YAML/JSON Validator Tool Project

Table of Contents

[Project Objectives 2](#_Toc140141740)

[Project Components 3](#_Toc140141741)

[Project Progression 5](#_Toc140141742)

[Version 0.1 5](#_Toc140141743)

[Version 1.0 10](#_Toc140141744)

[Project User Guide 16](#_Toc140141745)

[Understanding File Structure 16](#_Toc140141746)

[Prerequisites 17](#_Toc140141747)

[Running the Application 18](#_Toc140141748)

[Testing 18](#_Toc140141749)

[Project Roadmap 19](#_Toc140141750)

# 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 with 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
* Identify 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. Concepts such as debouncing and web sockets are used to make the validation process smoother by allowing the validation request to happen once there is a change in the input, removing the need to click a button.

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, with proper indentation and 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. This 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 expression 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 programming language in the world). For example, NLTK can be used for natural language processing and there are many spelling-checking libraries. However, this would also 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*

In terms of testing, 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 testing each and every important function in your program. 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 end-to-end testing. End-to-end 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. This 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 more 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 rules that are part of OAS3.0 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 the 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. This is thought to be the case initially. However, as the requirements require the implementation of the entire OAS3.0, 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 in its YAML form. This means that if the developer inputs YAML, we will validate YAML, and if the developer inputs JSON, we will convert it into YAML first before validating.

This started my research into how YAML can be validated and the result ended up with what I call the “schema method”. This schema method uses a template to crosscheck line-by-line with the YAML/JSON document to ensure that every field and value is what is expected (In the example below, “title” is the field, and the value must be a string). You can think of a schema as a blueprint for your YAML/JSON document. Additional rules can be added to restrict values such as data types, required values, etc. If any of the rules are violated, or there is an unexpected field, it is an error.

A screen shot of a computer program

Description automatically generated with medium confidence

*Schema for <Info> in OAS3.0*

A screenshot of a computer

Description automatically generated with low confidence

*<info> section in example YAML document*

It might be obvious that the best solution is to use what Swagger UI uses because we know it works. However, Swagger UI is using JavaScript libraries, not Python libraries. Remember that we are using Python to implement our validation logic. This would mean we would have to use NodeJS to implement our validation logic, which is a JavaScript runtime environment, which can be thought of as the JavaScript version of Flask but with more functionalities. Not wanting to add more technologies to this project, I explored the limits of Python to see if we can do the same thing without adding in JavaScript.

There are 2 well-documented and maintained libraries in Python that can help us with this schema method. These libraries allow you to create schemas of a certain format and the library will help you crosscheck and then output errors, if any:

1. [PyKwalify](https://pykwalify.readthedocs.io/en/unstable/)
2. [Cerberus](https://docs.python-cerberus.org/en/stable/)

It is not clear in the beginning which library is better than the other, so both are used to see which would fit OAS3.0 better. Note that OAS3.0 has some advanced features that are part of its specification. These features ultimately were the deciding factors on whether to continue using the library or not:

* References and links. This means the schema must support recursion. Recursion here means that it is possible that a field can have infinite other fields with a fixed schema. If the library does not support schema with references and links, we will have to hardcode the repeated fields until a limited depth, which is not what OAS3.0 wants
* “OR” logic. This is very similar to “IF-ELSE” statements where if a field is present, another field should not.
* Regular expressions. This applies especially to the fields, not so much to the values. Regular expressions are a specific format for how the fields should be named. If regular expressions are supported, it will greatly reduce the number of fields hardcoded in the schema
* Custom validation rules. If the schema does not allow custom validation rules, it cannot be used as it would bring no value to the project

After two weeks spent exploring the two libraries, the following results are determined:

