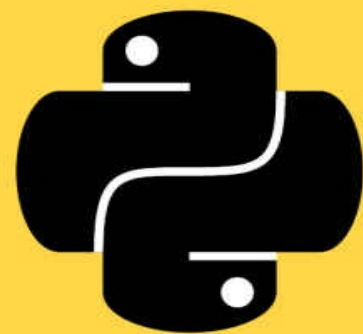


# Web API Development with Python

A Beginner's Guide using Flask and FastAPI

First Edition



 FastAPI

 Flask

REHAN HAIDER

# WEB API DEVELOPMENT WITH PYTHON

A Beginner's Guide using Flask and FastAPI

*(First Edition)*

By

*REHAN HAIDER*

Website: <https://CloudBytes.dev>

Email: [student@CloudBytes.dev](mailto:student@CloudBytes.dev)

# Copyright Notice

---

**Copyright © 2021 by Rehan Haider and CloudBytes. All rights reserved.**

*No part of this publication may be reproduced, stored, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise without written permission from the publisher. It is illegal to copy this book, post it to a website, or distribute it by any other means without permission.*

*Rehan Haider asserts the moral right to be identified as the author of this work.*

*Rehan Haider or CloudBytes has no responsibility for the persistence or accuracy of URLs for external or third-party Internet Websites referred to in this publication and does not guarantee that any content on such Websites is, or will remain, accurate or appropriate.*

*Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book and on its cover are trade names, service marks, trademarks, and registered trademarks of their respective owners.*

# Preface

---

This book is intended for beginners with basic programming knowledge, who are trying to understand advanced concepts and give them a hands-on experience of interacting with, and building an API using Python.

My method of learning is by doing and doing something over and over again till it becomes a habit. Thus, this book will follow that philosophy, talk about concepts that you need to know, but will be focused on providing hands-on examples that you follow along but not act as a complete reference.

The contents of this book are divided into three parts

1. **Introductory Concepts** : That explains terms and definitions that you will encounter
2. **Interacting with an API** : Explains how to use and interact with APIs. It's easier to build things than you one knows how to operate
3. **Building APIs** : Main focus of the book, building APIs using Python's Flask & FastAPI libraries

The topics and presentation will get progressively more difficult as you go deeper into the book and that is by design to get the readers to a stage where they are able self-learn any framework or library without requiring any help.

# Student Support

If you face any challenges or need help, you can email [student@CloudBytes.dev](mailto:student@CloudBytes.dev)

You can also create an issue on the [GitHub repository](#) of that particular exercise. In fact, this is a good practice.

## Additional Resources

Modern developers have only two loyal friends:

1. [Google](#)
2. [Stack Overflow](#)

Chances are your program will throw errors and you need to debug it. The best way to do that is start by Googling your error message and look through the links that Google suggests on Stack Overflow. Arguably over half of issues can be resolved this way. The rest requires looking through documentation, reading blog / tutorial posts, and talking to other developers.

# Table of Contents

## Table of Contents

### Prerequisites

1. Internet
2. Accounts
3. Operating System
4. Python
5. Terminal
6. Text Editor / IDE
7. Git for Version Control
8. Docker for Desktop (Optional)
9. Jupyter Notebook

### Chapter 1: Introduction to Web APIs

- 1.1 What is API?
- 1.2 Types of APIs
- 1.3 What is a Web API?
- 1.4 Getting our hands dirty
- 1.5 Getting hands dirty programmatically
- 1.6 JavaScript Object Notation (JSON)
- 1.7 Why are APIs needed?
- 1.8 API Design Patterns

### Chapter 2: Python & Working with APIs

- 2.1 Programmatically accessing an API

### Chapter 3: Building APIs with Flask

- 3.1 Initialise the development environment
- 3.2 Understanding the Starter Kit
- 3.3 Initialising the starter kit
- 3.3 A minimal Flask API
- 3.3 Explanation
- 3.5 Running the API
- 3.6 Call the API Programmatically
- 3.7 JSONIFY the response

### Chapter 4: Building interactive APIs

- 4.1 Capturing request arguments
- 4.2 Explanation

[4.3 Testing the API](#)

[4.4 Catching sneaky behaviour and errors](#)

[4.4 Handling incorrect API requests](#)

## **[Chapter 5: Multi-argument interactive API](#)**

[5.1 Capturing multiple arguments](#)

[5.2 Explanation](#)

[5.3 Testing the API](#)

[5.4 Reader Challenge](#)

## **[Chapter 6. Google search as an API](#)**

[6.1 An informal introduction to URL and Querystring](#)

[6.2 What can we do with this information?](#)

[6.3 Understanding the Starter Kit](#)

[6.4 Logic of the application](#)

[6.5 Rendering home page](#)

[6.6 Returning Search Results](#)

[6.7 Explanation](#)

[6.8 Student Challenge](#)

## **[Chapter 7: Building a Dictionary API](#)**

[7.1 Understanding the Starter Kit](#)

[7.2 Logic of the application](#)

[7.3 Handle incoming searches](#)

[7.4 Finding the definition of the word](#)

[7.5 Handling list of words](#)

[7.6 Testing the API](#)

[7.7 Student Challenge](#)

[7.8 Jupyter Notebook to test the API](#)

## **[Chapter 8: Building a POST API](#)**

[8.1 API to add Filters](#)

[8.2 Understanding the Starter Kit](#)

[8.3 Logic of the application](#)

[8.4 Implementing the Filter](#)

[8.5 Testing the API](#)

[8.6 Getting filtered image](#)

[8.7 Bonus Challenge #2](#)

## **[Chapter 9: Bonus Lesson: Deploying the API](#)**

[9.1 Configuring the CD Pipeline](#)

[9.2 Testing the API](#)

[9.3 Pro Tip: Testing using Terminal](#)

[9.4 Bonus Challenge](#)

[Chapter 10: Introducing FastAPI](#)

[10.1 Asynchronous Programming](#)

[10.2 Enter FastAPI](#)

[10.3 Understanding the Starter Kit](#)

[10.4 Saying Hello FastAPI](#)

[Chapter 11: Dictionary using FastAPI](#)

[11.1 Understanding the Starter Kit](#)

[11.2 Implementing the usage instructions](#)

[11.3 Implementing the dictionary](#)

[11.4 Testing the API](#)

[11.5 : FastAPI OpenAPI Docs and Swagger UI](#)

[11.6 Handling a list of words](#)

[11.7 Testing The API](#)

[Chapter 12: Image filters using FastAPI](#)

[12.1 Bonus: Deployment to Heroku](#)

[Chapter 13: FastAPI async / await](#)

[13.1 Making the program asynchronous](#)

[13.2 Making the filter API asynchronous](#)

[13.3 Test Scenario 1](#)

[13.4 Test Scenario 2](#)

[13.5 Test scenario 3](#)

[13.6 Test Scenario 4](#)

[13.7 Conclusion](#)

[13.8 Bonus Challenge](#)

[Chapter 14: Making a TODO API](#)

[14.1 API Specifications](#)

[14.2 Initialise the environment](#)

[14.3 Understanding the starter kit](#)

[14.4 Request Body and Data Validation](#)

[14.5 Data Model for TODO](#)

[14.6 Task manager](#)

[14.7 Building the API](#)

[14.8 Final API design](#)

[14.9 Testing the API](#)

[14.10 Where to from here?](#)

[Au Revoir](#)



[About the Author](#)  
[Acknowledgements](#)

# Prerequisites

## 1. Internet

Access to internet is needed to download resources

## 2. Accounts

You need to create the following accounts

1. **GitHub** : [..https://www.github.com](https://www.github.com)
2. **Google** : [..https://www.google.com](https://www.google.com)

## 3. Operating System

Any one of the following operating systems will do

1. *Windows 10 build 17063 and later (April 2018)*
2. *Linux (Ubuntu 18.04, etc.)*
3. *MacOS (10.12.6 or above)*

## 4. Python

Install the latest version of Python for your Operating System from. Make sure you add Python to the PATH variable by checking the highlighted boxes below.

```
https://www.python.org/downloads/
```

## 5. Terminal

For Windows:

CMD.exe is good enough for our purposes. You can open it by pressing **⊞ Windows Key + R** together and running “CMD”.

You can also use PowerShell; however, it will require you to remove the alias for curl which is mapped incorrectly to a cmdlet by running

```
Remove-Item alias:curl
```

*You will need to run the above every time you start a new PowerShell terminal.*

## For Linux / MacOS:

Most distributions of Linux & macOS come with terminals & cURL in-built.

## Common to all OS:

Make sure cURL is installed by running the below command

```
curl -V
```

*If you get any error or cURL is not installed download and install relevant version from the below link*

```
https://curl.haxx.se
```

## 6. Text Editor / IDE

You can use any Text Editor / IDE that you are comfortable with, but **VSCode** is recommended due to its tight integration with **GitHub** , **Docker** , and **WSL** .

Download the appropriate version of **VSCode** for your OS and install from:

```
https://code.visualstudio.com/download
```

In **VSCode** , go to “ *Extensions* ” search for **Python** and install the extension from **Microsoft** called **Pylance** .

Please ensure you login into the VSCode account using your **GitHub** Credentials this will enable you to use **VSCode** terminal with **GitHub** without typing in username and password repeatedly.

## 7. Git for Version Control

**Recommended for Beginners** : If you know absolutely nothing about **Git** , and are a Windows or macOS user, download the GitHub for Desktop from below.

```
https://desktop.github.com/
```

Otherwise for advanced users, download, install, and configure the command line utility.

**Downloading Git:** Check if Git is already available by running the below in the

terminal

```
git --version
```

If you see an output, you don't need to do anything. But if you get an error, proceed to the below URL, and download and install the appropriate version of Git for your operating system

```
https://git-scm.com/downloads
```

**Configure Git** : Check if Git is already available by running. Open the terminal and run the below commands using your account details

```
git config --global user.name "Your name here"  
git config --global user.email "your_email@example.com"
```

## 8. Docker for Desktop (Optional)

If you're running a CLI Linux, it's likely I don't need to explain this. For others, download the appropriate version of Docker for Desktop for your OS and install from:

```
https://www.docker.com/products/docker-desktop/
```

## 9. Jupyter Notebook

**Jupyter Notebook** is a browser based interactive **iPython** application that you can use to build your code by testing it step by step. Makes life infinitely easier.

**Jupyter** requires **Python & pip** to be installed. To install **Jupyter Notebook** , open the terminal and type the below to install Jupyter Notebook

```
pip install notebook
```

Once done just run the below command in the terminal to launch the Jupyter Notebook in your default browser.

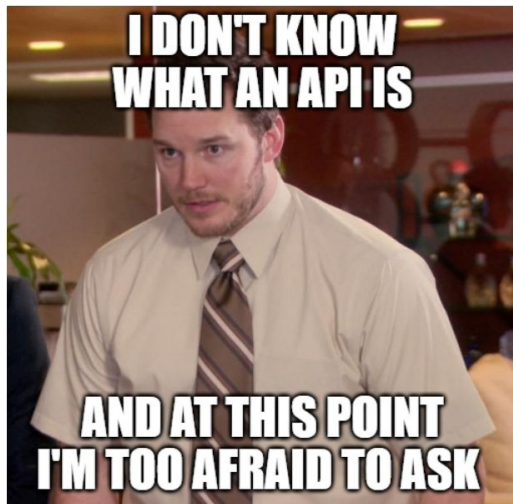
```
jupyter notebook
```

**Note: You can also use Jupyter Lab or Google Colab instead of Jupyter Notebook.**

# Chapter 1: Introduction to Web APIs

If you're a beginner even the thought of APIs is somewhat scary. What kind of dark magic is that? And why is everyone paying hundreds of thousands to API developers. Do they drink blood and sacrifice goats in a circle under moon light?

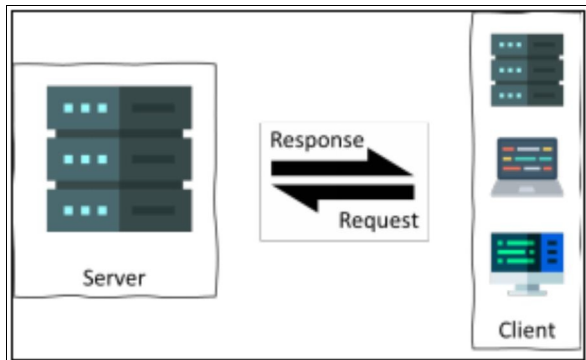
Pretty soon while the cool kids claim to be inventing calculus and you are out of the loop and feel like Andy below. If you identify with it, you're in luck. We are going to talk about APIs.



## 1.1 What is API?

**API** is an acronym for *A*pplication *P*rogramming *I*nterface, but nobody calls it just like nobody calls **USB** a **Universal Serial Bridge**. The technical meaning of **API** is that it is a set of definitions and protocols for building, communicating, and integrating application software(s), thus the term “*Interface*”.

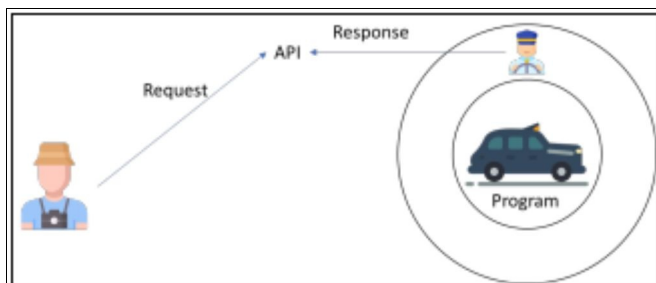
But technical jargons apart, **APIs** are just a way to interact with applications whose internal workings are not visible to external users. APIs allow the external users (clients) to “*request*” something from the application or server and get a corresponding “*response*”.



In fact, you have already used the real-world equivalent of **APIs** . For example, riding in an old-fashioned cab. To get a ride in a cab you would need to

1. “ **Request** ” a cab to pick you up, along with your contact and location details
2. In “ **response** ” to your request you get a cab details and the cab picks you up
3. You again “ **request** ” the cab driver to take you to a specific location
4. In “ **response** ” the cab driver drives you to your desired location

Now, you don’t need to know how to operate a car, to be able to get from one place to another, the cab driver acts like an **API** . You might not know anything about how a car operates but you can interact with the **API** layer, that is, the driver, and navigate to your desired outcome.



Another way to look at **APIs** is, it is a magic box that takes input in a very specific format and gives output in a very specific format. In application development this “ **standardisation** ” of input and output (I/O) is useful and makes it easy for a developer to interact with systems that they don’t control.

## 1.2 Types of APIs

If you’ve heard the term **API** , chances are it was used to refer to a very specific type of **API** called **Web API** . However, in general, the term **API** could have other meanings based on what the **API** is being used for.

Based on usage, **APIs** can be broadly divided into four categories as listed below

1. **Web APIs** are used to communicate between a server and a client over the internet. Web API as the name suggests is a very specific type of **API** used to interact and manipulate information or resources over internet.
2. **Remote APIs** define the standards of interaction for applications running on different machines. For example, **JDBC** connectivity **API** that connects a database to the program
3. **Libraries and Frameworks** acts as the interface to a software library is also a type of **API**
4. **Operating Systems** can specify **APIs** for applications to interact with the device. For example, Android devices with camera requires an **OS API** to enable control of the camera by any app

Each one these could have its own sub-categories. The scope of this book is limited to **Web APIs** .

## 1.3 What is a Web API?

**Web APIs** are used for communication between a server and a client over the internet or any network. Typically, **Web APIs** use **HTTP ( HyperText Transfer Protocol )** request methods, also known as **HTTP** verbs, to communicate with a server.

**HTTP** standards ( *RFCs 7231 & RFC 5789 but that is not important* ) specify a set of “ **request methods** ” that indicates the action that is to be performed.

These **HTTP** request methods are:

1. **GET** : “Gets” the specified resources from an endpoint. This results in a response from the server without any change in the state of the server
2. **POST** : Sends some data to an endpoint, typically resulting in an action that in turn changes the state of the server
3. **PUT** : Replaces some data on a server. Like **POST** but different in the sense that **PUT** requests will always produce the same result
4. **DELETE** : The **DELETE** method deletes the specified resource from server
5. **HEAD** : The **HEAD** method asks for a response like that of a **GET** request, but only the status line and header section
6. **CONNECT** : The **CONNECT** method establishes a tunnel to the server identified by the target resource
7. **OPTIONS** : The **OPTIONS** method is used to describe the communication options for the target resource.
8. **TRACE** : The **TRACE** method performs a message loop-back test along the path to the target resource.
9. **PATCH** : The **PATCH** method is used to apply partial modifications to a resource.



## 1.4 Getting our hands dirty

*(Please ensure the prerequisites as specified in the prerequisites chapter earlier are set up and satisfied before proceeding.)*

**CoinDesk** is a cryptocurrency news website, like **Yahoo Finance**, but for cryptocurrencies. CoinDesk offers a public and free API that provides the latest Bitcoin prices. We will interact with this **CoinDesk API** to fetch the latest Bitcoin prices using a classic utility **cURL**.

But before that, open your favourite web browser (Firefox, Chrome, Edge, etc.) and open the below website address.

```
https://api.coindesk.com/v1/bpi/currentprice.json
```

You would see some seemingly cryptic text displayed on your browser that will be like below.

```
{ "time" : { "updated" : "May 13, 2021 17:35:00 UTC" , "updatedISO" : "2021-05-13T17:35:00+00:00" ,  
"updateduk" : "May 13, 2021 at 18:35 BST" }, "disclaimer" : "This data was produced from the  
CoinDesk Bitcoin Price Index (USD). Non-USD currency data converted using hourly conversion  
rate from openexchangerates.org" , "chartName" : "Bitcoin" , "bpi" : { "USD" : { "code" : "USD" , "symbol"  
"&#36;" , "rate" : "47,190.4555" , "description" : "United States Dollar" , "rate_float" : 47190.4555 } , "GB"  
{ "code" : "GBP" , "symbol" : "&pound;" , "rate" : "33,618.8109" , "description" : "British Pound Sterling"  
"rate_float" : 33618.8109 } , "EUR" : { "code" : "EUR" , "symbol" : "&euro;" , "rate" : "39,064.2119" ,  
"description" : "Euro" , "rate_float" : 39064.2119 } } }
```

**What is this garbage?** It is a response that the **API** has sent.

When you opened the URL above using a browser, what you have actually done is

1. Used your browser to send a “**GET**” request to the endpoint
2. The API then sent a response back with two components
  - a. An **HTTP 200 OK** Status indicating the request was valid
  - b. A payload with data that is specified in the documentation
3. The browser then receives this data and renders it for you to see

So, your browser just interacted with **CoinDesk API**. But this is not useful if one wants to build an app to track Bitcoin prices because they will have to open a browser, type the **URL** and then copy the data from the browser to be able to use it and, of course,



## 1.5 Getting hands dirty programmatically

To use it in a program, we need to be able to get this data programmatically. So, let's go one level deeper and try that out.

