YAML - Quick Guide

Previous Page Next Page

YAML - Introduction

YAML Ain't Markup Language is a data serialization language that matches user's expectations about data. It designed to be human friendly and works perfectly with other programming languages. It is useful to manage data and includes Unicode printable characters. This chapter will give you an introduction to YAML and gives you an idea at its features.

Format

Consider the text shown below –

```
Quick brown fox jumped over the lazy dog.
```

The YAML text for this will be represented as shown below –

```
yaml.load(Quick brown fox jumped over the lazy dog.)
>>'Quick brown fox jumped over the lazy dog.'
```

Note that YAML takes the value in string format and represents the output as mentioned above.

Examples

Let us understand the formats in YAML with the help of the following examples –

Consider the following point number of "pi", which has a value of 3.1415926. In YAML, it is represented as a floating number as shown below –

```
>>> yaml.load('3.1415926536')
3.1415926536
```

Suppose, multiple values are to be loaded in specific data structure as mentioned below –

```
eggs
ham
```

```
spam
French basil salmon terrine
```

When you load this into YAML, the values are taken in an array data structure which is a form of list. The output is as shown below –

```
>>> yaml.load('''
   - eggs
   - ham
   - spam
   - French basil salmon terrine
   ''')
['eggs', 'ham', 'spam', 'French basil salmon terrine']
```

Features

YAML includes a markup language with important construct, to distinguish data-oriented language with the document markup. The design goals and features of YAML are given below –

- Matches native data structures of agile methodology and its languages such as Perl, Python, PHP, Ruby and JavaScript
- YAML data is portable between programming languages
- Includes data consistent data model
- Easily readable by humans
- Supports one-direction processing
- Ease of implementation and usage

YAML - Basics

Now that you have an idea about YAML and its features, let us learn its basics with syntax and other operations. Remember that YAML includes a human readable structured format.

Rules for Creating YAML file

When you are creating a file in YAML, you should remember the following basic rules –

- YAML is case sensitive
- The files should have **.yaml** as the extension
- YAML does not allow the use of tabs while creating YAML files; spaces are allowed instead

Basic Components of YAML File

The basic components of YAML are described below –

Conventional Block Format

This block format uses **hyphen+space** to begin a new item in a specified list. Observe the example shown below –

```
--- # Favorite movies
 - Casablanca
 - North by Northwest
 - The Man Who Wasn't There
```

Inline Format

Inline format is delimited with **comma and space** and the items are enclosed in JSON. Observe the example shown below –

```
--- # Shopping list
   [milk, groceries, eggs, juice, fruits]
```

Folded Text

Folded text converts newlines to spaces and removes the leading whitespace. Observe the example shown below –

```
- {name: John Smith, age: 33}
- name: Mary Smith
  age: 27
```

The structure which follows all the basic conventions of YAML is shown below –

```
men: [John Smith, Bill Jones]
women:
  - Mary Smith
  - Susan Williams
```

Synopsis of YAML Basic Elements

The synopsis of YAML basic elements is given here: Comments in YAML begins with the (**#**) character.

Comments must be separated from other tokens by whitespaces.

Indentation of whitespace is used to denote structure.

Tabs are not included as indentation for YAML files.

List members are denoted by a leading hyphen (**-**).

List members are enclosed in square brackets and separated by commas.

Associative arrays are represented using colon **(:)** in the format of key value pair. They are enclosed in curly braces **{}**.

Multiple documents with single streams are separated with 3 hyphens (---).

Repeated nodes in each file are initially denoted by an ampersand (**&**) and by an asterisk (*****) mark later.

YAML always requires colons and commas used as list separators followed by space with scalar values.

Nodes should be labelled with an exclamation mark (**!**) or double exclamation mark (**!!**), followed by string which can be expanded into an URI or URL.

YAML - Indentation and Separation

Indentation and separation are two main concepts when you are learning any programming language. This chapter talks about these two concepts related to YAML in detail.

Indentation of YAML

YAML does not include any mandatory spaces. Further, there is no need to be consistent. The valid YAML indentation is shown below –

```
a:
   b:
      - c
      -  d
      - e
f:
      "ghi"
```

You should remember the following rules while working with indentation in YAML:Flow blocks must be intended with at least some spaces with surrounding current block level.

Flow content of YAML spans multiple lines. The beginning of flow content begins with **{** or **[**.

Block list items include same indentation as the surrounding block level because - is considered as a part of indentation.

Example of Intended Block

Observe the following code that shows indentation with examples –

```
--- !clarkevans.com/^invoice
invoice: 34843
date   : 2001-01-23
bill-to: &id001
   given  : Chris
   family : Dumars
   address:
      lines: |
            458 Walkman Dr.
            Suite #292
      city    : Royal Oak
      state   : MI
      postal  : 48046
ship-to: *id001
product:
    - sku         : BL394D
      quantity    : 4
      description : Basketball
      price       : 450.00
   - sku         : BL4438H
      quantity    : 1
      description : Super Hoop
      price       : 2392.00
tax  : 251.42
total: 4443.52
comments: >
    Late afternoon is best.
    Backup contact is Nancy
    Billsmer @ 338-4338.
```

Separation of Strings

Strings are separated using double-quoted string. If you escape the newline characters in a given string, it is completely removed and translated into space value.

Example

In this example we have focused listing of animals listed as an array structure with data type of string. Every new element is listed with a prefix of hyphen as mentioned as prefix.

```
-
 - Cat
 - Dog
 - Goldfish
-
 - Python
 - Lion
 - Tiger
```

Another example to explain string representation in YAML is mentioned below.

```
 errors:
      messages:
         already_confirmed: "was already confirmed, please try signing in"
```

```
confirmation_period_expired: "needs to be confirmed within %{period}, please request a new one"
expired: "has expired, please request a new one"
not_found: "not found"
not_locked: "was not locked"
not_saved:
   one: "1 error prohibited this %{resource} from being saved:"
   other: "%{count} errors prohibited this %{resource} from being saved:"
```

This example refers to the set of error messages which a user can use just by mentioning the key aspect and to fetch the values accordingly. This pattern of YAML follows the structure of JSON which can be understood by user who is new to YAML.

YAML - Comments

Now that you are comfortable with the syntax and basics of YAML, let us proceed further into its details. In this chapter, we will see how to use comments in YAML.

YAML supports single line comments. Its structure is explained below with the help of an example –

```
# this is single line comment.
```

YAML does not support multi line comments. If you want to provide comments for multiple lines, you can do so as shown in the example below –

```
# this
# is a multiple
# line comment
```

Features of Comments

The features of comments in YAML are given below –

A commented block is skipped during execution.

Comments help to add description for specified code block.

