KDL is a small, pleasant document language with XML-like node semantics that looks like you're invoking a bunch of CLI commands! It's meant to be used both as a serialization format and a configuration language, much like JSON, YAML, or XML. It looks like this:

```
package {
  name my-pkg
  version "1.2.3"

  dependencies {
    // Nodes can have standalone values as well as
    // key/value pairs.
    lodash "^3.2.1" optional=#true alias=underscore
  }

  scripts {
    // "Raw" and dedented multi-line strings are supported.
    message """
      hello
      world
      """
    build #"""
      echo "foo"
      node -c "console.log('hello, world!');"
      echo "foo" > some-file.txt
      """#
  }

  // `\` breaks up a single node across multiple lines.
  the-matrix 1 2 3 \
             4 5 6 \
             7 8 9

  // "Slashdash" comments operate at the node level,
  // with just `/-`.
  /-this-is-commented {
    this entire node {
      is gone
    }
  }
}
```

For more details, see the overview below. Play with it in your browser!

There's a living specification, as well as various implementations. You can also check out the FAQ to answer all your burning questions!

The current version of the KDL spec is KDL 2.0.0. For legacy KDL, please refer to the KDL 1.0.0 spec. All users are encouraged to migrate. Migration is forward-and-backward-compatible and safe, and can be automated.

In addition to a spec for KDL itself, there are specifications for a KDL Query Language based on CSS selectors, and a KDL Schema Language loosely based on JSON Schema.

The language is based on SDLang, with a number of modifications and clarifications on its syntax and behavior. We are grateful for their work as an inspiration to ours.

Design and Discussion

KDL 2.0.0 has been finalized, and no further changes are expected. For questions about KDL and discussions, please see the discussions page. For minor editorial fixes or critical spec errata, please feel free to file an issue.

Used By

A lot of folks have started picking up KDL for both personal projects, and larger open source, and even proprietary projects! This section includes a list of some examples of KDL in the wild (either v1, v2, or both):

- Zellij - Terminal workspace/multiplexer
- Niri - Scrollable-tiling window manager for Wayland
- Bikeshed (here and here) - Specification pre-processor used by CSS, C++, WHATWG, various W3C working groups, and others.
- orogene - Lightning-fast JavaScript package manager
- Onyx - An efficient, procedural, and pragmatic programming language that compiles to WASM. Used for package manifests.
- Pop! OS/System76 Scheduler - Scheduling service which optimizes Linux's CPU scheduler and makes it go faster.
- ImStyle - ImGui application styling with Nim and KDL
- fmod-rs - Rust bindings to FMOD Core and FMOD Studio
- mise - dev tools, env vars, task runner
- Camping - Ruby web microframework
- Iron Vault - VTT (Virtual Tabletop) plugin for Obsidian for the Ironsworn family of games
- Microsoft TypeScript DOM Generator - Tool for generating DOM-related TypeScript and JavaScript library files
- You?

Implementations

- C - ckdl*
- C#/.NET - Kadlet
- C++ - kdlpp* (part of ckdl, requires C++20)
- Common Lisp - kdlcl
- Crystal - kdl-cr
- Dart - kdl-dart*
- Elixir - kuddle*
- Go -
  - gokdl
  - kdl-go
- Haskell -
  - Hustle
  - kdl-hs* (Format/comment-preserving parser)
- Java - kdl4j*
- JavaScript -
  - @bgotink/kdl* (Format/comment-preserving parser)
  - @virtualstate/kdl (query only, JSX based)
  - kdljs*
- Lua - kdlua
- Nim - kdl-nim
- OCaml - ocaml-kdl
- PHP - kdl-php
- Python -
  - ckdl*
  - cuddle
  - kdl-py*
- Ruby - kdl-rb*
- Rust -
  - kdl-rs (Format/comment-preserving parser)*
  - knus (Serde-*style* derive macros (not actual Serde))
- Swift - kdl-swift
- XSLT - xml2kdl

* Supports both KDL v1 and v2

Editor Support

- Intellij IDEA
- Sublime Text*
- TreeSitter (neovim, among others)
- VS Code*
- vim
- Kate*
- Zed

* Supports KDL 2.0.0

Overview

Basics

A KDL node is a node name string, followed by zero or more "arguments", and children.

```
title "Hello, World"
```

You can also have multiple values in a single node!

```
bookmarks 12 15 188 1234
```

Nodes can have properties, with string keys.

```
author "Alex Monad" email=alex@example.com active=#true
```

