2011.appspot.com/Referendum2012/results/csv/EG.csv. Over the wire and in non-Unicode-aware text editors, the CSV looks like:

```
|ا|ل|م|ح|ا|ف|ظ|ة|,|ن|س|ب|ة| |م|و|ا|ف|ق|,|ن|س|ب|ة| |غ|ي|ر| |م|و|ا|ف|ق|,|ع|د|د|
|ا|ل|ق|ل|ي|و|ب|ي|ة|,60.0,40.0,"2,639,808","853,125","15,224",32.9,"512,055","3
|ا|ل|ج|ي|ز|ة|,66.7,33.3,"4,383,701","1,493,092","24,105",34.6,"995,417","497,6
|ا|ل|ق|ا|ه|ر|ة|,43.2,56.8,"6,580,478","2,254,698","36,342",34.8,"974,371","1,2
|ق|ن|ا|,84.5,15.5,"1,629,713","364,509","6,743",22.8,"307,839","56,670"
...
```

Within this CSV file, the first column appears as the content of each line before the first comma and is named المحافظة (appearing at the start of each row as |ا|ل|م|ح|ا|ف|ظ|ة| in the example, which is displaying the relevant characters from left to right in the order they appear "on the wire").

The CSV translates to a table model that looks like:

Column / Row	column 1	column 2	column 3	column 4	column 5	column 6	column 7	column 8	column 9
column names	المحافظة	نسبة موافق	نسبة غير موافق	عدد الناخبين	الأصوات الصحيحة	الأصوات الباطلة	نسبة المشاركة	موافق	غير موافق
row 1	القليوبية	60.0	40.0	2,639,808	853,125	15,224	32.9	512,055	341,070
row 2	الجيزة	66.7	33.3	4,383,701	1,493,092	24,105	34.6	995,417	497,675
row 3	القاهرة	43.2	56.8	6,580,478	2,254,698	36,342	34.8	974,371	1,280,327
row 4	قنا	84.5	15.5	1,629,713	364,509	6,743	22.8	307,839	56,670

The fragment identifier #col=3 identifies the third of the columns, named نسبة غير موافق (appearing as |ن|س|ب|ة| |غ|ي|ر| |م|و|ا|ف|ق| in the example).

section 6.5.1 Bidirectional Tables defines how this table model should be displayed by compliant applications, and how metadata can affect the display. The default is for the display to be determined by the content of the table. For example, if this CSV were turned into an HTML table for display into a web page, it should be displayed with the first column on the right and the last on the left, as follows:

المحافظة	نسبة موافق	نسبة غير موافق	عدد الناخبين	الأصوات الصحيحة	الأصوات الباطلة	نسبة المشاركة	موافق	غير موافق
القليوبية	60.0	40.0	2,639,808	853,125	15,224	32.9	512,055	341,070
الجيزة	66.7	33.3	4,383,701	1,493,092	24,105	34.6	995,417	497,675
القاهرة	43.2	56.8	6,580,478	2,254,698	36,342	34.8	974,371	1,280,327
قنا	84.5	15.5	1,629,713	364,509	6,743	22.8	307,839	56,670

The fragment identifier `#col=3` still identifies the third of the columns, named `نسبة غير موافق`, which appears in the HTML display as the third column from the right and is what those who read right-to-left would think of as the third column.

Note that this display matches that shown on the original website.

8.2 Examples

8.2.1 Simple Example

A simple CSV file that complies to the constraints described in section 7. Best Practice CSV, at `http://example.org/tree-ops.csv`, might look like:

EXAMPLE 14: http://example.org/tree-ops.csv

```
GID,On Street,Species,Trim Cycle,Inventory Date
1,ADDISON AV,Celtis australis,Large Tree Routine Prune,10/18/2010
2,EMERSON ST,Liquidambar styraciflua,Large Tree Routine Prune,6/2/2010
```

Parsing this file results in an annotated tabular data model of a single table *T* with five columns and two rows. The columns have the annotations shown in the following table:

id	core annotations				
	table	**number**	**source number**	**cells**	**titles**
C1	*T*	1	1	*C1.1*, *C2.1*	`GID`
C2	*T*	2	2	*C1.2*, *C2.2*	`On Street`
C3	*T*	3	3	*C1.3*, *C2.3*	`Species`
C4	*T*	4	4	*C1.4*, *C2.4*	`Trim Cycle`
C5	*T*	5	5	*C1.5*, *C2.5*	`Inventory Date`

The extracted embedded metadata, as defined in [tabular-metadata], would look like:

EXAMPLE 15: tree-ops.csv Embedded Metadata

```
{
  "@type": "Table",
  "url": "http://example.org/tree-ops.csv",
  "tableSchema": {
    "columns": [
      {"titles": [ "GID" ]},
      {"titles": [ "On Street" ]},
      {"titles": [ "Species" ]},
      {"titles": [ "Trim Cycle" ]},
      {"titles": [ "Inventory Date" ]}
```

```
        ]
      }
    }
```

The rows have the annotations shown in the following table:

id	core annotations			
	table	number	source number	cells
R1	*T*	1	2	*C1.1*, *C1.2*, *C1.3*, *C1.4*, *C1.5*
R2	*T*	2	3	*C2.1*, *C2.2*, *C2.3*, *C2.4*, *C2.5*

NOTE

The source number of each row is offset by one from the number of each row because in the source CSV file, the header line is the first line. It is possible to reconstruct a [RFC7111] compliant reference to the first record in the original CSV file (`http://example.org/tree-ops.csv#row=2`) using the value of the row's source number. This enables implementations to retain provenance between the table model and the original file.