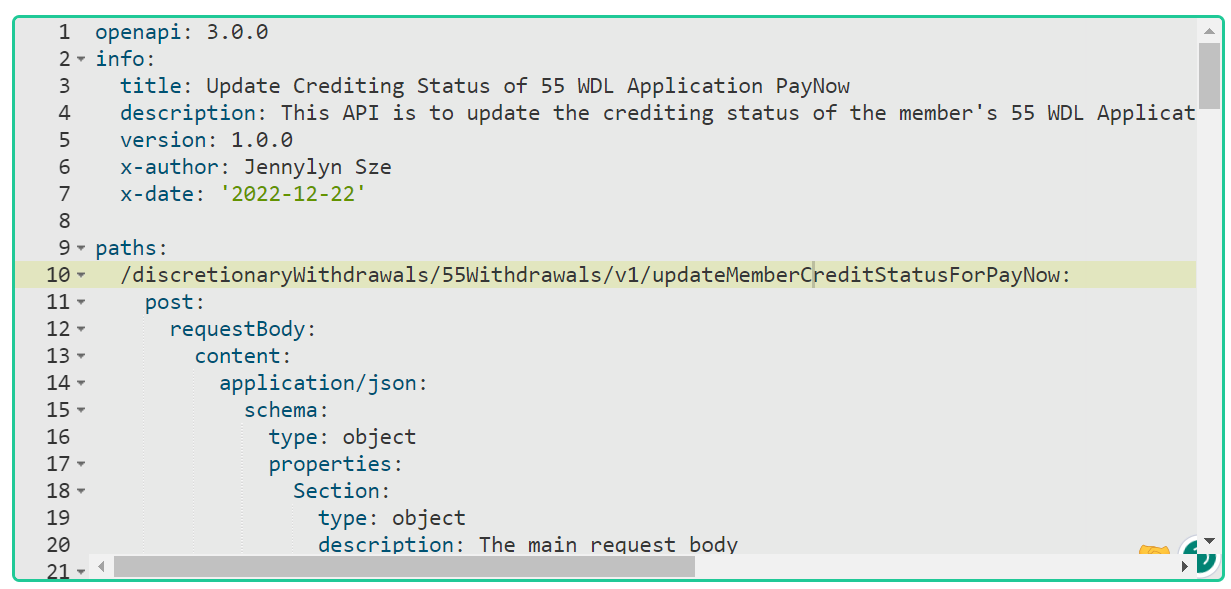
|  |  |  |
| --- | --- | --- |
|  | PyKwalify | Cerberus |
| References and links | Not possible | Not possible |
| “OR” logic | Possible | Possible |
| Regular expressions | Possible | Not possible |
| Custom validation rules | Not possible | Possible |

The conclusion is that both libraries are not fully compatible with OAS3.0, and thus are only good options for simpler YAML/JSON documents. PyKwalify cannot be used entirely because custom validation rules are very important to the project and its error reporting is not as useful as Cerberus. Cerberus, on the other hand, can be used but we need a lot more time to implement the whole OAS3.0 schema. These are the main reasons why:

* While Cerberus cannot support references and links directly, in the CPF context, we can arbitrarily say that our APIs will not have repeated fields up to 3 times. For example, in the example, YAML document given, fields are only repeated up to 2 times maximum.
* Regular expressions are inevitable, and can only be solved by taking more time to add fields specific to the CPF context.

At this point in time, the amount of time and effort needed to implement this schema is taken into consideration because it might not be possible to finish by the end of the internship. The schema, at the point of being given up, is at 70 thousand lines long without including the custom validation rules. This is highly unmaintainable and hard to modify if there are any schema changes. Therefore, there is a need to shift away from Python and go into JavaScript to implement our schema that satisfies all the features required.

Note that this does not mean nothing was achieved in the 2 weeks of exploring PyKwalify and Cerberus. These two libraries exposed the Python library called ruamel.yaml, which allows us to map each line of the YAML/JSON document to line numbers, and inspired the use of Ace Editor, which is a code editor that displays line numbers and with highlighting features that we can put on our website. This adds to the usability and appearance of the text input, which ultimately aids in providing value to the project.



*Ace Editor*

Implementing the schema in JavaScript is trivial, all we need to do is to see what library Swagger UI is using, and it is called [Another JSON Schema Validator (Ajv)](https://ajv.js.org/). Ajv is arguably not as well documented as Cerberus, but it is used by a lot more people in the community. Ajv can implement all the features we need and has very useful error messages. The schema was then finished in 1 week. However, to add Ajv as our validation method requires some technology changes in our back-end.

Before talking about the technology changes, we need to talk about the main differences between using Flask versus NodeJS as back-end servers. Both of them are generally the same in terms of functionalities, but it is the libraries that make a difference:

1. Flask uses pip while NodeJS uses npm or yarn, all of which are package managers to install, update, and remove dependencies and libraries. Pip is not compatible with npm or yarn, so if we use both Flask and NodeJS, there will be two package managers to handle. This adds to maintainability.
2. NodeJS is faster than Flask in terms of compiling
3. NodeJS is unable to do Routing while Flask does this easily
4. In general, JavaScript libraries are harder to maintain than Python libraries and are not as diverse as Python libraries, at least for those that apply to this project. This is due to a concept called “wheels” in pip, and it is not present in npm or yarn. This means that Python libraries can be updated with no conflicts easily while JavaScript libraries need to be monitored and versioning needs to be changed manually.

