GENERAL

YAML Tutorial : A Complete Language Guide with Examples

Omkar Birade

Updated 04 Sep 2024 · 17 min read

This tutorial will guide you through the fundamentals of YAML, starting from basic syntax to more advanced features. Whether you are a beginner looking to get started or an experienced developer aiming to refine your skills, this article will provide you with the essential knowledge to effectively utilize YAML in your projects.

Join us as we explore the key elements of YAML, including data types, structures, and common use cases, and learn how to integrate YAML into your workflows for better configuration management.

What we will cover:

What is YAML?

YAML is one of the most popular data serialization languages, and it is used mostly for writing configuration files. The YAML recursive acronym stands for YAML Ain't Markup Language. This language is designed with flexibility and accessibility in mind, so it's human-readable and simple to understand. YAML works with all modern programming languages and is widely used in data persistence, internet messaging, cross-language data sharing, and many other places. YAML files either have the extension .yaml or .yml.

All of these factors contribute to YAML's popularity as a configuration language in the DevOps domain, where it is widely used with well-known tools such as Kubernetes, Ansible, and Terraform.

What is a YAML used for?

YAML is often used for configuration files that are parsed and read by a programming language or framework. Its human-readable format makes it easy for developers and system administrators to understand and modify configuration settings.

Here are some of the most common use cases for YAML:

- **Configuration management (CM)** – Ansible uses yaml files to describe all CM configurations (playbooks, roles, etc.).
- **Infrastructure as code (IaC)** – OpenTofu, for example, can read yaml files and use them as input for different resources, data sources, and even outputs.
- **CI/CD** – Many CI/CD products rely on yaml to describe their pipelines (GitHub Actions, GitLab CI/CD, Azure DevOps, CircleCI)
- **Container orchestration (CO)** – K8s and Docker Compose rely heavily on yaml files to describe the infrastructure resources.
- **Data serialization** – YAML can be used to describe complex data types such as lists, maps, and objects.
- **APIs** – YAML can be used in defining API contracts and specifications (e.g. OpenAPI).

What does a regular YAML file look like?

Below, you can see a simple example of a YAML file:

```yaml
---
# A sample yaml file
company: spacelift
domain:
 - devops
 - devsecops
tutorial:
  - yaml:
      name: "YAML Ain't Markup Language"
      type: awesome
      born: 2001
  - json:
```

```
      type: good
      born: 1996
author: omkarbirade
published: true
```

Basic YAML syntax

A YAML format primarily uses three node types:

1. **Maps/Dictionaries** (YAML calls it *mapping*)
   The content of a *mapping* node is an unordered set of *key/value* node *pairs*, with the restriction that each of the keys is unique. YAML places no further restrictions on the nodes.
2. **Arrays/Lists** (YAML calls them *sequences*)
   The content of a *sequence* node is an ordered series of zero or more nodes. In particular, a sequence may contain the same node more than once. It could even contain itself.
3. **Literals** (Strings, numbers, boolean, etc.)
   The content of a *scalar* node is an opaque datum that can be presented as a series of zero or more *Unicode characters*.

Let us try to identify where these appear in the sample YAML file we saw earlier.

```
---
# key: value [mapping]
company: spacelift
# key: value is an array [sequence]
domain:
 - devops
 - devsecops
tutorial:
  - yaml:
      name: "YAML Ain't Markup Language" #string [literal]
      type: awesome #string [literal]
      born: 2001 #number [literal]
  - json:
      name: JavaScript Object Notation #string [literal]
      type: great #string [literal]
      born: 2001 #number [literal]
  - xml:
      name: Extensible Markup Language #string [literal]
      type: good #string [literal]
      born: 1996 #number [literal]
author: omkarbirade
published: true
```

1. Indentation

A YAML file relies on whitespace and indentation to indicate nesting. Notice the hierarchy and nesting is visible through a Python-like indentation style. It is critical to note that tab characters cannot be

```
  - yaml:  #nesting level 2 (2 spaces used for indentation)
      name: "YAML Ain't Markup Language" #string [literal] #nesting level 3 (4 spaces used for i
      type: awesome #string [literal]
      born: 2001 #number [literal]
```

