***YAML***

[YAML](http://yaml.org/) is an Open Source data serialization language designed for human readability and interaction with scripting languages. Use YAML to configure and define apps, dialogs, field types, message views and templates, media editor and renderers.

***Characteristics:***

* Whitespace indentation is used to denote structure; however tab characters are never allowed as indentation.
* Comments begin with the number sign (**#**), can start anywhere on a line and continue until the end of the line. Comments must be separated from other tokens by white space characters.
* Strings (scalars) are ordinarily unquoted, but may be enclosed in double-quotes ("), or single-quotes (').

*YAML Basic Building blocks:*

1. **Comments:**

So whenever you see the **#** character anywhere in the file that isn't first enclosed in '' (quotes) or "" (double quotes), it marks the beginning of a **comment**. This means all the text after it, up until the end of the line, is *completely ignored* and has no effect. You can use this to write notes on the file or temporarily disable one or more options.

These are all comments:

***#debug: false***

***#Hello there, how are you?***

***Debug: false # <- The line is read up to this character!***

This is not commented:

***name: '#change me'***

## Key-value pairs:

YAML keeps data stored as a map containing keys and values associated to those keys. This map is in no particular order, so you can reorder it at will. Each pair is in the format KEY: VALUE. For example:

***minecraft-tag: 'minecraft'***

***cancelled-tag: 'cancelled'***

Note the 'quotes' around the value. When the value is a text string, we use the quotes to make sure any special characters aren't given special meaning, and instead are all kept as part of the value. So even though they are optional, using them is highly recommended.

Other than text, numbers and true/false, the value associated to a key can also be another map of key-value pairs. To achieve this, omit the value and instead write the key-value pairs in the following lines. However, you must prefix them with at least one more SPACE character than the key.

YAML will consider that lines prefixed with more spaces than the parent key are contained inside it; Moreover, all lines must be prefixed with the same amount of spaces to belong to the same map. So this works:

formatting:

from-game:

chat: '(%sender%) %message%'

action: '\* %sender% %message%'

## Lists

There is one other data structure from YAML we need to use - The list. Lists are used to store a collection of ordered values. The values are not associated with a key, only with a positional index obtained from the order in which they are specified.

Like key-value pairs, list items are defined in the lines below the list key, all with the same amount of spaces prefixing them (at least as many spaces as the parent key). The difference is that they begin with a dash (**-**). Here's another valid list:

Example1:

***mylist:***

***- 'item 1'***

***- 'item 2'***

Example 2:

***# A list of food***

***- Sandwich***

***- Pizza***

***- Burrito***

### *Lists and Maps*

YAML files are always combinations of YAML maps and lists. This said, make sure you understand the concept of these two structures.

#### Lists

Members of a list are lines beginning at the same indentation level starting with leading hyphen (dash) and at least one space ( -  ). The number of spaces after the leading hyphen must be the same for all list members.

|  |
| --- |
| ***# A list of food***  ***- Sandwich***  ***- Pizza***  ***- Burrito***  ***- Chocolate cake*** |

#### Maps

Maps also known as dictionaries in YAML are represented in a simple key: value  form the colon must be followed by a space:

|  |
| --- |
| ***# An employee record***  ***name: John Doe***  ***job: Developer***  ***skill: Beginner*** |

#### Combination of Map and List

Let's combine maps and a list. It is a common use case in YAML files.

|  |
| --- |
| ***# An employee record***  ***name: John Doe***  ***job: Developer***  ***skill: Beginner***  ***employed: True***  ***food:***  ***- Sandwich***  ***- Pizza***  ***- Burrito***  ***- Chocolate cake***  ***drinks: [coke, beer, water, milk]*** *# drinks is another example of another notation of a list*  ***languages:***  ***groovy: Beginner***  ***java: Expert*** |

**NOTE:**

* The "root" structure of this file is a map.
* The value of the map entries with the keys food and drinks are lists
* The value of the map entry with the key languages again is a map.

YAML files are meant to configure or define "items" like apps, templates and dialogs. In a running system the data written in YAML is represented by a [Java Bean](http://en.wikipedia.org/wiki/JavaBeans).

***Working with YAML***

We have some basic libraries specially designed to deal with YAML file one of them is

**YAML Beans.**

***YAML Beans***

**Overview**

YamlBeans makes it easy to serialize and deserialize Java object graphs to and from YAML, a human-friendly data format. Replace XML and properties files with YAML for more expressive power (lists, maps, anchors, etc) and easier hand-editing.   
  
