# Ending The Data Format Wars

## Truce time

Carson Anderson
DevX-O, Weave

## Why Do We Care?

Having data in memory is great! But we often need to serialize that data out into a format that can be stored or shared between systems.

When that time comes we need to turn to a format like YAML, JSON, TOML, HCL, or any of the other common formats for sharing data.

Additionally, you are very likely to use at one or all of these formats to configure software or define infrastructure.

## The Contenders

These slides will focus on the 3 of the most common data formats.

- **JSON**
- **YAML**
- **TOML**

I have also included HCL since it is a common format used by popular DevOps tools like Vault and Terraform.

- **HCL**

## Disclaimer

No matter which format you use, actual functionality depends on the exact code you are using. Just because something is or is not in the spec of any format doesn't guarantee anything.

> Just because the spec says something, doesn't mean your parser will do it, or even do it right Test, test, test!

All examples executed here are done using popular Go parsers:

- `encoding/json` (built in!)
- `gopkg.in/yaml.v3`
- `https://github.com/BurntSushi/toml`
- `github.com/hashicorp/hcl`

# JSON: Primer

> JavaScript Object Notation

- Most rigid
- Whitespace is never important
- Commas and quotes are very important

---

JSON

```
{
    "str": "my string",
    "map": {
        "true": true,
        "num": false
    },
    "array": [1,2,3]
}
```

# YAML: Primer

> Yaml Ain't Markup Language

- Superset of JSON
- Arguably Most flexible

---

YAML

```yaml
str: my string
map:
  "true": true
  !!str false: false
array:
  - 1
  - 2
  - 3
jsonExample: {"str":"my string","map":{"true":true,"num":false},
```

# TOML: Primer

> Tom's Obvious Markup Language

- Simplest for basic use cases
- Less ambiguity than YAML
- Allows avoiding of nesting

---

TOML

```toml
str = "my string"
map.true = true
map.false = false
array = [1, 2, 3]
```

# HCL: Primer

> Hashicorp Configuration Language

- Mostly confined to Hashicorp tools
- Designed as a "syntax" more than a data format
- A decent mix of JSON/YAML/TOML

---

HCL

```
str = "my string"
map = {
    "true" = true
    "false" = false
}
array = [1, 2, 3]
```

## Basic Type Comparisons

All the comparisons will use a key/value set as the root type. Depending on the format this is going to be called something different

- map - YAML
- object - JSON, HCL
- table - TOML

> We will use this as the root type for all examples. This is because HCL and TOML are both "document" markup languages and require a key value pair as the root data structure

# objects/maps/tables - basic

| | |
|---|---|
| JSON | `{"key1":"value1", "key2":"value2"}` |
| YAML | `key1: value1`<br>`key2: value2` |
| TOML | `key1 = "value1"`<br>`key2="value2"` |
| HCL | `key1 = "value1"`<br>`key2="value2"` |

# objects/maps/tables - advanced

| | | |
|---|---|---|
| JSON | `{ "key1": "value1", "key 2": "value2", "1": "value3" }` | Trailing comma disallowed by strict parsers |
| YAML | `key1: value1`<br>`"key 2": value2`<br>`!!str 1: value3` | Can use a data type prefix to disambiguate |
| TOML | `key1 = "value1"`<br>`"key 2" = "value2"`<br>`1 = "value3"` | Quotes only required for keys with spaces, otherwise all keys are always read as strings |
| HCL | `key1 = "value1"`<br>`"key 2" = "value2"`<br>`"1" = "value3"` | Keys must be strings and must be quoted if they contain no letters |

# objects/maps/tables - nested

JSON

```
{
  "name": "Bob",
  "job": { "title": "Developer", "years": 10 }
}
```

YAML

```
name: Bob
job:
  title: Developer
  years: 10
```

TOML

```
name = "Bob"
[job]
title = "Developer"
years = 10
```

HCL

```
name = "Bob"
job {
  title = "Developer"
  years = 10
}
```

# objects/maps/tables - keys

Almost all formats require string keys for maps

| | | |
|---|---|---|
| JSON | <pre>{<br>  1: "value1",<br>}</pre> | 💀 invalid! |
| YAML | <pre>1: value1<br>true: value2<br>1.1: value3<br>"key": value4</pre> | This is the only format capable of representing maps with key values of a type other than strings |
| TOML | <pre>1 = "value1"</pre> | Will not fail, but the key will be forced to be a string |
| HCL | <pre>1 = "value1"</pre> | 💀 invalid! |