2. Mapping

Mappings are used to associate key/value pairs that are unordered. Maps in YAML files can be nested by increasing the indentation, or new maps can be created at the same level by resolving the previous one.

```
name: "YAML Ain't Markup Language" #mapping
type: awesome
born: 2001
```

3. Sequences

Sequences in YAML are represented by using the hyphen (-) and space. They are ordered and can be embedded inside a map using indentation.

```
languages:
#Sequence
  - YAML
  - JAVA
  - XML
  - Python
  - C
```

Tip: Remember that the order matters with sequences but not with mappings.

4. Literals—Strings

String literals in YAML do not need to be quoted. It is only important to quote them when they contain a value that can be mistaken for a special character.

Here is an example where the string has to be quoted as & is a special character.

```
message1: YAML & JSON # breaks as a & is a special character
message2: "YAML & JSON" # Works as the string is quoted
```

Folding Strings

Strings can also be written in blocks and be interpreted without the new line characters using the fold operator (greater than).

```
it is actually not
```

The above YAML snippet is interpreted as in the output below.

```
message: "even though it looks like this is a multiline message,it is actually not"
```

Block strings

Strings can be interpreted as blocks using the block (pipe) character.

```
message: |
 this is
 a real multiline
 message
```

This is interpreted with the new lines (\n) as below.

```
message: this is
 a real multiline
 message
```

Chomp characters

Multiline strings may end with whitespaces. Preserve chomp(+) and strip chomp operators can be used either to preserve or strip the whitespaces. They can be used with block and pipe characters.

- Preserving new line character

```
message: >+
 This block line
 Will be interpreted as a single
 line with a newline character at the
 end
```

The above snippet is interpreted as below in JSON

```
{
  "message": "This block line Will be interpreted as a single line with a newline character at t
}
```

- Stripping new line character

```
end
```

The above snippet is interpreted as below in JSON.

```
{
  "message": "This block line Will be interpreted as a single line without the newline character
}
```

5. Comments

Unlike JSON, the YAML file supports comments, which start with #. To learn more, see – how to add comments in YAML.

```
---
# Comments inside a YAML file can be added followed by the '#' character
company: spacelift
```

💡 **You might also like:**

- 16 DevOps Best Practices to Follow
- Top Most Useful CI/CD Tools for DevOps
- Why Generic CI/CD Tools Will Not Deliver Successful IaC

Advanced YAML syntax

Now that we know all the basics, let's look at more advanced YAML syntax features.

Documents

The above YAML snippet is called a document. A single YAML file can have more than one document. Each document can be interpreted as a separate YAML file which means multiple documents can contain the same or duplicate keys which are not allowed in the same document.

The beginning of a document is denoted by three hyphens —.

A YAML file with multiple documents would look like this, where each new document is indicated by `---`.

```
---
# document 1
codename: YAML
```

```
 - configuration language
 - data persistence
 - internet messaging
 - cross-language data sharing
---
# document 3
company: spacelift
domain:
 - devops
 - devsecops
tutorial:
   - name: yaml
   - type: awesome
   - rank: 1
   - born: 2001
author: omkarbirade
published: true
...
```

Triple dots are used to end a YAML document without starting a new one `...`

Before we learn more about YAML, this is a good time to practice writing your own YAML file. They can be validated here.

Now that we have seen an online YAML parser in action, it's time we learn about schemas and tags.

2. Schemas and Tags

Let's take a moment to consider how YAML will interpret the given document. Is the sequence's first literal a string or a boolean?

```
literals:
 - true
 - random
```

You are correct if you answer that the first item on the list is a boolean, and you are also correct if you answer that it is a string. The way it is resolved is determined by the YAML schema that the parser has implemented.