Our back-end will be using both Flask and NodeJS with separate responsibilities. Flask will handle the page routing and the line number mapping feature. This is because it is harder to do in JavaScript and performs slower than if it is implemented in Python. NodeJS will handle our validation logic and will contain most of the libraries used. The front-end will be served by Flask, but will only communicate with the NodeJS server. The NodeJS server will only call the line number mapping feature from the Flask server, and respond to the front-end validation requests.

A picture containing text, screenshot, diagram, font

Description automatically generated

*Project Architecture v1.0*

Now that we have refactored the validation logic to satisfy the project requirements and included more error reporting features, the last step is to deploy the application on Azure again. For version 0.1, since we only have one Flask server, we only needed one Dockerfile to deploy the application. A Dockerfile is simply a series of settings and steps to run the application on a virtual machine. Usually, we need one Dockerfile for one service. Services can be servers, databases, proxies, etc. As seen from the architecture diagram above, we have two servers so we need 2 Dockerfiles. This makes deploying on Azure more complicated because we need to somehow dockerize the two servers while still having the two servers communicate with each other. Locally, this is done by deploying both servers on localhost, which means they can communicate with each other on one IP address. On the cloud, there are two ways to do this (or the only two ways I learnt):

* Use Docker Compose to run two Dockerfiles on one virtual machine. Docker Compose is a plugin for Docker that allows multiple Dockerfiles to run together
* Deploy the two servers separately using two virtual machines

Since it makes more sense to run two Dockerfiles on one virtual machine, just like how we can run the whole application locally on our machines, I tried method 1 first. However, I was met with the main problem that the Flask Server cannot communicate with the Node Server. I have tried all configurations on Dockerfiles and docker-compose files, search countless GitHub repositories for examples, go on forums to search for answers, and read documentation, but to no avail. I also asked some developers in CPF, namely Akmal and Richard, but the result was the same. The conclusion reached is that Docker Compose does not support Flask and NodeJS servers to communicate with each other due to the lack of documentation for this, or that there is some configuration that I am not doing correctly. Furthermore, it appears that when the NodeJS server is hosted on the same website as the Flask server, the GSIB blocks most of the client-side JavaScript. Either way, it is easier to fall back on the second method and there are benefits for the second method. Docker Compose is still used to run the application locally because of its convenience, explained more in the Project User Guide in the next section.

The second method is simply dockerizing the Flask and NodeJS servers separately and deploying them on two websites. The benefits of this method include higher scaling for each of the servers and they are easier to manage. To facilitate this second method, we will need our NodeJS server to be able to serve websites, also known as Routing. Remember that we mentioned one of the differences between Flask and NodeJS is that NodeJS is unable to do Routing. That is why we will use an additional JavaScript library called ExpressJS, which adds Flask-like functionalities to NodeJS. Ultimately, the application is successfully deployed on Azure.

This schema method of validating YAML/JSON is not a typical method to validate fields and values because we are not the ones coding the function to validate whether the fields and values satisfy the rules applied to them, it is the library that does it. Therefore, we only need to test the custom validation rules that we created. This greatly reduces the amount of unit testing we need to perform. Also, recall that our validation logic is on JavaScript now, not Python. So, we have to change from PyTest to Jest, a JavaScript unit testing library. Further testing on currently approved AXP endpoint requests is also added to increase test coverage on actual data.

A screen shot of a computer code

Description automatically generated

*Unit tests on custom validation rules – checkPathCharacters*

Integration testing usually refers to how different parts of the code are applied together to obtain a combined result. In this project, this refers to how the validation part and the line number part are combined together to return a meaningful error message for the user. To ensure that this integration test passes, the line number needs to be output. However, due to the sheer number of steps to reach the line number output, I decided to combine integration testing and end-to-end testing, which will obtain all expected results in the end, including the line number. This would mean if the end-to-end testing passes, the integration tests should have passed. Note that this is not the usual way to perform integration testing, I just took this approach because it is a small application and there is only one function to test.

End-to-end testing in this version is performed exactly the same way as in version 0.1.

This concludes the programming progression update of project version 1.0. This is by no means a perfect release of the project, nor is it meant for production. To the best of my knowledge and ability, having performed only two iterations of the SDLC, I feel there are further improvements to the project, shown in the last section of this document.