Comments must not appear inside scalars.

YAML does not include any way to escape the hash symbol (#) so within multi-line string so there is no way to divide the comment from the raw string value.

The comments within a collection are shown below –

```
key: #comment 1
   - value line 1
   #comment 2
   - value line 2
```

```
   #comment 3
   - value line 3
```

The shortcut key combination for commenting YAML blocks is **Ctrl+Q**.

If you are using **Sublime Text editor**, the steps for commenting the block are mentioned below –

Select the block. Use "CTRL + /" on Linux and Windows and "CMD+/" for Mac operating system. Execute the block.

Note that the same steps are applicable if you are using **Visual Studio Code Editor**. It is always recommended to use **Sublime Text Editor** for creating YAML files as it supported by most operating systems and includes developer friendly shortcut keys.

YAML - Collections and Structures

YAML includes block collections which use indentation for scope. Here, each entry begins with a new line. Block sequences in collections indicate each entry with a **dash and space** (-). In YAML, block collections styles are not denoted by any specific indicator. Block collection in YAML can distinguished from other scalar quantities with an identification of key value pair included in them.

Mappings are the representation of key value as included in JSON structure. It is used often in multi-lingual support systems and creation of API in mobile applications. Mappings use key value pair representation with the usage of **colon and space** (**:**).

Examples

Consider an example of sequence of scalars, for example a list of ball players as shown below –

```
- Mark Joseph
- James Stephen
- Ken Griffey
```

The following example shows mapping scalars to scalars –

```
hr: 87
avg: 0.298
rbi: 149
```

The following example shows mapping scalars to sequences –

```
European:
- Boston Red Sox
- Detroit Tigers
- New York Yankees
```

```
national:
- New York Mets
- Chicago Cubs
- Atlanta Braves
```

Collections can be used for sequence mappings which are shown below –

```
-
name: Mark Joseph
hr: 87
avg: 0.278
-
name: James Stephen
hr: 63
avg: 0.288
```

With collections, YAML includes flow styles using explicit indicators instead of using indentation to denote space. The flow sequence in collections is written as comma separated list enclosed in square brackets. The best illustration for collection which is included in PHP frameworks like symphony.

```
[PHP, Perl, Python]
```

These collections are stored in documents. The separation of documents in YAML is denoted with three hyphens or dashes (---). The end of document is marked with three dots (…).

The separation of documents in YAML is denoted by three dashes (---). The end of document is represented with three dots (…).

The document representation is referred as structure format which is mentioned below –

```
# Ranking of 1998 home runs
---
- Mark Joseph
- James Stephen
- Ken Griffey

# Team ranking
---
- Chicago Cubs
- St Louis Cardinals
```

A question mark with a combination of space indicates a complex mapping in structure. Within a block collection, a user can include structure with a dash, colon and question mark. The following example shows the mapping between sequences –

```
- 2001-07-23
? [ New York Yankees,Atlanta Braves ]
: [ 2001-07-02, 2001-08-12, 2001-08-14]
```

YAML - Scalars and Tags

Scalars in YAML are written in block format using a literal type which is denoted as(**|**). It denotes line breaks count. In YAML, scalars are written in folded style (**>**) where each line denotes a folded space which ends with an **empty line** or **more indented** line.

New lines are preserved in literals are shown below –

```
ASCII Art
--- |
\//||\/||
// || ||__
```

The folded newlines are preserved for **more indented lines** and **blank lines** as shown below –

```
>
Sammy Sosa completed another
fine season with great stats.
63 Home Runs
0.288 Batting Average
What a year!
```

YAML flow scalars include plain styles and quoted styles. The double quoted style includes various escape sequences. Flow scalars can include multiple lines; line breaks are always folded in this structure.

```
plain:
This unquoted scalar
spans many lines.
quoted: "So does this
quoted scalar.\n"
```

In YAML, untagged nodes are specified with a specific type of the application. The examples of tags specification generally use **seq, map** and **str** types for YAML tag repository. The tags are represented as examples which are mentioned as below –

Integer tags

These tags include integer values in them. They are also called as numeric tags.

```
canonical: 12345
decimal: +12,345
sexagecimal: 3:25:45
```

```
octal: 014
hexadecimal: 0xC
```

Floating point numbers

These tags include decimal and exponential values. They are also called as exponential tags.

```
canonical: 1.23015e+3
exponential: 12.3015e+02
sexagecimal: 20:30.15
fixed: 1,230.15
negative infinity: -.inf
not a number: .NaN
```

Miscellaneous Tags

It includes a variety of integer, floating and string values embedded in them. Hence it is called miscellaneous tags.

```
null: ~
true: y
false: n
string: '12345'
```

YAML - Full Length Example

The following full-length example specifies the construct of YAML which includes symbols and various representations which will be helpful while converting or processing them in JSON format. These attributes are also called as key names in JSON documents. These notations are created for security purposes.

The above YAML format represents various attributes of defaults, adapter, and host with various other attributes. YAML also keeps a log of every file generated which maintains a track of error messages generated. On converting the specified YAML file in JSON format we get a desired output as mentioned below –

```
defaults: &defaults
   adapter:  postgres
   host:     localhost

development:
   database: myapp_development
   <<: *defaults

test:
   database: myapp_test
   <<: *defaults
```

Let’s convert the YAML to JSON format and check on the output.

```
{
   "defaults": {
      "adapter": "postgres",
      "host": "localhost"
   },
   "development": {
      "database": "myapp_development",
      "adapter": "postgres",
      "host": "localhost"
   },
   "test": {
      "database": "myapp_test",
      "adapter": "postgres",
      "host": "localhost"
   }
}
```

The defaults key with a prefix of “ <<: *” is included as and when required with no need to write the same code snippet repeatedly.