YAML schemas

Schemas can be thought of as the way a parser resolves or understands nodes (values) present in a YAML file. There are primarily three default schemas in YAML:

1. *FailSafe Schema* understands only maps, sequences, and strings and is guaranteed to work with any YAML file.
2. *JSON schema* understands all types supported within JSON, including boolean, null, int, and float, as well as those in the FailSafe schema.
3. *Core schema* is an extension of the JSON schema, making it more human-readable supporting the same types but in multiple forms.

Coming back to the original question, if the parser supports only the basic schema (FailSafe Schema), the first item will be evaluated as a string. Otherwise, it will be evaluated as a boolean. Read more about YAML schemas here.

YAML tags

What if we explicitly want a value to be parsed in a specific way?

Let's say from the same example that we want the first true value to be parsed as a string instead of a boolean, even when the parser uses the JSON or the core schema.

This is where tags come into the picture. Tags can be thought of as types in YAML.

Even though we explicitly didn't mention the tags/types in any of the YAML snippets we saw so far, they are inferred automatically by the YAML parser. For instance, the maps have the tag/type as tag:yaml.org,2002:map, sequences are tag:yaml.org,2002:seq and strings are tag:yaml.org,2002:str

The snippet below works perfectly fine, even when we specify the tags. It can be validated here.

```
---
# A sample yaml file
company: !!str spacelift
domain:
 - !!str devops
 - !!str devsecops
tutorial:
   - name: !!str yaml
   - type: !!str awesome
   - rank: !!int 1
   - born: !!int 2001
author: !!str omkarbirade
published: !!bool true
```

We can use these tags to explicitly specify a type. For our example, all we have to do is specify the type as a string, and the YAML parser will parse it as a string.

```
scalars:
 - !!str true
 - random
```

3. Anchors and alias

With a lot of configuration, configuration files can become quite large.

Anchors (&) and aliases (*) are used to avoid duplication in YAML files. When writing large configurations in YAML, it is common for a specific configuration to be repeated.

```
service1:
    config:
        env: prod
        retries: 3
        version: 4.8
service2:
    config:
        env: prod
        retries: 3
        version: 4.8
service3:
    config:
        env: prod
        retries: 3
        version: 4.8
...
```

As more and more things are repeated for large configuration files, this becomes tedious.

Anchors and aliases allow us to rewrite the same snippet without repeating any configuration.

Anchors (&) are used to define a chunk of configuration, and aliases refer to that chunk at a different part of the configuration.

```
---
vars:
   service1:
       config: &service_config
           env: prod
           retries: 3
           version: 4.8
   service2:
       config: *service_config
   service3:
       config: *service_config
...
```

Anchors and aliases here helped us cut down the repeated configuration.

But practically, configurations won't be completely identical they would vary here and there. For instance, what if all the above services are running on different versions? Does this mean we have to rewrite and repeat the whole config?

This is where overrides (<<:) come to the rescue. We can still use aliases and make the changes that we need.

```
---
vars:
   service1:
       config: &service_config
```

```
            <<: *service_config
            version: 5
    service3:
        config:
            <<: *service_config
            version: 4.2
...
```

YAML files treat `:` , `{` , `}` , `[` , `]` , `,` , `&` , `*` , `#` , `?` , `|` , `--` , `<` , `>` , `=` , `!` , `%` , `@` , `\`, etc, as special characters.

If special characters are a part of the data/value, they can be escaped using entities, Unicode, or quotes.

Entity Escapes

- space: `&#x20;`
- colon: `&#58;`
- ampersand: `&`

Unicode Escapes

- space: `"\u0020"`
- single-quote: `"\u0027"`
- double quote: `"\u0022"`

Quoted Escapes

1. Double quote in a single quote: ‘YAML is the “best” configuration language’
2. Single quote in a double quote: “ Yes, the ‘best’ “

YAML vs JSON

How is YAML different from JSON? While both YAML and JSON are data serialization formats, YAML offers greater readability, support for complex data types, and the ability to include comments, making it more suitable for configuration files and human-readable data representation. JSON, on the other hand, is simpler and more widely used for data exchange in web applications and APIs due to its native support in JavaScript and smaller file size.