**On windows: Press `⊞` Windows Key-> Search Terminal -> Open -> then run:**

```
curl -i -X GET "https://api.coindesk.com/v1/bpi/currentprice.json"
```

**On Linux / Mac: Open Terminal and run:**

```
curl -i -X GET "https://api.coindesk.com/v1/bpi/currentprice.json"
```

The **URLs** like the one above is also referred to as an “*endpoint*”.

**Once completed you will get a response like below:**

```
HTTP/ 1.1 200 OK
Content-Type: application/javascript
Content-Length: 679
Connection: keep-alive
Access-Control-Allow-Origin: *
Cache-Control: max-age= 15
Date: Mon, 10 May 2021 05 : 11 : 35 GMT
Expires: Mon, 10 May 2021 05 : 12 : 07 UTC
Server: nginx/ 1.18.0
X-Powered-By: Fat-Free Framework
X-Cache: Hit from cloudfront
Via: 1.1 b 83963 f 0701 c 4 af 7 f 684 fb 9 b 32 b 49 e 75 .cloudfront.net (CloudFront)
X-Amz-Cf-Pop: DEL 54 -C 3
X-Amz-Cf-Id: INdmxs 7 vfcYBcjYRBWz_-YE 8 T 2 ZJfKi 0 QeZP 4 Z 8 y 2 mP 8 bbo 62 lqitg==

{ "time" :{ "updated" : "May 10, 2021 05:11:00 UTC" , "updatedISO" : "2021-05-10T05:11:00+00:00" ,
"updateduk" : "May 10, 2021 at 06:11 BST" }, "disclaimer" : "This data was produced from the
CoinDesk Bitcoin Price Index (USD). Non-USD currency data converted using hourly conversion
rate from openexchangerates.org" , "chartName" : "Bitcoin" , "bpi" :{ "USD" :{ "code" : "USD" , "symbol"
```

```
"&#36;", "rate" : "59,387.4712" , "description" : "United States Dollar" , "rate_float" : 59387.4712 } , "GBP" : { "code" : "GBP" , "symbol" : "&pound;" , "rate" : "42,342.7919" , "description" : "British Pound Sterling" , "rate_float" : 42342.7919 } , "EUR" : { "code" : "EUR" , "symbol" : "&euro;" , "rate" : "48,872.1474" , "description" : "Euro" , "rate_float" : 48872.1474 } }
```

## 1.5.1 Explanation

**cURL** a command line utility developed almost 25 years ago, its main function is to fetch the response from a URL.

Using the flag ‘ *i* ’ with cURL includes protocol response headers in the output, and ‘ *x* ’ allows us to define the method used, in this case “ **GET** ”.

And you can see, the first part of the response above are the headers while the second part is the content. Headers are useful in understanding the behaviour of the endpoint.

The first line that says “ *HTTP/1.1 200 OK* ” is called a **response status code** . A “ *OK* ” response code means your request was valid and it was successfully responded to. If you change anything in the endpoint **URL** and try again, you will get an “ *HTTP/1.1 404 Not Found* ” error meaning your request couldn’t be processed by the server.

There are several such other response codes, most of them will be auto generated by our application. However, you can read about them here at [Mozilla](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status):

```
https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
```

The content part of the response in curly braces { } follows a standard format called **JSON** that is used to store and transmit data to and from the endpoint. In this example, the API responds by sending the price of **Bitcoin** in **USD** , **GBP** , a **EUR** along with several other important information such as timestamp, descriptions, etc.

Using this you can build a simple program that displays the price of Bitcoin to your users.

## 1.6 JavaScript Object Notation (JSON)

JSON is composed of “*key*” and “*value*” pairs separated by a colon ‘:’ in the format {“*key*”: “*value*” } , a visual demonstration below



The content in **section 1.5** above seems garbled but once *prettied* the pattern becomes quite clear and clearly follows the *key-value* pair method.

```
{
  "time" :{
    "updated" : "May 10, 2021 05:11:00 UTC" ,
    "updatedISO" : "2021-05-10T05:11:00+00:00" ,
    "updateduk" : "May 10, 2021 at 06:11 BST"
  },
  "disclaimer" : "This data was produced from the CoinDesk Bitcoin Price Index (USD). Non-USD
  currency data converted using hourly conversion rate from openexchangerates.org" ,
  "chartName" : "Bitcoin" ,
  "bpi" :{
    "USD" :{
      "code" : "USD" ,
      "symbol" : "&#36;" ,
      "rate" : "59,387.4712" ,
      "description" : "United States Dollar" ,
      "rate_float" : 59387.4712
    },
    "GBP" :{
      "code" : "GBP" ,
      "symbol" : "&pound;" ,
      "rate" : "42,342.7919" ,
      "description" : "British Pound Sterling" ,
      "rate_float" : 42342.7919
    },
    "EUR" :{
      "code" : "EUR" ,
      "symbol" : "&euro;" ,
      "rate" : "48,872.1474" ,
```

```
"description" : "Euro",  
"rate_float" : 48872.1474  
}  
}  
}
```

**JSON** is the most widespread means of transmitting data between server / client or vice versa, but it is not mandatory.

Data can also be sent as unformatted files, images and other multimedia, **XML** , or even plain text. However, **JSON** 's standardised format and lightweight nature combined with its easy interoperability with most programming languages has made it the de-facto standard for transmitting data. In fact, it is possible to add media files in **JSON** as well. **JSON** can be easily imported and used just like a **Python** dictionary.

**NOTE:** JSON format uses double inverted commas, while a Python Dictionary can be either single or double inverted commas.

## 1.7 Why are APIs needed?

There are innumerable reasons for using **APIs** , and several ways in which **APIs** are used, some of them not so good.



**Figure 1 - APIs: Helping developers connect to things they shouldn't**

Some key use cases for **APIs** are described below.

### 1.7.1 Integrating multiple applications

This scenario occurs when two applications need to communicate and exchange data. E.g. let's say a firm is using **Salesforce** as its **Customer Relationship Management (CRM)** system but is using **SAP** as its **Financial Account (FI)** sys. To present an accurate picture of growth the two systems need to talk to each other. This is typically achieved using API “adapters” that enable bi-directional data transfer.

### 1.7.2 Building cross-platform applications

You can use **Facebook** via web browser, **Android App** , and **iOS App** . **Facebook** needs to ensure that any updates from either of the 3 channels are reflected in other apps in real-time which is difficult to achieve if there are 3 separate apps with their own data.

To solve this, **Facebook** has consolidated their data in one distributed system and built their own **API** on top of that can be used by developers (both internal or external) to develop apps that can do anything from posting an update, uploading a pic or interacting with other users.

## 1.7.3 Enhancing functionality of applications

Have you logged into any **non-Google** app or website using your **Google** credentials?

This is enabled by **Google's OAuth API** that enables external apps to use **Google** authentication system to manage their user login.

Similarly, you can use **APIs** provided by **Cloud Service Providers** such as **Amazon**, **Azure**, or **GCP** to add certain individual features and build fully functional applications.

Do you need to store pictures? Use **AWS S3 API** to upload directly to the cloud.

Want to build a chat app that can automatically translate whatever you type. Use **GCP's Translation API**.

## 1.7.4 Provide external access to your application

You want to build an automated trading app that buys and sells stocks using your own magic algorithm? You need real-time and historical data to build these models. A broker or a financial data provider will provide you **API** access for you to be able to leverage and build applications.

## 1.7.5 As an app backend

**RESTful** (more on that ahead) backend designs are usually built as an **API** that can be called by the frontend. This is a foundational concept for building serverless apps.

# 1.8 API Design Patterns

We talked about **HTTP** methods in an earlier section. You can use the 9 **HTTP** methods and combine them in many ways and create systems that are either too complex to manage or are difficult to use.

To solve this problem, engineers have, over the years, developed several architectural patterns and best practices such as

## 1.8.1 RESTful APIs

**RESTful** or **REST APIs** are the most dominant architectural pattern in use today

The name comes from a doctoral dissertation by **Roy Fielding** in **2000** that introduced the term “**Re** presentational **S** tate **T** ransfer”. It uses a subset of **HTT** define 6 constraint guidelines that restrict the way a server can process and respond to a client’s requests. APIs that conform to these guidelines are called “**RESTful APIs**”.

**RESTful APIs** are typically built to be used by only 4 **HTTP** method, **GET** , **PC** , **PUT** , and **DELETE** and is the predominant architectural style for building websites and **Web APIs** .

- **GET** : Used to select or retrieve data from a server. Can be used to same limited data as well.
- **POST** : Used to send or write data to the server. Typically used to send sensitive information such as credentials, financial data, or large data sets such as files
- **PUT** : Used to update data that is already present on the server, e.g. updating database entries, replacing files, etc
- **DELETE** : Used to delete existing data from the server

**RESTful APIs** are also “**stateless**” meaning it doesn’t remember anything about your previous interaction. They behave like someone who owes you money and pretend to not remember anything about any previous talks.

Jokes aside, what this means, is when interacting with **RESTful APIs** , the client (you) have the responsibility to own and provide any historical data or context if needed.

For example, let’s say you login to **Google** website using its **oAuth2 API** , which is supposedly **RESTful** , then if you go open **Gmail** , you don’t need to login again, but **Google** doesn’t remember that you have already logged in. Instead, **Google** sets a cookie on your system that includes a “**token**” (technical term is **JavaScript Web Tokens** or **JWTs** ). When you go to **Gmail** , **Gmail** asks your browser if it already has a token, if it does it lets you pass, if not, you’re forced to login again.

Is it a good thing? Generally, but it depends. **Stateful-APIs** have their place and use cases, but in today’s mass user market the advantage of **Stateless APIs** is that because the responsibility of remembering session data is on the client or browser it makes it easier to scale.

We will mainly be focused on REST API patterns in the APIs that we build in this book.

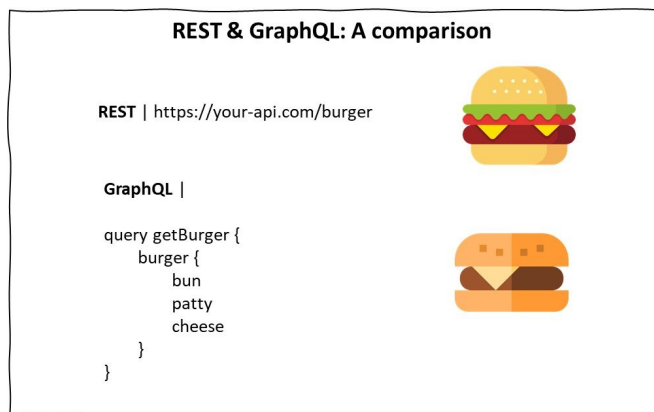
## 1.8.2 GraphQL APIs

Developed initially by **Facebook** but eventually released as an **Open Source** project. **GraphQL** allows the requestor to define the structure of data that is



required. **GraphQL** has since been touted as **REST** killer but just like all other such “killers” it hasn’t been able to replace **REST** .

**REST APIs** are quite inflexible when it comes to being able to handle the changing requirements from the clients that is accessing them. **GraphQL** is designed to solve this by allowing users to customise their request and picking the data they want in return.



The **CoinDesk** demo was an example of a **REST API** , however, if **CoinDesk** allowed a user to define exactly the data it needed, the date from which the prices are needed, and the currency, **REST API** design would have proved to be very complex. However, using **GraphQL** in these scenarios is much easier to build APIs as well as consume them.

### 1.8.3 CRUD APIs

Is typically used to build **APIs** for **Database** , it is mapped to the **REST** standard **CRUD** Stands for **C** REATE, **R** EAD, **U** PDATE, **D** ELETE. These are the action that can be performed by any database, and if you create a **REST API** that handles **DB** operations, you’re in effect created a **CRUD API** . Consequently, oft **REST & CRUD** are used interchangeably when talking about **APIs** that deals with **DBs** .

### 1.8.4 SOAP APIs

**S** imple **O** bject **A** ccess **P** rotocol or **SOAP** is one of the earliest **API** specificatic and was used to define messaging protocol specification for exchanging data between two systems.

**SOAP** has its use cases and advantages, but nobody uses it to develop **Web APIs**

nowadays. Well, nobody who is of sound mind and is not stuck with legacy systems that are built on **SOAP** .

The obsolescence of **SOAP APIs** is not due to its deficiencies but more to do with developer preferences where **REST** with its **JSON** based communication was easier to implement and interact.

# Chapter 2: Python & Working with APIs

Interacting with **APIs** using **cURL** and other utilities are fun, but in building applications and most other use cases they are too cumbersome.

In such cases, modern programming languages such as **Python** , **Go** , **Rust** , **C#** , **Ruby** , etc are used.

And before I get any hate, there are developers who can build full utilities using **PowerShell** or **Bash** , kudos to them, but we lesser mortals should stick to mortal languages.

## 2.1 Programmatically accessing an API

In the last chapter, we used a command line utility **cURL** to fetch **API** responses. Now let's try to do that with **Python** . We are going to use **Jupyter Notebook** to that inside of a program.

Open a **Jupyter Notebook** (or **Google Colab** ) and create a new Notebook.

**Run the following in Jupyter notebook**

```
#Jupyter Notebook/ In[1]
import requests

URL = "https://api.coindesk.com/v1/bpi/currentprice.json"

response = requests.get(URL)
data = json.loads(response.content.decode( "utf-8" ))
data
```

This snippet does the following

1. Imports **Python's** requests
2. Stores the **URL** of the endpoint in a string variable called “ **URL** ”
3. Sends a **GET** request to the **URL**
4. extract the response formatted as **JSON**
5. And then prints out the response data as below

```
{'time': {'updated': 'May 29 , 2021 14: 54 : 00 UTC',
'updatedISO': ' 2021-05-29 T14: 54 : 00 + 00 : 00 ',
'updateduk': 'May 29 , 2021 at 15: 54 BST'},
'disclaimer': 'This data was produced from the CoinDesk Bitcoin Price Index (USD). Non-USD
```

```
currency data converted using hourly conversion rate from openexchangerates.org',
'chartName': 'Bitcoin',
'bpi': {'USD': {'code': 'USD',
'symbol': '&# 36;',
'rate': ' 34 ,698.0350',
'description': 'United States Dollar',
'rate_float': 34698.035 },
'GBP': {'code': 'GBP',
'symbol': '&pound;',
'rate': ' 24 ,453.3361',
'description': 'British Pound Sterling',
'rate_float': 24453.3361 },
'EUR': {'code': 'EUR',
'symbol': '&euro;',
'rate': ' 28 ,458.3915',
'description': 'Euro',
'rate_float': 28458.3915 }}}}
```

This is exactly what we got in the last chapter.

**Brilliant! We have not got the JSON data from CoinDesk API programmatically using Python.**

## 2.2.1 Extracting the price of Bitcoin

What if we wanted to extract only the latest price of **Bitcoin** ?

As you can see from the previous section, the **Bitcoin** price is under `'bpi' -> 'USD' -> 'rate_float'` . To get this data, run the below in **Jupyter Notebook**

```
#Jupyter Notebook/Input#2
bitcoin_price = data[ "bpi" ][ "USD" ][ "rate_float" ]
bitcoin_price
```

This will throw out the latest **Bitcoin** price in **USD** , which at the time of writing this was “ **29561.038** ”.

**Congratulations! Now you can get the price of Bitcoin anytime you want.**

## 2.2.2 Bonus Challenge #1

Guess the date this section was written on based on the price of Bitcoin and send the answer to [student@CloudBytes.dev](mailto:student@CloudBytes.dev) with the subject as "Building Web APIs with Python: Bonus Challenge #1".

The first three correct responses will get \$50 Amazon gift vouchers.

# Chapter 3: Building APIs with Flask

**Flask** is a micro web framework written in **Python** . It is called a microframework because in the framework itself is no-frills added and comes without any feature bundled in.

More advanced web frameworks such as **Django** has almost everything needed to build a fully functioning website such as DB handling, user, and session management, etc.

**Flask** on the other hand, has almost nothing but supports extensions that can be used to bring in these features. Microframeworks are preferred by many because it is lightweight but still extensible to add any feature with ready to use plugins.

In subsequent chapters we will use Flask and its plugin to build a simple API that will respond with " *Hello, World!* "

## 3.1 Initialise the development environment

To process first, we need to initialise our development environment. Please follow the below instruction step by step

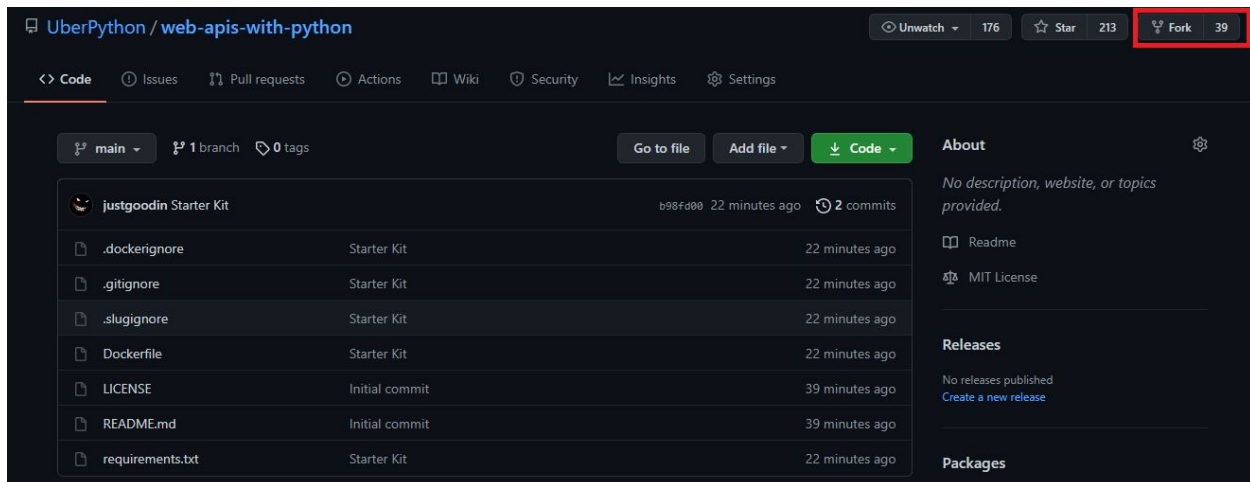
### 3.1.1 Fork the starter kit GitHub repository

This will create a copy of the repository in your own **GitHub** account that you can update.