The cells have the annotations shown in the following table (note that the values of all the cells in the table are strings, denoted by the double quotes in the table below):

id	core annotations				
	table	column	row	string value	value
C1.1	*T*	*C1*	*R1*	`"1"`	`"1"`
C1.2	*T*	*C2*	*R1*	`"ADDISON AV"`	`"ADDISON AV"`
C1.3	*T*	*C3*	*R1*	`"Celtis australis"`	`"Celtis australis"`
C1.4	*T*	*C4*	*R1*	`"Large Tree Routine Prune"`	`"Large Tree Routine Prune"`
C1.5	*T*	*C5*	*R1*	`"10/18/2010"`	`"10/18/2010"`
C2.1	*T*	*C1*	*R2*	`"2"`	`"2"`
C2.2	*T*	*C2*	*R2*	`"EMERSON ST"`	`"EMERSON ST"`
C2.3	*T*	*C3*	*R2*	`"Liquidambar styraciflua"`	`"Liquidambar styraciflua"`
C2.4	*T*	*C4*	*R2*	`"Large Tree Routine Prune"`	`"Large Tree Routine Prune"`
C2.5	*T*	*C5*	*R2*	`"6/2/2010"`	`"6/2/2010"`

8.2.1.1 Using Overriding Metadata

The tools that the consumer of this data uses may provide a mechanism for overriding the metadata that has been provided within the file itself. For example, they might enable the consumer to add machine-readable names to the columns, or to mark the fifth column as holding a date in the format `M/D/YYYY`. These facilities are implementation defined; the code for invoking a Javascript-based parser might look like:

EXAMPLE 16: Javascript implementation configuration

```
data.parse({
  "column-names": ["GID", "on_street", "species", "trim_cycle", "inventory_date
  "datatypes": ["string", "string", "string", "string", "date"],
  "formats": [null,null,null,null,"M/D/YYYY"]
});
```

This is equivalent to a metadata file expressed in the syntax defined in [tabular-metadata], looking like:

EXAMPLE 17: Equivalent metadata syntax

```
{
  "@type": "Table",
  "url": "http://example.org/tree-ops.csv",
  "tableSchema": {
    "columns": [{
      "name": "GID",
      "datatype": "string"
    }, {
      "name": "on_street",
      "datatype": "string"
    }, {
      "name": "species",
      "datatype": "string"
    }, {
      "name": "trim_cycle",
      "datatype": "string"
    }, {
      "name": "inventory_date",
      "datatype": {
        "base": "date",
        "format": "M/d/yyyy"
      }
    }]
  }
}
```

This would be merged with the embedded metadata found in the CSV file, providing the titles for the columns to create:

EXAMPLE 18: Merged metadata

```
{
  "@type": "Table",
  "url": "http://example.org/tree-ops.csv",
  "tableSchema": {
    "columns": [{
      "name": "GID",
      "titles": "GID",
      "datatype": "string"
    }, {
      "name": "on_street",
      "titles": "On Street",
      "datatype": "string"
    }, {
      "name": "species",
      "titles": "Species",
      "datatype": "string"
    }, {
      "name": "trim_cycle",
      "titles": "Trim Cycle",
      "datatype": "string"
    }, {
      "name": "inventory_date",
      "titles": "Inventory Date",
      "datatype": {
        "base": "date",
        "format": "M/d/yyyy"
      }
    }]
  }
}
```

The processor can then create an annotated tabular data model that included `name` annotations on the columns, and `datatype` annotations on the cells, and created cells whose values were of appropriate types (in the case of this Javascript implementation, the cells in the last column would be `Date` objects, for example).

Assuming this kind of implementation-defined parsing, the columns would then have the annotations shown in the following table:

id	core annotations						
	table	number	source number	cells	name	titles	datatype
C1	*T*	1	1	*C1.1*, *C2.1*	`GID`	`GID`	`string`
C2	*T*	2	2	*C1.2*, *C2.2*	`on_street`	`On Street`	`string`
C3	*T*	3	3	*C1.3*, *C2.3*	`species`	`Species`	`string`
C4	*T*	4	4	*C1.4*, *C2.4*	`trim_cycle`	`Trim Cycle`	`string`
C5	*T*	5	5	*C1.5*, *C2.5*	`inventory_date`	`Inventory Date`	`{ "base": "date", "format": "M/d/yyyy" }`

The cells have the annotations shown in the following table. Because of the overrides provided by the consumer to guide the parsing, and the way the parser works, the cells in the `Inventory Date` column (cells *C1.5* and *C2.5*) have values that are parsed dates rather than unparsed strings.