Let’s see an example.

The below code snippet of Kubernetes configuration is written in JSON. Don’t pay attention to what it does; just observe the file.

```
{
 "description": "APIService represents a server for a particular GroupVersion. Name must be \"ve
 "properties": {
   "apiVersion": {
```

```
},
"kind": {
  "description": "Kind is a string value representing the REST resource this object represent
  "type": [
    "string",
    "null"
  ],
  "enum": [
    "APIService"
  ]
},
"metadata": {
  "$ref": "https://kubernetesjsonschema.dev/master/_definitions.json#/definitions/io.k8s.apim
},
"spec": {
    "$ref": "https://kubernetesjsonschema.dev/master/_definitions.json#/definitions/io.k8s.kube
    "description": "Spec contains information for locating and communicating with a server"
  },
  "status": {
    "$ref": "https://kubernetesjsonschema.dev/master/_definitions.json#/definitions/io.k8s.kube
    "description": "Status contains derived information about an API server"
  }
 },
 "type": "object",
 "x-kubernetes-group-version-kind": [
  {
    "group": "apiregistration.k8s.io",
    "kind": "APIService",
    "version": "v1beta1"
  }
 ],
 "$schema": "http://json-schema.org/schema#"
}
```


Doesn't it look like a pure JSON file? Let's see if we can validate it in our YAML parser.

It's odd that the YAML parser didn't report the file as invalid. Does this imply that JSON is also YAML?

YAML is, in fact, a superset of JSON. All JSON files are valid YAML files, but not the other way around.

Can we combine JSON and YAML? Is it still a valid YAML file? Let's put this hypothesis to the test. Let us change some of the above snippets to make it look more like the YAML we are familiar with 😉


```
description: "APIService represents a server for a particular GroupVersion. Name must be \"versi
"properties": {
 "apiVersion": {
   "description": "APIVersion defines the versioned schema of this representation of an object.
   "type": [
     "string",
     "null"
   ]
 },
 "kind": {
   "description": "Kind is a string value representing the REST resource this object represents.
```

```
      "APIService"
    ]
  },
  "metadata": {
    "$ref": "https://kubernetesjsonschema.dev/master/_definitions.json#/definitions/io.k8s.apimac
  },
  "spec": {
    "$ref": "https://kubernetesjsonschema.dev/master/_definitions.json#/definitions/io.k8s.kube-a
    "description": "Spec contains information for locating and communicating with a server"
  },
  "status": {
    "$ref": "https://kubernetesjsonschema.dev/master/_definitions.json#/definitions/io.k8s.kube-a
    "description": "Status contains derived information about an API server"
  }
}
```

```
"type": "object"
"x-kubernetes-group-version-kind": [
  {
    "group": "apiregistration.k8s.io",
    "kind": "APIService",
    "version": "v1beta1"
  }
]
"$schema": "http://json-schema.org/schema#"
```


Notice that there isn't a root JSON wrapper {} anymore; there are just maps at the root level, but most of it is still JSON. Validate the file once more in a YAML parser. It is a valid YAML file, but when we try to validate it in a JSON parser, it says it is invalid. That's because the file is no longer JSON but rather YAML. This demonstrates that YAML is, in fact, the superset of JSON.

Examples: How to use YAML files?

We learned a lot about YAML and saw that it works great as a configuration language. Let us see it in action with some of the most famous tools.

Using YAML in Ansible playbooks

Ansible playbooks are used to automate repeated tasks that execute actions automatically. Playbooks are expressed in *YAML format* and perform any action defined in plays. To learn more about Ansible playbooks, see our article: Working with Ansible Playbooks – Tips & Tricks with Examples.

Here is a simple Ansible playbook that installs Nginx, applies the specified template to replace the existing default Nginx landing page, and finally enables TCP access on port 80.

```
---
- hosts: all
  become: yes
  vars:
    page_title: Spacelift
    page_description: Spacelift is a sophisticated CI/CD platform for Terraform, CloudFormation,
  tasks:
```

```
- name: Apply Page Template
  template:
    src: files/spacelift-intro.j2
    dest: /var/www/html/index.nginx-debian.html

- name: Allow all access to tcp port 80
  ufw:
    rule: allow
    port: '80'
    proto: tcp
```

Kubernetes

Kubernetes, also known as K8s, is an open-source system for automating the deployment, scaling, and management of containerized applications.