**Step 1** : Login to [GitHub](#) and navigate to [the starter kit repository](#)

```
https://github.com/CloudBytesDotDev/web-apis-with-python
```

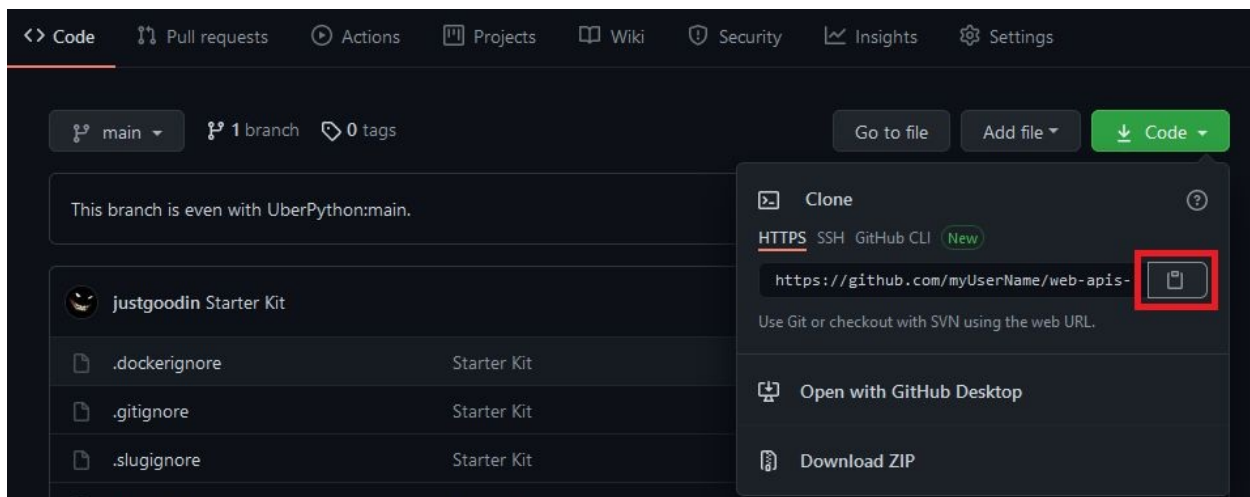
**Step 2** : Create a Fork of the repository by clicking on the fork button on top right side of the webpage as shown below



### 3.1.1 Clone the repository

Open your terminal, navigate to the folder where you want to save the git repository, e.g. I typically keep them in “ *C:\Users\MyName\Documents\GitHub* ”.

Clone this new repository in your account. To copy the **Git URL** press on the green “ **Code** ” button and then click on the clipboard icon as shown below



Then run the following command from your terminal with Git installed, replacing “ *<myUserName>* ” with your actual **GitHub** username

```
git clone https://github.com/<myUserName>/web-apis-with-python.git
```

To learn basics of how to use **Git & GitHub** , do a quick read of the **README** available in the starter kit on the **GitHub** , or use the below link.

```
https://github.com/CloudBytesDotDev/web-apis-with-python/blob/main/README.md
```

## 3.1.2 Creating a Python Virtual Environment

It is recommended to create a different virtual environment for each repository. You can do so by:

### a) Open the terminal and navigate to the repository folder

E.g. if you cloned the above example in “`C:\Users\MyName\Documents\GitHub\hello-api-python-flask`” , then navigate to that folder by running

```
cd C:\Users\MyName\Documents\GitHub\hello-api-python-flask
```

### b) Create a virtual environment by running the following command

*On Windows :*

```
python -m venv env
```

*On Linux/macOS*

```
python3 -m venv env
```

“ **env** ” is the name of the virtual environment you have created.

## 3.1.3 Activate the virtual environment

Run the below at the prompt from the folder where you created the virtual environment

```
./env/Scripts/activate
```

Once successfully activated the terminal prompt changes to

```
(env) PS C:\Users\myName\Documents\GitHub\project>
```

Notice the (env) in the beginning that highlights that the env virtual environment is active.

## 3.1.4 Install Python Dependencies

The **Python** libraries that are required for this example are listed in “ `requirements.t` ”, one of the files that were cloned from **GitHub** .

To install these libraries, in the terminal run

```
pip3 install -r requirements.txt
```

**Note** : You can also use docker files available in the repository to run the environment in a docker container directly using VSCODE. This avoids the hassle of creating and managing virtual environments.

**Congratulations! The development environment for this section is configured.**

You can use the same instructions for other exercises as well by changing the URL of the GitHub repository

### 3.1.5 Deactivating the virtual environment

You should create a separate virtual environment for each project in the project folder, to do so, you will need to first deactivate, to do so, just type and run

```
Deactivate
```

## 3.2 Understanding the Starter Kit

Before we begin, all the code that will be used in this book is provided in the book, so you don't really need to do anything with **Git** , but the intent of the starter kit is to give you a feel of how is it done in real world scenarios.

**Repository Structure** : The repository contains several branches that contain the starter kit for each problem that we are going to solve. They are listed as below

```
git
├── p1-hello-api-flask
├── s1-hello-api-flask
├── p2-byog-flask
├── s2-byog-flask
├── p3-dictionary-api-flask
├── s3-part1-dictionary-api-flask
├── s3-part2-dictionary-api-flask
├── s3-bonus-dictionary-api-flask
├── p4-image-filter-flask
└── s4-image-filter-flask
```

“ *p 1* ” as the prefix denotes the problem-1, and “ *s 1* ” denotes solution to the problem 1 that has been provided as a reference.

Each problem has one or more solution scenarios as you can see from the above

## 3.3 Initialising the starter kit

Run the below command to open the first problem starter

---



```
git checkout p1-hello-api-flask
```

This will automatically fetch the files under “ *p1-hello-api-flask* ” branch which contains

```
├── requirements.txt
└── app.py
```

*(Ignore the other files that you may see as they are not important at the moment )*

**a) requirements.txt** : contains the list of **Python** libraries that are required to be installed for this exercise. We did this in the previous section.

**b) app.py** : Is a starter template which will contain our **Flask** application and associated logic. The contents of the file are below .

### app.py

```
from flask import Flask, jsonify, request

# Initialise the app
app = Flask( __name__ )

# Define what the app does
@app . get ( "/greet" )
def index () :
    """
    TODO :
    1. Capture first name & last name
    2. If either is not provided: respond with an error
    3. If first name is not provided and second name is provided:
       respond with "Hello Mr. <second-name>!"
    4. If first name is provided but second name is not provided:
       respond with "Hello, <first-name>!"
    5. If both names are provided: respond with a question,
       "Is your name <first-name> <second-name>"
    """
    return jsonify ( "TODO" )
```

## 3.3 A minimal Flask API

Our minimal API will simply say " *Hello, World !* " when it receives a get request. Update the *app.py* to the below.

## app.py

```
from flask import Flask, jsonify, request

# Initialise the app
app = Flask(__name__)

# Define what the app does
@app.get("/greet")
def index():
    """
    TODO :
    1. Capture first name & last name
    2. If either is not provided: respond with an error
    3. If first name is not provided and second name is provided:
       respond with "Hello Mr. <second-name>!"
    4. If first name is provided but second name is not provided:
       respond with "Hello, <first-name>!"
    5. If both names are provided: respond with a question,
       "Is your name <first-name> <second-name>"
    """
    return "Hello, World!"
```

## 3.3 Explanation

In the first section we import the headers required

```
from flask import Flask, jsonify, request
```

*Flask* : is the base class for **Flask** app that we will use in this example

Ignore *jsonify* & *request* for now, it will be explained in subsequent section

```
# Initialise the app
app = Flask(__name__)
```

As the comment says, the **Flask** app is initialised using this code

```
@app.get("/greet")
```

This defines the **URL** endpoint where we need to send the **API** request to. The "@" sign before it is a **Python** syntax called decorator. It is used to apply properties of one method to another. In this case, the properties of `app.route()` is applied to the function defined in the next line, i.e. `index()`.

```
@app.get("/greet")
def index():
    return "Hello, World!"
```

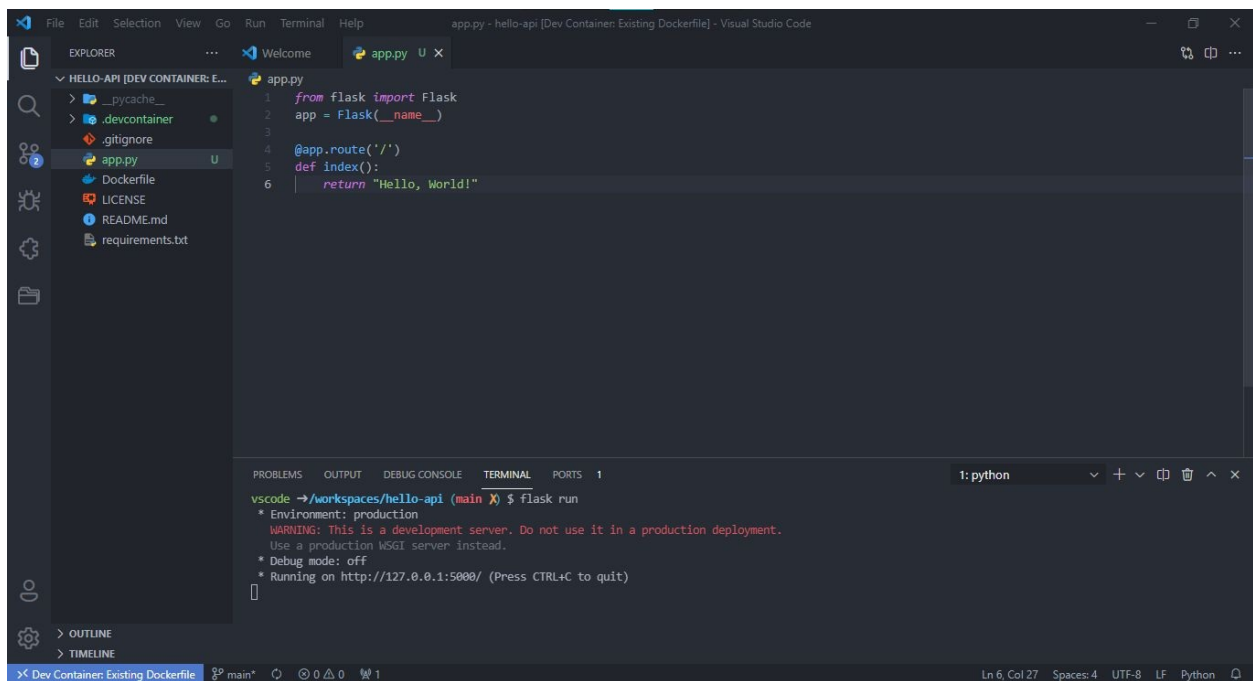
In this snippet we defined the `index()` method that will respond to any **GET** request at the `"/greet "` endpoints.

## 3.5 Running the API

To run the **API**, open the **VSCODE** Terminal at and run the following command

```
flask run
```

This should produce an output as shown below

The image shows a screenshot of the Visual Studio Code editor. The Explorer panel on the left shows a project named 'HELLO-API [DEV CONTAINER: Existing Dockerfile]'. The main editor area displays the code for 'app.py', which includes the Flask import, app creation, and the @app.route decorator for the index function. The Terminal panel at the bottom shows the command 'flask run' being executed, with output indicating the environment is production, debug mode is off, and the server is running on http://127.0.0.1:5000/.

```
File Edit Selection View Go Run Terminal Help app.py - hello-api [Dev Container: Existing Dockerfile] - Visual Studio Code
EXPLORER
HELLO-API [DEV CONTAINER: Existing Dockerfile]
  __pycache__
  .devcontainer
  .gitignore
  app.py
  Dockerfile
  LICENSE
  README.md
  requirements.txt
  app.py
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route('/')
5 def index():
6     return "Hello, World!"
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 1
vscode -> /workspaces/hello-api (main) $ flask run
* Environment: production
* WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Note the output at the bottom that states " **Running on** [\\_http://127.0.0.1/5000\\_](http://127.0.0.1/5000)". The **Flas** app has started and is serving at the address mentioned on the last line of the terminal output, "<http://127.0.0.1:5000>".

Now open your browser and enter the below address and press Enter

```
http:// 127.0.0.1 : 500 /greet
```

---

You should see " *Hello, World !* "

### 3.5.1 What just happened?

A brief run-down is, we asked Flask to

1. Listen to the route ' */greet* ' using `@app.get("/greet")` which represents the home for domain *U* this case ' <https://127.0.0.1:5000/greet> '.
2. Define and tell the listener at ' */greet* ' how to respond using `def index()`
3. Instructions for the program to follow if a request is received at " */greet* ", i.e. `return "Hello, World!"`

So when you opened the address in the browser, the browser sent a **GET** request to ' <https://127.0.0.1:5000/greet> ' which was captured by the listener at ' / ', then the listener looked for the instructions under `@app.route('/')` and found the function `index()` . It then executed the instruction in `index()` which was to return the phrase " *Hello World !* ". That response is what you see rendered in your browser.

## 3.6 Call the API Programmatically

Let's open the terminal and run

```
curl -i -X GET "http://127.0.0.1:5000/greet"
```

The response from the **API** will be like the below

```
HTTP/ 1.0 200 OK
Content-Type: text/html; charset=utf -8
Content-Length: 13
Server: Werkzeug/ 2.0.1 Python/ 3.9.4
Date: Sat, 22 May 2021 11 : 32 : 46 GMT

Hello, World!
```

**Congratulations! You now have made your first API.**

**NOTE:** The Flask App will not automatically update if you change the code. To do that you have to shut down by pressing "CTRL+C" to quit first and then restart it by running "flask run" again.

## 3.7 JSONIFY the response

If you noticed, we stated earlier that **JSON** is the de-facto standard form of communication over the web. But this response displayed on the browser, does not look like a **JSON** .

How do we convert it into a **JSON** and send the response back?

We need to restructure our program by

1. *Importing jsonify plugin from flask*
2. *Adding descriptions and*
3. *Creating a response format*

by changing **app.py** program to the following

### app.py

```
#hello-api-python-flask/app.py
from flask import Flask, jsonify, request

# Initialise the app
app = Flask(__name__)

# Define what the app does
@app.get("/greet")
def index():
    """
    TODO :
    1. Capture first name & last name
    2. If either is not provided: respond with an error
    3. If first name is not provided and second name is provided:
       respond with "Hello Mr. <second-name>!"
    4. If first name is provided but second name is not provided:
       respond with "Hello, <first-name>!"
    5. If both names are provided: respond with a question,
       "Is your name <fist-name> <second-name>
    """
    response = { "data" : "Hello, World!" }
    return jsonify(response)
```

Note we have added “ *jsonify* ” module import and change the response to a Python dictionary. We then convert this dictionary to a JSON response using *jsonify()* .

Then close the running instance of flask by pressing “ *Ctrl + C* ” in the **VSCode** terminal where the flask is running. Then restart the app by running “ *flask run* ”.

Now, if you go back to your **Windows / macOS / Linux** terminal, and rerun the following command

```
curl -i -X GET "http://127.0.0.1:5000/"
```

---

Now you see a **JSONified** response as described below:

```
HTTP/ 1.0 200 OK
Content-Type: application/json
Content-Length: 25
Server: Werkzeug/ 2.0.1 Python/ 3.9.4
Date: Sat, 22 May 2021 11 : 46 : 39 GMT

{ "data" : "Hello, World!" }
```

# Chapter 4: Building interactive APIs

Now let's use what we learnt in the last chapter and build on top of it to make it interactive.

What we want to do is, we will send a get request with a name (Jason) as a parameter, and the API will respond back with “ *Hello, <name> !* ”.

## 4.1 Capturing request arguments

To do this, let's change our `app.py` to the below:

### app.py

```
#hello-api-python-flask/app.py
from flask import Flask, jsonify, request

# Initialise the app
app = Flask(__name__)

# Define what the app does
@app.get("/greet")
def index():
    """
    TODO :
    1. Capture first name & last name
    2. If either is not provided: respond with an error
    3. If first name is not provided and second name is provided:
       respond with "Hello Mr. <second-name>!"
    4. If first name is provided but second name is not provided:
       respond with "Hello, <first-name>!"
    5. If both names are provided: respond with a question,
       "Is your name <first-name> <second-name>
    """
    name = request.args.get( "name" )
    response = { "data" : f"Hello, {name} !" }
    return jsonify(response)
```

## 4.2 Explanation

We have added a new command to our program

---

```
name = request.args.get("name")
```

This uses **Flask's** plugin request to capture the arguments that are passed as part of the querystring.

## 4.3 Testing the API

Now restart the **Flask** app, and run the below command from terminal

```
curl -X GET "http://127.0.0.1:5000/greet?name=Jason"
```

You should get the below

```
{ "data" : "Hello, Jason!" }
```

As you can see,

1. We dropped “*-i*” which is used to ask for headers in response, thus we did not get headers.
2. The `request.args.get()` method parsed the GET request and isolated the name and the value of the argument from the querystring “*greet?name=Jason*”
3. Then we created a response template to include the name
4. And finally, JSONified our response back

## 4.4 Catching sneaky behaviour and errors

What happens if you send incorrect arguments? Let's try this, instead of “*name*” we change the argument to “*fname*”.