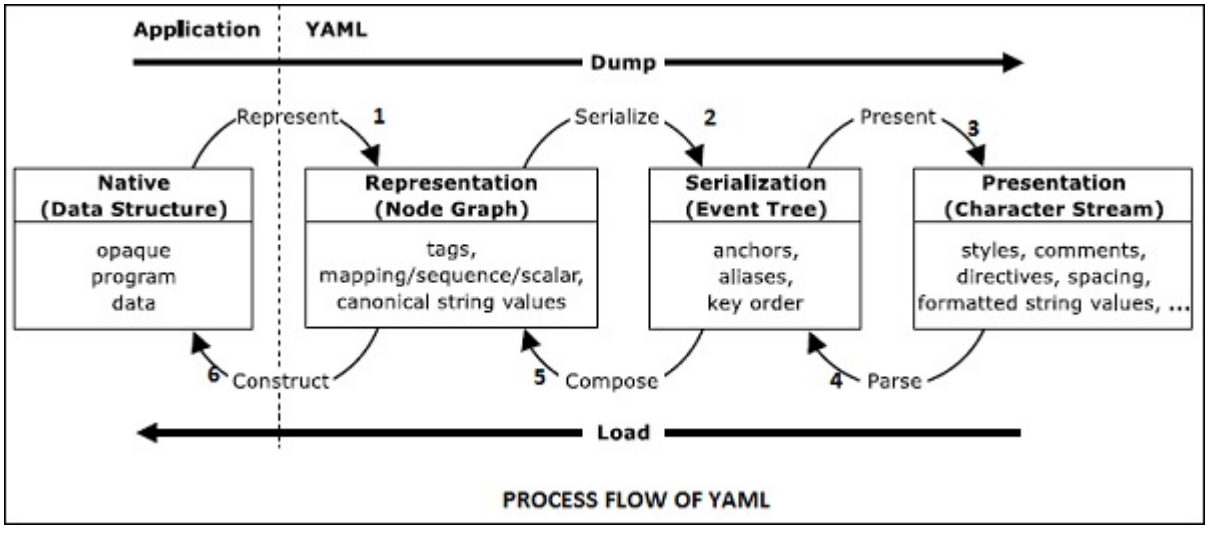
YAML - Processes

YAML follows a standard procedure for Process flow. The native data structure in YAML includes simple representations such as nodes. It is also called as Representation Node Graph.

It includes mapping, sequence and scalar quantities which is being serialized to create a serialization tree. With serialization the objects are converted with stream of bytes.

The serialization event tree helps in creating presentation of character streams as represented in the following diagram.

The reverse procedure parses the stream of bytes into serialized event tree. Later, the nodes are converted into node graph. These values are later converted in YAML native data structure. The figure below explains this –



The information in YAML is used in two ways: **machine processing** and **human consumption**. The processor in YAML is used as a tool for the procedure of converting information between complementary views in the diagram given above. This chapter describes the information structures a YAML processor must provide within a given application.

YAML includes a serialization procedure for representing data objects in serial format. The processing of YAML information includes three stages: **Representation, Serialization, Presentation and parsing**. Let us discuss each of them in detail.

Representation

YAML represents the data structure using three kinds of nodes: **sequence, mapping** and **scalar**.

Sequence

Sequence refers to the ordered number of entries, which maps the unordered association of key value pair. It corresponds to the Perl or Python array list.

The code shown below is an example of sequence representation −

```
product:
   - sku         : BL394D
     quantity    : 4
     description : Football
     price       : 450.00
   - sku         : BL4438H
     quantity    : 1
     description : Super Hoop
     price       : 2392.00
```

Mapping

Mapping on the other hand represents dictionary data structure or hash table. An example for the same is mentioned below −

```
batchLimit: 1000
threadCountLimit: 2
key: value
keyMapping: <What goes here?>
```

Scalars

Scalars represent standard values of strings, integers, dates and atomic data types. Note that YAML also includes nodes which specify the data type structure. For more information on scalars, please refer to the chapter 6 of this tutorial.

Serialization

Serialization process is required in YAML that eases human friendly key order and anchor names. The result of serialization is a YAML serialization tree. It can be traversed to produce a series of event calls of YAML data.

An example for serialization is given below –

```
consumer:
   class: 'AppBundle\Entity\consumer'
   attributes:
      filters: ['customer.search', 'customer.order', 'customer.boolean']
   collectionOperations:
      get:
         method: 'GET'
         normalization_context:
       groups: ['customer_list']
   itemOperations:
      get:
         method: 'GET'
         normalization_context:
            groups: ['customer_get']
```

Presentation

The final output of YAML serialization is called presentation. It represents a character stream in a human friendly manner. YAML processor includes various presentation details for creating stream, handling indentation and formatting content. This complete process is guided by the preferences of user.

An example for YAML presentation process is the result of JSON value created. Observe the code given below for a better understanding –

```
{
   "consumer": {
      "class": "AppBundle\\Entity\\consumer",
      "attributes": {
         "filters": [
            "customer.search",
            "customer.order",
            "customer.boolean"
         ]
      },
      "collectionOperations": {
         "get": {
            "method": "GET",
            "normalization_context": {
               "groups": [
                  "customer_list"
               ]
            }
         }
```

```
      },
      "itemOperations": {
         "get": {
            "method": "GET",
            "normalization_context": {
               "groups": [
                  "customer_get"
               ]
            }
         }
      }
   }
}
```

Parsing

Parsing is the inverse process of presentation; it includes a stream of characters and creates a series of events. It discards the details introduced in the presentation process which causes serialization events. Parsing procedure can fail due to ill-formed input. It is basically a procedure to check whether YAML is well-formed or not.

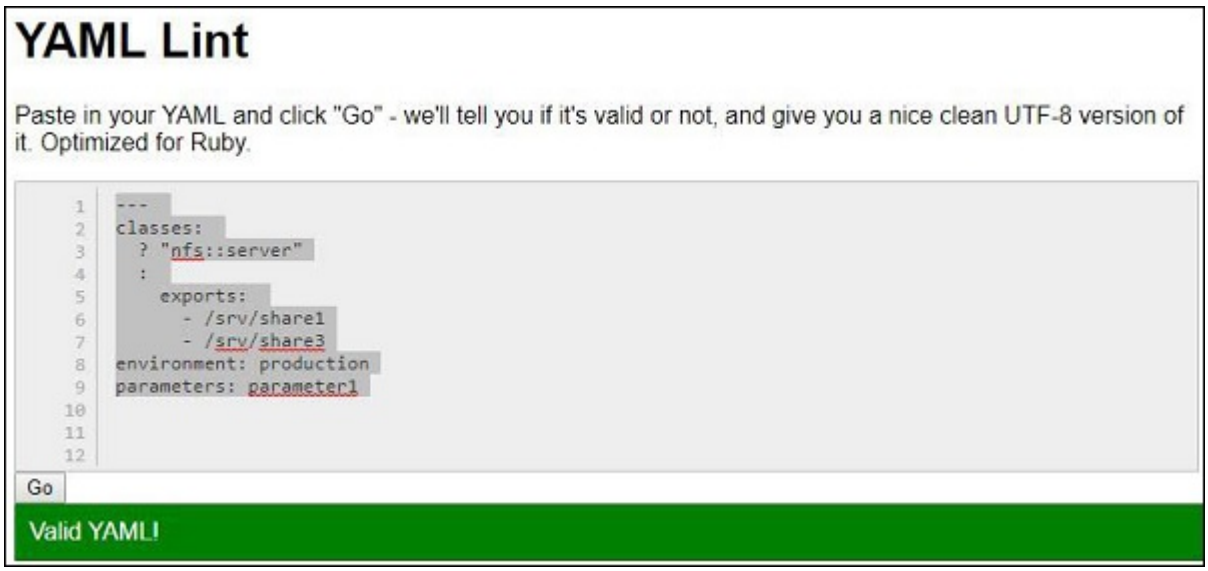
Consider a YAML example which is mentioned below –

```
---
   environment: production
   classes:
      nfs::server:
         exports:
            - /srv/share1
            - /srv/share3
   parameters:
      paramter1
```

With three hyphens, it represents the start of document with various attributes later defined in it.