Kubernetes works based on a state model where it tries to reach the desired state from the current state in a declarative way. Kubernetes uses *YAML files* to define the Kubernetes object, which is applied to the cluster to create resources like pods, services, and deployments.

Here is a YAML file that describes a deployment that runs Nginx.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
spec:
 selector:
   matchLabels:
     app: nginx
replicas: 2 # tells deployment to run 2 pods matching the template
template:
   metadata:
     labels:
       app: nginx
spec:
     containers:
       - name: nginx
image: nginx:1.14.2
ports:
   - containerPort: 80
```

Best practices for writing YAML files

YAML relies heavily on indentation, so if your indentation is not correct, the yaml file will be automatically invalid when you try to use it in your automations. Here are some of the best practices you can use when writing yaml files:

1. **Use consistent indentation** – You should always use spaces and be consistent with the number of spaces you use throughout your file (usually, this should be two or four).

people, the key names used throughout the file should be meaningful.

4. **Structure complex data** – Lists are denoted by a dash (“-”), while maps are key-value pairs.
5. **Use comments** – Comments will help you and others understand what that data is used for. To write a comment in a YAML file, you can prefix the line with the “#” character
6. **Use validations** – Validate your YAML files regularly using special tools for this purpose to catch syntax errors fast.
7. **Use quotes for strings** – This is especially useful if your string contains special characters or starts with a number.

Interesting facts about YAML

YAML works great as a configuration language, but it is important to be aware of certain challenges as well when using it.

The curious case of the Norway problem

Imagine listing the abbreviation of all the countries where it snows.

```
countries:
- GB # Great britain
- IE # Ireland
- FR # France
- DE # Denmark
- NO # Norway
```

All looks good, right? But when you try to read this YAML file in Python, we see NO being read as False instead of ‘NO’.

```
>>> from pyyaml import load
>>> load(the_configuration)
{'countries': ['GB', 'IE', 'FR', 'DE', False]}
```

So why does this happen?

Remember the core schema which interprets NULL | null the same way? The same schema interprets FALSE | F | NO the same way. So, instead of parsing NO as a string, it parses it as a boolean. This can be easily solved by quoting NO.

```
countries:
- GB # Great Britain
- IE # Ireland
- FR # France
- DE # Denmark
- 'NO' # Norway
```

Find the code here.

Key Points

Congratulations on completing the article. You are now on your way to becoming a YAML expert. Isn't YAML fantastic?

YAML is an important language that finds its uses almost everywhere where writing configuration is required. Kubernetes, Ansible, docker-compose, and other tools are excellent examples.

Then, you can use Spacelift to mix and match Terraform, Pulumi, AWS CloudFormation, Kubernetes, and Ansible Stacks and have them talk to one another. For example, you can set up Terraform Stacks to provision the required infrastructure (like an ECS/EKS cluster with all its dependencies) and then deploy the following via a Kubernetes Stack. Check it out for free by creating a trial account.

Hope you enjoyed reading 🙂

Written by

Omkar Birade

Omkar is a CoFounder at Interleap and a seasoned DevSecOps engineer with a passion for DevOps and security. He specializes in AWS DevOps, Terraform, CDKTF and Docker and has held roles ranging from a Fullstack engineer to a DevSecOps Engineer. In his spare time, he takes pleasure in learning new things and sharing his learnings on his personal blog at omkarbirade.medium.com.

Read also

GENERAL 14 min read

Writing .gitlab-ci.yml File with Examples [Tutorial]

ANSIBLE 12 min read

44 Ansible Best Practices to Follow [Tips & Tricks]

KUBERNETES 19 min read

17 Kubernetes Best Practices Every Developer Should Know