# strings - YAML

Strings may be declared in many formats: unquoted, double quoted, single quoted, declared with a type prefix, or as a "block scalar"

---

YAML

```yaml
str1: my string # no quotes required for many string values
str2: "my string"
str3: 'my string'
str4: !!str my string
str5: >-
  my string

bool1: 'true'
bool2: "true"
bool3: !!str true
bool4: >-
  true

unicode: "\u03B1\u03B2\u03B3"

complex1: '''single'' "double" quotes' # Note the '', not \'
complex2: "'single' \"double\" quotes" # supports \ sequences
complex3: >-
  'single' "double" quotes
```

# long strings

```
This is a very long string that has no newlines but it includes 'single' and "double" quotes
```

---

> This is a very long string that has no newlines but it includes 'single' and "double" quotes

# long strings

JSON

```
{"key1":"This is a very long string that has no newlines but it
```

YAML

```
key1: This is a very long string that has no newlines but it inc
key2: "This is a very long string that has no newlines but it in
key3: >-
  This is a very long string that
  has no newlines but it includes
  'single' and "double" quotes
```

TOML

```
key1 = "This is a very long string that has no newlines but it i
key2 = """
This is a very long string that \
has no newlines but it includes \
'single' and "double" quotes"""
```

HCL

```
key1 = "This is a very long string that has no newlines but it i
```

# multiline strings

We will start with rendering a basic example:

```
line1
line2
```

# multiline strings

| | | |
|---|---|---|
| JSON | `{"key1":"line1\n1line2"}` | Must use escapes |
| YAML | `key1: "line1\nline2"`<br>`key2: \|-`<br>`  line1`<br>`  line2` | See also: https://yaml-multiline.info/ |
| TOML | `key1 = "line1\nline2"`<br>`key2 = """`<br>`line1`<br>`line2"""` | |
| HCL | `key1 = "line1\nline2"`<br>`# heredocs *almost* work`<br>`key2 = <<EOF`<br>`line1`<br>`line2`<br>`EOF` | ⚠️ Not exactly the same! Leaves a trailing newline and needs a *Terraform* `trim` function which is not pure HCL |

# multiline strings - complex

Now a more complex example

```
line1
  line2
    line3
    < tab indented
```

# multiline strings - complex

JSON

```
{"key1":"line1\n  line2\n    line3\n\t< tab indented"}
```

YAML

```
key1: "line1\n  line2\n    line3\n\t< tab indent
key2: |-
  line1
    line2
      line3
      < tab indented
```

key2 == 😱

TOML

```
key1 = "line1\n  line2\n    line3\n\t< tab indented"
key2 = """
line1
  line2
    line3
    < tab indented"""
```

HCL

```
key1 = "line1\n  line2\n    line3\n\t< tab indented"
# again, heredocs only *almost* work so they are not shown here
```

## lists

In all formats lists are *ordered* and can contain zero to N of any type.

---

Task - make a list of string `"value"` and bool `false`

- value
- `false`

# lists

JSON

```
{ "myList": [ "value", false ] }
```

YAML

```
myList1: [ "value", false ]
myList2: [
    value, false,
]
myList3:
  - value
  - !!bool false # type tag not required but valid
```

TOML

```
myList1 = ["value",false]
myList2 = [
  "value",
  false,
]
```

Trailing comma allowed, whitespace discouraged when inline

HCL

```
myList1 = [ "value", false ]
myList2 = [
  "value1",
  false,
]
```

Trailing comma allowed, no whitespace required

## bools

Task - make a map:

- string key `"true"` to bool value `true`
- string key `"false"` to bool value `false`

