TOML

Tom's Obvious, Minimal Language

Tom Preston-Werner, Pradyun Gedam, et al.

2020-02-12

Preface

Latest tagged version: v0.5.0.

NOTE: The `master` branch of this repository tracks the very latest development and may contain features and changes that do not exist on any released version. To find the spec for a specific version, look in the `versions` subdirectory.

As of version 0.5.0, TOML should be considered extremely stable. The goal is for version 1.0.0 to be backwards compatible (as much as humanly possible) with version 0.5.0. All implementations are strongly encouraged to become 0.5.0 compatible so that the transition to 1.0.0 will be simple when that happens.

Objectives

TOML aims to be a minimal configuration file format that's easy to read due to obvious semantics. TOML is designed to map unambiguously to a hash table. TOML should be easy to parse into data structures in a wide variety of languages.

Contents

1 Example

```
# This is a TOML document.

title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00 # First class dates

[database]
server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true

[servers]

  # Indentation (tabs and/or spaces) is allowed but not required
  [servers.alpha]
  ip = "10.0.0.1"
  dc = "eqdc10"

  [servers.beta]
  ip = "10.0.0.2"
  dc = "eqdc10"

[clients]
data = [ ["gamma", "delta"], [1, 2] ]

# Line breaks are OK when inside arrays
hosts = [
  "alpha",
  "omega"
]
```

2 Spec

- TOML is case sensitive.
- A TOML file must be a valid UTF-8 encoded Unicode document.
- Whitespace means tab (0x09) or space (0x20).
- Newline means LF (0x0A) or CRLF (0x0D 0x0A).

Comment

A hash symbol marks the rest of the line as a comment, except when inside a string.

```
# This is a full-line comment
key = "value"  # This is a comment at the end of a line
another = "# This is not a comment"
```

Control characters other than tab (U+0000 to U+0008, U+000A to U+001F, U+007F) are not permitted in comments.

Basic Types

Boolean

Booleans are just the tokens you're used to. Always lowercase.

```
bool1 = true
bool2 = false
```

Integer

Integers are whole numbers. Positive numbers may be prefixed with a plus sign. Negative numbers are prefixed with a minus sign.

```
int1 = +99
int2 = 42
int3 = 0
```

```
int4 = -17
```

For large numbers, you may use underscores between digits to enhance readability. Each underscore must be surrounded by at least one digit on each side.

```
int5 = 1_000
int6 = 5_349_221
int7 = 1_2_3_4_5     # VALID but discouraged
```

Leading zeros are not allowed. Integer values `-0` and `+0` are valid and identical to an unprefixed zero.

Non-negative integer values may also be expressed in hexadecimal, octal, or binary. In these formats, leading `+` is not allowed and leading zeros are allowed (after the prefix). Hex values are case insensitive. Underscores are allowed between digits (but not between the prefix and the value).

```
# hexadecimal with prefix `0x`
hex1 = 0xDEADBEEF
hex2 = 0xdeadbeef
hex3 = 0xdead_beef

# octal with prefix `0o`
oct1 = 0o01234567
oct2 = 0o755 # useful for Unix file permissions

# binary with prefix `0b`
bin1 = 0b11010110
```

64 bit (signed long) range expected (−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807).

Float

Floats should be implemented as IEEE 754 binary64 values.

A float consists of an integer part (which follows the same rules as decimal integer values) followed by a fractional part and/or an exponent part. If both a fractional part and exponent part are present, the fractional part must precede the exponent part.

```
# fractional
flt1 = +1.0
flt2 = 3.1415
flt3 = -0.01

# exponent
flt4 = 5e+22
flt5 = 1e06
```

```
flt6 = -2E-2

# both
flt7 = 6.626e-34
```

A fractional part is a decimal point followed by one or more digits.

An exponent part is an E (upper or lower case) followed by an integer part (which follows the same rules as decimal integer values but may include leading zeros).

Similar to integers, you may use underscores to enhance readability. Each underscore must be surrounded by at least one digit.

```
flt8 = 224_617.445_991_228
```

Float values `-0.0` and `+0.0` are valid and should map according to IEEE 754.