**Basic deserialization**   
  
The YamlReader class is used to deserialize YAML to Java objects. The following YAML defines a Map with four entries. The "phone numbers" entry is a List of two items, each of which is a Map.

name: Chaitanya Kattunga

age: 27

address: Madhapur

phone numbers:

- name: Home

number: 206-555-5138

- name: Work

number: 425-555-2306

The "read" method reads the next YAML document and deserializes it into HashMaps, ArrayLists, and Strings. Since we know the root object defined in the YAML of our example is a Map, we can cast the object and make use of it.

YamlReader reader = new YamlReader(new FileReader("contact.yml"));

Object object = reader.read();

System.out.println(object);

Map map = (Map)object;

System.out.println(map.get("address"));

**Multiple objects**   
  
A stream of YAML can contain more than one YAML document. Each call to YamlReader#read() deserializes the next document into an object. YAML documents are delimited by "---" (this is optional for the first document).

name: Nathan Sweet

age: 28

---

name: Jenny Alarco

age: 25

This prints the String "28" then "25":

YamlReader reader = new YamlReader(new FileReader("contact.yml"));

while (true) {

Map contact = reader.read();

if (contact == null) break;

System.out.println(contact.get("age"));

}

**Deserializing other classes**   
  
There are two ways to deserialize something other than HashMaps, ArrayLists, and Strings. Imagine this YAML document and Java class:

name: Nathan Sweet

age: 28

public class Contact {

public String name;

public int age;

}

The "read" method can be passed a class, so the YamlReader knows what it is deserializing:

YamlReader reader = new YamlReader(new FileReader("contact.yml"));

Contact contact = reader.read(Contact.class);

System.out.println(contact.age);

The YamlReader creates an instance of the Contact class and sets the "name" and "age" fields. The YamlReader determines the "age" value in the YAML needs to be converted into a int. Deserialization would have failed if the age was not a valid int. The YamlReader can set public fields and bean setter methods.   
  
Instead of telling the YamlReader what type to deserialize, the type can alternatively be specified in the YAML using a tag:

!com.example.Contact

name: Nathan Sweet

age: 28

**Serializing objects**   
  
The YamlWriter class is used to serialize Java objects to YAML. The "write" method automatically handles this by recognizing public fields and bean getter methods.

Contact contact = new Contact();

contact.name = "Nathan Sweet";

contact.age = 28;

YamlWriter writer = new YamlWriter(new FileWriter("output.yml"));

writer.write(contact);

writer.close();

This outputs:

!com.example.Contact

name: Nathan Sweet

age: 28

The tags are automatically output as needed so that the YamlReader class will be able to reconstruct the object graph. For example, serializing this ArrayList does not output any tag for the list because YamlReader uses an ArrayList by default.

List list = new ArrayList();

list.add("moo");

list.add("cow");

- moo

- cow

If the list was a LinkedList, then YamlWriter knows that a tag is needed and outputs:

List list = new LinkedList();

list.add("moo");

list.add("cow");

!java.util.LinkedList

- moo

- cow

**Complex graphs**   
  
YamlBeans can serialize any object graph.

public class Contact {

public String name;

public int age;

public List phoneNumbers;

}

public class Phone {

public String name;

public String number;

}

friends:

- !com.example.Contact

name: Bob

age: 29

phoneNumbers:

- !com.example.Phone

name: Home

number: 206-555-1234

- !com.example.Phone

name: Work

number: 206-555-5678

- !com.example.Contact

name: Mike

age: 31

phoneNumbers:

- !com.example.Phone

number: 206-555-4321

enemies:

- !com.example.Contact

name: Bill

phoneNumbers:

- !com.example.Phone

name: Cell

number: 206-555-1234

This is a map of lists of contacts, each with a list of phone numbers. Again, the public fields could also have been bean properties.   
  
  
**Tag shortcuts**   
  
Tags can be lengthy sometimes and can clutter up the YAML. Alternate tags can be defined for a class and will be used instead of the full class name.

YamlWriter writer = new YamlWriter(new FileWriter("output.yml"));

writer.getConfig().setClassTag("contact", Contact.class);

writer.write(contact);

writer.close();

The output no longer contains the full class name for the Contact class.