id	core annotations				
	table	column	row	string value	value
C1.1	*T*	*C1*	*R1*	`"1"`	`"1"`
C1.2	*T*	*C2*	*R1*	`"ADDISON AV"`	`"ADDISON AV"`
C1.3	*T*	*C3*	*R1*	`"Celtis australis"`	`"Celtis australis"`
C1.4	*T*	*C4*	*R1*	`"Large Tree Routine Prune"`	`"Large Tree Routine Prune"`
C1.5	*T*	*C5*	*R1*	`"10/18/2010"`	`2010-10-18`
C2.1	*T*	*C1*	*R2*	`"2"`	`"2"`
C2.2	*T*	*C2*	*R2*	`"EMERSON ST"`	`"EMERSON ST"`

id	core annotations				
	table	column	row	string value	value
C2.3	*T*	*C3*	*R2*	"Liquidambar styraciflua"	"Liquidambar styraciflua"
C2.4	*T*	*C4*	*R2*	"Large Tree Routine Prune"	"Large Tree Routine Prune"
C2.5	*T*	*C5*	*R2*	"6/2/2010"	2010-06-02

8.2.1.2 Using a Metadata File

A similar set of annotations could be provided through a metadata file, located as discussed in section 5. Locating Metadata and defined in [tabular-metadata]. For example, this might look like:

EXAMPLE 19: http://example.org/tree-ops.csv-metadata.json

```
{
  "@context": ["http://www.w3.org/ns/csvw", {"@language": "en"}],
  "url": "tree-ops.csv",
  "dc:title": "Tree Operations",
  "dcat:keyword": ["tree", "street", "maintenance"],
  "dc:publisher": {
    "schema:name": "Example Municipality",
    "schema:url": {"@id": "http://example.org"}
  },
  "dc:license": {"@id": "http://opendefinition.org/licenses/cc-by/"},
  "dc:modified": {"@value": "2010-12-31", "@type": "xsd:date"},
  "tableSchema": {
    "columns": [{
      "name": "GID",
      "titles": ["GID", "Generic Identifier"],
      "dc:description": "An identifier for the operation on a tree.",
      "datatype": "string",
      "required": true
    }, {
      "name": "on_street",
      "titles": "On Street",
      "dc:description": "The street that the tree is on.",
      "datatype": "string"
    }, {
      "name": "species",
      "titles": "Species",
      "dc:description": "The species of the tree.",
      "datatype": "string"
    }, {
      "name": "trim_cycle",
      "titles": "Trim Cycle",
      "dc:description": "The operation performed on the tree.",
      "datatype": "string"
    }, {
      "name": "inventory_date",
      "titles": "Inventory Date",
      "dc:description": "The date of the operation that was performed.",
      "datatype": {"base": "date", "format": "M/d/yyyy"}
    }],
    "primaryKey": "GID",
    "aboutUrl": "#gid-{GID}"
```

```
      }
    }
```

The annotated tabular data model generated from this would be more sophisticated again. The table itself would have the following annotations:

dc:title
    {"@value": "Tree Operations", "@language": "en"}
dcat:keyword
    [{"@value": "tree", "@language", "en"}, {"@value": "street", "@language": "en"}, {"@value": "maintenance", "@language": "en"}]
dc:publisher
    [{ "schema:name": "Example Municipality", "schema:url": {"@id": "http://example.org"} }]
dc:license
    {"@id": "http://opendefinition.org/licenses/cc-by/"}
dc:modified
    {"@value": "2010-12-31", "@type": "date"}

The columns would have the annotations shown in the following table:

id	core annotations							other annotations
	table	number	source number	cells	name	titles	datatype	**dc:description**
C1	*T*	1	1	*C1.1*, *C2.1*	GID	GID, Generic Identifier	string	An identifier for the operation on a tree.
C2	*T*	2	2	*C1.2*, *C2.2*	on_street	On Street	string	The street that the tree is on.
C3	*T*	3	3	*C1.3*, *C2.3*	species	Species	string	The species of the tree.
C4	*T*	4	4	*C1.4*, *C2.4*	trim_cycle	Trim Cycle	string	The operation performed on the tree.
C5	*T*	5	5	*C1.5*, *C2.5*	inventory_date	Inventory Date	{ "base": "date", "format": "M/d/yyyy" }	The date of the operation that was performed.

The rows have an additional primary key annotation, as shown in the following table:

id	core annotations				
	table	number	source number	cells	primary key
R1	*T*	1	2	*C1.1*, *C1.2*, *C1.3*, *C1.4*, *C1.5*	C1.1
R2	*T*	2	3	*C2.1*, *C2.2*, *C2.3*, *C2.4*, *C2.5*	C2.1

Thanks to the provided metadata, the cells again have the annotations shown in the following table. The metadata file has provided the information to supplement the model with additional annotations but also, for the `Inventory Date` column (cells *C1.5* and *C2.5*), have a value that is a parsed date rather than an unparsed string.