YAML lint is the online parser of YAML and helps in parsing the YAML structure to check whether it is valid or not. The official link for YAML lint is mentioned below: http://www.yamllint.com/

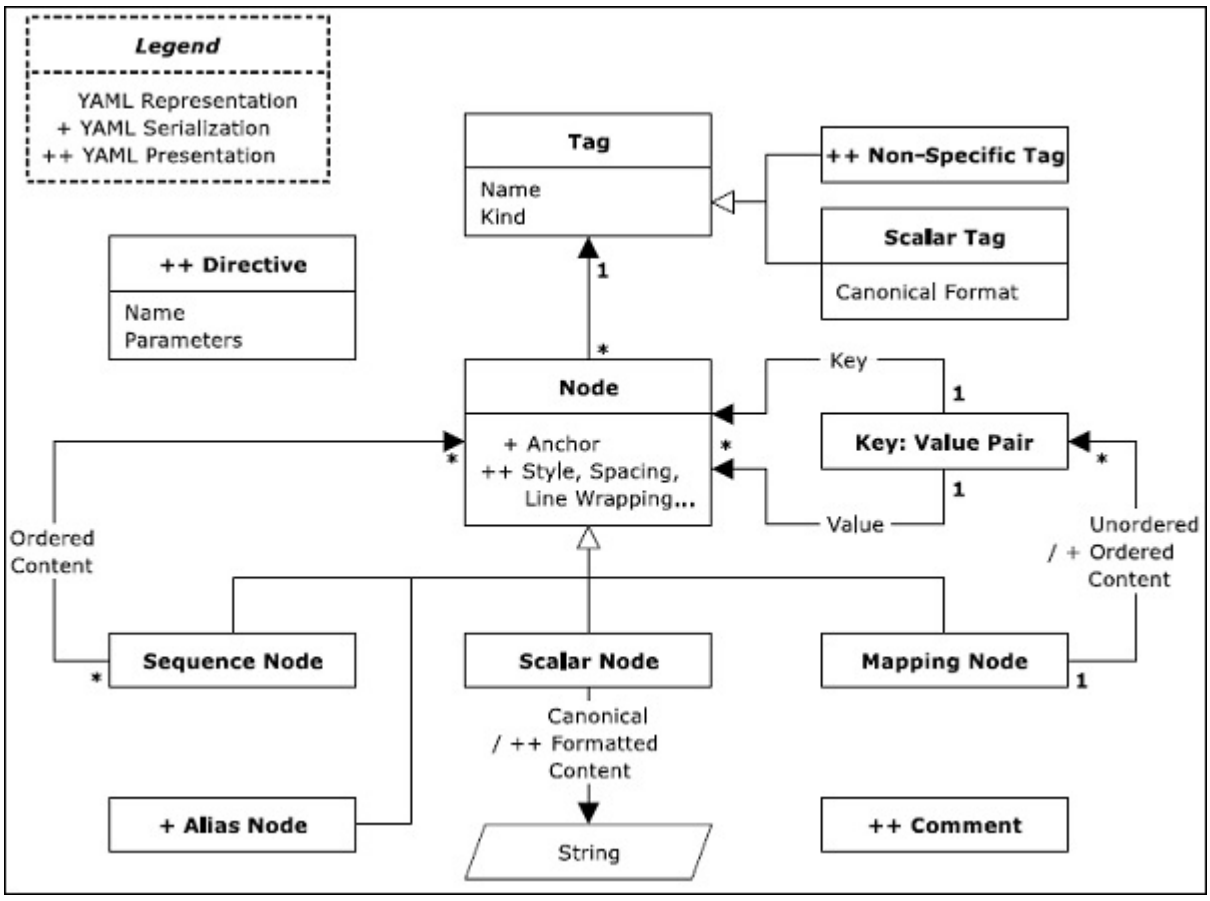
You can see the output of parsing as shown below –



YAML - Information Models

This chapter will explain the detail about the procedures and processes that we discussed in last chapter. Information Models in YAML will specify the features of serialization and presentation procedure in a systematic format using a specific diagram.

For an information model, it is important to represent the application information which are portable between programming environments.