!contact

name: Chaitanya kattunga

age: 27

**Lists and maps**   
  
When reading or writing a List or Map, Yaml Beans cannot know what type of objects are supposed to be in the List or Map, so it will write out a tag.

!com.example.Contact

name: Bill

phoneNumbers:

- !com.example.Phone

number: 206-555-1234

- !com.example.Phone

number: 206-555-5678

- !com.example.Phone

number: 206-555-7654

This can make the YAML less readable. To improve this, you may define what element type should be expected for a List or Map field on your object.

YamlWriter writer = new YamlWriter(new FileWriter("output.yml"));

writer.getConfig().setPropertyElementType(Contact.class, "phoneNumbers", Phone.class);

writer.write(contact);

writer.close();

Now YamlBeans knows what to expect for elements of the "phoneNumbers" field, so extra tags will not be output.

!com.example.Contact

name: Bill

phoneNumbers:

- number: 206-555-1234

- number: 206-555-5678

- number: 206-555-7654

Setting the element type for a Map field tells YamlBeans what to expect for values in the Map. Keys in a Map are always Strings.   
  
**Anchors**   
  
When an object graph contains multiple references to the same object, an anchor may be used so that the object only needs to be defined once in the YAML.

oldest friend:

&1 !contact

name: Bob

age: 29

best friend: \*1

In this map, the "oldest friend" and "best friend" keys reference the same object. The YamlReader automatically handles the anchors in the YAML when rebuilding the object graph. By default, the YamlWriter automatically outputs anchors when writing objects.

Contact contact = new Contact();

contact.name = "Bob";

contact.age = 29;

Map map = new HashMap();

map.put("oldest friend", contact);

map.put("best friend", contact);

**Swagger - YAML Integration**

As of now, we have seen the what are the basic building blocks involved in yaml and how to work with yaml files with YANL Beans library, but we are supposed to define API specification in yaml language in order to work with swagger API(Open API Specification).

***What is Swagger?***

Swagger is a specification for documenting REST API. It specifies the format (URL, method, and representation) to describe REST web services. Swagger is meant to enable the service producer to update the service documentation in real time so that client and documentation systems are moving at the same pace as the server. The methods, parameters, and models description are tightly integrated into the server code, thereby maintaining the synchronization in APIs and its documentation.

The goal of Swagger is to define a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined via Swagger, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interfaces have done for lower-level programming, Swagger removes the guesswork in calling the service.

***Advantages***

* With the Swagger framework, the server, client and documentation team can be in synchronization simultaneously.
* As Swagger is a language-agnostic specification, with its declarative resource specification, clients can easily understand and consume services without any prior knowledge of server implementation or access to the server code.
* The Swagger UI framework allows both implementers and users to interact with the API. It gives clear insight into how the API responds to parameters and options.
* Swagger responses are in JSON and XML, with additional formats in progress.
* Swagger implementations are available for various technologies like Scala, Java, and HTML5.
* Client generators are currently available for Scala, Java, JavaScript, Ruby, PHP, and ActionScript 3, with more client support underway.

## *How do I get started?*

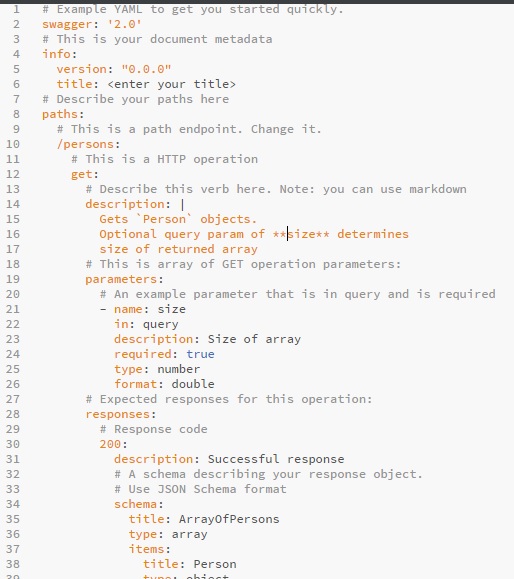
If you're an API provider and want to use Swagger to describe your APIs - there are several approaches available:

* ***A top-down approach*** where you would use the [*Swagger Editor*](http://editor.swagger.io/) to create your Swagger definition and then use the integrated [Swagger Codegen](http://swagger.io/getting-started/swagger-codegen) tools to generate server implementation.

