YAML/JSON Validator Tool Project

Table of Contents

[Project Objectives 2](#_Toc137562705)

[Project Progress 3](#_Toc137562706)

[Learning Points 4](#_Toc137562707)

[Project Constraints 5](#_Toc137562708)

[Proposed Actions Going Forward 6](#_Toc137562709)

# Project Objectives

This project is part of the effort to incorporate the provision of the OpenAPI Specification (OAS) validation and standardization into CPF’s current API processes that require support from ITPMs.

As a summary of the analysis of the different API formats, OAS is chosen for this project due to its simple, yet extensible structure with many modern features that other formats do not support such as updated parameter types, improved examples, references, and links, and callbacks among many others.

Benefits of incorporating the OAS standard include increasing the value of CPF’s APIs by designing APIs strategically and being more reusable. This project aims to refactor existing and new APIs in a streamlined manner to meet industry standards and onboard APIs securely onto the API Exchange Platform (AXP)

# Project Components

Two tools in the AXP Dev Portal website are part of this project:

1. YAML/JSON OAS3.0 Validation Tool and
2. YAML/JSON API Preview Tool.

The validation tool will take the developer’s YAML/JSON document as input, validate it based on the [OAS 3.0.3 specification](https://swagger.io/specification/v3/), and output any structural errors, if any. Additional information on the errors is also displayed to give the developer advice on how to correct the errors.

A screen shot of a computer program

Description automatically generated with medium confidence

*YAML Document Example*

The preview tool will allow anyone – be it the development team or end consumers – to visualize and interact with the API’s resources without having any of the implementation logic in place. It is essentially a concise overview of a website server with examples of website responses if an end user were to use it. This preview page is automatically generated from the YAML/JSON document provided using Swagger UI plugin.

A screenshot of a computer

Description automatically generated with medium confidence

*API Preview Example*

These two tools will be presented together on a website.

It is important to note that these two tools exist and are maintained presently in the open-source community, with the main inspiration being [Swagger UI](https://swagger.io/). As a result, there is a need for these in-house developed tools to be different to bring value. Two main additional features will be added on top of the validation tool:

1. Custom Validation Rules
2. Spelling Checking

The custom validation rules check the following:

* No special characters in the API path
* Path subtier length must conform to CPF’s API requirements
* Path version must match the info version
* Subtier must be in camel case format
* Subtier spelling mistakes
* First word of subtier must be a verb
* Properties that are stated to be required must be present

The spelling checking feature will check for any spelling mistakes in parts of the YAML/JSON document that have heavy sentencing. This will ignore parts that are intended to have improper English jargon and focus on parts that should be spelling-checked.

# Project Progression

This section will go into detail on how the tools are developed and the concerns encountered along the way.

## Version 0.1

My initial thought on this project is simply that it is already done by Swagger UI, which is a company/community large enough to maintain this sort of application with very high standards. There will be no need to create another similar one. After being given the existing codebase and one template example of a YAML file, and quick research on Swagger UI, there are two obstacles on why we cannot just use the already-developed tool by Swagger UI:

1. We need to pay to add custom validation rules on Swagger UI’s tool
2. There is no in-built spelling-checking feature. We will have to use Grammarly, which is a spelling-checking application, as a browser plugin or desktop application. This cannot be expected of the user by default.

Therefore, we will need to code the application from scratch. While both tools are open-source on Github, which means we can essentially copy their code, it does not make sense to do so as we will have no information on whether custom validation rules can be added or not. It will be clearer if we code it ourselves, which will make the application easier to maintain and customize in the future.

Note that these concerns are only on the validation tool, not the preview tool so we will just be using the preview plugin from Swagger UI, which is a part of the Swagger UI preview tool. This will reduce the time taken to program our own API preview GUI and I think it looks great already. Therefore, most of the time will be spent on developing the validation tool.

Version 0.1 of the validation tool is designed to use Flask to route two websites for the two tools respectively. Flask is a web framework written in Python and it is a lightweight framework as it does not require many dependencies, which makes it very fast in performance. It will serve as our back-end server, serving our routes and creating the validation logic of our YAML/JSON document.

A screen shot of a computer program

Description automatically generated with low confidence

*Flask routing*

The websites themselves are the front-end and will be using simple HTML, CSS, and client-side JavaScript to reduce the dependencies needed for faster performance. JQuery is the library used in JavaScript to bring the back-end and front-end together. It is used to call the routes created in Flask to validate the YAML/JSON document provided by the front-end. It will then respond with either a “Success” message if there are no errors, or a list of the errors, if any.