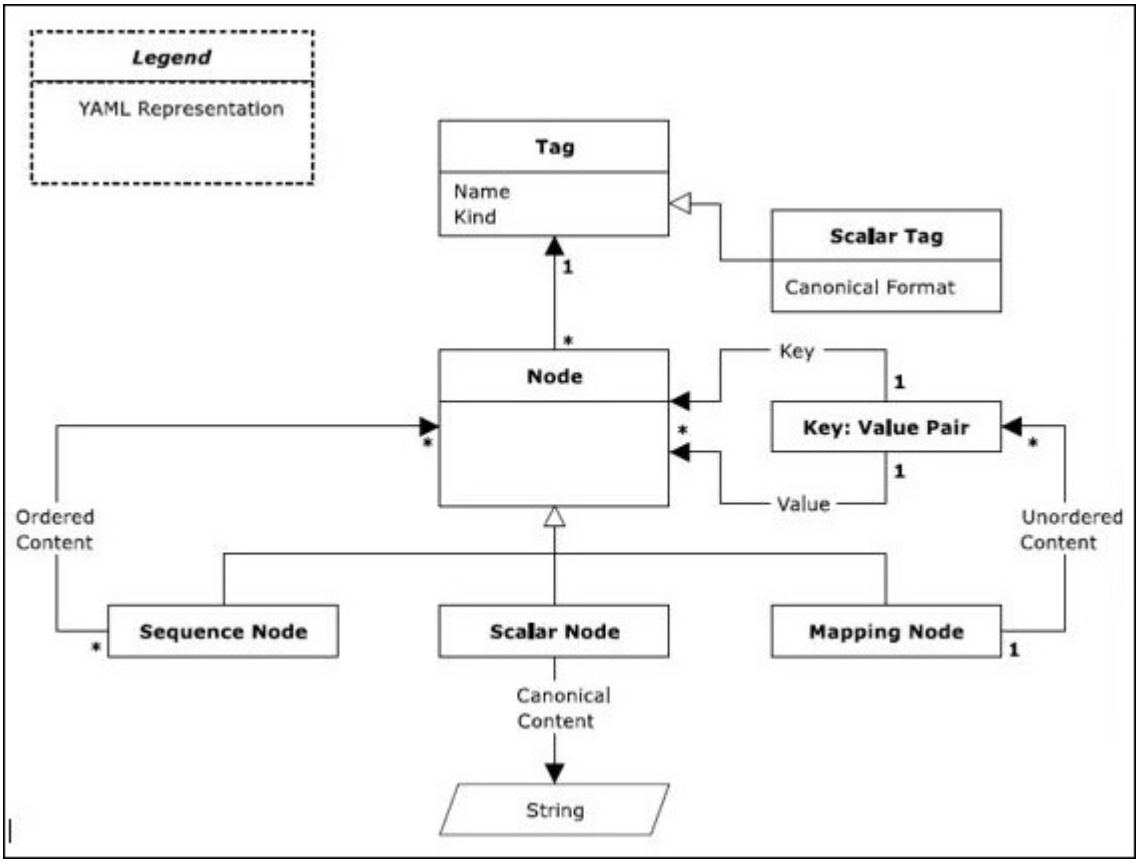


The diagram shown above represents a normal information model which is represented in graph format. In YAML, the representation of native data is rooted, connected and is

directed graph of tagged nodes. If we mention directed graph, it includes a set of nodes with directed graph. As mentioned in the information model, YAML supports three kinds of nodes namely –

- Sequences
- Scalars
- Mappings

The basic definitions of these representation nodes were discussed in last chapter. In this chapter, we will focus on schematic view of these terms. The following sequence diagram represents the workflow of legends with various types of tags and mapping nodes.



There are three types of nodes: **sequence node, scalar node** and **mapping node**.

Sequences

Sequence node follows a sequential architecture and includes an ordered series of zero or more nodes. A YAML sequence may contain the same node repeatedly or a single node.

Scalars

The content of scalars in YAML includes Unicode characters which can be represented in the format with a series of zero. In general, scalar node includes scalar quantities.

Mapping

Mapping node includes the key value pair representation. The content of mapping node includes a combination of key-value pair with a mandatory condition that key name should be maintained unique. Sequences and mappings collectively form a collection.

Note that as represented in the diagram shown above, scalars, sequences and mappings are represented in a systematic format.

YAML - Syntax Characters

Various types of characters are used for various functionalities. This chapter talks in detail about syntax used in YAML and focuses on character manipulation.

Indicator Characters

Indicator characters include a special semantics used to describe the content of YAML document. The following table shows this in detail.

Sr.No.	Character & Functionality
1	**–** It denotes a block sequence entry
2	**?** It denotes a mapping key
3	**:** It denotes a mapping value
4	**,** It denotes flow collection entry
5	**[** It starts a flow sequence
6	**]** It ends a flow sequence
7	**{** It starts a flow mapping

8	**}** It ends a flow mapping
9	**#** It denotes the comments
10	**&** It denotes node’s anchor property
11	***** It denotes alias node
12	**!** It denotes node’s tag
13	**|** It denotes a literal block scalar
14	**>** It denotes a folded block scalar
15	**`** Single quote surrounds a quoted flow scalar
16	**"** Double quote surrounds double quoted flow scalar
17	**%** It denotes the directive used

The following example shows the characters used in syntax –

```
%YAML 1.1
---
!!map {
   ? !!str "sequence"
   : !!seq [
```

```
      !!str "one", !!str "two"
   ],
   ? !!str "mapping"
   : !!map {
      ? !!str "sky" : !!str "blue",
      ? !!str "sea" : !!str "green",
   }
}

# This represents
# only comments.
---
!!map1 {
   ? !!str "anchored"
   : !local &A1 "value",
   ? !!str "alias"
   : *A1,
}
!!str "text"
```

YAML - Syntax Primitives

In this chapter you will learn about the following aspects of syntax primitives in YAML –

- Production parameters
- Indentation Spaces
- Separation Spaces
- Ignored Line Prefix
- Line folding

Let us understand each aspect in detail.

Production Parameters

Production parameters include a set of parameters and the range of allowed values which are used on a specific production. The following list of production parameters are used in YAML –

Indentation

It is denoted by character **n** or **m** Character stream depends on the indentation level of blocks included in it. Many productions have parameterized these features.

Context

It is denoted by **c**. YAML supports two groups of contexts: **block styles** and **flow styles**.

Style

It is denoted by s. Scalar content may be presented in one of the five styles: **plain, double quoted and single quoted flow, literal and folded block.**

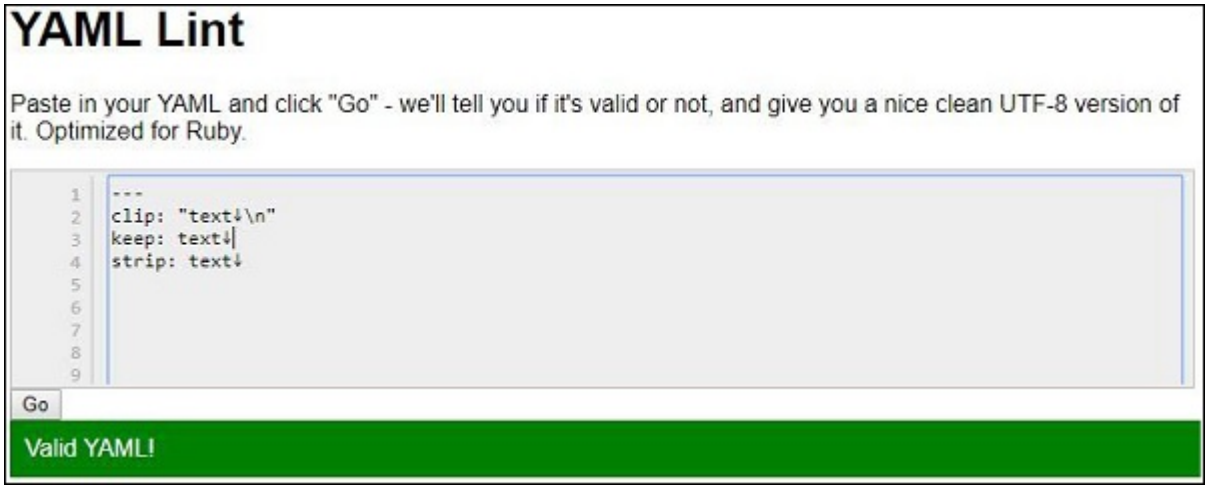
Chomping

It is denoted by **t**. Block scalars offer many mechanisms which help in trimming the block: **strip, clip** and **keep**. Chomping helps in formatting new line strings. It is used Block style representation. Chomping process happens with the help of indicators. The indicators controls what output should be produced with newlines of string. The newlines are removed with **(-)** operator and newlines are added with **(+)** operator.