And they can have nested child nodes, too!

```
contents {
  section "First section" {
    paragraph "This is the first paragraph"
    paragraph "This is the second paragraph"
  }
}
```

Nodes without children are terminated by a newline, a semicolon, or the end of a file stream:

```
node1; node2; node3
```

Values

KDL supports 4 data types:

- Strings: `unquoted`, `"hello world"`, or `#"hello world"#`
- Numbers: `123.45`, `0xdeadbeef`, `#inf`, `#-inf`, `#nan`
- Booleans: `#true` and `#false`
- Null: `#null`

Strings

It supports three different formats for string input: unquoted, quoted, and raw.

```
node1 this-is-a-string
node2 "this\nhas\tescapes"
node3 #"C:\Users\zkat\raw\string"#
```

You don't have to quote strings unless any the following apply:

- The string contains whitespace.
- The string contains any of `[]{}()\/#";=`.
- The string is one of `true`, `false`, `null`, `inf`, `-inf`, or `nan`.
- The strings starts with a digit, or `+` / `-` / `.` / `-.`, `+.` and a digit. (aka "looks like a number")

In essence, if it can get confused for other KDL or KQL syntax, it needs quotes.

Both types of quoted string can be written across multiple lines by using triple quotes (`"""`) followed immediately by a newline. Additionally, common indentation shared with the line containing the closing quotes will be stripped/dedented:

```
string """
  my
    multiline
  value
  """
```

Raw strings, which do not support `\` escapes and can be used when you want certain kinds of strings to look nicer without having to escape a lot:

```
exec #"""
  echo "foo"
  echo "bar"
  cd C:\path\to\dir
  """#

regex #"\d{3} "[^/"]+""#
```

You can add any number of `#`s before and after the opening and closing `#` to disambiguate literal closing `#"` sequences:

```
other-raw ##"hello#"world"##
```

Numbers

There are 4 ways to represent numbers in KDL. KDL does not prescribe any representation for these numbers, and it's entirely up to individual implementations whether to represent all numbers with a single type, or to have different representations for different forms.

KDL has regular decimal-radix numbers, with optional decimal part, as well as an optional exponent.

```
num 1.234e-42
```

And using the appropriate prefix, you can also enter hexadecimal, octal, and binary literals:

```
my-hex 0xdeadbeef
my-octal 0o755
my-binary 0b10101101
```

Finally, all numbers can have underscores to help readability:

```
bignum 1_000_000
```

Comments

KDL supports C-style comments, both line-based and multiline. Multiline comments can be nested.

```
// C style

/*
C style multiline
*/

tag /*foo=#true*/ bar=#false

/*/*
hello
*/*/
```

On top of that, KDL supports `/-` "slashdash" comments, which can be used to comment out individual nodes, entries, or child blocks:

```
// This entire node and its children are all commented out.
/-mynode foo key=1 {
  a
  b
  c
}

mynode /-commented "not commented" /-key=value /-{
  a
  b
}
// The above is equivalent to:
mynode "not commented"
```

Type Annotations

KDL supports type annotations on both values and nodes. These can be arbitrary, but can be used by individual implementations or use-cases to constrain KDL's basic types. A number of type names are also reserved to have specific meanings.

```
numbers (u8)10 (i32)20 myfloat=(f32)1.5 {
  strings (uuid)"123e4567-e89b-12d3-a456-426614174000" (date)"20
  (author)person name=Alex
}
```

More Details

```
// Nodes can be separated into multiple lines
title \
  "Some title"


// Files must be utf8 encoded!
smile 😁

// Node names and property keys are just strings, so you can wri
// quoted or raw strings, too!
"illegal(){}[]/\\=#;identifier" #"1.2.3"# "#false"=#true

// Identifiers are very flexible. The following is a legal bare
-<123~!$@%^&*,.:'`|?+>

// And you can also use non-ASCII unicode!
ノード　お名前=ℌ^•ﻌ•^ℌ

// kdl specifically allows properties and values to be
// interspersed with each other, much like CLI commands.
foo bar=#true baz quux=#false 1 2 3
```

Design Principles

1. Human Maintainability
2. Flexibility
3. Cognitive Simplicity and Learnability
4. Ease of de/serialization
5. Ease of implementation

Compatibility with JSON and XML

There are two specifications for writing KDL that can be losslessly translated between it and JSON or XML. These specifications define a stricter *subset* of KDL that, even if not entirely idiomatic, is still valid and fits into the data models of the other two languages:

- JSON in KDL
- XML in KDL

FAQ

How do you pronounce KDL?