# bools

JSON

```
{
  "true": true,
  "false": false
}
```

YAML

```
"true": true
true2: yes
true3: Y
!!str false: false
false2: no
false3: N
```

y|Y|yes|Yes|YES |n|N|no|No|NO |true|True|TRUE |false|False|FALSE |on|On|ON|off|Off|OFF

TOML

```
true = true
false = false
```

Keys are always interpreted as strings, so no need to quote

HCL

```
"true" = true
"false" = false
```

Quotes required

# null

Sometimes we want to include keys in objects but exlicitly set them to *nothing*

---

Task - make a map of string "key" to `null` 29

# null

| | | |
|---|---|---|
| JSON | `{"key":null}` | |
| YAML | `key1: null`<br>`key2: Null`<br>`key3: NULL`<br>`key4: ~`<br>`key5:` | |
| TOML | `key = null` | 💀 explicitly not supported |
| HCL | `key = null` | ⚠️ null is in the spec but is not well supported |

# comments

JSON

```
{}
```

Not supported

YAML

```yaml
# comment on a line
str1: "value" # comments can come after
str2: value # comments can come after
str3: "use quotes to keep # comments that come after"
str4: >-
  use a block scalar to keep # comments that come after

bool: true # comments can come after
num: 1 # comments can come after
```

TOML

```toml
# key is the key
key1 = "value"
key2 = 'value' # comments can come after
key3 = true # comments can come after
```

HCL

```
# key is the key
key1 = "value"
key2 = "value" # comments can come after
```

# numbers

JSON

```
{
  "number1":1,
  "number2":2.0,
  "number3":3.0e1,
  "number4":1000000
}
```

No distinction is made between ints or floats

YAML

```
number1: 1
number2: 2.0
number3: 3.0e1
number4: 1_000_000
```

TOML

```
number1 = 1
number2 = 2.0
number3 = 3.0e1
number4 = 1_000_000
```

HCL

```
number1 = 1
number2 = 2.0
number3 = 3.0e1
number4 = 1000000
```

Does not support _ in numbers

# datetimes

JSON

```
{"date":"1979-05-27T07:32:00Z"}
```

No native support

YAML

```
canonical: 1979-05-27T07:32:00Z
iso8601: 1979-05-27T07:32:00-05:00
date: !!timestamp 1979-05-27

# in the spec but not commonly supported
spaced: 1979-05-27 T07:32:00 -5
```

TOML

```
# RFC 3339 with T or space separator
date1 = 1979-05-27
date2 = 1979-05-27T07:32:00Z
date3 = 1979-05-27T00:32:00-07:00
date4 = 1979-05-27 00:32:00-07:00
```

HCL

```
date = "1979-05-27T07:32:00Z"
```

No native support

# Complex Structure Comparisons

So far we have simply compared basic types, but most use cases involve much more complex data types.

## Objects with Arrays

Create a document in each format that contains the following:

- A list of bools
- A list of numbers
- A list of floats
- A list of names

# Objects with Arrays - JSON / YAML

JSON

```json
{
  "bools": [ true, false, true ],
  "numbers": [ 1, 2, 3, 4, 5 ],
  "floats": [ 1.1, 2.2, 3.3 ],
  "names": [ "Alice", "Bob", "Carol" ]
}
```

YAML

```yaml
bools: [ true, false, true ]
numbers: # no spaces
- 1
- 2
- 3
floats:  # two spaces
  - 1.1
  - 2.2
  - 3.3
names:  # four spaces (any indentation works)
    - Alice
    - Bob
    - Carol
```

# Objects with Arrays - TOML / HCL

TOML

```
bools = [ true, false, true ]
numbers = [ 1, 2, 3, 4, 5 ]
floats = [ 1.1, 2.2, 3.3   ] # whitespace doesn't matter
names = [ # can span multiple lines
  "Alice",
  "Bob",
  "Carol", # trailing comma is ok
]
```

HCL

```
bools = [ true, false, true ]
numbers = [ 1, 2, 3, 4, 5 ]
floats = [ 1.1, 2.2, 3.3   ] # whitespace doesn't matter
names = [ # can span multiple lines
  "Alice",
  "Bob",
  "Carol", # trailing comma is ok
]
```