An example for chomping process is shown below –

```
strip: |-
   text↓
clip: |
   text↓
keep: |+
   text↓
```

The output after parsing the specified YAML example is as follows –



Indentation Spaces

In YAML character stream, indentation is defined as a line break character by zero or more characters. The most important point to be kept in mind is that indentation must not contain any tab characters. The characters in indentation should never be considered as a part of node's content information. Observe the following code for better understanding –

```
%YAML 1.1
---
!!map {
   ? !!str "Not indented"
   : !!map {
      ? !!str "By one space"
      : !!str "By four\n spaces\n",
      ? !!str "Flow style"
      : !!seq [
         !!str "By two",
         !!str "Still by two",
         !!str "Again by two",
```

```
      ]
   }
}
```

The output that you can see after indentation is as follows –

```
{
   "Not indented": {
      "By one space": "By four\n spaces\n",
      "Flow style": [
         "By two",
         "Still by two",
         "Again by two"
      ]
   }
}
```

Separation Spaces

YAML uses space characters for separation between tokens. The most important note is that separation in YAML should not contain tab characters.

The following lone of code shows the usage of separation spaces –

```
{ · first: · Sammy, · last: · Sosa · }
```

The syntax shown above gives you the following output:

```
{
   "\u00b7 last": "\u00b7 Sosa \u00b7",
   "\u00b7 first": "\u00b7 Sammy"
}
```

Ignored Line Prefix

Empty prefix always includes indentation depending on the scalar type which also includes a leading whitespace. Plain scalars should not contain any tab characters. On the other hand, quoted scalars may contain tab characters. Block scalars completely depend on indentation.

The following example shows the working of ignored line prefix in a systematic manner –

```
%YAML 1.1
---
!!map {
   ? !!str "plain"
   : !!str "text lines",
   ? !!str "quoted"
   : !!str "text lines",
   ? !!str "block"
```

```
   : !!str "text·®lines\n"
}
```

The output achieved for the block streams is as follows –

```
{
   "plain": "text lines",
   "quoted": "text lines",
   "block": "text\u00b7\u00aelines\n"
}
```

Line Folding

Line Folding allows breaking long lines for readability. More amounts of short lines mean better readability. Line folding is achieved by noting original semantics of long line. The following example demonstrates line folding –

```
%YAML 1.1
--- !!str
"specific\L\
trimmed\n\n\n\
as space"
```

You can see the output for line folding in JSON format as follows –

```
"specific\u2028trimmed\n\n\nas space"
```

YAML - Character Streams

In YAML, you come across various character streams as follows –

- Directives
- Document Boundary Markers
- Documents
- Complete Stream

In this chapter, we will discuss them in detail.

Directives

Directives are basic instructions used in YAML processor. Directives are the presentation details like comments which are not reflected in serialization tree. In YAML, there is no way to define private directives. This section discusses various types of directives with relevant examples –

Reserved Directives

Reserved directives are initialized with three hyphen characters (---) as shown in the example below. The reserved directives are converted into specific value of JSON.

```
%YAML 1.1
--- !!str
"foo"
```

YAML Directive

YAML Directives are default directives. If converted in JSON, the value fetched includes forward slash character in preceding and terminating characters.

```
%YAML 1.1
---
!!str "foo"
```

Document Boundary Markers

YAML uses these markers to allow more than one document to be contained in one stream. These markers are specially used to convey the structure of YAML document. Note that a line beginning with "---"is used to start a new document.

The following code explains about this with examples –

```
%YAML 1.1
---
!!str "foo"
%YAML 1.1
---
!!str "bar"
%YAML 1.1
---
!!str "baz"
```

Documents

YAML document is considered as a single native data structure presented as a single root node. The presentation details in YAML document such as directives, comments, indentation and styles are not considered as contents included in them.

There are two types of documents used in YAML. They are explained in this section –

Explicit Documents

It begins with the document start marker followed by the presentation of the root node. The example of YAML explicit declaration is given below –

```
---

some: yaml

...
```

It includes an explicit start and end markers which is "---"and "…" in given example. On converting the specified YAML in JSON format, we get the output as shown below –

```
{
   "some": "yaml"
}
```

Implicit Documents

These documents do not begin with a document start marker. Observe the code given below –

```
fruits:
   - Apple
   - Orange
   - Pineapple
   - Mango
```

Converting these values in JSON format we get the output as a simple JSON object as given below –

```
{
   "fruits": [
      "Apple",
      "Orange",
      "Pineapple",
      "Mango"
   ]
}
```

Complete Stream

YAML includes a sequence of bytes called as character stream. The stream begins with a prefix containing a byte order denoting a character encoding. The complete stream begins with a prefix containing a character encoding, followed by comments.

An example of complete stream (character stream) is shown below –

```
%YAML 1.1
---
!!str "Text content\n"
```

YAML - Node Properties

Each presentation node includes two major characteristics called **anchor** and **tag**. Node properties may be specified with node content, omitted from the character stream.

The basic example of node representation is as follows –

```
%YAML 1.1
---
!!map {
   ? &A1 !!str "foo"
   : !!str "bar",
   ? !!str &A2 "baz"
   : *a1
}
```

Node Anchors

The anchor property represents a node for future reference. The character stream of YAML representation in node is denoted with the **ampersand (&)** indicator. The YAML processor need not preserve the anchor name with the representation details composed in it. The following code explains this –

```
%YAML 1.1
---
!!map {
   ? !!str "First occurence"
   : &A !!str "Value",
   ? !!str "Second occurence"
   : *A
}
```

The output of YAML generated with anchor nodes is shown below –

```
---
!!map {
   ? !!str "First occurence"
   : !!str "Value",
   ? !!str "Second occurence"
   : !!str "Value",
}
```

Node Tags

The tag property represents the type of native data structure which defines a node completely. A tag is represented with the (**!**) indicator. Tags are considered as an inherent part of the representation graph. The following example of explains node tags in detail –

```
%YAML 1.1
---
!!map {
   ? !<tag:yaml.org,2002:str> "foo"
   : !<!bar> "baz"
}
```

Node Content

Node content can be represented in a flow content or block format. Block content extends to the end of line and uses indentation to denote structure. Each collection kind can be represented in a specific single flow collection style or can be considered as a single block. The following code explains this in detail –

```
%YAML 1.1
---
!!map {
   ? !!str "foo"
   : !!str "bar baz"
}

%YAML 1.1
---
!!str "foo bar"

%YAML 1.1
---
!!str "foo bar"

%YAML 1.1
---
!!str "foo bar\n"
```

YAML - Block Scalar Header

In this chapter, we will focus on various scalar types which are used for representing the content. In YAML, comments may either precede or follow scalar content. It is important to note that comments should not be included within scalar content.