Special float values can also be expressed. They are always lowercase.

```
# infinity
sf1 = inf  # positive infinity
sf2 = +inf # positive infinity
sf3 = -inf # negative infinity

# not a number
sf4 = nan  # actual sNaN/qNaN encoding is implementation specific
sf5 = +nan # same as `nan`
sf6 = -nan # valid, actual encoding is implementation specific
```

String

There are four ways to express strings: basic, multi-line basic, literal, and multi-line literal. All strings must contain only valid UTF-8 characters.

Basic strings

Basic strings are surrounded by quotation marks. Any Unicode character may be used except those that must be escaped: quotation mark, backslash, and the control characters other than tab (U+0000 to U+0008, U+000A to U+001F, U+007F).

```
str = "I'm a string. \"You can quote me\". Name\tJos\u00E9\nLocation\tSF."
```

For convenience, some popular characters have a compact escape sequence.

```
\b         - backspace       (U+0008)
\t         - tab             (U+0009)
\n         - linefeed        (U+000A)
```

```
\f         - form feed       (U+000C)
\r         - carriage return (U+000D)
\"         - quote           (U+0022)
\\         - backslash       (U+005C)
\uXXXX     - unicode         (U+XXXX)
\UXXXXXXXX - unicode         (U+XXXXXXXX)
```

Any Unicode character may be escaped with the `\uXXXX` or `\UXXXXXXXX` forms. The escape codes must be valid Unicode scalar values.

All other escape sequences not listed above are reserved and, if used, TOML should produce an error.

Sometimes you need to express passages of text (e.g. translation files) or would like to break up a very long string into multiple lines. TOML makes this easy.

Multi-line basic strings

Multi-line basic strings are surrounded by three quotation marks on each side and allow newlines. A newline immediately following the opening delimiter will be trimmed. All other whitespace and newline characters remain intact.

```
str1 = """
Roses are red
Violets are blue"""
```

TOML parsers should feel free to normalize newline to whatever makes sense for their platform.

```
# On a Unix system, the above multi-line string will most likely be the same as:
str2 = "Roses are red\nViolets are blue"

# On a Windows system, it will most likely be equivalent to:
str3 = "Roses are red\r\nViolets are blue"
```

For writing long strings without introducing extraneous whitespace, use a "line ending backslash". When the last non-whitespace character on a line is a `\`, it will be trimmed along with all whitespace (including newlines) up to the next non-whitespace character or closing delimiter. All of the escape sequences that are valid for basic strings are also valid for multi-line basic strings.

```
# The following strings are byte-for-byte equivalent:
str1 = "The quick brown fox jumps over the lazy dog."

str2 = """
The quick brown \
```

```
  fox jumps over \
    the lazy dog."""

str3 = """\
       The quick brown \
       fox jumps over \
       the lazy dog.\
       """
```

Any Unicode character may be used except those that must be escaped: backslash and the control characters other than tab, line feed, and carriage return (U+0000 to U+0008, U+000B, U+000C, U+000E to U+001F, U+007F).

You can write a quotation mark, or two adjacent quotation marks, anywhere inside a multi-line basic string. They can also be written just inside the delimiters.

```
str4 = """Here are two quotation marks: "". Simple enough."""
# str5 = """Here are three quotation marks: """."""  # INVALID
str5 = """Here are three quotation marks: ""\"."""
str6 = """Here are fifteen quotation marks: ""\"""\"""\"""\"""\"".."""

# "This," she said, "is just a pointless statement."
str7 = """"This," she said, "is just a pointless statement.""""
```

If you're a frequent specifier of Windows paths or regular expressions, then having to escape backslashes quickly becomes tedious and error prone. To help, TOML supports literal strings which do not allow escaping at all.

Literal strings

Literal strings are surrounded by single quotes. Like basic strings, they must appear on a single line:

```
# What you see is what you get.
winpath  = 'C:\Users\nodejs\templates'
winpath2 = '\\ServerX\admin$\system32\'
quoted   = 'Tom "Dubs" Preston-Werner'
regex    = '<\i\c*\s*>'
```

Since there is no escaping, there is no way to write a single quote inside a literal string enclosed by single quotes. Luckily, TOML supports a multi-line version of literal strings that solves this problem.