## Array Of Objects/Maps - JSON / YAML

Create a document in each format that contains the following:

- An ordered list of people, with each having
  - name - `string`
  - age - `number`
  - verified - `bool`

# Array Of Objects/Maps - JSON / YAML

JSON

```json
{
    "people": [
        { "name": "Alice", "age": 30, "verified": true },
        { "name": "Bob", "age": 31, "verified": false }
    ]
}
```

YAML

```yaml
people:
  - name: Alice
    age: 30
    verified: true
  - name: Bob
    age: 31
    verified: false
```

# Array Of Objects - TOML / HCL

TOML

```toml
[[people]]
name = "Alice"
age = 30
verified = true

[[people]]
name = "Bob"
age = 31
verified = false
```

Repeat the map key name in double braces to start a new item.
https://toml.io/en/v1.0.0#array-of-tables

HCL

```hcl
people = [
    { name = "Alice", age = 30, verified = true },
    {
        name = "Bob",
        age = 31,
        verified = false,
    },
]
```

# Real World Example - AWS RunInstancesRequest

Let's create a portion of the AWS `RunInstances` request body in all formats

```
{
    "ImageId": STRING,
    "InstanceType": STRING,
    "Placement": {
         "AvailabilityZone": STRING
    },
    "BlockDeviceMappings": [
       {
          "DeviceName": STRING,
          "Ebs": {
             "DeleteOnTermination": BOOLEAN,
             "VolumeSize": NUMBER
          }
       }
    ]
}
```

# RunInstancesRequest - Go Types

For a more real-world use case, we are going to use our own custom go types for the example

```
type RunInstancesInput struct {
    ImageId             string              `json:"ImageId" yaml:"ImageId", toml:"ImageId", hcl:"ImageI
    InstanceType        string              `json:"InstanceType" yaml:"InstanceType", toml:"InstanceTyp
    Placement           Placement           `json:"Placement" yaml:"Placement", toml:"Placement", hcl:"
    BlockDeviceMappings []BlockDeviceMapping `json:"BlockDeviceMappings" yaml:"BlockDeviceMappings", tom
}

type Placement struct {
    AvailabilityZone string `json:"AvailabilityZone" yaml:"AvailabilityZone", toml:"AvailabilityZone", h
}

type BlockDeviceMapping struct {
    DeviceName string `json:"DeviceName" yaml:"DeviceName", toml:"DeviceName", hcl:"DeviceName"`
    Ebs        Ebs    `json:"Ebs" yaml:"Ebs", toml:"Ebs", hcl:"Ebs"`
}

type Ebs struct {
    DeleteOnTermination bool `json:"DeleteOnTermination" yaml:"DeleteOnTermination", toml:"DeleteOnTermi
    VolumeSize          int  `json:"VolumeSize" yaml:"VolumeSize", toml:"VolumeSize", hcl:"VolumeSize"`
}
```

# RunInstancesRequest - JSON

JSON

```
{
   "ImageId": "ami-12345678",
   "InstanceType": "t2.micro",
   "Placement": {
      "AvailabilityZone": "us-east-1a"
   },
   "BlockDeviceMappings": [
      {
         "DeviceName": "/dev/sda1",
         "Ebs": {
            "DeleteOnTermination": true,
            "VolumeSize": 8
         }
      },
      {
         "DeviceName": "/dev/sdb",
         "Ebs": { "DeleteOnTermination": true, "VolumeSize": 20 }
      }
   ]
}
```

# RunInstancesRequest - YAML (from JSON)

Remember, you can always just use JSON in YAML

YAML

```
{
   "ImageId": "ami-12345678",
   "InstanceType": "t2.micro",
   "Placement": {
      "AvailabilityZone": "us-east-1a"
   },
   "BlockDeviceMappings": [
      {
         "DeviceName": "/dev/sda1",
         "Ebs": {
            "DeleteOnTermination": true,
            "VolumeSize": 8
         }
      },
      {
         "DeviceName": "/dev/sdb",
         "Ebs": { "DeleteOnTermination": true, "VolumeSize": 20 }
      }
   ]
}
```