Note that all flow scalar styles can include multiple lines, except with usage in multiple keys.

The representation of scalars is given below –

```
%YAML 1.1
---
!!map {
   ? !!str "simple key"
   : !!map {
      ? !!str "also simple"
      : !!str "value",
      ? !!str "not a simple key"
      : !!str "any value"
```

```
   }
}
```

The generated output of block scalar headers is shown below –

```
{
   "simple key": {
      "not a simple key": "any value",
      "also simple": "value"
   }
}
```

Document Marker Scalar Content

All characters in this example are considered as content, including the inner space characters.

```
%YAML 1.1
---
!!map {
   ? !!str "---"
   : !!str "foo",
   ? !!str "...",
   : !!str "bar"
}

%YAML 1.1
---
!!seq [
   !!str "---",
   !!str "...",
   !!map {
      ? !!str "---"
      : !!str "..."
   }
]
```

The plain line breaks are represented with the example given below –

```
%YAML 1.1
---
!!str "as space \
trimmed\n\
specific\L\n\
none"
```

The corresponding JSON output for the same is mentioned below –

```
"as space trimmed\nspecific\u2028\nnone"
```

YAML - Flow Styles

Flow styles in YAML can be thought of as a natural extension of JSON to cover the folding content lines for better readable feature which uses anchors and aliases to create the

object instances. In this chapter, we will focus on flow representation of the following concepts –

- Alias Nodes
- Empty Nodes
- Flow Scalar styles
- Flow collection styles
- Flow nodes

The example of alias nodes is shown below –

```
%YAML 1.2
---
!!map {
   ? !!str "First occurrence"
   : &A !!str "Foo",
   ? !!str "Override anchor"
   : &B !!str "Bar",
   ? !!str "Second occurrence"
   : *A,
   ? !!str "Reuse anchor"
   : *B,
}
```

The JSON output of the code given above is given below –

```
{
   "First occurrence": "Foo",
   "Second occurrence": "Foo",
   "Override anchor": "Bar",
   "Reuse anchor": "Bar"
}
```

Nodes with empty content are considered as empty nodes. The following example shows this –

```
%YAML 1.2
---
!!map {
   ? !!str "foo" : !!str "",
   ? !!str "" : !!str "bar",
}
```

The output of empty nodes in JSON is represented as below –

```
{
   "": "bar",
   "foo": ""
}
```

Flow scalar styles include double-quoted, single-quoted and plain types. The basic example for the same is given below −

```
%YAML 1.2
---
!!map {
   ? !!str "implicit block key"
   : !!seq [
      !!map {
         ? !!str "implicit flow key"
         : !!str "value",
      }
   ]
}
```

The output in JSON format for the example given above is shown below −

```
{
   "implicit block key": [
      {
         "implicit flow key": "value"
      }
   ]
}
```

Flow collection in YAML is nested with a block collection within another flow collection. Flow collection entries are terminated with **comma** (**,**) indicator. The following example explains the flow collection block in detail −

```
%YAML 1.2
---
!!seq [
   !!seq [
      !!str "one",
      !!str "two",
   ],

   !!seq [
      !!str "three",
      !!str "four",
   ],
]
```

The output for flow collection in JSON is shown below −

```
[
   [
      "one",
      "two"
   ],
   [
      "three",
```

```
      "four"
   ]
]
```

Flow styles like JSON include start and end indicators. The only flow style that does not have any property is the plain scalar.

```
%YAML 1.2
---
!!seq [
!!seq [ !!str "a", !!str "b" ],
!!map { ? !!str "a" : !!str "b" },
!!str "a",
!!str "b",
!!str "c",]
```

The output for the code shown above in JSON format is given below –

```
[
   [
      "a",
      "b"
   ],

   {
      "a": "b"
   },

   "a",
   "b",
   "c"
]
```

YAML - Block Styles

YAML includes two block scalar styles: **literal** and **folded**. Block scalars are controlled with few indicators with a header preceding the content itself. An example of block scalar headers is given below –

```
%YAML 1.2
---
!!seq [
   !!str "literal\n",
   !!str "·folded\n",
   !!str "keep\n\n",
   !!str "·strip",
]
```

The output in JSON format with a default behavior is given below –

```
[
   "literal\n",
   "\u00b7folded\n",
   "keep\n\n",
   "\u00b7strip"
]
```

Types of Block Styles

There are four types of block styles: **literal, folded, keep** and **strip** styles. These block styles are defined with the help of Block Chomping scenario. An example of block chomping scenario is given below –

```
%YAML 1.2
---
!!map {
   ? !!str "strip"
   : !!str "# text",
   ? !!str "clip"
   : !!str "# text\n",
   ? !!str "keep"
   : !!str "# text\n",
}
```

You can see the output generated with three formats in JSON as given below –

```
{
   "strip": "# text",
   "clip": "# text\n",
   "keep": "# text\n"
}
```

Chomping in YAML controls the final breaks and trailing empty lines which are interpreted in various forms.

Stripping

In this case, the final line break and empty lines are excluded for scalar content. It is specified by the chomping indicator "-".

Clipping

Clipping is considered as a default behavior if no explicit chomping indicator is specified. The final break character is preserved in the scalar's content. The best example of clipping is demonstrated in the example above. It terminates with newline **"\n"** character.

Keeping

Keeping refers to the addition with representation of "+" chomping indicator. Additional lines created are not subject to folding. The additional lines are not subject to folding.

YAML - Sequence Styles

To understand sequence styles, it is important to understand collections. The concept of collections and sequence styles work in parallel. The collection in YAML is represented with proper sequence styles. If you want to refer proper sequencing of tags, always refer to collections. Collections in YAML are indexed by sequential integers starting with zero as represented in arrays. The focus of sequence styles begins with collections.