id	core annotations					
	table	column	row	string value	value	about URL
C1.1	*T*	*C1*	*R1*	`"1"`	`"1"`	`http://example.org/tree-ops.csv#gid-1`
C1.2	*T*	*C2*	*R1*	`"ADDISON AV"`	`"ADDISON AV"`	`http://example.org/tree-ops.csv#gid-1`
C1.3	*T*	*C3*	*R1*	`"Celtis australis"`	`"Celtis australis"`	`http://example.org/tree-ops.csv#gid-1`
C1.4	*T*	*C4*	*R1*	`"Large Tree Routine Prune"`	`"Large Tree Routine Prune"`	`http://example.org/tree-ops.csv#gid-1`
C1.5	*T*	*C5*	*R1*	`"10/18/2010"`	`2010-10-18`	`http://example.org/tree-ops.csv#gid-1`
C2.1	*T*	*C1*	*R2*	`"2"`	`"2"`	`http://example.org/tree-ops.csv#gid-2`
C2.2	*T*	*C2*	*R2*	`"EMERSON ST"`	`"EMERSON ST"`	`http://example.org/tree-ops.csv#gid-2`
C2.3	*T*	*C3*	*R2*	`"Liquidambar styraciflua"`	`"Liquidambar styraciflua"`	`http://example.org/tree-ops.csv#gid-2`
C2.4	*T*	*C4*	*R2*	`"Large Tree Routine Prune"`	`"Large Tree Routine Prune"`	`http://example.org/tree-ops.csv#gid-2`
C2.5	*T*	*C5*	*R2*	`"6/2/2010"`	`2010-06-02`	`http://example.org/tree-ops.csv#gid-2`

8.2.2 Empty and Quoted Cells

The following slightly amended CSV file contains quoted and missing cell values:

EXAMPLE 20: CSV file containing quoted and missing cell values

```
GID,On Street,Species,Trim Cycle,Inventory Date
1,ADDISON AV,"Celtis australis","Large Tree Routine Prune",10/18/2010
2,,"Liquidambar styraciflua","Large Tree Routine Prune",
```

Parsing this file similarly results in an annotated tabular data model of a single table *T* with five columns and two rows. The columns and rows have exactly the same annotations as previously, but there are two `null` cell values for *C2.2* and *C2.5*. Note that the quoting of values within the CSV makes no difference to either the string value or value of the cell.

id	core annotations				
	table	column	row	string value	value
C1.1	*T*	*C1*	*R1*	`"1"`	`"1"`
C1.2	*T*	*C2*	*R1*	`"ADDISON AV"`	`"ADDISON AV"`
C1.3	*T*	*C3*	*R1*	`"Celtis australis"`	`"Celtis australis"`
C1.4	*T*	*C4*	*R1*	`"Large Tree Routine Prune"`	`"Large Tree Routine Prune"`
C1.5	*T*	*C5*	*R1*	`"10/18/2010"`	`"10/18/2010"`
C2.1	*T*	*C1*	*R2*	`"2"`	`"2"`
C2.2	*T*	*C2*	*R2*	`""`	`null`
C2.3	*T*	*C3*	*R2*	`"Liquidambar styraciflua"`	`"Liquidambar styraciflua"`

id	core annotations				
	table	column	row	string value	value
C2.4	*T*	*C4*	*R2*	`"Large Tree Routine Prune"`	`"Large Tree Routine Prune"`
C2.5	*T*	*C5*	*R2*	`""`	`null`

8.2.3 Tabular Data Embedding Annotations

The following example illustrates some of the complexities that can be involved in parsing tabular data, how the flags described above can be used, and how new tabular data formats could be defined that embed additional annotations into the tabular data model.

In this example, the publishers of the data are using an internal convention to supply additional metadata about the tabular data embedded within the file itself. They are also using a tab as a separator rather than a comma.

EXAMPLE 21: Tab-separated file containing embedded metadata

```
#        publisher        City of Palo Alto
#        updated        12/31/2010
#name        GID        on_street        species        trim_cycle        inve
#datatype        string        string        string        string        date:
        GID        On Street        Species        Trim Cycle        Inventory
        1        ADDISON AV        Celtis australis        Large Tree Routine
        2        EMERSON ST        Liquidambar styraciflua        Large Tree R
```

8.2.3.1 Naive Parsing

Naive parsing of the above data will assume a comma separator and thus results in a single table *T* with a single column and six rows. The column has the annotations shown in the following table:

id	core annotations				
	table	number	source number	cells	titles
C1	*T*	1	1	*C1.1*, *C2.1*, *C3.1*, *C4.1*, *C5.1*	`# publisher City of Palo Alto`

The rows have the annotations shown in the following table:

id	core annotations			
	table	number	source number	cells
R1	*T*	1	2	*C1.1*
R2	*T*	2	3	*C2.1*
R3	*T*	3	4	*C3.1*
R4	*T*	4	5	*C4.1*
R5	*T*	5	6	*C5.1*
R6	*T*	6	7	*C6.1*

The cells have the annotations shown in the following table (note that the values of all the cells in the table are strings, denoted by the double quotes in the table below):

id	core annotations				
	table	column	row	string value	value
C1.1	*T*	*C1*	*R1*	`"# updated 12/31/2010"`	`"# updated 12/31/2010"`

id	core annotations				
	table	column	row	string value	value
C1.1	*T*	*C1*	*R1*	`"#name GID on_street species trim_cycle inventory_date"`	`"#name GID on_street species trim_cycle inventory_date"`
C2.1	*T*	*C1*	*R2*	`"#datatype string string string string date:M/D/YYYY"`	`"#datatype string string string string date:M/D/YYYY"`
C3.1	*T*	*C1*	*R3*	`" GID On Street Species Trim Cycle Inventory Date"`	`" GID On Street Species Trim Cycle Inventory Date"`
C4.1	*T*	*C1*	*R4*	`" 1 ADDISON AV Celtis australis Large Tree Routine Prune 10/18/2010"`	`" 1 ADDISON AV Celtis australis Large Tree Routine Prune 10/18/2010"`
C5.1	*T*	*C1*	*R5*	`" 2 EMERSON ST Liquidambar styraciflua Large Tree Routine Prune 6/2/2010"`	`" 2 EMERSON ST Liquidambar styraciflua Large Tree Routine Prune 6/2/2010"`