# RunInstancesRequest - YAML

YAML

```yaml
ImageId: ami-12345678
InstanceType: t2.micro
Placement:
  AvailabilityZone: us-east-1a
BlockDeviceMappings:
  - DeviceName: /dev/sda1
    Ebs:
      DeleteOnTermination: true
      VolumeSize: 8
  - DeviceName: /dev/sdb
    Ebs: { DeleteOnTermination: true, VolumeSize: 20 }
```

# RunInstancesRequest - TOML (inline)

TOML

```toml
ImageId = "ami-12345678"
InstanceType = "t2.micro"
Placement = { AvailabilityZone = "us-east-1a" }
BlockDeviceMappings = [
   # each inline table must be on a single line to be valid TOML
   {  DeviceName = "/dev/sda", Ebs = { DeleteOnTermination = tru
   {  DeviceName = "/dev/sdb", Ebs = { DeleteOnTermination = tru
]
```

This works, but over-use of inline tables is can lead to ugly files

# RunInstancesRequest - TOML (Invalid)

TOML

```toml
ImageId = "ami-12345678"
InstanceType = "t2.micro"
Placement = {
   AvailabilityZone = "us-east-1a"
}
BlockDeviceMappings = [
   {
      DeviceName = "/dev/sda1"
      Ebs = {
         DeleteOnTermination = true
         VolumeSize = 8
      }
   },
   {
      DeviceName = "/dev/sdb"
      Ebs = { DeleteOnTermination = true, VolumeSize = 20 }
   }
]
```

---

This is **invalid** toml, but you might be tempted to think it would work. However, if you like this format, don't worry we will come back to it

# RunInstancesRequest - TOML (Canonical)

TOML

```toml
AmiId = "ami-0ff8a91507f77f867"
InstanceType = "t2.micro"

[Placement]
AvailabilityZone = "us-east-1a"

[[BlockDeviceMappings]] # device 1
DeviceName = "/dev/sda1"

# you can use dot notation for nested tables
Ebs.DeleteOnTermination = true
Ebs.VolumeSize = 8

[[BlockDeviceMappings]] # device 2
DeviceName = "/dev/sdb"

# or you can use inline tables
Ebs = { VolumeSize = 20, DeleteOnTermination = true }
```

This is embracing TOML rather than working against it

# RunInstancesRequest - HCL

HCL

```
ImageId = "ami-12345678"
InstanceType = "t2.micro"
Placement = {
   AvailabilityZone = "us-east-1a"
}
BlockDeviceMappings = [
   {
      DeviceName = "/dev/sda1"
      Ebs = {
         DeleteOnTermination = true
         VolumeSize = 8
      }
   },
   {
      DeviceName = "/dev/sdb"
      Ebs = { DeleteOnTermination = true, VolumeSize = 20 }
   }
]
```

---

Interestingly, the invalid TOML example is actually valid and canonical HCL.

# Special Features

Now, lets cover some features not supported across every format

## One line documents - JSON & YAML

This can be incredibly useful for things like logs. It makes it possible to emit structured logs but still read them line by line.

---

JSON excels at this use case

JSON

```
{"ImageId":"ami-12345678","InstanceType":"t2.micro","Placement":
```

Yaml also it this (if you just give it JSON, but you really shouldn't do in the second format since it is error prone for large or complex data

YAML

```
# You can of course use the json as-is as yaml
{"ImageId":"ami-12345678","InstanceType":"t2.micro","Placement":
```

YAML

```
# Or you can drop the quotes but be warned:
# every single bit of whitespace here is required!
{ ImageId: "ami-12345678", InstanceType: "t2.micro", Placement:
```

# One line(ish) documents - TOML & HCL

TOML and HCL can **almost** do our single line example. However both formats *must* have a key/value pair (map/table/etc) as the root data type of the document.