Same as "cuddle".

Why yet another document language?

Because nothing out there felt quite right. The closest one I found was SDLang, but that had some design choices I disagreed with.

Ok, then, why not SDLang?

SDLang is an excellent base, but I wanted some details ironed out, and some things removed that only really made sense for SDLang's current use-cases, including some restrictions about data representation. KDL is very similar in many ways, except:

- The grammar and expected semantics are well-defined and specified. This was the original impetus for working on KDL, followed by details that seemed like they could be improved.
- There is only one "number" type. KDL does not prescribe representations, but does have keywords for NaN, infinity, and negative infinity if decimal numbers are intended to be represented as IEEE754 floats.
- Slashdash (`/-`) comments are great and useful!
- Quoteless "identifier" strings (e.g. `node foo=bar`, vs `node foo="bar"`).
- KDL does not have first-class date or binary data types. Instead, it supports arbitrary type annotations for any custom data type you might need: `(date)"2021-02-03"`, `(binary)"deadbeefbadc0ffee"`.
- Values and properties can be interspersed with each other, rather than one having to follow the other. It was not clear whether this was actually allowed in SDLang.
- Multi-line strings are supported using """<newline> and their lines are automatically "dedented" to match their closing quotes' indentation level.
- Raw strings are written with `#` (`#"foo\bar"#`), instead of backticks. This, while more verbose, allows embedding of languages, especially scripting languages, that use this syntax on a regular basis, without additional escaping (e.g. bash and JavaScript).
- KDL identifiers can use a wide range of UTF-8 and are much more lax about valid characters than SDLang.
- KDL does not support "anonymous" nodes. Instead, any string can be used as a node name. For lists of arbitrary values, there is a convention of naming the nodes simply `-`.
- Namespaces are not supported, but `:` is a legal identifier character, and applications can choose to implement namespaces as they see fit.
- KDL supports arbitrary identifiers for node names and attribute names, meaning you can use arbitrary strings for those: `"123" "value"=1` is a valid node, for example. This makes it easier to use KDL for representing arbitrary key/value pairs using child nodes.

Have you seen that one XKCD comic about standards?

Yes. I have. Please stop linking me to it.

What about YAML?

YAML is a great, widespread language. Unlike KDL, which is node-based (like XML or HTML), it's based on map and array data structures, which can provide an easier serialization experience in some cases.

At the same time, YAML can be ambiguous about what types the data written into it is. There's also a persistent issue where very large YAML files become unmanageable, especially due to the significant indentation feature.

KDL is designed to avoid these particular pitfalls by always being explicit about types, and having clearly-delimited scope (and the ability to auto-indent/auto-format). Syntax errors are easier to catch, and large files are (hopefully!) much more manageable.

What about JSON?

JSON is a great serialization language, but it can be very difficult to use as a human configuration language. This is largely due to its very specific, very strict syntax, as well as its lack of support for comments.

KDL, on the other hand, has great comment support, and has a much more forgiving syntax without being so flexible as to allow certain classes of unfortunate mistakes. It also has much more flexibility around how to represent data.

If you need to interoperate with a service that consumes or emits JSON, or for some other reason have need to write "JSON in KDL", we have JiK, an official microsyntax for losslessly encoding JSON.

What about TOML?

It nests very poorly. It doesn't fare well with large files. Also, I felt some discomfort continuing to use and promote something by its creator.

What about XML?

XML is actually pretty fantastic, and has long been a standard for data exchange across many industries. At the same time, XML is known to be very verbose, and editing it involves writing (and updating) matching tags. Another large pitfall with XML is its lack of direct support for arbitrary string key/value pairs, so what would be a simple `foo: x` in some languages has to be represented as <entry name="foo" value="x" /> or something similar. XML also functions great as a **markup** language. That is, it is easy to intersperse with text, like HTML.

KDL, just like XML, is a node/element-based language, but with much more lightweight syntax. It also adds the ability to apply anonymous values directly to a node, rather than as children. That is, `nodename 1 2 3` instead of <element><child>1</child><child>2</child>(etc)</element>. This can make it much more manageable and readable as a human configuration language, and is also less verbose when exchanging documents across APIs!

Finally, KDL is **not** a markup language. XML or HTML do a much better job of "marking up" a text document with special tags, although KDL can still be useful for templating engines that want to be more strict about text fragments.

If you need to interoperate with a service that consumes or emits XML, or for some other reason have need to write "XML in KDL", we have XiK, an official microsyntax for losslessly encoding XML.