```
curl -X GET "http://127.0.0.1:5000/greet?fname=Jason"
```

You should get the response

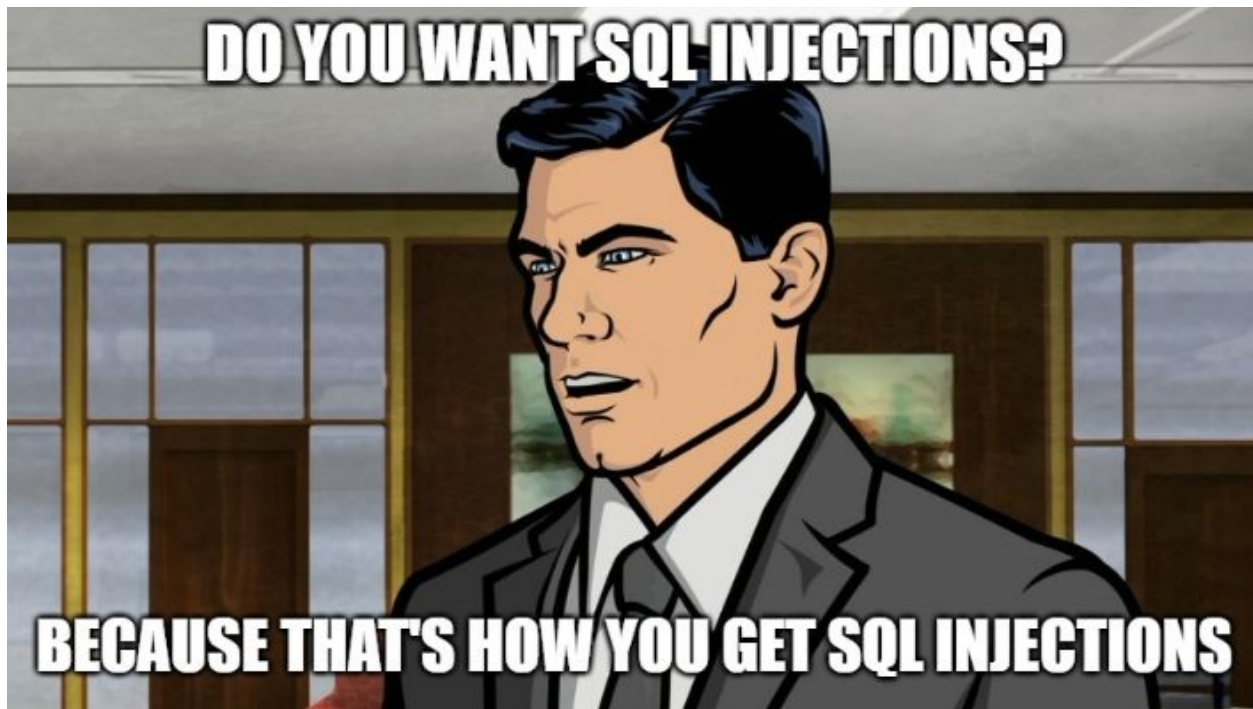
```
{ "data" : "Hello, None!" }
```

This looks harmless, but just looking at this particular response an experienced programmer will guess your code, i.e.

1. You are capturing only “*name*” and not any other parameter
2. You are constructing a response by using fstrings

### 4.4.1 Why is this bad?





## 4.4 Handling incorrect API requests

We want to ensure a proper response is provided only if the **API** request has the correct parameters, otherwise an error will be shown.

To do this, we need to add conditional logic to ensure “name” argument is being passed if not we just send an error response. Change the *app.py* to the below

### app.py

```
#hello-api-python-flask/app.py
from flask import Flask, jsonify, request

# Initialise the app
app = Flask(__name__)

# Define what the app does
@app.get("/greet")
def index():
    """
    TODO :
    1. Capture first name & last name
    2. If either is not provided: respond with an error
    3. If first name is not provided and second name is provided:
       respond with "Hello Mr. <second-name>!"
```

4. If first name is provided but second name is not provided:  
respond with "Hello, <first-name>!"

5. If both names are provided: respond with a question,  
"Is your name <first-name> <second-name>

"""

```
name = request.args.get( "name" )
if not name:
    return jsonify({ "status" : "error" })

response = { "data" : f"Hello, {name} !" }
return jsonify(response)
```

We added another code block in this version that does two things

1. *Validates if a name argument has been provided as part of the querystring*
2. *If not, then it returns an error*

Now let's try to test the **API** again

```
curl -X GET "http://127.0.0.1:5000/greet?fname=Jason"
```

As expected, you will get an error message as below:

```
{ "status" : "error" }
```

Brilliant! You are now ensuring people will not randomly try different keywords and attempts to send incorrect requests are dealt with appropriately.

Our error message is not very descriptive and that is intentional. Depending upon the consumers of the **API** the error messages can range from prescriptive to generic as in above. It depends on what the **API** is.

E.g. for someone trying to authenticate to a bank, the bank wouldn't want to give too many details about why the authentication failed. However, if I was a stockbroker developing APIs for my users, I'd want to tell them why their order was rejected.

# Chapter 5: Multi-argument interactive API

What if we wanted the program to greet us with either our first name or our last name or both, depending upon what we send in the querystring?

Consider the scenario where if Jason call the **API** with his

1. *First name only, the API responds with “ Hello, Jason ! ”, indicating familiarity*
2. *Last name only, the API responds with a respectful “ Hello, Mr. Statham ! ”*
3. *Both first and last name, the API responds with an annoyed “ Is your name Jason Statham ? ”*

## 5.1 Capturing multiple arguments

Let’s try to introduce that into our `app.py` by updating it to the below. It is as simple as adding another `request.args.get()` and searching for the name of the arguments.

### app.py

```
#hello-api-python-flask/app.py
from flask import Flask, jsonify, request

# Initialise the app
app = Flask(__name__)

# Define what the app does
@app.get("/greet")
def index():
    """
    TODO :
    1. Capture first name & last name
    2. If either is not provided: respond with an error
    3. If first name is not provided and second name is provided:
       respond with "Hello Mr. <second-name>!"
    4. If first name is provided but second name is not provided:
       respond with "Hello, <first-name>!"
    5. If both names are provided: respond with a question,
       "Is your name <fist-name> <second-name>
    """
```

```

fname = request.args.get( "fname" )
lname = request.args.get( "lname" )

if not fname and not lname:
    # If both first name and last name are missing, return an error
    return jsonify( { "status" : "error" } )
elif fname and not lname:
    # If first name is present but last name is missing
    response = { "data" : f"Hello, {fname} !" }
elif not fname and lname:
    # If first name is missing but last name is present
    response = { "data" : f"Hello, Mr. {lname} !" }
else :
    # if none of the above is true, then both names must be present
    response = { "data" : f"Is your name {fname} {lname} ?" }

return jsonify(response)

```

## 5.2 Explanation

We first captured two arguments from the get request, fname and lname, representing first name and last name using this snippet

```

fname = request.args.get( "fname" )
lname = request.args.get( "lname" )

```

Then we implemented our workflow using the following conditional logic

1. *If either is not provided: respond with an error*
2. *If the first name is not provided and the second name is provided: respond with " **Hello Mr <second-name> !** "*
3. *If the first name is provided by the second name is not provided: respond with " **Hello, <first-name> !** "*
4. *If both names are provided: respond with a question, " **Is your name <first-name> <second-name> ?** "*

```

if not fname and not lname:
    # If both first name and last name are missing, return an
    return jsonify( { "status" : "error" } )
elif fname and not lname:
    # If first name is present but last name is missing
    response = { "data" : f"Hello, {fname} !" }

```

```
elif not fname and lname:
    # If first name is missing but last name is present
    response = { "data" : f"Hello, Mr. {lname} !" }
else :
    # if none of the above is true, then both names must be present
    response = { "data" : f"Is your name {fname} {lname} ?" }
```

Finally, we  *jsonify*  our response and return it to the requestor.

## 5.3 Testing the API

Now restart the flask app and try all the three scenarios.

### a. Try with first name only

```
curl -i -X GET "http://127.0.0.1:5000/greet?fname=Jason"
```

### b. Try with last name only

```
curl -i -X GET "http://127.0.0.1:5000/greet?lname=Statham"
```

### c. Try with both first name and last name

```
curl -i -X GET "http://127.0.0.1:5000/greet?fname=Jason&lname=Statham"
```

**Awesome! Now you have created an interactive multi-argument API.**

**Note:** The solution is also available for viewing on the GitHub repository on a branch named “solution”.

## 5.4 Reader Challenge

Read the documentation of [\\_Flask](#) and build a front-end for this **API** . You can find the documentation here:

```
https://flask.palletsprojects.com/en/2.0.x/
```

# Chapter 6. Google search as an API

You've gotten through here and that's a fantastic achievement. Now, let's have some fun!

So, you've learnt everything that you need to learn about **APIs** and are a master who has ambitions to build the next Google. But you're also too lazy to build it from scratch.

How do you build your own **Lo-Fi Google** ?



How can we get VC funding for this?

## 6.1 An informal introduction to URL and Querystring

URLs ( *Uniform Resource Locator* ) are better known as web address or website address. So, something like “ <https://www.google.com> ”, “ <https://CloudBytes.dev/books> ”, or “ <https://example.com/over/there?name=ferret> ” is an example of a URL.

A typical URL will contain:

1. *Protocol* : Typically, `http` or `https`
2. *Domain name* : `google.com` or `CloudBytes.dev` , are examples of domain name
3. *File name* : E. g . `index.php` or `books.html` , etc.
4. *Querystring* “ `there?name=ferret` ” or “ `greet?name=Jason` ” are examples

## 6.2 What can we do with this information?

Let's try to **Google** the term " **cat memes** " and look at the address bar in the browser. The address of the search results page should resemble something like the below

```
https://www.google.com/search?q=cat+memes&...
```

**Google** searches words by sending **GET** requests to it's the **URL** with the term th is being searched as a querystring parameter. So, does that mean it is an **API** . Technically, no, but that doesn't mean we can't treat it like an **API** and do funky stuff with it.

So, let's start by changing the branch to the starter kit for this exercise.

```
git checkout p2-byog-flask
```

Then use the instructions provided previously under the section " **Initialise the development environment** " to

1. Initialise a **Python** virtual environment,
2. Activate the virtual environment and
3. Install the dependencies.

## 6.3 Understanding the Starter Kit

**Repository Structure** : The cloned files from **GitHub** are organised as

```
.
├── app.py
├── templates
│   └── index.html
├── static
│   ├── logo.png
│   └── styles.css
└── requirements.txt
```

Before getting into the details of what is included in the individual files, let's understand the different folders and their use.

1. **templates** : **Flask** application looks for **HTML** pages that will be displayed the browser in this folder by default
2. **static** : Is used to keep any static files that are used in the templates. This includes images, logos, **JavaScript** files, and **CSS Stylesheets**

*app.py*

Contains the template for our program, the contents of which are

```
#byog-python-flask/app.py
from flask import Flask, request, render_template, redirect

app = Flask(__name__)

@app.get("/")
def index():
    """
    TODO: Render the home page provided under templates/index.html in the repository
    """
    return "TODO"

@app.get("/search")
def search():
    """
    TODO:
    1. Capture the word that is being searched
    2. Search for the word on Google and display results
    """
    return "TODO"

if __name__ == "__main__":
    app.run()
```

## templates/index.html

This template page is designed to look like a Google homepage. We will use **Flask's** inbuilt method `render_template()` to render this website. We don't need to make any changes to this **HTML** code.

```
<!-- #byog-python-flask/templates/index.html -->
<!DOCTYPE html>
<html lang = "en" >

< head >
  < meta charset = "UTF-8" >
  < meta http-equiv = "X-UA-Compatible" content = "IE=edge" >
  < meta name = "viewport" content = "width=device-width, initial-scale=1.0" >
  < link rel = "shortcut icon" href =
"https://www.google.com/images/branding/googleg/1x/googleg_standard_color_128dp.png"
  type = "image/x-icon" >
  < title >Goggle</ title >
  < link rel = "stylesheet" href =
"https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css" >
  < link rel = "stylesheet" href = "static/styles.css" >
</ head >

< body >
```



```

< div class = "main-wrapper" >
  < div class = "logo-container" >
    < img src = "static/logo.png" alt = "Lazy man's Google" >
  </ div >
  < div class = "search-bar" >
    < form action = "/search" method = "get" >
      < input type = "search" name = "q" autocomplete = "false" autofocus >
      < button type = "submit" >Goggle Search</ button >
      < button type = "submit" >I'm Feeling Lucky</ button >
    </ form >
  </ div >
</ div >
</ body >

</ html >

```

## static/logo.png

It is the logo for our search engine, named " **Goggle** ", you can use any other logo

## static/style.css

Contains the styling for our **HTML** page. No changes are required for the **CSS Stylesheet** as well.

```

/* #byog-python-flask/static/style.css */
byog-python-flask / app .py
.logo-container > img {
  max-height : 80% ;
  max-width : 80% ;
  margin : auto;
  text-align : -webkit-center;
  margin-top : 47px ;
}

.search-bar > form > input {
  display : flex;
  width : 100% ;
  height : 44px ;
  border-radius : 24px ;
  border : 1px solid #dfe1e5 ;
  margin : 0 auto;
  max-width : 482px ;
  margin-bottom : 1em ;
}

.main-wrapper {
  justify-content : center;
  align-items : center;
  margin : auto;
}

```

```

display : inline-block;
text-align : center;
margin : 2% 25% 0% 25% ;
}

body {
background-color : #f6f6f6 ;
font-family : Arial, sans-serif;
font-size : small;
margin : 0 ;
padding : 0 ;
min-height : 80% ;
position : relative;
}

button {
margin : 1em 1em 1em 1em ;
background-image : -webkit-linear-gradient (top,#f5f5f563,#f1f1f1);
background-color : #f2f2f200 ;
border-radius : 16px ;
line-height : 27px ;
height : 36px ;
min-width : 54px ;
text-align : center;
cursor : pointer;
color : #5F6368 ;
}

button :hover {
box-shadow : 0 1px 6px rgba (0, 0, 0, 1);
background-image : -webkit-linear-gradient (top,#f8f8f8,#f1f1f1);
background-color : #f8f8f8 ;
border : 1px solid #c6c6c6 ;
color : #222 ;
border-radius : 16px ;
}

```

## 6.4 Logic of the application

For simplicity, a basic html with a form is already provided in “ *index.html* ”, however, you are free to go crazy and learn from **UI** development skills by trying to copy the google page.

Our objective is to provide a search box to the “ **Gogglers\*** ”, who when entered set of words, is redirected to **Google.com** (because we’re lazy) and shown the results for the search.

This will work following the below pseudocode

1. Render the homepage with a search box and submit button
2. User submission is sent to API as a GET request
3. The API backend captures the GET request and parses the query
4. The API then redirects to Google with appropriate querystring to search for .

## 6.5 Rendering home page

To render the home page, we need to modify the route logic for home, i.e. " / ". We do that by changing `app.py` and updating the `index()` method as per below.

### app.py

```
#byog-python-flask/app.py
from flask import Flask, request, render_template, redirect

app = Flask(__name__)

@app.get("/")
def index():
    # Render the index.html template and return it
    return render_template("index.html")

@app.get("/search")
def search():
    """
    TODO:
    1. Capture the word that is being searched
    2. Search for the word on Google and display results
    """
    return "TODO"

if __name__ == "__main__":
    app.run()
```

We used the `render_template()` method to render the " `index.html` " page and then returned it to the requestor.

### 6.5.1 Returning an HTML page

An appropriate route for the homepage, i.e. " / " is defined along with the handle function named `index()`.

By using `render_template()` method, we will return the rendered **HTML** page based the " `index.html` " file that is available in the templates folder.

Now if you start the flask app by running " `flask run` " and navigate to the **URL** ( <http://127.0.0.1/> ) in the browser, you will see your own **Google** page with two

buttons, “ *Goggle Searc h* ” and “ *I'm Feeling Luck y* ”.

If you press any of these buttons you will encounter an *Error 404 Not Foun d* .

## 6.5.2 Why are we getting a 404 error?

To answer this question, you need to look at this declaration in the *template/index.htm l* file that defines the form containing the two buttons.