However, if we use a wrapper type we can get to a single line

```
type wrapper struct {
    Document types.RunInstancesInput `toml:"Document" hcl:"Document"`
}
```

TOML

```
Document = {ImageId="ami-12345678",InstanceType="t2.micro",Placer
```

HCL

```
Document = {ImageId="ami-12345678",InstanceType="t2.micro",Placer
```

# Serializing any type - JSON & YAML

Both formats support serializing any data type directly, not just key value pairs

| | |
|---|---|
| JSON | `"my string"` |
| JSON | `1` |
| JSON | `true` |
| YAML | `my string` |
| YAML | `1` |
| YAML | `true` |
| YAML | `# empty is valid in YAML but not JSON` |

# Serializing any type - TOML & HCL

Not supported! Root type must be key/value pair

TOML

```
"my string"
```

HCL

```
"my string"
```

# Anchors - YAML Magic - Simple

Any node can be given an id with `&` and referenced with `*`

YAML

```yaml
name: &name Carson
accounts:
  - id: 1
    name: *name
  - {id: 2, name: *name}
```

# Anchors - YAML Magic - Complex

YAML

```yaml
name: &name Carson
users:
  - &carson
    id: 35
    name: *name
accounts:
  - &account1
    user: *carson
    balance: 100
  - user: *carson
    balance: 200
defaultAccount: *account1
```

# Anchors - YAML Magic - Practical

A notable real-world user of this is `docker-compose`

docs.docker.com/compose/compose-file/10-fragments/#example-4 (https://docs.docker.com/compose/compose-file/10-fragments/#example-4)

YAML

```yaml
# deduplicate env vars using anchors
x-base-env: &base-env
  FOO: BAR

services:
  first:
    image: my-image:latest
    environment: &first-env
      <<: *base-env
      FIZZ: BUZZ
  second:
    image: another-image:latest
    environment:
      <<: *first-env
      ZOT: QUIX
```

# Document Separators - YAML Streaming

Yaml can use `---` to separate documents in a stream

YAML

```yaml
# leading document separator is optional but encouraged
---
name: Carson
id: 35
---
name: Tami
id: 14

# do not use trailing document separator
---
```

# Stream Separators - YAML Streaming

Yaml can use `...` to end a stream. This removes abiguity around missing data or connection closures

YAML

```yaml
---
name: Carson
id: 35
---
name: Tami
id: 14

# Note the ... to say
# "end of stream" rather than "start of document"
#
# Without this, the stream wouldn't be able to tell
# the difference between missing data and an and of data
...
```

# Document Separators - YAML Streaming

Multiple streams are possible in one data source. This is useful for things like bursts of messages over a long-lived connection

YAML

```yaml
---
name: Carson
id: 35
---
name: Tami
id: 14
...
---
name: Raul
id: 15
---
name: Ella
id: 23
...
```

## Document Separators - Practical

Starting files with `---` makes it possible to easily use concatination for composition of resurces. This is why I almost always do it in every yaml I write

YAML
```yaml
---
{ name: Carson, id: 35 }
```

YAML
```yaml
---
{ name: Tami, id: 14 }
```

YAML
```yaml
---
{ name: Raul, id: 15 }
```

Easy composition by just concating files.

```
cat user1.yaml user2.yaml user3.yaml | kubectl apply -f -
```

## Nesting

Occasionally you might want to use long strings to nest data structures inside others. This kind of string based nesting is a common escape hatch for strict data structures.

---

Task: make a map with

- "type" is a string containing the type of data.
- "data" is a nested map containing
  - "str" with a string value of `"Hello world"` (literal quotes are in the string)
  - "bool" containing a bool value of `true`

JSON

```json
{
    "type": "json",
    "data": { "str": "\"Hello World\"", "bool": true }
}
```

# Nesting - Examples

JSON

```json
{
  "type": "json",
  "data": { "str": "\"Hello World\"", "bool": true }
}
```

YAML

```yaml
type: yaml
data:
  str: '"Hello World"'
  bool: true
```

TOML

```toml
type = "toml"
[[data]]
str = '"Hello World"'
bool = true
```

HCL

```hcl
type = "hcl"
data = {
  str = "\"Hello World\"",
  bool = true
}
```

# Nesting - JSON

JSON


```
{
  "json":
    "{\"type\":\"json\",\"data\":{\"str\":\"\\\"Hello World\\\"\\
  "yaml":
    "type: yaml\ndata:\n  str: '\"Hello World\"'\n  bool: true",
  "toml":
    "type = \"toml\"\n[[data]]\nstr = '\"Hello World\"'\nbool = 
  "hcl":
    "type = \"hcl\"\ndata = {\n  str = \"\\\"Hello World\\\"\",\n
}
```