8.2.3.2 Parsing with Flags

The consumer of the data may use the flags described above to create a more useful set of data from this file. Specifically, they could set:

- delimiter to a tab character
- skip rows to 4
- skip columns to 1
- comment prefix to `#`

Setting these is done in an implementation-defined way. It could be done, for example, by sniffing the contents of the file itself, through command-line options, or by embedding a dialect description into a metadata file associated with the tabular data, which would look like:

EXAMPLE 22: Dialect description

```
{
  "delimiter": "\t",
  "skipRows": 4,
  "skipColumns": 1,
  "commentPrefix": "#"
}
```

With these flags in operation, parsing this file results in an annotated tabular data model of a single table *T* with five columns and two rows which is largely the same as that created from the original simple example described in section 8.2.1 Simple Example. There are three differences.

First, because the four skipped rows began with the comment prefix, the table itself now has four `rdfs:comment` annotations, with the values:

1. `publisher City of Palo Alto`
2. `updated 12/31/2010`
3. `name GID on_street species trim_cycle inventory_date`
4. `datatype string string string string date:M/D/YYYY`

Second, because the first column has been skipped, the source number of each of the columns is offset by one from the number of each column:

id	core annotations				
	table	number	source number	cells	titles
C1	*T*	1	2	*C1.1*, *C2.1*	`GID`
C2	*T*	2	3	*C1.2*, *C2.2*	`On Street`
C3	*T*	3	4	*C1.3*, *C2.3*	`Species`
C4	*T*	4	5	*C1.4*, *C2.4*	`Trim Cycle`
C5	*T*	5	6	*C1.5*, *C2.5*	`Inventory Date`

Finally, because four additional rows have been skipped, the source number of each of the rows is offset by five from the row number (the four skipped rows plus the single header row):

id	core annotations			
	table	number	source number	cells
R1	*T*	1	6	*C1.1*, *C1.2*, *C1.3*, *C1.4*, *C1.5*
R2	*T*	2	7	*C2.1*, *C2.2*, *C2.3*, *C2.4*, *C2.5*

8.2.3.3 Recognizing Tabular Data Formats

The conventions used in this data (invented for the purpose of this example) are in fact intended to create an annotated tabular data model which includes named annotations on the table itself, on the columns, and on the cells. The creator of these conventions could create a specification for this particular tabular data syntax and register a media type for it. The specification would include statements like:

- A tab delimiter is always used.
- The first column is always ignored.
- When the first column of a row has the value `"#"`, the second column is the name of an annotation on the table and the values of the remaining columns are concatenated to create the value of that annotation.
- When the first column of a row has the value `#name`, the remaining cells in the row provide a `name` annotation for each column in the table.
- When the first column of a row has the value `#datatype`, the remaining cells in the row provide `datatype`/`format` annotations for the cells within the relevant column, and these are interpreted to create the value for each cell in that column.
- The first row where the first column is empty is a row of headers; these provide `title` annotations on the columns in the table.
- The remaining rows make up the data of the table.

Parsers that recognized the format could then build a more sophisticated annotated tabular data model using only the embedded information in the tabular data file. They would extract embedded metadata looking like:

EXAMPLE 23: Embedded metadata in the format of the annotated tabular model

```
{
  "@context": "http://www.w3.org/ns/csvw",
  "url": "tree-ops.csv",
  "dc:publisher": "City of Palo Alto",
  "dc:updated": "12/31/2010",
  "tableSchema": {
    "columns": [{
      "name": "GID",
      "titles": "GID",
      "datatype": "string",
    }, {
      "name": "on_street",
      "titles": "On Street",
      "datatype": "string"
    }, {
      "name": "species",
      "titles": "Species",
      "datatype": "string"
    }, {
      "name": "trim_cycle",
      "titles": "Trim Cycle",
      "datatype": "string"
    }, {
      "name": "inventory_date",
      "titles": "Inventory Date",
      "datatype": {
        "base": "date",
        "format": "M/d/yyyy"
      }
    }]
  }
}
```

As before, the result would be a single table *T* with five columns and two rows. The table itself would have two annotations:

`dc:publisher`
    `{"@value": "City of Palo Alto"}`
`dc:updated`
    `{"@value": "12/31/2010"}`

The columns have the annotations shown in the following table:

id	core annotations					
	table	**number**	**source number**	**cells**	**name**	**titles**
C1	*T*	1	2	*C1.1*, *C2.1*	`GID`	`GID`
C2	*T*	2	3	*C1.2*, *C2.2*	`on_street`	`On Street`
C3	*T*	3	4	*C1.3*, *C2.3*	`species`	`Species`
C4	*T*	4	5	*C1.4*, *C2.4*	`trim_cycle`	`Trim Cycle`
C5	*T*	5	6	*C1.5*, *C2.5*	`inventory_date`	`Inventory Date`

