TAML

Tamme Schichler

Mar 22, 2022

CONTENTS

TAML is a configuration file format combining some aspects of Markdown, CSV, TOML, YAML and Rust.

As configuration language, TAML's main design goals are to be:

- Human-writeable
- Human-readable
- Unambiguous and Debuggable
- Computer-readable

One central feature is that it uses headings rather than indentation or far-spanning nested brackets to denote complex data structures. Another is the relatively strong distinction between data types.

In addition to this, implementations of the file format *should* make it easy to make it easy for software end(!) users to learn about and correct mistakes in configuration files. A number of error codes and descriptions are documented here which will ideally be largely shared between implementations.

Please refer to the table of contents to the left for examples and details.

Note: TAML is explicitly not a data transfer format.

Most notably, it is **not streamable**, as repeated non-list fields are not valid and *must* lead to a parsing error before any data becomes effective. *This includes unrelated preceding data in the same document.*

CHAPTER

ONE

TAML BY EXAMPLE

Hint: This is a non-normative quickstart guide.

For the thorough format reference, see grammar_reference.

The most simple TAML document is empty:

```
```

While this is valid TAML, such a configuration file will usually be unsupported in practice, as fields are not always optional.

1.1 Key-Value Pairs

To define structural fields (including in the implicit top-level context), you can write them as key-value pairs as follows:

```
// This is a comment. The parser will ignore it.
a_string: "This is Unicode text. You can escape \\ and \"."
some_data: <Some-Encoding:This is a data literal. You can escape \\ and \>.>
an_integer: 5
negative: -0
decimal: 0.0
negative_decimal: -10.0
list: ("Inline lists may contain heterogeneous data but no line breaks.", 1, 2.0, ())

`You can quote identifiers and escape \\ and \` within.`: ()
```

Note: Integers and decimals are disjoint! If a decimal value is expected, write `1.0` instead of `1`.

1.2 Tabular List

Lists may also be written as tables with a single column:

```
# [[items]]
"This is a list in tabular form."

1
2
3
4
5

"This is still part of the list."
```

Lists continue up to the next heading or end of document. They may not contain nested sections.

Each row in the table creates a new column in the list assigned to the `items` field, while empty lines and lines with only a comment are ignored.

1.3 Enums

Enum values are written as variant identifier, optionally followed by a list:

```
unit_variant: Unit
empty_variant: Empty()
newtype_variant: SameAsBefore("This is a nested value.")
tuple_variant: Tuple(1, 2.0, 3, 4, 5)
```

Hint: Booleans are normally treated as enumeration type with the unit variants `false` and `true`.

Hint: Nullability, as for example expressed by the `Option<...>` type in Rust, should translate to optional fields rather explicit enum values.

1.3.1 Structural Variant

Structural variants can't be expressed inline. Instead, `:Variant` is attached to the respective field name in a heading:

```
# a_field:AVariant
a: ()
b: ()
```

1.4 Structural Section

Complex data structures can be represented in TAML as follows:

```
top_level_field: ()

# outer_structural_field
inner_field: ()

## inner_structural_field
deeply_nested: ()

#
another_top_level_field: ()
```

This is equivalent to the following JSON:

```
{
        "top_level_field": [],
        "outer_structural_field": {
                "inner_field": [],
                "inner_structural_field": {
                        "deeply_nested": []
                }
        },
        "another_top_level_field": []
}
```

1.5 Structures in Lists

Structure headings create list items whenever identifiers are wrapped in square brackets (`[...]`):

```
# [items]
a: 1
b: 2

# [items]
a: 3
b: 4
c: 5
```

equals

```
"items": [
        {
                "a": 1,
                "b": 2
        },
        {
                "a": 3,
                "b": 4,
                "c": 5
        }
]
```

Note: Fields that are defined twice are normally invalid. However, adding items to an existing list is possible as above.

1.6 Path Heading

The following are equivalent:

```
# a
## [b]
### c
d: 1
e: 2

## f
### g
#### [h]
##### [[j]]
1
2
3
4
5

# k
## l
### m
### n

// Illegal, would redefine `a`:
// # a
// ## o
```

```
# a
## [b].c
d: 1
e: 2

## f.g.[h].[[j]]
1
2
3
4
5

# k.l
## m
## n

// Illegal, would redefine `a`:
// # a.o
```

1.7 Multi-Column Table

The following are equivalent:

```
# [a]
b: 1
## [c]
## d
e: 2
f: 3
##
g: 4

# [a]
b: 5
## [[c]]
6
7
## d
e: 8
f: 9
##
g: 10
```

```
# [[a].{b, c, d.{e, f}, g}]
1, (), 2, 3, 4
5, (6, 7), 8, 9, 10
```

Hint: I don't recommend manually aligning table cells here, as some people (including me) use proportional fonts almost everywhere.