Multi-line literal strings

Multi-line literal strings are surrounded by three single quotes on each side and allow newlines. Like literal strings, there is no escaping whatsoever. A newline immediately following the opening delimiter will be trimmed. All other content between the delimiters is interpreted as-is without modification.

```
regex2 = '''I [dw]on't need \d{2} apples'''
lines  = '''
The first newline is
trimmed in raw strings.
   All other whitespace
   is preserved.
'''
```

You can write 1 or 2 single quotes anywhere within a multi-line literal string, but sequences of three or more single quotes are not permitted.

```
quot15 = '''Here are fifteen quotation marks: """""""""""""""'''

# apos15 = '''Here are fifteen apostrophes: '''''''''''''''''  # INVALID
apos15 = "Here are fifteen apostrophes: '''''''''''''''"

# 'That's still pointless', she said.
str = ''''That's still pointless', she said.'''
```

Control characters other than tab are not permitted in a literal string. Thus, for binary data it is recommended that you use Base64 or another suitable ASCII or UTF-8 encoding. The handling of that encoding will be application specific.

Table

Key/Value Pair

The primary building block of a TOML document is the key/value pair.

Keys are on the left of the equals sign and values are on the right. Whitespace is ignored around key names and values. The key, equals sign, and value must be on the same line (though some values can be broken over multiple lines).

```
key = "value"
```

Values must have one of the following types.

- String
- Integer

- Float
- Boolean
- Offset Date-Time
- Local Date-Time
- Local Date
- Local Time
- Array
- Inline Table

Unspecified values are invalid.

```
key = # INVALID
```

There must be a newline after a key/value pair. (See Inline Table for exceptions.)

```
first = "Tom" last = "Preston-Werner" # INVALID
```

Keys

A key may be either bare, quoted or dotted.

Bare keys

Bare keys may only contain ASCII letters, ASCII digits, underscores, and dashes (`A-Za-z0-9_-`). Note that bare keys are allowed to be composed of only ASCII digits, e.g. `1234`, but are always interpreted as strings.

```
key = "value"
bare_key = "value"
bare-key = "value"
1234 = "value"
```

Quoted keys

Quoted keys follow the exact same rules as either basic strings or literal strings and allow you to use a much broader set of key names. Best practice is to use bare keys except when absolutely necessary.

```
"127.0.0.1" = "value"
"character encoding" = "value"
" ə " = "value"
'key2' = "value"
'quoted "value"' = "value"
```

A bare key must be non-empty, but an empty quoted key is allowed (though discouraged).

```
= "no key name"  # INVALID
"" = "blank"     # VALID but discouraged
'' = 'blank'     # VALID but discouraged
```

Dotted keys

Dotted keys are a sequence of bare or quoted keys joined with a dot. This allows for grouping similar properties together:

```
name = "Orange"
physical.color = "orange"
physical.shape = "round"
site."google.com" = true
```

In JSON land, that would give you the following structure:

```
{
  "name": "Orange",
  "physical": {
    "color": "orange",
    "shape": "round"
  },
  "site": {
    "google.com": true
  }
}
```

Whitespace around dot-separated parts is ignored, however, best practice is to not use any extraneous whitespace.

Defining a key multiple times is invalid.

```
# DO NOT DO THIS
name = "Tom"
name = "Pradyun"
```

Since bare keys are allowed to compose of only ASCII integers, it is possible to write dotted keys that look like floats but are 2-part dotted keys. Don't do this unless you have a good reason to (you probably don't).

```
3.14159 = "pi"
```

The above TOML maps to the following JSON.

```
{ "3": { "14159": "pi" } }
```

As long as a key hasn't been directly defined, you may still write to it and to names within it.