***A bottom-up approach*** where you have an existing REST API for which you want to create a Swagger definition. Either you create the definition manually (using the same Swagger Editor mentioned above),

Here YAML Comes into picture right now ,we are supposed to provide the yaml file to Swagger Editor ,and it showcase us relevant REST API documentation in Swagger UI.

Please have a look at below Sample yaml template for designing any REST API .



It is the Standard template anything we can define with the file .It looks like **key-value** pair we just need to change the values in the above file which is in the black in color(Right side), Don’t make any changes in the left side(key).Care should be taken care while giving spaces.

**Example:**



**NOTE:** Observations for this yaml file, I have just used the same sample template to define my own Rest API for Persons .

**title** : “Provided my own API name”

**paths :** defined own request Mapping URI ,here in my case (“/person”) .

**parameters:** We can define following properties under this section.

1.we can define the nature of the Request Mapping .(“Get”)

2. Required parameters with (“name”) tag

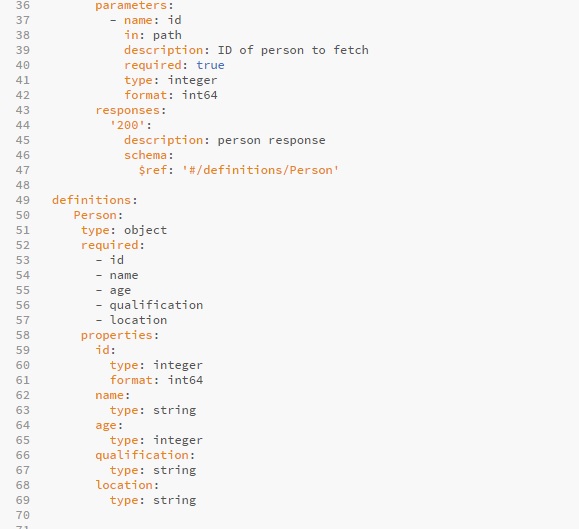
3.Description of the Request Mapping Service with (“description”) tag

4. Number of records need to fetch with (“minimum” ,maximum”, “default”).

**responses:**

This section will defines response message along with response type.

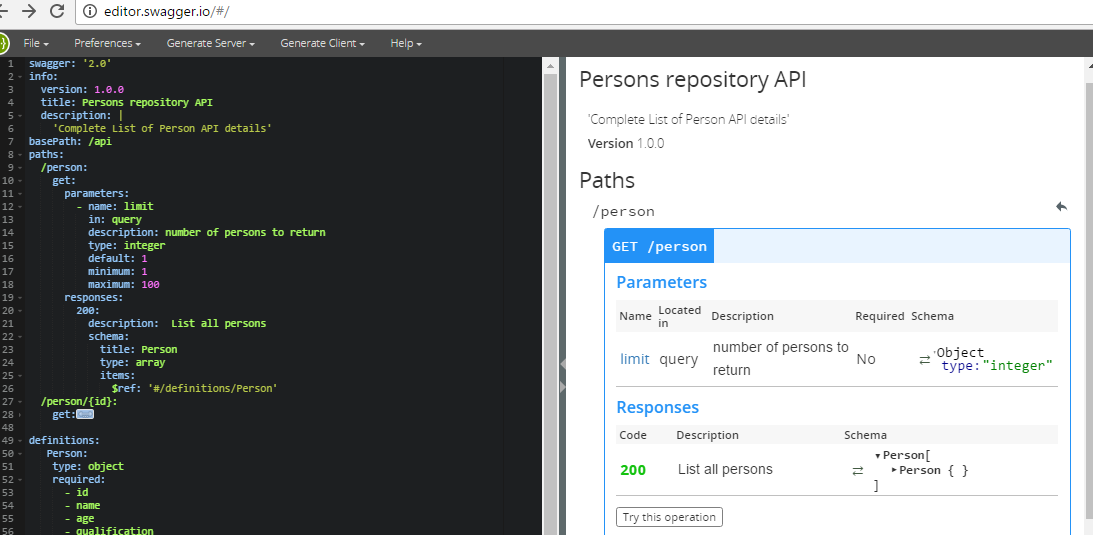
**Image Continuation…**



**definitions:**

This section defines the model object and its attributes with type.

Once we give this above file to swagger editor ,it will provide us the documentation.



Resources:

<http://editor.swagger.io/#/>

<https://github.com/EsotericSoftware/yamlbeans>