# Nesting - YAML

JSON

```yaml
json: >-
  {"type":"json","data":{"str":"\"Hello World\"","bool": true}}
yaml: |-
  type: yaml
  data:
    str: '"Hello World"'
    bool: true
toml: |-
  type = "toml"
  [[data]]
  str = '"Hello World"'
  bool = true
hcl: |-
  type = "hcl"
  data = {
    str = "\"Hello World\"",
    bool = true,
  }
```

# Nesting - TOML

TOML

```
json = '{"type":"json","data":{"str":"\"Hello World\"","bool":tru

yaml = '''
type: yaml
data:
  str: '"Hello World"'
  bool: true'''

toml = '''
type = "toml"
[[data]]
str = '"Hello World"'
bool = true'''

hcl = '''
type = "hcl"
data = {
  str = "Hello World",
  bool = true,
}'''
```

# Nesting - HCL - Exact

The only way to get the *exact* same map of strings in HCL is to use double quotes and escapes.

HCL

```
json = "{\"type\":\"json\",\"data\":{\"str\":\"\\\"Hello World\\\
yaml = "type: yaml\ndata:\n  str: '\"Hello World\"'\n  bool: tru
toml = "type = \"toml\"\n[[data]]\nstr = '\"Hello World\"'\nbool
hcl = "type = \"hcl\"\ndata = {\n  str = \"\\\"Hello World\\\"\"
```

# Nesting - HCL - Added Trailing Newlines

HCL

```
json = <<EOJSON
{ type = "json", data = { str = "\"Hello World\"", bool = true }
EOJSON

yaml = <<EOYAML
type: yaml
data:
  str: '"Hello World"'
  bool: true
EOYAML

toml = <<EOTOML
type = "toml"
[[data]]
str = '"Hello World"'
bool = true
EOTOML

hcl = <<EOHCL
type = "hcl"
data = {
  str = "\"Hello World\"",
  bool = true,
}
EOHCL
```

## Bonus - jq & jy

While not a feature of the formats themselves. `jq`, and to a lesser extent `yq`, are both massively useful tools that you should be using.

jqlang.github.io/jq/ (https://jqlang.github.io/jq/)

github.com/mikefarah/yq (https://github.com/mikefarah/yq)

If you are handling json or yaml files on the command line, you *need* these tools

# Wrapping Up - JSON

- One way to JSON, get used to it or get out
- Can be readable enough with only short and simple strings

# Wrapping Up - YAML

- Very flexible
- Maybe too flexible
- When in doubt, disambiguate/quote
- Can always just put JSON in any field
- Anchors are amazing
- Native concept of streaming

# Wrapping Up - TOML

- Great for simple use cases
- Dot notation is super useful
- Lists of objects can be a bit weird

# Wrapping Up - HCL

- Pretty good mix of all others

# Questions?

# Playgrounds

The next 4 slides have editing enabled to facilitate answering questions

# JSON Playground

```
{
    "str": "my string",
    "map": {
        "true": true,
        "num": false
    },
    "array": [1,2,3]
}
```

# YAML Playground

```
str: my string
map:
  "true": true
  !!str false: false
array:
  - 1
  - 2
  - 3
jsonExample: {"str":"my string","map":{"true":true,"num":false},"array":[1,2,3]}
```

# TOML Playground

```
str = "my string"
map.true = true
map.false = false
array = [1, 2, 3]
```

# HCL Playground

```
str = "my string"
map = {
    "true" = true
    "false" = false
}
array = [1, 2, 3]
```

# Thank you

Carson Anderson
DevX-O, Weave
@carson_ops (http://twitter.com/carson_ops)

https://github.com/carsonoid/talk-ending-the-data-format-wars (https://github.com/carsonoid/talk-ending-the-data-format-wars)