Following JSON analogy, you should note that dotted keys creates nested Hash Tables (see Table section). So you may define nested structures in a flat way as long as all types is compatible.

```
# THIS IS VALID:
#    fruit is Table with apple and orange,
#    apple is Table, and orange is Integer - there is no conflicts
fruit.apple.smooth = true
fruit.orange = 2
```

```
# THIS IS INVALID
# fruit is Table with apple,
# but apple CAN NOT be an Integer and a Table at the same time
fruit.apple = 1
fruit.apple.smooth = true
```

Defining dotted keys out-of-order is discouraged.

```
# VALID BUT DISCOURAGED

apple.type = "fruit"
orange.type = "fruit"

apple.skin = "thin"
orange.skin = "thick"

apple.color = "red"
orange.color = "orange"
```

```
# RECOMMENDED

apple.type = "fruit"
apple.skin = "thin"
apple.color = "red"

orange.type = "fruit"
orange.skin = "thick"
orange.color = "orange"
```

Table definition (continue)

Tables (also known as hash tables or dictionaries) are collections of key/value pairs. They appear in square brackets on a line by themselves. You can tell them apart from arrays because arrays are only ever values.

```
[table]
```

Under that, and until the next table or EOF are the key/values of that table. Key/value pairs within tables are not guaranteed to be in any specific order.

```
[table-1]
key1 = "some string"
key2 = 123

[table-2]
key1 = "another string"
key2 = 456
```

Naming rules for tables are the same as for keys (see definition of Keys above).

```
[dog."tater.man"]
type.name = "pug"
```

In JSON land, that would give you the following structure:

```
{ "dog": { "tater.man": { "type": { "name": "pug" } } } }
```

Whitespace around the key is ignored, however, best practice is to not use any extraneous whitespace.

```
[a.b.c]            # this is best practice
[ d.e.f ]          # same as [d.e.f]
[ g .  h  . i ]    # same as [g.h.i]
[ j . " " . 'l' ]  # same as [j." ".'l']
```

You don't need to specify all the super-tables if you don't want to. TOML knows how to do it for you.

```
# [x] you
# [x.y] don't
# [x.y.z] need these
[x.y.z.w] # for this to work

[x] # defining a super-table afterwards is ok
```

Empty tables are allowed and simply have no key/value pairs within them.

Like keys, you cannot define any table more than once. Doing so is invalid.

```
# DO NOT DO THIS

[fruit]
apple = "red"

[fruit]
orange = "orange"
```

```
# DO NOT DO THIS EITHER

[fruit]
apple = "red"

[fruit.apple]
texture = "smooth"
```

Defining tables out-of-order is discouraged.

```
# VALID BUT DISCOURAGED
[fruit.apple]
[animal]
[fruit.orange]
```

```
# RECOMMENDED
[fruit.apple]
[fruit.orange]
[animal]
```

Dotted keys define everything to the left of each dot as a table. Since tables cannot be defined more than once, redefining such tables using a `[table]` header is not allowed. Likewise, using dotted keys to redefine tables already defined in `[table]` form is not allowed.

The `[table]` form can, however, be used to define sub-tables within tables defined via dotted keys.

```
[fruit]
apple.color = "red"
apple.taste.sweet = true

# [fruit.apple]  # INVALID
# [fruit.apple.taste]  # INVALID

[fruit.apple.texture]  # you can add sub-tables
smooth = true
```

Inline Table (special syntax)

Inline tables provide a more compact syntax for expressing tables. They are especially useful for grouped data that can otherwise quickly become verbose. Inline tables are enclosed in curly braces `{` and `}`. Within the braces, zero or more comma separated key/value pairs may appear. Key/value pairs take the same form as key/value pairs in standard tables. All value types are allowed, including inline tables.