```
< form action = "/search" method = "get" >
  < input type = "search" name = "search" id = "search" >
  < button type = "submit" >Goggle Search</ button >
  < button type = "submit" >I'm Feeling Lucky</ button >
</ form >
```

As seen above, there is an action defined to the route “ */searc h* ”, but in our program *app.p y* , we did haven’t yet defined the “ */searc h* ” route yet, thus when browser sends a **GET** request to the **API** , it sends a *404 Not Foun d* error since the route doesn’t exist .

## 6.6 Returning Search Results

As per our pseudocode, pressing the buttons should send a **GET** request, which it is, now we need to listen to it, and parse the request. We do that in the following manner:

### **app.py**

```
#byog-python-flask/app.py
from flask import Flask, request, render_template, redirect

app = Flask(__name__)

@app.get("/")
def index ():
    return render_template( "index.html" )

@app.get("/search")
def search ():
    args = request.args.get( "q" )
    return redirect( f"https://google.com/search?q= {args} " )

if __name__ == "__main__" :
    app.run()
```

## 6.7 Explanation

a) Import the `redirect()` method from “ `flask` ” library to enable us to redirect to another **URL**

```
from flask import Flask, request, render_template, redirect
```

b) Define the login of the route “ `/search` ”, that will be used to listen and respond to **GET** requests

```
@app.get("/search")
def search():
    args = request.args.get("q")
    return redirect(f"https://google.com/search?q={args}")
```

The above method

1. Parses the argument “ `q` ”, from the incoming **GET** request querystring
2. Then redirects to the **Google** website with the querystring of Google search and adding the argument

Now restart your **Flask** application and open it in your browser. Type something in the search bar and press “ `Goggle Search` ”, this will take you to the **Google** website with the terms you had searched for.

So now you have your own awesome personal **Google**.

## 6.8 Student Challenge

Implement the “ `I'm Feeling Lucky` ” button.

# Chapter 7: Building a Dictionary API

For this exercise we will build an **API** that will act as dictionary for **English** language, the endpoint that we will define will have the following features:

1. Respond with an exact match for a word when requested
2. Respond with approximate matches for a word when no match is found
3. Respond to list of words and not just a single word

We will also organise our code as per professional standards making use of **MVC** concepts.

So, let's begin by checking out the starter kit for this exercise

```
git checkout p3-dictionary-api-flask
```

Then use the instructions provided previously under the section "**Initialise the development environment**" to

1. Initialise a **Python** virtual environment,
2. Activate the virtual environment and
3. Install the dependencies.

Remember, each project has its own dependencies and a separate virtual environment needs to be created.

## 7.1 Understanding the Starter Kit

**Repository Structure** : The cloned repository includes the following

```
.
├── app.py
├── data
│   └── dictionary.db
├── model
│   └── dbHandler.py
└── requirements.txt
```

### app.py

Contains the template for our program, the contents of which are

```

#dictionary-api-python-flask/app.py
from flask import Flask, request, jsonify
from model.dbHandler import match_exact, match_like

app = Flask(__name__)

@app.get("/")
def index ():
    """
    DEFAULT ROUTE
    This method will
    1. Provide usage instructions formatted as JSON
    """
    return "TODO"

@app.get("/dict")
def dict ():
    """
    DEFAULT ROUTE
    This method will
    1. Accept a word from the request
    2. Try to find an exact match and return it if found
    3. If not found, find all approximate matches and return
    """
    return "TODO"

if __name__ == "__main__" :
    app.run()

```

In line #3 We are importing `match_exact()` and `match_like()` functions which are defi under `model/dbHandler.p y` file.

## model/dbHandler.py

Models, the V from **MVC** design pattern, are used to model and handle data. While **MVC** is a good read and an interesting topic, but to summarise

1. **Model** is the central component and that contains the application logic and data handling.
2. **View** defines what the user sees or the frontend, in the **Goggle** example, it will include everything under templates and static folders.
3. **Controller** manager user interaction, in our examples **app.p y** is acting like a controller with its `@app.route()` invocations.

The contents of our starter model are

```

#dictionary-api-python-flask/model/dbHandler.py
import sqlite3 as SQL

def match_exact (word: str) -> list:

```

```

"""
This method will:
1. Accept a string
2. Search the dictionary for an exact match
3. If success return the definition
4. If not return an empty list
"""

return "TODO"

def match_like (word: str) -> list:
"""
This method will:
1. Accept a string
2. Search the dictionary for approximate matches
3. If success return the definition as a list
4. If not return an empty list
"""

return "TODO"

```

## data/dictionary.db

This is the database of **English** words that we are going to use. It is formatted as an **SQLite DB** , which is a lightweight relational DB engine. A good practice is to keep all the database in one place, this has been segregated in its folder.

The schema of dictionary.db is described below

Column	Type	Schema
word	varchar( 25 )	"word" varchar( 25 ) NOT NULL
wordtype	varchar( 20 )	"wordtype" varchar( 20 ) NOT NULL
definition	text	"definition" text NOT NULL

## 7.2 Logic of the application

First, we need to update the *index()* method to handle incoming requests and respond back with usage instructions. To do this update the method as per below

```

#Update the index method under app.py
@app.get("/")
def index ():
"""
DEFAULT ROUTE
This method will
1. Provide usage instructions formatted as JSON
"""

response = { "usage" : "/dict?=<word>" }
return jsonify(response)

```



A very sweet and simple method that will tell us what the usage pattern is.

## 7.3 Handle incoming searches

We first need to accept the word being searched from the incoming request. As per our feature specification, the app should be able to handle either a single word or a list of words in the querystring.

a) Let's first build the logic to handle a single word. We will update our *dictionary()* method to capture a get request and store it using. As in previous instances, we can do that by using this snippet

```
word = request.args.get( "word" )
```

b) Then we build our usual sanity check to ensure a proper input has been provided

```
if not word:  
    return jsonify( { "data" : "Not a valid word or no word provided" } )
```

Thus, if the incoming request doesn't include a parameter "word" or the querystring is malformed, our API will return this error

c) Now let's call the *match\_exact()* method by passing the word and capture the result in a variable. After that we will check if the response is an empty list which would signify the word doesn't exist in our dictionary. If an exact match is found, we will return a response with the definition.

```
definitions = match_exact(word)  
if definitions:  
    return jsonify( { "data" : definitions } )
```

d) If an exact match is not found, we search for an approximate match using *match\_like()* method and return the definitions. If an approximate match is not found as well, an error status is returned.

```
definitions = match_like(word)  
if definitions:  
    return jsonify( { "data" : definitions } )  
else :  
    return jsonify( { "data" : "word not found" } )
```

app.py

Thus the *app.py* will look like below

```
# dictionary-api-python-flask/app.py
from flask import Flask, request, jsonify
from model.dbHandler import match_exact, match_like

app = Flask(__name__)

@app.get("/")
def index ():
    """
    DEFAULT ROUTE
    This method will
    1. Provide usage instructions formatted as JSON
    """
    response = { "usage" : "/dict?=<word>" }
    return jsonify(response)

@app.get("/dict")
def dictionary ():
    """
    SEARCH ROUTE
    This method will
    1. Accept a word from the request
    2. Try to find an exact match and return it if found
    3. If not found, find all approximate matches and return
    """
    word = request.args.get( "word" )

    # Return an error querystring is malformed
    if not word:
        return jsonify({ "status" : "error" , "data" : "word not found" })

    # Try to find an exact match
    definitions = match_exact(word)
    if definitions:
        return jsonify({ "status" : "success" , "data" : definitions})

    # Try to find an approximate match
    definitions = match_like(word)
    if definitions:
        return jsonify({ "status" : "partial" , "data" : definitions})
    else :
        return jsonify({ "status" : "error" , "data" : "word not found" })

if __name__ == "__main__" :
    app.run()
```

### 7.3.1 Testing our changes

**Home Route** : Let's do a quick test of our app so far, open terminal and run

```
curl -X GET http:// 127.0.0.1 : 5000 /
```

This should produce

```
{ "usage" : "/dict?=<word>" }
```

**Dictionary Route** : Similarly let's test the dictionary route.

```
curl -X GET http:// 127.0.0.1 : 5000 /dict?word=data
```

The response should be

```
{ "data" : "TODO" , "status" : "success" , "word" : "data" }
```

The output is not correct because we haven't implemented our models yet.

## 7.4 Finding the definition of the word

We need to use *Python's sqlite3* library to interact with the database and extract the matches.

First, we will find all exact matches to the provided word by

1. *Establish connection to the dictionary*
2. *Query the database for exact matches*
3. *Close the connection to the database*
4. *Return the response*

We do that by updating the method `match_exact()` to the following

```
def match_exact (word: str) -> list:
    """
    This method will:
    1. Accept a string
    2. Search the dictionary for an exact match
    3. If success return the definition
    4. If not return an empty list
    """
    # Establish connection to the dictionary database
    db = SQL.connect( "data/dictionary.db" )

    # Query the database for exact matches
    sql_query = "SELECT * from entries WHERE word=?"
    match = db.execute(sql_query, (word,)).fetchall()
```

```
# Clone the connection to the database  
db.close()
```

```
# Return the results  
return match
```

Next, we need to follow similar steps but find and return all approximate matches. We do that by updating `match_like()` method to

```
def match_like (word: str) -> list:  
    """  
    This method will:  
    1. Accept a string  
    2. Search the dictionary for approximate matches  
    3. If success return the definition as a list  
    4. If not return an empty list  
    """  
  
    # Establish connection to the dictionary database  
    db = SQL.connect( "data/dictionary.db" )  
  
    # Query the database for exact matches  
    sql_query = "SELECT * from entries WHERE word LIKE ?"  
    match = db.execute(sql_query, ( "%" + word + "%" ,)).fetchall()  
  
    # Clone the connection to the database  
    db.close()  
  
    # Return the results  
    return match
```

## model/dbHandler.py

The final model now includes both completed methods, `match_exact()` and `match_like` and looks like the below

```
# dictionary-api-python-flask/models/dbHandler.py  
import sqlite3 as SQL  
  
def match_exact (word: str) -> list:  
    """  
    This method will:  
    1. Accept a string  
    2. Search the dictionary for an exact match  
    3. If success return the definition  
    4. If not return an empty list  
    """  
  
    # Establish connection to the dictionary database  
    db = SQL.connect( "data/dictionary.db" )  
    sql_query = "SELECT * from entries WHERE word=?"
```

```

# Query the database for exact matches
match = db.execute(sql_query, (word,)).fetchall()
# Clone the connection to the database
db.close()

# Return the results
return match

def match_like (word: str) -> list:
    """
    This method will:
    1. Accept a string
    2. Search the dictionary for approximate matches
    3. If success return the definition as a list
    4. If not return an empty list
    """
    # Establish connection to the dictionary database
    db = SQL.connect( "data/dictionary.db" )

    # Query the database for exact matches
    sql_query = "SELECT * from entries WHERE word LIKE ?"
    match = db.execute(sql_query, ( "%" + word + "%" ,)).fetchall()

    # Clone the connection to the database
    db.close()
    # Return the results
    return match

```

## 7.4.1 Testing our changes

To test restart **Flask** and run the following in **Terminal**

```
curl -X GET http:// 127.0 . 0.1 : 5000 /dict?word= data
```

Our **API** server will look at the query string, parse the word parameter as “ *tes t* ” and search for exact matches. This will produce a **JSON** that is composed of a list of lists under the key “ *dat a* ” and “ *statu s* ” as “ *succes s* ”.

```

{
  "data" :[
    [ "data" , "n. pl." , "See Datum." ],
    [ "data" , "pl. " , "of Datum" ]
  ],
  "status" : "success" ,
  "word" : "data"
}

```

Note the “ *dat a* ” in the above is a list of lists. This is because one word can have

multiple meanings and the dictionary **API** will rightly respond with a list where there are multiple meanings.

Similarly, let's search for the approximate matches as well by running

```
curl -X GET http:// 127.0.0.1 : 5000 /dict?word=datar
```

The server will look for a word, “ *data r* ” in the dictionary, and when it is not found, it will try to find similar matches. This will produce a **JSON** that is composed of a list of lists under the key “ *dat a* ” and “ *statu s* ” as “ *partia l* ”.

```
{
  "data": [
    [ "commendatory", "n.", "One who holds a living in commendam." ],
    [ "dataria", "n.", "Formerly, a part of the Roman chancery; now, a separate\n  office from which are sent graces or favors, cognizable in foro\n  externo, such as appointments to benefices. The name is derived from\n  the word datum, given or dated (with the indications of the time and\n  place of granting the gift or favor)." ],
    [ "datary", "n.", "An officer in the pope's court, having charge of the\n  Dataria." ],
    [ "datary", "n.", "The office or employment of a datary." ],
    [ "mandatary", "n.", "One to whom a command or charge is given; hence,\n  specifically, a person to whom the pope has, by his prerogative, given\n  a mandate or order for his benefice." ],
    [ "mandatary", "n.", "One who undertakes to discharge a specific business\n  commission; a mandatory." ]
  ],
  "status": "partial",
  "word": "datar"
}
```

**Congratulations! The first step of our API is ready.**

## 7.5 Handling list of words

While our app is working fine, but if we try to send multiple words together using a command like below,

```
curl -X GET "http://127.0.0.1:5000/?word=data&word=datar"
```

It will respond with only the definition of “ *dat a* ” because our API doesn’t have mechanism to accept multiple parameters. Hence, we need to modify our app to look for a list of words provided as part of the query string.

To solve this, there are two options

1. Use `str.split()` method to split the input string by ‘,’ into a list of strings
2. Use the Flask’s in-built method, `request.args.getlist()`

While the first option works and is semantically easier to implement, it is not best-practice. Use of the second method is recommended due to its prevalence, **REST API** standards, and browser implementations.

Hence, in the `app.py`, change the line

```
word = request.args.get( "word" )
```

to the below using `request.args.getlist()` method

```
words = request.args.getlist( "word" )
```

Now the rest of the program will need to be changed to use a loop for each element in the list using the below algorithm

1. For each word in the list of words
  1. Search for an exact match
  2. If an exact match is found: append the data in response
  3. If an exact match is not found
    1. Search for an approx. match
    2. If an approx. match is found: append the data in response
    3. If approx. match is not found: Add error in the response
2. Return the final response

So, we modify our `app.py` method `dictionary()` as per below

## app.py

```
# dictionary-api-python-flask/app.py
from flask import Flask, request, jsonify
from model.dbHandler import match_exact, match_like

app = Flask(__name__)

@app.get("/")
def index():
    """
    DEFAULT ROUTE
    This method will
    1. Provide usage instructions formatted as JSON
    """
    response = { "usage" : "/dict?=<word>" }
    return jsonify(response)

@app.get("/dict")
def dictionary():
    """
    DEFAULT ROUTE
```

```

This method will
1. Accept a word from the request
2. Try to find an exact match and return it if found
3. If not found, find all approximate matches and return
"""
words = request.args.getlist( "word" )

# Return an error querystring is malformed
if not words:
    response = { "status": "error" , "word" : words, "data" : "word not found" }
    return jsonify(response)

# Initialise the response
response = { "words" : []}

for word in words:
    # Try to find an exact match
    definitions = match_exact(word)
    if definitions:
        response[ "words" ].append({ "status": "success" , "word" : word, "data" : definitions})
    else :
        # Try to find an approximate match
        definitions = match_like(word)
        if definitions:
            response[ "words" ].append({ "status": "partial" , "word" : word, "data" : definitions})
        else :
            response[words].append({ "status": "error" , "word" : word, "data" : "word not found" })

# Return the response after processing all words
return jsonify(response)

if __name__ == "__main__" :
    app.run()

```

## 7.6 Testing the API

We send two words in the **GET** request now using Terminal

```
curl -X GET "http://127.0.0.1:5000/?word=data&word=datar"
```

And now we should response that includes a " *success* " message for " *data* " because it was an exact match in our dictionary, and a " *partial* " status for " *datar* " since got only partial matches.

```

{ "words" :[
  { "data" :[[ "data" , "n. pl." , "See Datum." ],[ "data" , "pl. " , "of Datum" ]],
    "status" : "success" ,
    "word" : "data" },
  { "data" :[[ "commendatory" , "n." , "One who holds a living in commendam." ],[ "dataria" , "n." ,
    "Formerly, a part of the Roman chancery; now, a separate\n  office from which are sent graces or

```



```
favors, cognizable in foro\
  externo, such as appointments to benefices. The name is derived
  from\
  the word datum, given or dated (with the indications of the time and\
  place of granting the gift or favor)." ],[ "datary" , "n." , "An officer in the pope's court, having charge of the\
  Dataria." ],[
  "datary" , "n." , "The office or employment of a datary." ],[ "mandatary" , "n." , "One to whom a command
  or charge is given; hence,\
  specifically, a person to whom the pope has, by his prerogative,
  given\
  a mandate or order for his benefice." ],[ "mandatary" , "n." , "One who undertakes to discharge
  a specific business\
  commission; a mandatary." ]],
  "status" : "partial" ,
  "word" : "datar" ]}}
```

## 7.7 Student Challenge

Build a front end for this program using what you learnt in **Goggle** exercise, it should include

1. A simple homepage with a search box, submit button
2. It should be able to search only 1 word at a time. You can do a list of words as well, but that requires **JavaScript** .
3. A page to display the results where the word and match status is highlighted as a header, and each definition is shown as an unordered list underneath

**NOTE: The reference solution is available on GitHub Repository under the branch " s3-bonus-dictionary-api-flask "**

## 7.8 Jupyter Notebook to test the API

**Jupyter Notebook** is a very handy tool loved by almost anybody who uses Python regularly. It is an interactive browser-based **Python IDE** that takes inputs in "cells" and outputs immediately below the cell.

In the above example, *In[1]* is the first cell where a Python command was entered then executed by pressing " *Ctrl + Enter* " or " *Shift + Enter* ". This immediately executes the command and prints " *Hello, Students !* " below it.

Then we stored a string " *Hello, Students !* " in a variable *message*, to see the content of that variable, just type the variable name and press " *Ctrl + Enter* ". This displays the content in *Out[2]*.

While doing data or other forms of analysis using **Python** , **Pandas** , and **NumPy** this is a very handy feature that allows the analyst to explore the contents of data in real-time.

**Note:** By default, the Jupyter Notebook will use the system-wide Python installation, so you may need to install certain libraries using " *pip3 install <library-name>* " syntax in case you get errors

Let's use **Jupyter Notebook** to test our dictionary **API** . Type the below in cell 1,

```
#Jupyter Notebook/ In[1]
import requests
```

This will import the **Python** library that is used to send **cURL** requests, you will not see any output. Then, we will use *requests.get()* method to fetch a response.