(`taml fmt` would undo it by default, too.)

Hint: You can write `.{}` in a table heading to assign an empty structure to a field in each row.

Or as JSON:

```
{
        "a": [
                {
                        "b": 1,
                        "c": [],
                        "d": {
                                "e": 2,
                                "f": 3
                        },
                        "g": 4
                },
                {
                        "b": 5,
                        "c": [
                                6,
                                7
```

(continues on next page)

(continued from previous page)

```
                        ],
                        "d": {
                                "e": 8,
                                "f": 9
                        },
                        "g": 10
                }
        ]
}
```

CHAPTER

TWO

TAML GRAMMAR REFERENCE

Hint: This page is aimed at format support implementors.

For a user manual (even when using a TAML library as developer), see *TAML by Example*.

TK: Use singular for headings.

All grammar is defined in terms of Unicode codepoint identity.

Where available, the canonical binary or at-rest encoding of TAML is UTF-8, while its runtime text-API representation should use the canonical representation of arbitrary Unicode strings in the target ecosystem.

Note: Where no standard Unicode text representation exists, it's likely best to provide only a binary UTF-8 API.

2.1 Whitespace

Note: TK: Format as regex section

```
[ \t]+
```

Whitespace is meaningless except when separating otherwise-joined tokens.

Note that *line breaks* are not included here.

2.2 Comment

Note: TK: Format as regex section

```
//[^\r\n]+
```

At (nearly) any point in the document, a line comment can be written as follows:

```
// This is a comment. It stretches for the rest of the line.
// This is another comment.
```

The only limitation to comment placement is that the line up to that point must be otherwise complete.

2.3 Line break

Note: TK: Format as regex section

```
\r?\n
```

TAML does not use commas to delineate values, outside of *inline lists* and *rows*.

Instead, line breaks are a grammar token that separates *comments*, *headings*, *key-value pairs* and *table rows*.

Note: "Line break" more specifically refers to Unicode code point U+000A LINE FEED (LF), which can optionally be prefixed with a single U+000D CARRIAGE RETURN (CR).

This is the only position in which verbatim carriage return characters are legal. Note that occurrences of the line feed character in quotes are not considered to be a line break token! Correct the literal in question by either replacing all verbatim carriage return characters with `\r` or deleting them.

Empty lines outside of quotes and lines containing only a comment always can be removed without changing the structure or contents of the document.

Hint: `taml fmt` preserves single empty lines but collapses longer blank parts of the document.

`taml fix` can fix your line endings for you without changing the meaning of quotes. (TODO) It warns about any occurrence of the character it doesn't fix by default, in either sense. (TODO)

2.4 Identifier

Note: TK: Format as regex section

```
[a-zA-Z_][a-zA-Z\-_0-9]*
```

```
`([^\\`\r]|\\\\|\\`|\\r)*`
```

Identifiers in TAML are arbitrary Unicode strings and can appear in two forms, verbatim and quoted:

2.4.1 Verbatim

Verbatim identifiers must start with an ASCII-letter or underscore (_). They may contain only those codepoints plus ASCII digits and the hypen-minus character (-).

Hint: Support for - is a compatibility affordance.

When outlining a new configuration structure, I recommend for example `a_b` over `a-b`, as the former is treated as single "word" by most text editors. (Try double-clicking each.)

2.4.2 Quoted

Backtick (`)-quoted identifiers are parsed as **completely arbitrary** Unicode strings.

Only the following characters are backlash-escaped:

- \ as \\
- ` as \`

All other sequences starting with a backslash are invalid in quoted strings and *must* lead to an error.

> **Warning:** Identifiers formally may be empty or contain U+0000 NULL.
>
> However, parsers for ecosystems where this cannot be safely supported are free to limit support here, as long as this limitation is prominently declared.
>
> (A parser written in for example C# or Rust very much should support both, though. A parser written in C or C++ should consider not supporting NULL due to its common special meaning.)
>
> TK: Define an error code that should be used here. Something like TAML-L0001?

2.5 Key

Only *identifiers* may be keys. Keys appear in *section* headers, enum *variants* and as part of key-value pairs like the following:

```
key: value
```

(`value` is a *unit variant* here, but could be replaced with any other *value*.)

2.6 Value

A value is any one of the following:

`data literal`_, *decimal*, **`enum variant`_**, *integer*, *list*, *string*, **struct_**.