Inline tables are intended to appear on a single line. A terminating comma (also called trailing comma) is not permitted after the last key/value pair in an inline table. No newlines are allowed between the curly braces unless they are valid within a value. Even so, it is strongly discouraged to break an inline table onto multiples lines. If you find yourself gripped with this desire, it means you should be using standard tables.

```
name = { first = "Tom", last = "Preston-Werner" }
point = { x = 1, y = 2 }
animal = { type.name = "pug" }
```

The inline tables above are identical to the following standard table definitions:

```
[name]
first = "Tom"
last = "Preston-Werner"

[point]
x = 1
y = 2

[animal]
type.name = "pug"
```

Inline tables fully define the keys and sub-tables within them. New keys and sub-tables cannot be added to them.

```
[product]
type = { name = "Nail" }
# type.edible = false  # INVALID
```

Similarly, inline tables can not be used to add keys or sub-tables to an already-defined table.

```
[product]
type.name = "Nail"
# type = { edible = false }  # INVALID
```

Date and Time

Offset Date-Time

To unambiguously represent a specific instant in time, you may use an RFC 3339 formatted date-time with offset.

```
odt1 = 1979-05-27T07:32:00Z
odt2 = 1979-05-27T00:32:00-07:00
```

```
odt3 = 1979-05-27T00:32:00.999999-07:00
```

For the sake of readability, you may replace the T delimiter between date and time with a space (as permitted by RFC 3339 section 5.6).

```
odt4 = 1979-05-27 07:32:00Z
```

The precision of fractional seconds is implementation specific, but at least millisecond precision is expected. If the value contains greater precision than the implementation can support, the additional precision must be truncated, not rounded.

Local Date-Time

If you omit the offset from an RFC 3339 formatted date-time, it will represent the given date-time without any relation to an offset or timezone. It cannot be converted to an instant in time without additional information. Conversion to an instant, if required, is implementation specific.

```
ldt1 = 1979-05-27T07:32:00
ldt2 = 1979-05-27T00:32:00.999999
```

The precision of fractional seconds is implementation specific, but at least millisecond precision is expected. If the value contains greater precision than the implementation can support, the additional precision must be truncated, not rounded.

Local Date

If you include only the date portion of an RFC 3339 formatted date-time, it will represent that entire day without any relation to an offset or timezone.

```
ld1 = 1979-05-27
```

Local Time

If you include only the time portion of an RFC 3339 formatted date-time, it will represent that time of day without any relation to a specific day or any offset or timezone.

```
lt1 = 07:32:00
lt2 = 00:32:00.999999
```

The precision of fractional seconds is implementation specific, but at least millisecond precision is expected. If the value contains greater precision than the implementation can support, the additional precision must be truncated, not rounded.

Array

Arrays are square brackets with values inside. Whitespace is ignored. Elements are separated by commas. Arrays can contain values of the same data types as allowed in key/value pairs. Values of different types may be mixed.

```
integers = [ 1, 2, 3 ]
colors = [ "red", "yellow", "green" ]
nested_array_of_int = [ [ 1, 2 ], [3, 4, 5] ]
nested_mixed_array = [ [ 1, 2 ], ["a", "b", "c"] ]
string_array = [ "all", 'strings', """are the same""", '''type''' ]

# Mixed-type arrays are allowed
numbers = [ 0.1, 0.2, 0.5, 1, 2, 5 ]
contributors = [
  "Foo Bar <foo@example.com>",
  { name = "Baz Qux", email = "bazqux@example.com", url = "https://example.com/bazqux
]
```

Arrays can span multiple lines. A terminating comma (also called trailing comma) is ok after the last value of the array. There can be an arbitrary number of newlines and comments before a value and before the closing bracket.

```
integers2 = [
  1, 2, 3
]

integers3 = [
  1,
  2, # this is ok
]
```

Array of Tables (special syntax)

The last type that has not yet been expressed is an array of tables. These can be expressed by using a table name in double brackets. Under that, and until the next table or EOF are the key/values of that table. Each table with the same double bracketed name will be an element in the array of tables. The tables are inserted in the order encountered. A double bracketed table without any key/value pairs will be considered an empty table.

```
[[products]]
name = "Hammer"
sku = 738594937
```

```
[[products]]

[[products]]
name = "Nail"
sku = 284758393

color = "gray"
```

In JSON land, that would give you the following structure.

```
{
  "products": [
    { "name": "Hammer", "sku": 738594937 },
    { },
    { "name": "Nail", "sku": 284758393, "color": "gray" }
  ]
}
```