```
# Jupyter Notebook/ In[2]
URL = "http://127.0.0.1:5000/"
response = requests.get(URL)
response.raise_for_status
```

We check the response for status using " *raise\_for\_status* ", which will tell us if the response was successful. This will output

```
< bound method Response.raise_for_status of < Response [ 200 ]>>
```

A *[200] Status* means the **API** request was successfully processed. To see the contents of the response, we need to run

```
# Jupyter Notebook/ In[3]
response.content
```

This will print out the output for a **GET** request at the **URL** <https://127.0.0.1:5000>, which in this case is the usage instructions

```
b'{ "usage" : "/dict?word=<word>" }'
```

The " *b* " denotes that the contents are encoded in binary but printed out to the console.

**Why does this matter** ? Because any data in binary is made of 0's and 1's in a literal sense, and if we wanted to just check the value of "usage", we would not be able to do that.

Thus, we need to

1. Decode the response into string using " *utf-8* " specifications
2. Convert the string into a *JSON* object

```
# Jupyter Notebook / In[4]
import json
data = response.content.decode( "utf-9" )
data = json.loads(data)
data[usage]
```

This will print out

```
"/dict?word=<word>"
```

## 7.8.1 Searching the dictionary using Jupyter

Based on the above, we need to run the following

```
# Jupyter Notebook / In[1]
import requests
import json

URL = "http://127.0.0.1:5000/"
word = data

response = request.get( f" {URL} dict?word= {word} " )

data = json.loads(response.content.decode( "utf-8" ))
data
```

This will print out the definitions for the word " *data* "

# Chapter 8: Building a POST API

We have created a few **APIs** that respond to the **GET** request. How about we create an **API** that can handle **POST** requests. We will build an **API** that accepts image via a **POST** request. In this case, we cannot use a **GET** request because all the data in the **GET** request is part of the " **querystring** " or headers, both of which have a maximum limit.

Once the **API** accepts the image sent as a payload in a **POST** request, our app will apply filters to it, and then send it back to the requestor.

## 8.1 API to add Filters

In this example, we are going to learn to fork a repository from **GitHub** . This will be needed in the next chapter.

a) To get started, switch to the starter kit for this exercise by running

```
git checkout p4-image-filter-flask
```

Then using instructions in previous chapters,

1. *Create a virtual environment*
2. *Activate the virtual environment*
3. *Install the dependencies*
4. *Open the folder in VSCode*

## 8.2 Understanding the Starter Kit

**Repository Structure** : The cloned files from **GitHub** are organised as

```
.
├── app.py
├── bin
│   └── filters.py
├── requirements.txt
└── sample.jpg
```

**app.py**

```
from flask import Flask, request, send_file, jsonify
```

```

from bin.filters import apply_filter

app = Flask(__name__)

filters_available = []

@app.route("/", methods=["GET", "POST"])
def index ():
    """
    TODO:
    1. Return the usage instructions that
    a) specifies which filters are available,
    b) specifies and the method format
    """
    pass

@app.post("/<filter>")
def image_filter (filter):
    """
    TODO:
    1. Checks if the provided filter is available, if not, return an error
    2. Check if a file has been provided in the POST request, if not return an error
    3. Apply the filter using apply_filter() method from bin.filters
    4. Return the filtered image as response
    """
    pass

if __name__ == "__main__" :
    app.run()

```

## bin/filters.py

```

from PIL import Image, ImageFilter
import io

def apply_filter (file: object, filter: str) -> object:
    """
    TODO:
    1. Accept the image as file object, and the filter type as string
    2. Open the as an PIL Image object
    3. Apply the filter
    4. Convert the PIL Image object to file object
    5. Return the file object
    """
    pass

```

## sample.jpg

It is an image provided to ease the testing of this **API** . You can use your own images instead.

## 8.3 Logic of the application

First, we will define the route for “ / ”, but unlike previous cases this route accepts requests using both **GET** and **POST** methods. The `index()` method then returns a response that explains

1. *The filters that are available*
2. *Usage instructions on how to use the filter*

### 8.3.1 Available Filters

We are going to use the **PIL (Python Imaging Library)** to handle and apply filters. If you read the [\\_PIL Documentation](https://pillow.readthedocs.io/en/stable/) at <https://pillow.readthedocs.io/en/stable/> , you can find the filters that are available out of the box. We list these in the `available_filters` in the `app.py` .

```
# Filters available in PIL
filters_available = [
    "blur" ,
    "contour" ,
    "detail" ,
    "edge_enhance" ,
    "edge_enhance_more" ,
    "emboss" ,
    "find_edges" ,
    "sharpen" ,
    "smooth" ,
    "smooth_more" ,
]
```

### 8.3.2 Home Route

```
@app.route("/", methods=["GET", "POST"])
def index ():
    """
    TODO:
    1. Return the usage instructions that
    a) specifies which filters are available,
    b) specifies and the method format
    """
    response = {
        "filters_available": filters_available,
        "usage": { "http_method": "POST" , "URL" : "/<filter_available>" },
```

```
}  
  
return jsonify(response)
```

`@app.route` is used to provide two **HTTP** methods at the same time, both **GET** and **POST**. Then we define the response that contains the usage instructions and the filters available and return it to the requestor.

### 8.3.3 Filter Route

```
@app.post("/<filter>")  
def image_filter (filter):  
    """  
    TODO:  
    1. Checks if the provided filter is available, if not, return an error  
    2. Check if a file has been provided in the POST request, if not return an error  
    3. Apply the filter using apply_filter() method from bin.filters  
    4. Return the filtered image as response  
    """  
    if filter not in filters_available:  
        response = { "error" : "incorrect filter" }  
        return jsonify(response)  
  
    file = request.files[ "image" ]  
    if not file:  
        response = { "error" : "no file provided" }  
        return jsonify(response)  
  
    filtered_image = apply_filter(file, filter)  
  
    return send_file(filtered_image, mimetype= "image/JPEG" )
```

`@app.post("<filter>")` is used to post requests, but notice the route?

This route is expressed as a parameter, so if the **POST** request is sent to the “URL/blur” using the `<filter>` parameter will capture “ *blur* ” as a variable. This occurs in the next line where “ *filter* ” is the argument for the `image_filter()` method.

#### Ensuring the provided filter is available

```
if filter not in filters_available:  
    response = { "error" : "incorrect filter" }  
    return jsonify(response)
```

The above snippet checks if the provided “ *filter* ” is not available in the list of available filters, and if it is not available, it returns an error message stating the filter is incorrect.

## Ensuring a file is provided

```
file = request.files[ "image" ]
if not file:
    response = { "error" : "no file provided" }
    return jsonify(response)
```

A quick check to see if the request includes any file with the name “ *image* ”. If the file is not provided in the **POST** request, it will return an error message stating the file was not provided.

## If all constraints are met, apply the filter, and return the file

```
filtered_image = apply_filter(file, filter)
return send_file(filtered_image, mimetype= "image/JPEG" )
```

We use the *apply\_filter()* method and pass it the file object and filter string as arguments, and the method returns a file object with filter applied. We will implement the logic for *apply\_filter()* in the next section.

Then we return the filtered image using *send\_file()* method and specify the *mimetype* as “ *image/JPEG* ” so that the requestor knows what kind of data it has received.

## 8.3.4 Combining all changes

After all these changes, our *app.py* looks like below

### app.py

```
from flask import Flask, request, send_file, jsonify
from bin.filters import apply_filter

app = Flask(__name__)

filters_available = [
    "blur",
    "contour",
    "detail",
    "edge_enhance",
    "edge_enhance_more",
    "emboss",
    "find_edges",
    "sharpen",
    "smooth",
    "smooth_more",
]
```



```

@app.route("/", methods=["GET", "POST"])
def index ():
    """
    TODO:
    1. Return the usage instructions that
    a) specifies which filters are available,
    b) specifies and the method format
    """
    response = {
        "filters_available": filters_available,
        "usage": { "http_method": "POST", "URL": "/<filter_available>" },
    }
    return jsonify(response)

@app.post("/<filter>")
def image_filter (filter):
    """
    TODO:
    1. Checks if the provided filter is available, if not, return an error
    2. Check if a file has been provided in the POST request, if not return an error
    3. Apply the filter using apply_filter() method from bin.filters
    4. Return the filtered image as response
    """
    if filter not in filters_available:
        response = { "error": "incorrect filter" }
        return jsonify(response)

    file = request.files[ "image" ]
    if not file:
        response = { "error": "no file provided" }
        return jsonify(response)

    filtered_image = apply_filter(file, filter)

    return send_file(filtered_image, mimetype= "image/JPEG" )

if __name__ == "__main__" :
    app.run()

```

But our app is not ready yet. The `apply_filter()` method stills needs to be completed

## 8.4 Implementing the Filter

So now that we have passed the image to the `apply_filter()` method, we need to implement the logic for it.

### bin/filters.py

```

from PIL import Image, ImageFilter
import io

```

```

def apply_filter (file: object, filter: str) -> object:
    """
    TODO:
    1. Accept the image as file object, and the filter type as string
    2. Open the as an PIL Image object
    3. Apply the filter
    4. Convert the PIL Image object to file object
    5. Return the file object
    """
    image = Image.open(file)
    image = image.filter(eval( f"ImageFilter. {filter.upper()} " ))

    file = io.BytesIO()
    image.save(file, "JPEG" )
    file.seek( 0 )

    return file

```

## 8.4.1 Explanation

From the **PIL (Python Imaging Library)** we have imported two methods, *Image()* & *ImageFilter*.

*Image()* is used to store the image data, while *ImageFilter* is used to apply the pre-configured filter transformations.

To implement the *apply\_filter()* method, which takes two arguments

- a. *file* : The image that was received as a file object
- b. *filter* : The filter that will be applied as a string

We first open the image as a **PIL** object by

```
image = Image.open(file)
```

Once the file is opened as a **PIL** Image, we can apply a filter such as “ *blur* ” using the below command:

```
image = image.filter(ImageFilter.BLUR)
```

Using the above syntax, we can create filters for other filters as well, but that would mean we have to create a very long if-else logic, one if clause for each filter.

Instead we’re going to use a python method called *eval()* to convert a string to a python function call

```
image = image.filter(eval( f'ImageFilter. {filter.upper()} ' ))
```

Here, we converted the variable `filter` to uppercase, then appended the filter to “`ImageFilter` .” Finally, we convert that to a valid Python code using the “`eval()`” method, which converts a string to a **Python** command for execution.

Now, our filter is applied but we need to convert the **PIL Image Object** to a file object before returning, that is done by

```
file = io.BytesIO()
image.save(file, "JPEG" )
file.seek( 0 )

return file
```

We first create an empty buffer in memory using `io.BytesIO()` and pass the pointer to the buffer to the `image.save()` method. This will store the file in memory, finally, we reset the pointer to start with `file.seek(0)` , and finally return the file object.

## 8.5 Testing the API

Start the Flask app by running “`flask run`” in the terminal. Then open another terminal window and fire up the Jupyter Notebook by running “`jupyter notebook`”

### Getting usage instructions

Let’s first test if the **API** is accessible, to do that run the below code snippet in **Jupyter Notebook** .

```
# Jupyter Notebook/ In[1]
import requests
response = requests.post( "http://127.0.0.1:5000/" )
response.raise_for_status
```

This will produce the below output

```
< bound method Response.raise_for_status of < Response [ 200 ]>>
```

The “`200`” response code means our request was successfully processed. Now run the below snippet to see that the response was

```
# Jupyter Notebook/ In[2]
import json
json.loads(response.content.decode( "utf-8" ))
```

Here, we first have to decode the response because our **API** response was a binary

stream. Then we load that as a **JSON** object. This will produce the output

```
{'filters_available': ['blur',  
'contour',  
'detail',  
'edge_enhance',  
'edge_enhance_more',  
'emboss',  
'find_edges',  
'sharpen',  
'smooth',  
'smooth_more'],  
'usage': {'URL': '/<filter_available>', 'http_method': 'POST'}}
```

We have the usage instructions, now we can proceed.

## 8.6 Getting filtered image

To get the filtered image, run the following and replace “ *sample.jpg* ” with the path to the file of your choice

```
# Jupyter Notebook/ In[3]  
file = { "image" : open( "sample.jpg" , "rb" )}  
headers = { "type" : "multipart/image" }  
  
URL = "http://127.0.0.1:5000"  
filter = "contour"  
  
response = requests.post( f" {URL} / {filter} " ,headers=headers, files=file)  
response.raise_for_status
```

### 8.6.1 Explanation

We first open the image that needs to be filtered using the `open()` method, then add it to a **Python** dictionary with the name “ *image* ”.

Then we define the header that tells the **API** server that kind of file this is. Headers are optional, but good practice.

We then define the parameters of the **URL** and querystring and then finally send the request along with headers, and the file.

When we ask the response for status, it will show

```
< bound method Response.raise_for_status of < Response [ 200 ]>>
```

This means our **POST** request was processed successfully.

But wait, there's more!

We have received the filtered image, but we haven't stored or viewed it yet. To do that run

```
# Jupyter Notebook/ In[4]
from PIL import Image
import io

image = Image.open(io.BytesIO(response.content))
image.save( "response.jpg" , "JPEG" )

image
```

We again import the **PIL** library and **io** library to handle the image. The response is converted to file object in memory using `io.BytesIO()` and then loaded as an image using `Image.open()`

Finally, the file is saved using `image.save()` , you can change the “ `response.jpg` ” to path of the file you want to save to.

Then we load the image into **Jupyter** , this smooth operator!

**Congratulations! You have made your own building blocks for an Instagram.**

## 8.7 Bonus Challenge #2

Using this solution as a starting point, update the program to include more filters such as *sepia* , *black & white* , etc.

You can mail the response to [student@CloudBytes.dev](mailto:student@CloudBytes.dev) with Subject line " **Bonus Challenge #2** ". The first three working solutions will get a \$100 Amazon gift voucher

# Chapter 9: Bonus Lesson: Deploying the API

We have created our image **API** , brilliant! But it is still running on our local machine. To access it over the internet, we need to set up a **CI/CD** pipeline that will automatically take our updates, configure the environment, and then deploy these API on any hosting provider.

In this example, we will use Heroku for deployment. Below is a simple representation of how this works,



We will configure our environment in a way that, our code being pushed to GitHub will trigger an event to Heroku, which will build the program and then deploy it to their cloud which will be accessible to the world.

## 9.1 Configuring the CD Pipeline

This assumes you have created a fork of the starter kit provided in prerequisites and have updated your solution in the branch name “ *p4-image-filter-flask* ”

**a) Create Heroku Account** : First, go to the [Heroku signup page](https://signup.heroku.com/) provided below website and register for a free account. This won't require any credit card.

```
https://signup.heroku.com/
```

After you have confirmed your email and set your password. Login to the Heroku site by visiting

```
https://id.heroku.com/login
```

**b) Create New App** : Once logged in, click on “ *Create new app* ”, then choose the app name, and a region. This will be the region where your **API** will be hosted and can be any country.

The app name must be unique and will form the **URL** for your app.

Then click on “ *Add to pipeline* ” below the region, followed by selecting “ *Choose a*

*pipeline* ". Finally select " *Create new pipeline* ". This should ask you for the name of the pipeline and the stage, leave this unchanged, and then click on " *Create app* " .

**c) Configure the pipeline :** Under " *Deployment method* " choose " *Connect to GitHub* " then scroll below and click on the " *Connect to GitHub* " button.

**d) Authenticate GitHub :** You will be redirected to **GitHub** login; login using your credentials and allow **Heroku** to connect to **GitHub** .

**e) Choose a repository :** On the right of your username, type the name of the forked repository ( *image-filter-api-python-flask* ) and then press *Search* button. This will list the repository below

**f) Connect the app to the repository :** Click on the " *Connect* " button to the right of the repository name. Then scroll below to the *Automatic deploy* section.

**g) Enable Automatic Deploys :** Under the " *Automatic Deploys* " section, click on the dropdown " *Choose the branch to deploy* " and choose " *p4-image-filter-flask* ", assuming you have made all the updates to only in this branch. Then click on the " *Enable Automatic Deploys* " button.

**i) Manual Deploy :** This is required only for the first time. After that the program will auto-deploy from the "main" branch every time you push updates to it.

**j) Connect the pipeline to GitHub again :** There is a bug on **Heroku** which sometimes will require you to connect the app to **GitHub** again. Do so, go back to the dashboard by clicking the **Heroku** Logo on top left, or using the **URL** below.

```
https://dashboard.heroku.com/apps
```

Then click on your app, if you see a message asking to connect the pipeline again, go ahead and do that by clicking on " *Connect to GitHub* ", you will need to select the repository again.

**NOTE:** Heroku uses a Procfile that defines the web-app configuration. This is provided as part of the starter kit. Without the Procfile, the app will not deploy correctly on Heroku.

## 9.2 Testing the API

Your **API** app is up and running! Note the **URL** of your **API** , it will be unique based on the name you have given but should look like

```
https://api-demo-cloudbytes.herokuapp.com/
```

Replace the " *api-demo-cloudbyte s* " with the name you had provided.

To test, run the below in **Jupyter Notebook**

```
# Jupyter Notebook/ In[1]
import requests
import json
# Change the below URL to the URL of your app
URL = "https://api-demo-cloudbytes.herokuapp.com/"
response = requests.get(URL)
response.raise_for_status
json.loads(response.content.decode( "utf-8" ))
```

Now, you can try the filter **API** using **POST** method

```
# Jupyter Notebook/ In[2]
file = { "image" : open( "sample.jpg" , "rb" )}
headers = { "type" : "multipart/image" }

filter = "contour"

response = requests.post( f" {URL} / {filter} " ,headers=headers, files=file)
response.raise_for_status
```

And print the output by running the below