# Project User Guide

This section will serve as a complete and detailed guide on how to navigate the project source files and how to deploy the project locally and on the cloud. For people who have experience with the technologies used and would like to get the application running ASAP, refer to the README.md file from <https://github.com/DanielxDante/Project-First-Integritas>

## Understanding File Structure

.

|--- .gitignore

|--- LICENSE.txt

|--- Makefile # Convenient local deployment without using Docker

|--- README.md

|--- azure-pipelines.yml # Azure Pipelines yaml file

|--- docker-compose.yml # Local deployment using Docker Compose

└--- src

|--- \_\_init\_\_.py

|--- app.py # Code for Flask and line number mapping and response

|--- requirements.txt # Python dependencies

|--- Dockerfile # Dockerfile to build Flask server

|--- e2e\_test

| |--- \_\_init\_\_.py

| |--- chromedriver

| └--- e2e\_test.py # Integration and End-to-End testing selenium code

|--- examples

| |--- example1.py # YAML document with no errors

| |--- example2.py # YAML document with validation errors

| └--- example3.py # YAML document with syntax errors

|--- node

| |--- package.json # JavaScript dependencies

| |--- validator.js # Code for Express and validation logic

| |--- oas3.0\_schema.yaml # OAS 3.0 schema for Ajv

| |--- Dockerfile # Dockerfile to build ExpressJS server

| |--- typojs\_dictionaries/en\_GB-ise # UK dictionary (downloaded externally)

| └--- unit\_test

| |--- jest.config.js # Jest configuration file

| |--- unit\_test.js # Rewritten validation logic for unit testing

| |--- unit\_test.test.js # Jest test cases

| └--- uris.txt # Approved URIs from AXP Sharepoint

|--- static

| |--- css

| | |--- bootstrap.min.css # Bootstrap CSS

| | |--- framework.min.css

| | |--- highlight.min.css # Highlight CSS

| | |--- site.css # General CSS for all templates

| | └--- swagger-ui.css # Swagger CSS

| |--- dictionary

| | |--- customDictionary.csv # Custom dictionary in csv format

| | |--- customDictionary.txt # Custom dictionary in txt format

| | |--- customDictionary.xlsx # Custom dictionary in xlsx format

| | └--- dictionary.txt # Dictionary saved from project handover

| |--- images

| | |--- Ajax-loader.gif

| | |--- Error.PNG

| | |--- Home.PNG

| | |--- Success.PNG

| | |--- Logo.png

| | └--- cpfLogo.png

| └--- js

| | |--- bootstrap.bundle.min.js.download # Bootstrap JS

| | |--- jquery.min.js.download # Jquery JS

| | |--- js-yaml-3.13.1.min.js.download # JSYAML

| | |--- kong.utils.js.download

| | |--- swagger-ui-bundle.js # Part of Swagger Preview

| | |--- swagger-ui-standalone-preset.js # Part of Swagger Preview

| | └--- script.js # Code for browser interactions and Fetch

└--- templates

|--- 404.html # Catchall page

└--- API Exchange Developer Portal.html # Home page

## Prerequisites

Make sure you have the following prerequisites installed on your machine:

* Docker V2: [Installation Guide](https://docs.docker.com/engine/install/)
* Docker Compose (needed only if Docker V1 is already installed on your machine): [Installation Guide](https://docs.docker.com/compose/install/linux/)
* Git and Git Bash (needed only if you do not have the source code on your machine): [Installation Guide](https://git-scm.com/downloads)

## Running the Application

Follow these steps to run the application using Docker:

1. Clone the repository from Github to your local machine (skip if you already have the source code)

git clone https://github.com/DanielxDante/Project-First-Integritas.git

1. Navigate to the project directory

cd Project-First-Integritas

1. Build the Docker images using Docker Compose

docker-compose up -d

1. Open your web browser and visit <http://localhost/> to access the application
2. To stop the application, run:

docker-compose down

If you wish to deploy the application onto Azure and the run the application there, fork the repository from Github or create a Github repository with the source code and use the azure-pipelines.yml provided to create a CICD pipeline deploying the website. Make sure to change the appropriate variables in the azure-pipeline.yml file as commented.

It is not recommended to run the application locally without using Docker due to dependencies, especially if you are using OSes other than Windows. If you still wish to deploy the application locally without Docker, there are two ways to do so:

Follow these steps to run the application using Makefile:

Follow these steps to run the application using Bash:

## Testing

332 initial unit tests have been created in unit\_test.test.js. After testing on Docker in Azure, all 332 have passed with >90% line and branch coverage.

A screenshot of a computer

Description automatically generated

13 initial end-to-end tests have been created in e2e\_test.py. After testing on the deployed web applications using Azure, all 13 have passed.

A screen shot of a computer

Description automatically generated

# Project Roadmap

Improvements that can be made:

* Find a way implement YAML/JSON to line number mapping in JavaScript to completely remove Flask. This will reduce redundant complexity in the back-end and architecture
* Improve GUI based on your preferences
* Multi-language support