> **Warning:** TAML processors should be as strict as at all sensible regarding value types. For example, if a string is expected, don't accept an integer and vice versa.

In some cases, remapping TAML value types is a good idea, like when parsing rust_decimal values using Serde, which should still be written as *decimals* in TAML but internally processed as strings. Such remappings should be done explicitly on a case-by-case basis.

2.7 Integer

Note: TK: Format as regex section

```
-?(0|[1-9]\d*)
```

A whole number with base 10. Note that `-0` is legal and *may* be interpreted differently from `0`.

Additional leading zeroes are disallowed to avoid confusion with languages and/or parsing systems where this would denote base 8.

Hint: If your configuration requires setting a bitfield, consider accepting it as data literal e.g. like this instead:

```
some_bitfield: <bits:1000_0001 1111_0000>
another_encoding: <hex:81 F0>
```

2.8 Decimal

Note: TK: Format as regex section

```
-?(0|[1-9]\d*)\.\d+
```

A fractional base 10 number. Note that `-0` is legal and *may* be interpreted differently from `0`.

Additional leading zeroes are disallowed for consistency with integers. Additional trailing zeroes are considered idempotent and **must not make a difference when parsing a value**.

Note: Integers and decimals *should* be considered disjoint. Don't accept one for the other unless not doing so would be unusually inconvenient.

Note: Decimals, like integers, are not required to fit any particular binary representation.

For example, they could be parsed and processed with arbitrary precision rather than as IEEE 754 float.

Warning: `taml fmt` removes idempotent trailing zeroes from decimals.

`serde_taml` excludes them while lexing, which also affects `reserde`.

> Absolutely do not make any distinction regarding additional trailing zeroes in decimals when writing a lexer or parser.

2.9 String

Note: TK: Format as regex section

```
"([^\\"\r]|\\\\|\\"|\\r)*"
```

Strings are written as quoted Unicode literals. The characters `\`, `"` and U+000D CARRIAGE RETURN (CR) must be escaped as `\\`, `\"` and `\r`, respectively.

The character U+0000 NULL may be unsupported in environments where processing it would be unreasonably error-prone.

2.10 Enum Variants

TK

2.10.1 Unit Variant

Unit variants are written as single *identifiers*.

Notable unit variants are the boolean values `true` and `false`, which are not associated with more specific grammar in TAML.

2.11 List

TK

2.11.1 Inline Lists

2.12 Sections

TAML's grammar is, roughly speaking, split into three contexts:

- structural sections
- headings
- tabular sections

2.12.1 Structural Sections

The initial context is a structural section. Structural sections can contain key-value pairs and nested sections, which can be structural sections.

```
first: 1
second: 2

# third
first: 3.1
second: 3.2
```

Each nested section is introduced by a heading nested *exactly* one deeper than the surrounding section's.

It continues until a heading with at most equal depth is encountered or up to the end of the file. An empty nested heading can be used to semantically (but not grammatically!) return to its immediately surrounding structural section.

```
first: 1
second: 2

# third
first: 3.1
second: 3.2

## third
first: "3.3.1"
second: "3.3.2"

## fourth
first: "3.4.1"
second: "3.4.2"

#
fourth: 4
```

2.12.2 Headings

2.12.3 Tabular Sections

Tabular sections are a special shorthand to quickly define lists with structured content.

The following are equivalent:

```
# [[dishes].{id, name, [price].{currency, amount}]
<luid:d6fce69d-9c9d>, "A", EUR, 10.95
<luid:c37dcc6a-2002>, "B", EUR, 5.50
<luid:00000000-0000>, "Test Item", EUR, 0.0
```

```
# [dishes]
id: <luid:d6fce69d-9c9d>
name: "A"
## price
currency: EUR
amount: 10.95

# [dishes]
```

(continues on next page)

(continued from previous page)

```
id: <luid:c37dcc6a-2002>
name: "B"
## price
currency: EUR
amount: 5.50

# [dishes]
id: <luid:00000000-0000>
name: "Test Item"
## price
currency: EUR
amount: 0.0
```

Hint: As of right now, there is intentionally no way to define common values once per table.

I haven't found a way to express this that both is intuitive and won't make copy/paste errors much more likely.

Row

TK

CHAPTER

THREE

TAML DIAGNOSTICS

CHAPTER

FOUR

FORMATTING TAML

As a general rule, TAML code in this documentation follows the recommended formatting rules and would stay unchanged if `taml fmt` was used on it. Exceptions are explicitly noted.