To begin coding the validation logic, one needs to understand the structure of YAML and JSON. YAML and JSON are both information formats for data to be sent across websites and servers. Without these formats, information sent from one website to another would not be standardized so one website might not understand the information sent by another. YAML is the more human-understandable format as it has proper indentation and proper syntax. JSON is the more machine-understandable format as it is in the format of a dictionary, which is a data structure. In theory and practice, this makes validating JSON much easier than YAML because one just needs to go through the dictionary without needing to parse YAML.

Therefore, Version 0.1 chose to validate using JSON which first converts YAML to JSON before implementing the validation logic. Each field and its corresponding value of the JSON document can then be extracted easily, as shown in the example. For example, if the field “openapi” appears in the document, its value will be doc\_json[“openapi”].

A screen shot of a computer code

Description automatically generated with low confidence

*Extracting fields from JSON*

The simple structure of the example YAML/JSON document provided makes this method viable because there won’t be many fields to extract. The naïve approach to implementing the validation logic is simply using functions. This method will be known as the “functional method” for convenience. The functions will take each of the extracted values as input and validate them. As long as the field exists and is extracted, this opens up many possibilities for how each of the values can be validated, such as checking if it is present or not, whether it is in a certain string format or not, etc. This makes adding custom validation rules very easy.

Another attractive reason to use this functional method is that Python has a lot of well-developed and documented libraries for almost every functionality that exists (arguably even more modern libraries than JavaScript, even though JavaScript is the most used language). For example, NLTK can be used for natural language processing and there are many spelling-checking libraries. This would mean that our server will have a lot of dependencies to process, which impacts performance the most. The more complex features we need to implement, the lower the performance.

A picture containing text, screenshot

Description automatically generated

*Functional methods to validate fields*

This functional method makes unit testing possible and easy because the inputs to each of these functions can be made up and tested individually. Unit testing is done using another Python library called PyTest.

A screen shot of a computer program

Description automatically generated with low confidence

*Unit testing example*

Once we have tested the validation logic, we can integrate our back-end with the front-end and perform integration testing. Integration testing directly tests the interaction between the end user and the whole application. This will cover everything that the end user can click, input, or scroll on the website and how the website will respond to those actions. Integration is done using another Python library called Selenium.

A screen shot of a computer code

Description automatically generated with low confidence

*Integration testing example*

Once we have finished the first iteration of the Software Development Lifecycle (SDLC) of the application, we can then deploy our application onto a URL that anyone can access and use because just running the application on a local machine is not scalable. This is done by hosting our application on a cloud server, which is running 24/7. But before we can put our application onto a cloud server, we have to make sure our application can run on every machine. This has been a tricky problem for many years because different operating systems have different names for the dependencies we are using. If we simply move the application files from a Windows machine to a MacOS machine, some dependencies might not be identified, hence causing the application to run improperly. This is solved by [Docker](https://docs.docker.com/get-started/overview/), a Software-As-A-Service to package and OS-level virtualize software applications from its infrastructure so that applications can run on any machine. Applications packaged by Docker are called containers and once they run, are called images, which is essentially a snapshot of the application that is running correctly.

Azure is mainly used in CPF so we will be using that as our cloud service. While we can just deploy our application container onto an Azure container registry and create an application service with a public URL, this is not scalable because what if we change some code in our application? We will need to redeploy everything again as the updated code will not be on Azure. Azure Pipeline is a DevOps tool in Azure used to automate the Software Development Lifecycle (SLDC) as well as deploy applications among many other features. It is capable of tracking changes in your code and performing the whole process of testing and deploying without manually doing so.

A picture containing text, screenshot, line, font

Description automatically generated

*4-step pipeline*

This concludes Version 0.1 of the project (unfortunately I don’t have screenshots of this version). Concerns with this version:

1. Back-end performance worsens with more possible fields to extract. This is inevitable with the functional method because each extraction will need to iterate more of the JSON document. This will cause slow responses (lag) from the server when validating.
2. Not fully implementing OAS 3.0, which is one of the core components of the project. We want to add on top of OAS 3.0, not compromising its base specifications for the custom validation rules. Many of the complex features of OAS 3.0 such as references and links cannot be implemented easily using the functional method as it involves recursion for each pair of fields to be extracted. This will worsen the back-end performance exponentially.
3. Error messages are not beneficial enough to the developer. By transforming the YAML document into JSON, we lose beneficial information such as line number of the error or the parent field that caused the error.

The validation logic of the YAML/JSON document is the challenging part of this project depending on how extensive and exhaustive the validation needs to be. For many small applications that are just validating a small YAML/JSON document that does not follow OAS3.0, the functional method is still viable. However, for this project, we need to change the validation approach.

## Version 1.0

To avoid iterating the JSON document multiple times and recursion between the pairs of fields, we have to validate the YAML/JSON document without

# Learning Points

# Project Constraints

# Proposed Actions Going Forward