```
# Jupyter Notebook/ In[3]
from PIL import Image
import io

image = Image.open(io.BytesIO(response.content))
image.save( "response.jpg" , "JPEG" )

image
```

## 9.3 Pro Tip: Testing using Terminal

Do this from terminal

```
curl -F "image=@sample.jpg" "https://api-demo-cloudbytes.herokuapp.com/blur" --output
"heroku.jpg"
```

## 9.4 Bonus Challenge

Build a frontend for this **API** . Share the link to your repository at [student@CloudBytes.dev](mailto:student@CloudBytes.dev) with Subject " *Image Bonus Challenge* ". The top three designs before 31st August 2021 will get an Amazon gift voucher worth \$100.



# Chapter 10: Introducing FastAPI

**Flask** is brilliant, it is quite easy to learn, and by now you have somewhat mastered building APIs using **Flask** . But **Flask** has several large limitations the biggest one being scalability.

Make no mistake, **Flask** is scalable, in fact, portions of **Airbnb** run on **Flask** and cannot get bigger than that, except, it can!

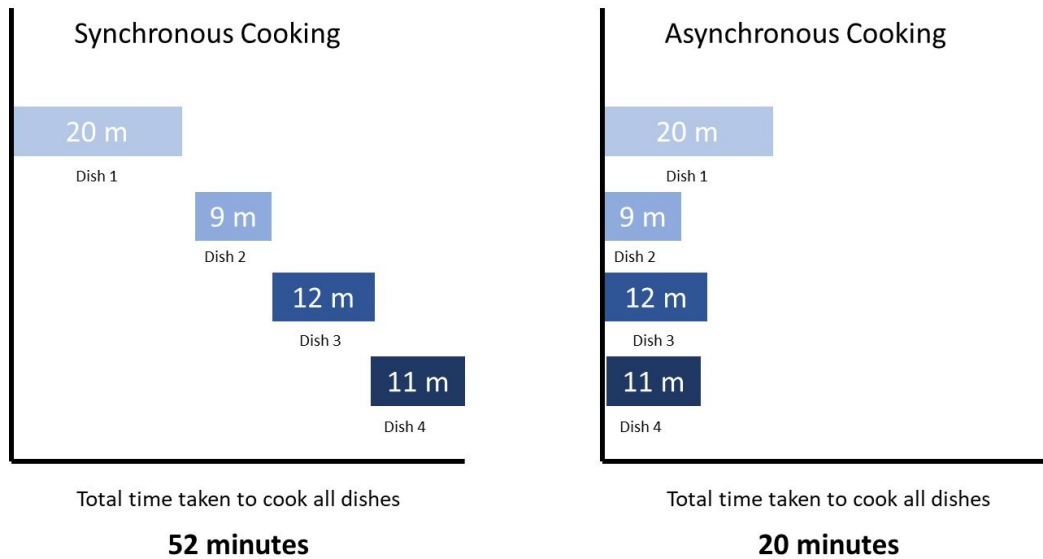
[Netflix](#) , [Reddit](#) , [Uber](#) , [Mozilla](#) , [Samsung](#) , all of these use **Flask** as part of their technology stack. **Netflix** , who typically are on the cutting edge of technology innovation in fact chose **Flask** for multiple applications internally, but the keyword is internally. To understand why, we first need to understand Asynchronous Programming

## 10.1 Asynchronous Programming

The scalability problem with **Flask** is with managing multiple requests together and serving them at the same time. Making an **API** request is like getting a takeout for a group of friends. Suppose you are hosting a few friends for a party and want to order something to eat.

You choose a restaurant and order 4 dishes, however, since the restaurant has only one cook, it can prepare only one dish at a time. It will quite a while for the cook to finish your order because they must make them one after the other in sequence.

But if the restaurant had 4 cooks, all four dishes could be prepared simultaneously, making the whole transaction faster.



Replace restaurant with **API** , food with response and order with request in the above scenario and you have got what is called “ **Asynchronous** ” **API** . What this means is you place multiple requests at the same time, the **API** prepares all the responses simultaneously, but sends it back only when you ask for the next one.

The example is of course is an oversimplification, but a good demonstration of the concept.

And this brings us back to **Flask** , while it is possible to perform asynchronous task programming with **Flask** , it is based on **Flask** is based on **WSGI** , which is synchronous by default. Thus, making it inherently slow especially compared to framework and languages that are designed for speed or excel at asynchronous programming such as NodeJS, Go, etc. And therefore, **Flask** is rarely used for external facing mass-user platforms.

And we haven't even got to the other issues with **Flask** such as data validation, web-socket support, and automatic documentation.

## 10.2 Enter FastAPI

[FastAPI](#) is a relative newcomer but a modern framework that is purpose-built using the latest that **Python** has to offer. It was launched only about 3 years ago but has seen steady adoption and increasing is replacing **Flask** , including being [integrated with Microsoft products in their ML APIs](#) .

It has taken off because of two reasons, firstly, it is fast, very fast. This is because of the out-of-the box support for async feature that was introduced in

**Python 3.6+ . Flask** on the other hand, relies on hacks and other libraries to achieve the same.

Secondly, it solves many of other problems that make **Flask** hard to your without writing a lot of additional code such as data validation, or even automated documentation. I might be overstating it but **FastAPI** is to **Flask** what a supercomputer is to **TI-82** calculator.

So, let's go ahead and experiment with **FastAPI** .

If you have installed the dependencies from the ' *requirements.txt* ' from the starter kit, FastAPI should already be installed.

To switch to the problem branch for this exercise run the below command

```
git checkout p5-hello-fastapi
```

## 10.3 Understanding the Starter Kit

**Repository Structure:** Once you have checked out the branch, you should see the following file structure (ignore the other files)

```
├── main.py
└── requirements.txt
```

### a) main.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def index ():
    return "TODO"
```

There are 2 points that needs to be noted

1. *The convention is to call the main program file “ **main.py** ”*
2. *The syntax is exactly like Flask, except instead of initialising a **Flask** app we initialise a **FastAPI** app*

## 10.4 Saying Hello FastAPI

To do so, all we need to do is change the **TODO** .

## main.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def index ():
    return { "hello" : "FastAPI" }
```

That's it. It's like using **Flask** , but it's not. We'll see the differences as we move further.

To run the program, from the terminal execute

```
uvicorn main:app --reload
```

This will start the app at <http://127.0.0.1:8000/>. That you can open with a browser  
Alternatively, test the **API** using the **cURL** utility by running

```
curl -X GET "http://127.0.0.1:8000/"
```

# Chapter 11: Dictionary using FastAPI

In this chapter, we will do the following

1. *Reimplement our dictionary API using FastAPI*
2. *Make the API asynchronous and see it in action.*

To get started switch to the problem branch by running the below

```
git checkout p6-dictionary-fastapi
```

This will clone the starter kit, which is very similar to our **Flask** dictionary starter kit

## 11.1 Understanding the Starter Kit

**Repository Structure** : The repository will contain the following files

```
.
├── main.py
├── data
│   └── dictionary.db
├── model
│   └── dbHandler.py
└── requirements.txt
```

### main.py

Looks very similar to the **Flask** application but it has a few additional headers

```
from typing import List, Optional
from fastapi import FastAPI, Query
from fastapi.encoders import jsonable_encoder
from model.dbHandler import match_exact, match_like
```

```
app = FastAPI ()
```

```
@ app . get ( "/" )
```

```
def index ():
```

```
    """
```

```
    DEFAULT ROUTE
```

```
    This method will
```

```
    1. Provide usage instructions formatted as JSON
```

```

"""
    return "TODO"

@app . get ( "/dict" )
def dictionary ():
    """
    DEFAULT ROUTE
    This method will
    1. Accept a word from the request
    2. Try to find an exact match, and return it if found
    3. If not found, find all approximate matches and return
    """
    return "TODO"

```

### model/dbHandler.py

This file remains unchanged from the **Flask** example

### data/dictionary.db

This too, remains unchanged.

## 11.2 Implementing the usage instructions

The only change we will have to do in our earlier Flask code, is instead of using *jsonify()* method, we need to use *jsonable\_encoder()* method.

```

@app . get ( "/" )
def index ():
    """
    DEFAULT ROUTE
    This method will
    1. Provide usage instructions formatted as JSON
    """
    response = { "usage" : "/dict?=<word>" }
    return jsonable_encoder ( response )

```

## 11.3 Implementing the dictionary

To complete our *dictionary()* method, again, we need to make only two changes

1. Use *jsonable\_encoder()* to encode our responses

2. Instead of using `request.args.get()` to capture the “word” parameter from query, just put “word” the `dictionary()` method arguments as show below

```
@ app . get ( "/dict" )
def dictionary ( word : str ):
    """
    DEFAULT ROUTE
    This method will
    1. Accept a word from the request
    2. Try to find an exact match, and return it if found
    3. If not found, find all approximate matches and return
    """

    if not word :
        response = { "status" : "error" , "word" : word , "data" : "word not found" }
        return jsonable_encoder ( response )

    definitions = match_exact ( word )
    if definitions :
        response = { "status" : "success" , "word" : word , "data" : definitions }
        return jsonable_encoder ( response )

    # Try to find an approximate match
    definitions = match_like ( word )
    if definitions :
        response = { "status" : "partial" , "word" : word , "data" : definitions }
        return jsonable_encoder ( response )
    else :
        response = { "status" : "error" , "word" : word , "data" : "word not found" }
        return jsonable_encoder ( response )
```

As you can see from above, we defined the method `dictionary()` to handle GET requests at the `/dict` path. But we also added `word` as a parameter.

```
def dictionary ( word : str ):
```

**FastAPI** will interpret this as a request for an argument (like `request.get.args()` ).

**WARNING** : The name of this argument and the name of a parameter path should be different.

i.e. you cannot have a path like the below

```
@ app . get ( "{word}" )
def dictionary ( word : str ):
```

**FastAPI** will interpret the above as a path and not an argument. You will instead need to use something like

```
@ app . get ( "{word_path}" )
def dictionary ( word_path : str, word_arg : str ):
```

## 11.4 Testing the API

The final program would look like below

### main.py

```
from typing import List , Optional
from fastapi import FastAPI , Query
from fastapi . encoders import jsonable_encoder
from model . dbHandler import match_exact , match_like

app = FastAPI ()

@ app . get ( "/" )
def index ():
    """
    DEFAULT ROUTE
    This method will
    1. Provide usage instructions formatted as JSON
    """
    response = { "usage" : "/dict?=<word>" }
    return jsonable_encoder ( response )

@ app . get ( "/dict" )
def dictionary ( word : str ):
    """
    DEFAULT ROUTE
    This method will
    1. Accept a word from the request
    2. Try to find an exact match, and return it if found
    3. If not found, find all approximate matches and return
    """
```



```

if not word :
    response = { "status" : "error" , "word" : word , "data" : "word not found" }
    return jsonable_encoder ( response )

definitions = match_exact ( word )
if definitions :
    response = { "status" : "success" , "word" : word , "data" : definitions }
    return jsonable_encoder ( response )

# Try to find an approximate match
definitions = match_like ( word )
if definitions :
    response = { "status" : "partial" , "word" : word , "data" : definitions }
    return jsonable_encoder ( response )
else :
    response = { "status" : "error" , "word" : word , "data" : "word not found" }
    return jsonable_encoder ( response )

```

Restart the program by running

```
uvicorn main:app --reload
```

To test out program, go to your terminal and run

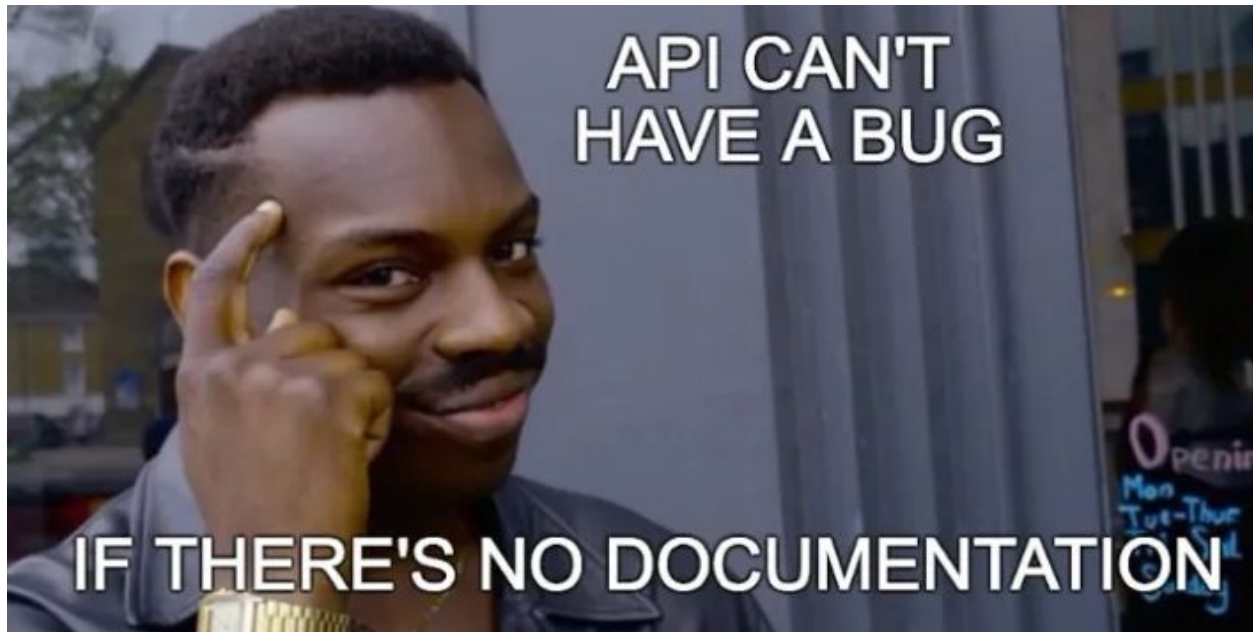
```
curl -X GET "http://127.0.0.1:8000/dict?word=datar"
```

This will throw out the same result as in **Flask** example.

## 11.5 : FastAPI OpenAPI Docs and Swagger UI

All developers hate building documentation, and if anyone says otherwise rest assured, they are lying. And there are good reasons, for starters, it is time consuming. Secondly, **API** documentation's main purpose is to allow other users to write applications using these **APIs** .

Thus, the more documentation is available, the better the chances of the developers understanding how to use **API** , the higher the chances of their finding bugs.



But seriously, having good documentation solves a lot of problems. E.g. if you are a backend developer building an **API** for the frontend developer to use, good documentation will help them understand the behaviour of the **API** reducing errors and development timeframes. But as I mentioned, building good documentation is time consuming, until developers figured out how to automate it.

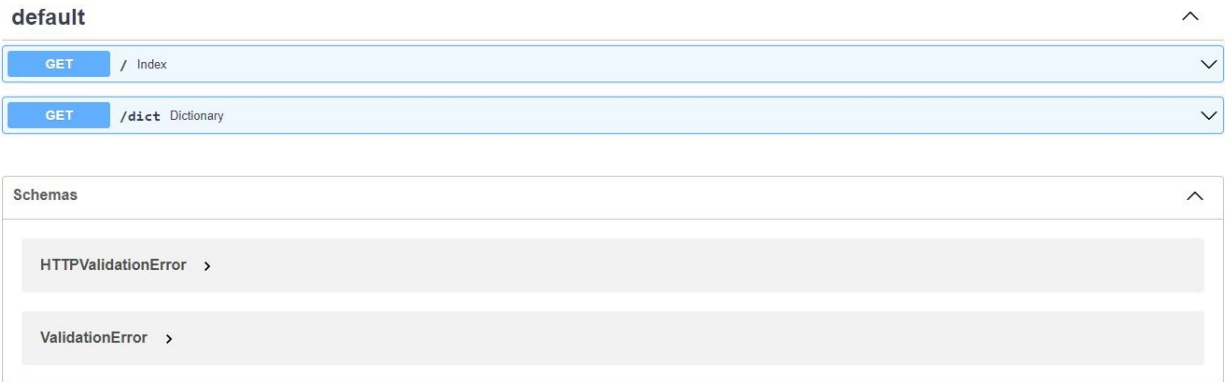
This is where **FastAPI's** automatic documentation engine makes life a lot easier by generating **APIs** documentation using the Docstring and the method declarations by automatically converting these into a **UI** that can be used by developers.

### 11.5.1 Working with Swagger UI

While the **FastAPI** app is running start your browser and go to the below **URL**