You can create nested arrays of tables as well. Just use the same double bracket syntax on sub-tables. Each double-bracketed sub-table will belong to the most recently defined table element. Normal sub-tables (not arrays) likewise belong to the most recently defined table element.

```
[[fruit]]
  name = "apple"

  [fruit.physical]  # subtable
    color = "red"
    shape = "round"

  [[fruit.variety]]  # nested array of tables
    name = "red delicious"

  [[fruit.variety]]
    name = "granny smith"

[[fruit]]
  name = "banana"

  [[fruit.variety]]
    name = "plantain"
```

The above TOML maps to the following JSON.

```
{
  "fruit": [
```

```
    {
      "name": "apple",
      "physical": {
        "color": "red",
        "shape": "round"
      },
      "variety": [
        { "name": "red delicious" },
        { "name": "granny smith" }
      ]
    },
    {
      "name": "banana",
      "variety": [
        { "name": "plantain" }
      ]
    }
  ]
}
```

If the parent of a table or array of tables is an array element, that element must already have been defined before the child can be defined. Attempts to reverse that ordering must produce an error at parse time.

```
# INVALID TOML DOC
[fruit.physical]  # subtable, but to which parent element should it belong?
  color = "red"
  shape = "round"

[[fruit]]  # parser must throw an error upon discovering that "fruit" is
           # an array rather than a table
  name = "apple"
```

Attempting to append to a statically defined array, even if that array is empty or of compatible type, must produce an error at parse time.

```
# INVALID TOML DOC
fruit = []

[[fruit]] # Not allowed
```

Attempting to define a normal table with the same name as an already established array must produce an error at parse time. Attempting to redefine a normal table as an array must likewise produce a parse-time error.

```
# INVALID TOML DOC
[[fruit]]
```

```
  name = "apple"

  [[fruit.variety]]
    name = "red delicious"

  # INVALID: This table conflicts with the previous array of tables
  [fruit.variety]
    name = "granny smith"

  [fruit.physical]
    color = "red"
    shape = "round"

  # INVALID: This array of tables conflicts with the previous table
  [[fruit.physical]]
    color = "green"
```

You may also use inline tables where appropriate:

```
points = [ { x = 1, y = 2, z = 3 },
           { x = 7, y = 8, z = 9 },
           { x = 2, y = 4, z = 8 } ]
```

3 Environment related points

Filename Extension

TOML files should use the extension `.toml`.

MIME Type

When transferring TOML files over the internet, the appropriate MIME type is `application/toml`.

4 Comparison with Other Formats

TOML shares traits with other file formats used for application configuration and data serialization, such as YAML and JSON. TOML and JSON both are simple and use ubiquitous data types, making them easy to code for or parse with machines. TOML and YAML both emphasize human readability features, like comments that make it easier to understand the purpose of a given line. TOML differs in combining these, allowing comments (unlike JSON) but preserving simplicity (unlike YAML).

Because TOML is explicitly intended as a configuration file format, parsing it is easy, but it is not intended for serializing arbitrary data structures. TOML always has a hash table at the top level of the file, which can easily have data nested inside its keys, but it doesn't permit top-level arrays or floats, so it cannot directly serialize some data. There is also no standard identifying the start or end of a TOML file, which can complicate sending it through a stream. These details must be negotiated on the application layer.

INI files are frequently compared to TOML for their similarities in syntax and use as configuration files. However, there is no standardized format for INI and they do not gracefully handle more than one or two levels of nesting.

Further reading:

- YAML spec: https://yaml.org/spec/1.2/spec.html
- JSON spec: https://tools.ietf.org/html/rfc8259
- Wikipedia on INI files: https://en.wikipedia.org/wiki/INI_file

5 Community

Get Involved

Documentation, bug reports, pull requests, and all other contributions are welcome!

Wiki

We have an Official TOML Wiki that catalogs the following:

- Projects using TOML
- Implementations
- Validators
- Language agnostic test suite for TOML decoders and encoders
- Editor support
- Encoders
- Converters

Please take a look if you'd like to view or add to that list. Thanks for being a part of the TOML community!