The rows have the annotations shown in the following table, exactly as in previous examples:

id	core annotations			
	table	**number**	**source number**	**cells**
R1	*T*	1	6	*C1.1*, *C1.2*, *C1.3*, *C1.4*, *C1.5*
R2	*T*	2	7	*C2.1*, *C2.2*, *C2.3*, *C2.4*, *C2.5*

The cells have the annotations shown in the following table. Because of the way the particular tabular data format has been specified, these include additional annotations but also, for the `Inventory Date` column (cells *C1.5* and *C2.5*), have a value that is a parsed date rather than an unparsed string.

id	core annotations				
	table	column	row	string value	value
C1.1	*T*	*C1*	*R1*	`"1"`	`"1"`
C1.2	*T*	*C2*	*R1*	`"ADDISON AV"`	`"ADDISON AV"`
C1.3	*T*	*C3*	*R1*	`"Celtis australis"`	`"Celtis australis"`
C1.4	*T*	*C4*	*R1*	`"Large Tree Routine Prune"`	`"Large Tree Routine Prune"`
C1.5	*T*	*C5*	*R1*	`"10/18/2010"`	`2010-10-18`
C2.1	*T*	*C1*	*R2*	`"2"`	`"2"`
C2.2	*T*	*C2*	*R2*	`"EMERSON ST"`	`"EMERSON ST"`
C2.3	*T*	*C3*	*R2*	`"Liquidambar styraciflua"`	`"Liquidambar styraciflua"`
C2.4	*T*	*C4*	*R2*	`"Large Tree Routine Prune"`	`"Large Tree Routine Prune"`
C2.5	*T*	*C5*	*R2*	`"6/2/2010"`	`2010-06-02`

8.2.4 Parsing Multiple Header Lines

The following example shows a CSV file with multiple header lines:

EXAMPLE 24: CSV file with multiple header lines

```
Who,What,,Where,
Organization,Sector,Subsector,Department,Municipality
#org,#sector,#subsector,#adm1,#adm2
UNICEF,Education,Teacher training,Chocó,Quidbó
UNICEF,Education,Teacher training,Chocó,Bojayá
```

Here, the first line contains some grouping titles in the first line, which are not particularly helpful. The lines following those contain useful titles for the columns. Thus the appropriate configuration for a dialect description is:

EXAMPLE 25: Dialect description for multiple header lines

```
{
  "skipRows": 1,
  "headerRowCount": 2
}
```

With this configuration, the table model contains five columns, each of which have two titles, summarized in the following table:

id	core annotations				
	table	number	source number	cells	titles
C1	*T*	1	1	*C1.1*, *C2.1*	`Organization`, `#org`
C2	*T*	2	2	*C1.2*, *C2.2*	`Sector`, `#sector`
C3	*T*	3	3	*C1.3*, *C2.3*	`Subsector`, `#subsector`
C4	*T*	4	4	*C1.4*, *C2.4*	`Department`, `#adm1`
C5	*T*	5	5	*C1.5*, *C2.5*	`Municipality`, `#adm2`

As metadata, this would look like:

EXAMPLE 26: Extracted metadata

```
{
  "tableSchema": {
    "columns": [
      { "titles": ["Organization", "#org"] },
      { "titles": ["Sector", "#sector"] },
      { "titles": ["Subsector", "#subsector"] },
      { "titles": ["Department", "#adm1"] },
      { "titles": ["Municipality", "#adm2"] },
    ]
  }
}
```

A separate metadata file could contain just the second of each of these titles, for example:

EXAMPLE 27: Metadata file

```
{
  "tableSchema": {
    "columns": [
      { "name": "org", "titles": #org" },
      { "name": "sector", "titles": #sector" },
      { "name": "subsector", "titles": #subsector" },
      { "name": "adm1", "titles": #adm1" },
      { "name": "adm2", "titles": #adm2" },
    ]
  }
}
```

This enables people from multiple jurisdictions to use the same tabular data structures without having to use exactly the same titles within their documents.

A. IANA Considerations

/.well-known/csvm

URI suffix:
csvm

Change controller:
W3C

Specification document(s):
This document, section 5.3 Default Locations and Site-wide Location Configuration

B. Existing Standards

This section is non-normative.

This appendix outlines various ways in which CSV is defined.

B.1 RFC 4180

[RFC4180] defines CSV with the following ABNF grammar:

```
file = [header CRLF] record *(CRLF record) [CRLF]
header = name *(COMMA name)
record = field *(COMMA field)
name = field
field = (escaped / non-escaped)
escaped = DQUOTE *(TEXTDATA / COMMA / CR / LF / 2DQUOTE) DQUOTE
non-escaped = *TEXTDATA
COMMA = %x2C
CR = %x0D
DQUOTE =  %x22
LF = %x0A
CRLF = CR LF
TEXTDATA =  %x20-21 / %x23-2B / %x2D-7E
```

Of particular note here are:

- The production for `TEXTDATA` indicates that only non-control ASCII characters are permitted within a CSV file. This restriction is routinely ignored in practice, and is impractical on the international web.
- Lines should be ended with `CRLF`. This makes it harder to produce CSV files on Unix-based systems where the usual line ending is `LF`.
- The header line is optional; a `header` parameter on the media type indicates whether the header is present or not.
- Fields may be escaped by wrapping them in double quotes; any double quotes within the field must be escaped with two double quotes (`""`).