Example

Let us consider the number of planets in universe as a sequence which can be created as a collection. The following code shows how to represent the sequence styles of planets in universe –

```
# Ordered sequence of nodes in YAML STRUCTURE
Block style: !!seq
- Mercury   # Rotates - no light/dark sides.
- Venus     # Deadliest. Aptly named.
- Earth     # Mostly dirt.
- Mars      # Seems empty.
- Jupiter   # The king.
- Saturn    # Pretty.
- Uranus    # Where the sun hardly shines.
- Neptune   # Boring. No rings.
- Pluto     # You call this a planet?
Flow style: !!seq [ Mercury, Venus, Earth, Mars,      # Rocks
                    Jupiter, Saturn, Uranus, Neptune, # Gas
                    Pluto ]                           # Overrated
```

Then, you can see the following output for ordered sequence in JSON format –

```
{
   "Flow style": [
      "Mercury",
      "Venus",
      "Earth",
      "Mars",
      "Jupiter",
      "Saturn",
      "Uranus",
      "Neptune",
      "Pluto"
   ],

   "Block style": [
```

```
      "Mercury",
      "Venus",
      "Earth",
      "Mars",
      "Jupiter",
      "Saturn",
      "Uranus",
      "Neptune",
      "Pluto"
   ]
}
```

YAML - Flow Mappings

Flow mappings in YAML represent the unordered collection of key value pairs. They are also called as mapping node. Note that keys should be maintained unique. If there is a duplication of keys in flow mapping structure, it will generate an error. The key order is generated in serialization tree.

Example

An example of flow mapping structure is shown below –

```
%YAML 1.1
paper:
   uuid: 8a8cbf60-e067-11e3-8b68-0800200c9a66
   name: On formally undecidable propositions of  Principia Mathematica and related systems I.
   author: Kurt Gödel.
tags:
   - tag:
      uuid: 98fb0d90-e067-11e3-8b68-0800200c9a66
      name: Mathematics
   - tag:
      uuid: 3f25f680-e068-11e3-8b68-0800200c9a66
      name: Logic
```

The output of mapped sequence (unordered list) in JSON format is as shown below –

```
{
   "paper": {
      "uuid": "8a8cbf60-e067-11e3-8b68-0800200c9a66",
      "name": "On formally undecidable propositions of Principia Mathematica and related systems I.",
      "author": "Kurt Gödel."
   },
   "tags": [
      {
         "tag": {
            "uuid": "98fb0d90-e067-11e3-8b68-0800200c9a66",
```

```
            "name": "Mathematics"
         }
      },
      {
         "tag": {
            "uuid": "3f25f680-e068-11e3-8b68-0800200c9a66",
            "name": "Logic"
         }
      }
   ]
}
```

If you observe this output as shown above, it is observed that the key names are maintained unique in YAML mapping structure.

YAML - Block Sequences

The block sequences of YAML represent a series of nodes. Each item is denoted by a leading "-" indicator. Note that the "-" indicator in YAML should be separated from the node with a white space.

The basic representation of block sequence is given below –

```
block sequence:
··- one↓
  - two : three↓
```

Example

Observe the following examples for a better understanding of block sequences.

Example 1

```
port: &ports
  adapter:  postgres
  host:     localhost

development:
  database: myapp_development
  <<: *ports
```

The output of block sequences in JSON format is given below –

```
{
   "port": {
      "adapter": "postgres",
      "host": "localhost"
   },
   "development": {
```

```
      "database": "myapp_development",
      "adapter": "postgres",
      "host": "localhost"
   }
}
```

YAML - Failsafe Schema

A YAML schema is defined as a combination of set of tags and includes a mechanism for resolving non-specific tags. The failsafe schema in YAML is created in such a manner that it can be used with any YAML document. It is also considered as a recommended schema for a generic YAML document.

Types

There are two types of failsafe schema: **Generic Mapping** and **Generic Sequence**

Generic Mapping

It represents an associative container. Here, each key is unique in the association and mapped to exactly one value. YAML includes no restrictions for key definitions.

An example for representing generic mapping is given below –

```
Clark : Evans
Ingy : döt Net
Oren : Ben-Kiki
Flow style: !!map { Clark: Evans, Ingy: döt Net, Oren: Ben-Kiki }
```

The output of generic mapping structure in JSON format is shown below –

```
{
   "Oren": "Ben-Kiki",
   "Ingy": "d\u00f6t Net",
   "Clark": "Evans",
   "Flow style": {
      "Oren": "Ben-Kiki",
      "Ingy": "d\u00f6t Net",
      "Clark": "Evans"
   }
}
```

Generic Sequence

It represents a type of sequence. It includes a collection indexed by sequential integers starting with zero. It is represented with **!!seq** tag.

```
Clark : Evans
Ingy : döt Net
```

```
Oren : Ben-Kiki
Flow style: !!seq { Clark: Evans, Ingy: döt Net, Oren: Ben-Kiki }
```

The output for this generic sequence of failsafe

```
schema is shown below:
{
   "Oren": "Ben-Kiki",
   "Ingy": "d\u00f6t Net",
   "Clark": "Evans",
   "Flow style": {
      "Oren": "Ben-Kiki",
      "Ingy": "d\u00f6t Net",
      "Clark": "Evans"
   }
}
```

YAML - JSON Schema

JSON schema in YAML is considered as the common denominator of most modern computer languages. It allows parsing JSON files. It is strongly recommended in YAML that other schemas should be considered on JSON schema. The primary reason for this is that it includes key value combination which are user friendly. The messages can be encoded as key and can be used as and when needed.

The JSON schema is scalar and lacks a value. A mapping entry in JSON schema is represented in the format of some key and value pair where null is treated as valid.

Example

A null JSON schema is represented as shown below –

```
!!null null: value for null key
key with null value: !!null null
```

The output of JSON representation is mentioned below –

```
{
   "null": "value for null key",
   "key with null value": null
}
```

Example

The following example represents the Boolean JSON schema –

```
YAML is a superset of JSON: !!bool true
Pluto is a planet: !!bool false
```

The following is the output for the same in JSON format –

```
{
   "YAML is a superset of JSON": true,
   "Pluto is a planet": false
}
```

Example

The following example represents the integer JSON schema –

```
negative: !!int -12
zero: !!int 0
positive: !!int 34
```

The output of integer generated JSON schema is shown below:

```
{
   "positive": 34,
   "zero": 0,
   "negative": -12
}
```

Example

The tags in JSON schema is represented with following example –

```
A null: null
Booleans: [ true, false ]
Integers: [ 0, -0, 3, -19 ]
Floats: [ 0., -0.0, 12e03, -2E+05 ]
Invalid: [ True, Null, 0o7, 0x3A, +12.3 ]
```

You can find the JSON Output as shown below –

```
{
   "Integers": [
      0,
      0,
      3,
      -19
   ],

   "Booleans": [
      true,
      false
   ],
   "A null": null,

   "Invalid": [
        true,
```

```
        null,
        "0o7",
        58,
        12.300000000000001
    ],

    "Floats": [
       0.0,
       -0.0,
       "12e03",
       "-2E+05"
    ]
}
```

Previous Page Next Page

Enter email for newsletter go