B.2 Excel

Excel is a common tool for both creating and reading CSV documents, and therefore the CSV that it produces is a de facto standard.

NOTE

The following describes the behavior of Microsoft Excel for Mac 2011 with an English locale. Further testing is needed to see the behavior of Excel in other situations.

B.2.1 Saved CSV

Excel generates CSV files encoded using Windows-1252 with `LF` line endings. Characters that cannot be represented within Windows-1252 are replaced by underscores. Only those cells that need escaping (e.g. because they contain commas or double quotes) are escaped, and double quotes are escaped with two double quotes.

Dates and numbers are formatted as displayed, which means that formatting can lead to information being lost or becoming inconsistent.

B.2.2 Opened CSV

When opening CSV files, Excel interprets CSV files saved in UTF-8 as being encoded as Windows-1252 (whether or not a BOM is present). It correctly deals with double quoted cells, except that it converts line breaks within cells into spaces. It understands `CRLF` as a line break. It detects dates (formatted as `YYYY-MM-DD`) and formats them in the default date formatting for files.

B.2.3 Imported CSV

Excel provides more control when importing CSV files into Excel. However, it does not properly understand UTF-8 (with or without BOM). It does however properly understand UTF-16 and can read non-ASCII characters from a UTF-16-encoded file.

A particular quirk in the importing of CSV is that if a cell contains a line break, the final double quote that escapes the cell will be included within it.

B.2.4 Copied Tabular Data

When tabular data is copied from Excel, it is copied in a tab-delimited format, with `LF` line breaks.

B.3 Google Spreadsheets

B.3.1 Downloading CSV

Downloaded CSV files are encoded in UTF-8, without a BOM, and with `LF` line endings. Dates and numbers are formatted as they appear within the spreadsheet.

B.3.2 Importing CSV

CSV files can be imported as UTF-8 (with or without BOM). `CRLF` line endings are correctly recognized. Dates are reformatted to the default date format on load.

B.4 CSV Files in a Tabular Data Package

Tabular Data Packages place the following restrictions on CSV files:

> As a starting point, CSV files included in a Tabular Data Package package must conform to the RFC for CSV (4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files). In addition:
>
> - File names MUST end with `.csv`.
> - Files MUST be encoded as UTF-8.
> - Files MUST have a single header row. This row MUST be the first row in the file.
>   - Terminology: each column in the CSV file is termed a *field* and its `name` is the string in that column in the header row.
>   - The `name` MUST be unique amongst fields, MUST contain at least one character, and MUST conform to the character restrictions defined for the *`name`* property.
> - Rows in the file MUST NOT contain more fields than are in the header row (though they may contain less).
> - Each file MUST have an entry in the `tables` array in the `datapackage.json` file.
> - The resource metadata MUST include a `tableSchema` attribute whose value MUST be a valid *schema description*.

- All fields in the CSV files MUST be described in the *schema description*.

CSV files generated by different applications often vary in their syntax, e.g. use of quoting characters, delimiters, etc. To encourage conformance, CSV files in a CSV files in a Tabular Data Package SHOULD:

- Use "," as field delimiters.
- Use `CRLF` (`U+000D U+000A`) or `LF` (`U+000A`) as line terminators.

If a CSV file does not follow these rules then its specific CSV dialect MUST be documented. The resource hash for the resource in the `datapackage.json` descriptor MUST:

- Include a `dialect` key that conforms to that described in the CSV Dialect Description Format.

Applications processing the CSV file SHOULD read use the `dialect` of the CSV file to guide parsing.

NOTE

To replicate the findings above, test files which include non-ASCII characters, double quotes, and line breaks within cells are:

- test-utf8.csv
- test-utf8-bom.csv
- test-utf16.csv
- test-utf16-bom.csv
- test.xslx
- test.xsl

C. Acknowledgements

This section is non-normative.

At the time of publication, the following individuals had participated in the Working Group, in the order of their first name: Adam Retter, Alf Eaton, Anastasia Dimou, Andy Seaborne, Axel Polleres, Christopher Gutteridge, Dan Brickley, Davide Ceolin, Eric Stephan, Erik Mannens, Gregg Kellogg, Ivan Herman, Jeni Tennison, Jeremy Tandy, Jürgen Umbrich, Rufus Pollock, Stasinos Konstantopoulos, William Ingram, and Yakov Shafranovich.

D. Changes from previous drafts

D.1 Changes since the candidate recommendation of 16 July 2015

- Use `text/tab-separated-values` instead of the un-registered `text/tsv`.
- `/.well-known/csvm` has been registered at IANA

D.2 Changes since the working draft of 16 April 2015

- Merging of metadata files has been removed as it was determined not to be necessary.
- Embedded metadata now used for compatibility check only, or as metadata if no other is found.
- The titles annotation has been added to rows, and a section added describing the way in which screen readers should announce rows and columns to users

- A Datatype description may have an id annotation to reference an external datatype definition in XSD, OWL, or some other format.
- Renamed the direction annotation to table direction.
- The built-in locations for locating metadata files were removed in favor of a site-wide configuration file, which uses the original values for file-specific and directory-specific metadata locations as the default value. See section 5.3 Default Locations and Site-wide Location Configuration.
- The pattern for numeric types is now a number format pattern rather than a regular expression.

D.3 Changes since the working draft of 08 January 2015

The document has undergone substantial changes since the last working draft. Below are some of the changes made:

- Describe all core annotations on a group of tables, tables, columns, rows, and datatypes.
- Removed the *Core Tabular Data Model*, as this can be derived from the Annotated Tabular Data Model with an appropriate *dialect descriptiondialect description*.
- Describe the process of locating metadata, using this to create embedded metadata, processing tabular data files, and creating annotated tables.
- Move details on datatypes and parsing cells. Datatypes include detailed formats for different datatypes.

E. References

E.1 Normative references

[BCP47]
A. Phillips; M. Davis. *Tags for Identifying Languages*. September 2009. IETF Best Current Practice. URL: https://tools.ietf.org/html/bcp47

[BIDI]
Mark Davis; Aharon Lanin; Andrew Glass. *Unicode Bidirectional Algorithm*. 5 June 2014. Unicode Standard Annex #9. URL: http://www.unicode.org/reports/tr9/

[ECMASCRIPT]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[ISO8601]
Representation of dates and times. International Organization for Standardization. 2004. ISO 8601:2004. URL: http://www.iso.org/iso/catalogue_detail?csnumber=40874

[JSON-LD]
Manu Sporny; Gregg Kellogg; Markus Lanthaler. *JSON-LD 1.0*. 16 January 2014. W3C Recommendation. URL: http://www.w3.org/TR/json-ld/

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC3968]
G. Camarillo. *The Internet Assigned Number Authority (IANA) Header Field Parameter Registry for the Session Initiation Protocol (SIP)*. December 2004. Best Current Practice. URL: https://tools.ietf.org/html/rfc3968

[RFC4180]
Y. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. October 2005. Informational. URL: https://tools.ietf.org/html/rfc4180

[RFC5785]
M. Nottingham; E. Hammer-Lahav. *Defining Well-Known Uniform Resource Identifiers (URIs)*. April 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5785

[UAX35]
Mark Davis; CLDR committee members. *Unicode Locale Data Markup Language (LDML)*. 15 March 2013. Unicode Standard Annex #35. URL: http://www.unicode.org/reports/tr35/tr35-31/tr35.html

[UNICODE]
The Unicode Standard. URL: http://www.unicode.org/versions/latest/

[URI-TEMPLATE]
J. Gregorio; R. Fielding; M. Hadley; M. Nottingham; D. Orchard. *URI Template*. March 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6570

[tabular-metadata]
Jeni Tennison; Gregg Kellogg. *Metadata Vocabulary for Tabular Data*. W3C Proposed Recommendation. URL: http://www.w3.org/TR/2015/PR-tabular-metadata-20151117/

[xmlschema11-2]
David Peterson; Sandy Gao; Ashok Malhotra; Michael Sperberg-McQueen; Henry Thompson; Paul V. Biron et al. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. 5 April 2012. W3C Recommendation. URL: http://www.w3.org/TR/xmlschema11-2/

E.2 Informative references

[EBNF-NOTATION]
Tim Bray; Jean Paoli; C. Michael Sperberg-McQueen; Eve Maler; François Yergau. *EBNF Notation*. W3C Recommendation. URL: http://www.w3.org/TR/xml/#sec-notation

[RFC7111]
M. Hausenblas; E. Wilde; J. Tennison. *URI Fragment Identifiers for the text/csv Media Type*. January 2014. Informational. URL: https://tools.ietf.org/html/rfc7111

[UAX15]
Mark Davis; Ken Whistler. *Unicode Normalization Forms*. 31 August 2012. Unicode Standard Annex #15. URL: http://www.unicode.org/reports/tr15

[annotation-model]
Robert Sanderson; Paolo Ciccarese; Benjamin Young. *Web Annotation Data Model*. 15 October 2015. W3C Working Draft. URL: http://www.w3.org/TR/annotation-model/

[csv2json]
Jeremy Tandy; Ivan Herman. *Generating JSON from Tabular Data on the Web*. W3C Proposed Recommendation. URL: http://www.w3.org/TR/2015/PR-csv2json-20151117/

[csv2rdf]
Jeremy Tandy; Ivan Herman; Gregg Kellogg. *Generating RDF from Tabular Data on the Web*. W3C Proposed Recommendation. URL: http://www.w3.org/TR/2015/PR-csv2rdf-20151117/

[encoding]
Anne van Kesteren; Joshua Bell; Addison Phillips. *Encoding*. 20 October 2015. W3C Candidate Recommendation. URL: http://www.w3.org/TR/encoding/

[vocab-data-cube]
Richard Cyganiak; Dave Reynolds. *The RDF Data Cube Vocabulary*. 16 January 2014. W3C Recommendation. URL: http://www.w3.org/TR/vocab-data-cube/