```
http://127.0.0.1:8000/docs
```

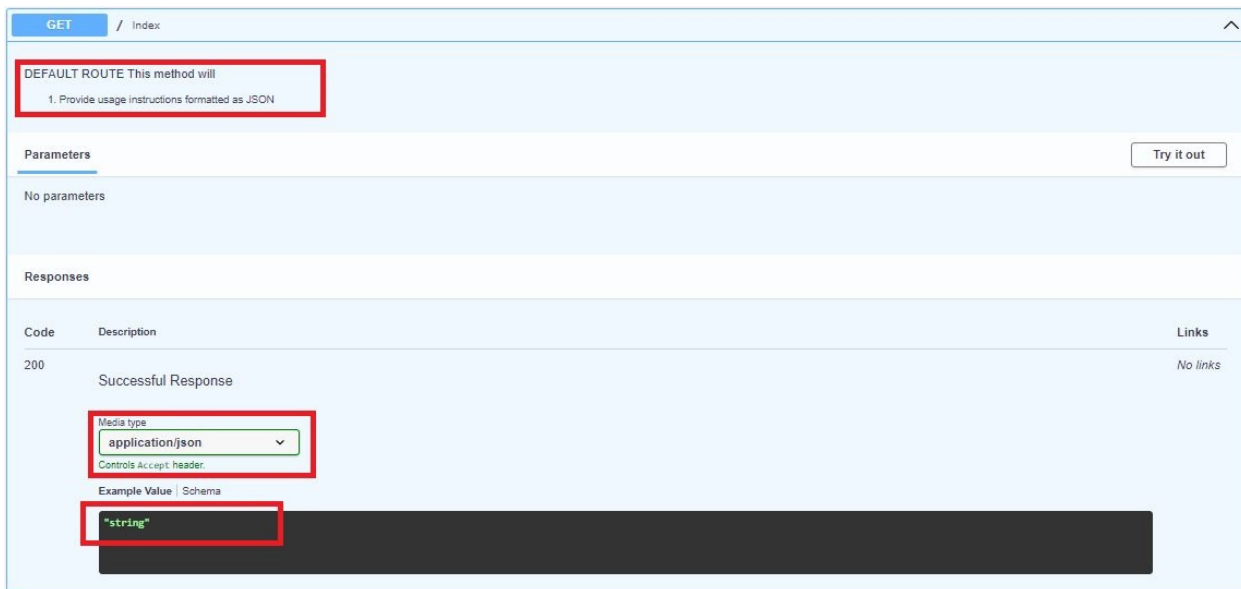
You will see a page like the one below



This is a UI interface of the automated documentation that has been generated. You can clearly see there are two paths that map to what we had coded in our “main.py”

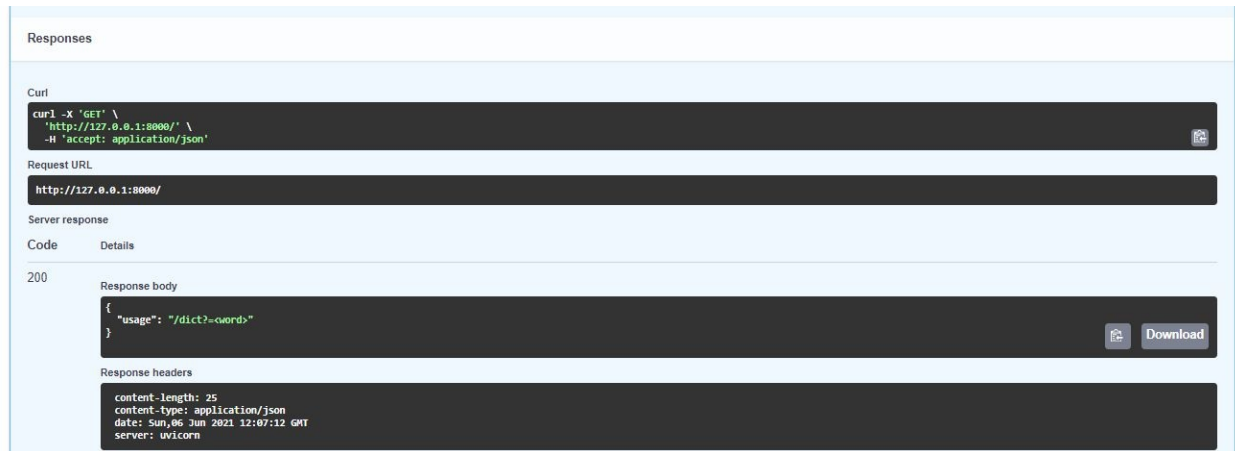
1. “ / ” : which has an “ **index** ” method and responds to “ **GET** ” method
2. “ /dic t ” : which has a “ **dictionary** ” method and responds to “ **GET** ” method

Additionally, if you expand either of the methods in the browser by clicking on the downwards facing arrowhead, you will see it has the comments that we had put in our code, see the areas highlighted in in the next figure



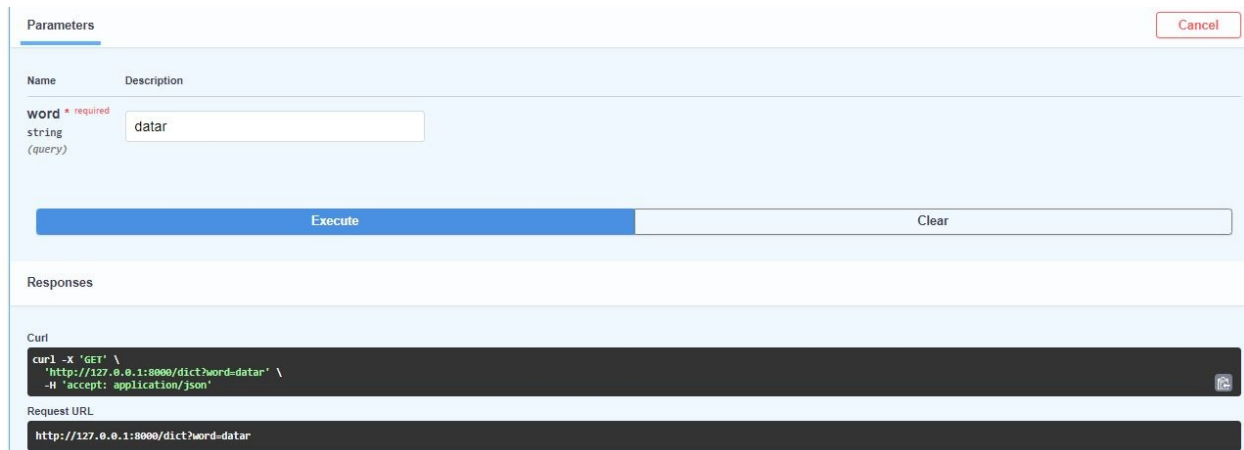
It also tells you what kind of data it sends back, and what kind of input it takes.

Now if you click on the “ *Try it out* ” button on the top left, and then click on execute, it will send a **cURL** request and present the response as well.

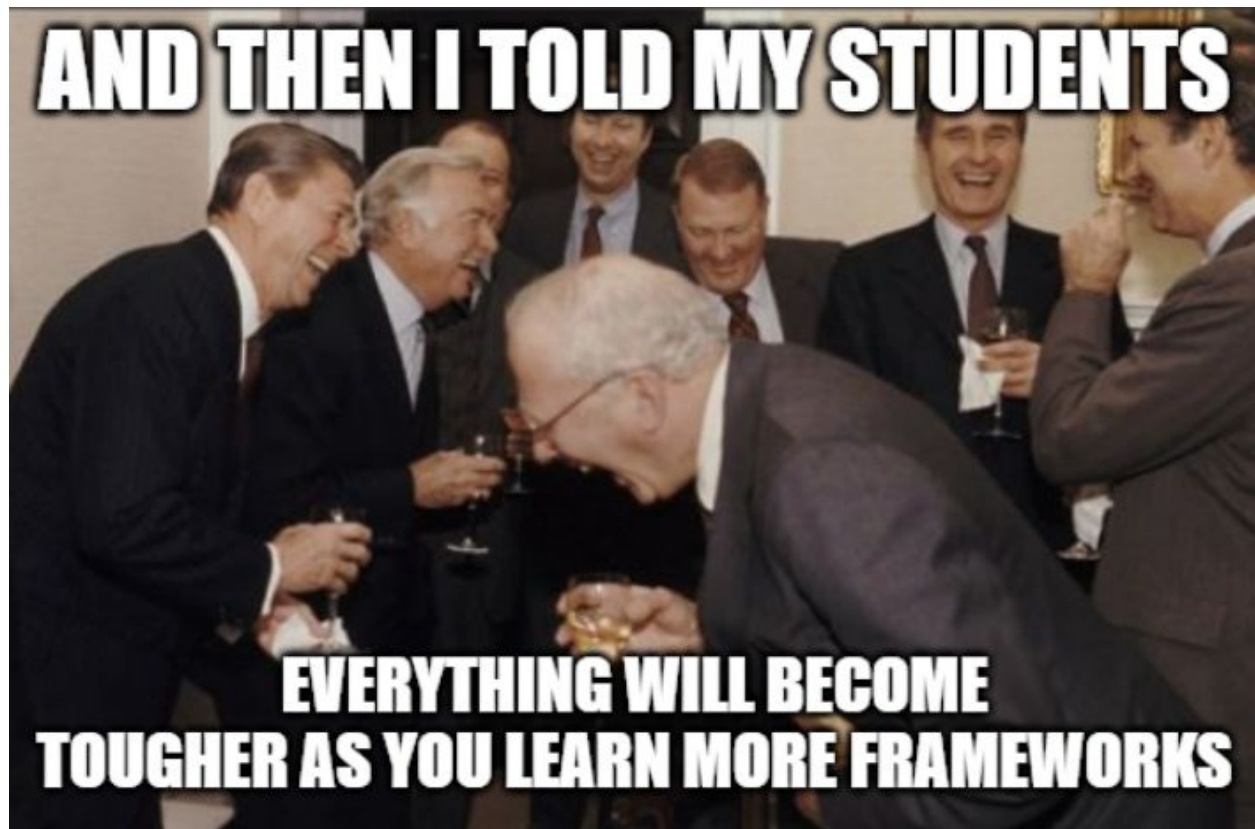


That is stunning! This makes a developer’s life a lot easier where they do not have to know **cURL** commands or write scripts to test their **APIs** during the development.

But we’re not done yet. If you expand on the second path “ *dic t* ”, and then click *Try it out* ”, the **UI** will automatically create a text box with the parameter that we had coded.



If you want to send angry tweets to [@rehanhaider](https://twitter.com/rehanhaider), about why didn’t we make it so easy in the beginning of book, I will totally understand. If not, enjoy this meme.



## 11.6 Handling a list of words

We will change our dictionary **API** to and make it able to handle multiple words. To do that we need to change the `dictionary()` method as per the below

### main.py

```
# s6-part2-dictionary-fastapi/main.py
from typing import List, Optional
from fastapi import FastAPI, Query
from fastapi.encoders import jsonable_encoder
from model.dbHandler import match_exact, match_like

app = FastAPI ()

@ app . get ( "/" )
def index ():
    """
    DEFAULT ROUTE
    This method will
    1. Provide usage instructions formatted as JSON
```

```

"""
response = { "usage" : "/dict?=<word>" }
return jsonable_encoder ( response )

@ app . get ( "/dict" )
def dictionary ( words : List [ str ] = Query ( None )):
    """
    DICTIONARY ROUTE
    This method will
    1. Accept a word from the request
    2. Try to find an exact match, and return it if found
    3. If not found, find all approximate matches and return
    """
    if not words :
        response = {
            "status" : "error" ,
            "word" : words ,
            "data" : "word not found"
        }
        return jsonable_encoder ( response )

    # Initialise the response
    response = { "words" : []}

    for word in words :
        # Try to find an exact match
        definitions = match_exact ( word )
        if definitions :
            response [ "words" ]. append ( {
                "status" : "success" ,
                "word" : word ,
                "data" : definitions
            })
        else :
            # Try to find an approximate match
            definitions = match_like ( word )
            if definitions :
                response [ "words" ]. append ( {
                    "status" : "partial" ,
                    "word" : word ,
                    "data" : definitions
                })

```

```

    })
    else :
        response [ words ]. append ( {
            "status" : "error" ,
            "word" : word ,
            "data" : "word not found"
        })

    return jsonable_encoder ( response )

```

## Explanation

As you can see, almost all the program is same, except we changed the `dictionary()` method to

```
def dictionary ( words : List [ str ] = Query ( None ) )
```

Lots to unpack here so let's first see what does the “ : ” after the variable words mean.

This is a feature introduced in **Python 3.6+** called type hinting. Let's say we declare a variable `x` that is supposed to be an `int`. It can be expressed as the declaration below

```
x : int = 15
```

This means that `x` is a variable of type “ `int` ” which is assigned value 15. Not digressing from the topic and going too deep into **Python** as a language, but type hinting helps editors, and frameworks put in tighter checking and ensure that datatypes are consistent.

Type hinting is part of the “ `typing` ” module, which is imported in the first line in the header.

Also, type hinting is NOT MANDATORY for most python programs. But in our case, because **FastAPI** does data validation, it is mandatory (sort of).

Thus, what the line means is that `word` is an argument, which is a list of strings, and whose value is the output for `Query()` method. `Query()` is used to capture querystring.

This is how we catch multiple lists in **FastAPI** .

## 11.7 Testing The API

Run the program by executing below, however, you may not need to do that unless you have manually stopped the program.

```
uvicorn main:app --reload
```

The “ *reload* ” flag means any changes that you are making to the **API** are automatically executed in real-time.

So, open the browser and navigate to the autogenerated API documentation by visiting the URL

```
http://127.0.0.1:8000/docs
```

You can use the “ *Try it out* ” feature from the browser and then press on “ *Add string item* ” to add a new line where you can form your query and send multiple words

GET /dict Dictionary

DICTIONARY ROUTE This method will

1. Accept a word from the request
2. Try to find an exact match, and return it if found
3. If not found, find all approximate matches and return

Parameters

Name	Description
words	array[string] (query)

data -

estimate -

Add string item

Execute

Alternatively, you can always do this via curl.

```
curl -X GET 'http://127.0.0.1:8000/dict?words=data&words=estimate'
```



# Chapter 12: Image filters using FastAPI

Now at the risk of becoming repetitive, let's quickly rebuild our image filter API. We start by switching to the branch that contains the problem starter kit. To do that, run the below command

```
git checkout p7-filter-fastapi
```

Like our previous examples, we simply make small changes to the **Flask** code in the following manner

## main.py

```
from bin . filters import apply_filter
from typing import List , Optional
from fastapi import FastAPI , File , UploadFile
from fastapi . encoders import jsonable_encoder
from fastapi . responses import StreamingResponse
import io

app = FastAPI ()

# Read the PIL document to find out which filters are available out-of-the box
filters_available = [
    "blur" ,
    "contour" ,
    "detail" ,
    "edge_enhance" ,
    "edge_enhance_more" ,
    "emboss" ,
    "find_edges" ,
    "sharpen" ,
    "smooth" ,
    "smooth_more" ,
]

@ app . api_route ( "/" , methods = [ "GET" , "POST" ])
def index ():
    """
```

```

Return the usage instructions that specifies
1. which filters are available, and
2. the method format
"""

response = {
    "filters_available": filters_available ,
    "usage": { "http_method": "POST" , "URL" : "<filter_available>/" },
}

return jsonable_encoder ( response )

@app . post ( "/" {filter} " )
def image_filter ( filter : str , img : UploadFile = File ( ... )):
    """
    TODO :
    1. Checks if the provided filter is available, if not, return an error
    2. Check if a file has been provided in the POST request,
       if not return an error
    3. Apply the filter using apply_filter() method from bin.filters
    4. Return the filtered image as response
    """

    if filter not in filters_available :
        response = { "error" : "incorrect filter" }
        return jsonable_encoder ( response )

    filtered_image = apply_filter ( img . file , filter )

    return StreamingResponse ( filtered_image , media_type = "image/jpeg" )

```

## Explanation

For the “ / ” route we changed “ `route()` ” method that is used in Flask, to “ `api_r` ” method which performs the same function in **FastAPI** .

The `/filter` route becomes even simpler, albeit with a few tricky changes. To receive the file as part of **POST** request, we created an argument “ `img` ” and used a method `File(...)` . This is **FastAPI** method to receive multipart form-data, or in other words, the binary data that the **cURL** sends to the **API** .

Multipart form-data is just a fancy term for organising the data that you’re sending as part of your **HTTP** request/response. It is multipart because the body is broken into small chunks, form-data signifies this is a **POST** request coming out from an **HTML** form but is typically used to signify a file is being sent.

By default, **FastAPI** returns a string, but we want to send back a file. So, we need to stream the binary data that we have to the client. For this we use the *StreamingResponse()* method.

A key part to notice, which is a big advantage with **FastAPI**, is the fact that we did not need to check if the file hasn't been sent. This is because once we declare that “ *img* ” is an uploaded file coming as multipart form-data, **FastAPI** does the check itself and throws a generic error message. We can still catch this error and send a more specific response but not required really.

## 12.1 Bonus: Deployment to Heroku

Use the same instructions as in chapter 9. However, deploying **FastAPI** requires the Procfile command to be

```
web: uvicorn main:app --host=0.0.0.0 --port=${PORT:-5000}
```

# Chapter 13: FastAPI async / await

We talked about asynchronous programming a lot in previously, but so far, we have been using **FastAPI** exactly like **Flask** . In this chapter, we will make our filter **API** asynchronous and test if it improves functionality.

## 13.1 Making the program asynchronous

To begin you should know, starting **Python 3.6+** , `async/await` keywords are available natively which makes it quite intuitive to define asynchronous code. This makes it look like a normal program and make it asynchronous by adding `async` declaration and awaiting at the right moments.

But what are `async/await`? Let's look at the below pseudocode

```
async def get_dish ( orders ):
    # Do some asynchronous food cooking
    return dishes
```

**async**: is a keyword to declare an asynchronous function. By adding `async` we tell Python that inside the function, it must be aware of multiple dishes cooking at the same time and be ready to move to a different dish while one is cooking before coming back.

When you want to call a function that starts with **async** , you must **await** it. So, if we tried to run

```
dishes = get_dish ( orders )
```

It will not work, because `get_dish()` was defined as an `async` function so it is expecting to be awaited till it completes its coroutines

Instead you need to run the following o get the results

```
dishes = await get_dish ( orders )
```

Keep in mind, if you call an asynchronous method it must be awaited otherwise it will throw an error.

However, **await** keyword can only be used inside an **async** function. What that means is you get into a chicken and egg situation where you need to create an

async function to call another async function, but to execute the first async function you need another async function.

Luckily, in **FastAPI** you don't need to worry about that as the path operation handles that for you.

Outside of that, you will need to run the coroutines using an event loop, we will see an example in our tests.

Finally, you heard the term **coroutine** a few times above, it is just a very fancy term for what is being returned by any **async** function . Sometimes, **coroutines** term is also used to summarise using asynchronous code with **async** and **await**

## 13.2 Making the filter API asynchronous

The real advantage of asynchronous program doesn't lie just in the program itself. The program can also do **multitasking / multithreading** by itself. However, the advantage is in being able to respond to several requests that are arriving at the same time ( **parallelism** ). As we proceed, we will test this and see how our **API** performance improves by almost 50-80%.

Then change the `index()` function definition in `main.py` as per below

```
async def index():
    """
    Return the usage instructions that specifies
    1. which filters are available, and
    2. the method format
    """
    response = {
        "filters_available": filters_available,
        "usage": { "http_method": "POST", "URL": "/<filter_available>/" },
    }
    return jsonable_encoder(response)
```

### 13.2.1 Test scenarios

Our scenario that we send **10 API** requests to our **API** endpoint under the following conditions

1. *Sequential API calls to sequential API endpoint*

2. *Asynchronous API calls to sequential API endpoint*
3. *Sequential API calls to asynchronous API endpoint*
4. *Asynchronous API calls to asynchronous API endpoint*

## 13.3 Test Scenario 1

### Sequential API calls to sequential API endpoint

Switch back to the filter branch by running, assuming this is the branch you are using for the API

```
git checkout p7-filter-fastapi
```

Start the program by running

```
uvicorn main:app --reload
```

Now, before you open jupyter notebook we need to install a few libraries to the system-wide python installation. By default, the **Jupyter Notebook** will use the system-wide **Python** installation. Install the following libraries using the commands below one

```
Pip3 install aiohttp[speedups]
```

We will use *aiohttp* for our asynchronous **API** calls.

Now open a new **Jupyter Notebook** and import the libraries required by running the below

```
# Jupyter Notebook/ In[1]
import aiohttp
import time
import asyncio
import requests
```

Then let's send the **10 API** calls and measure the time taken for the operation to complete by running the below

```
# Jupyter Notebook/ In[2]
URL = "http://127.0.0.1:8000"
start = time.time ()
results = []
```

```
for i in range ( 10 ):
    results . append ( requests . post ( URL ).content)
print ( f "Time: { time . time () - start } " )
```

### Explanation :

This snippet will execute the below steps 10 times sequentially

1. Send an API call
2. Capture the response
3. Add the contents response into a list named “ **result s** ”

For me it took ~14.0 seconds to complete.

## 13.4 Test Scenario 2

### Asynchronous API calls to sequential API endpoint

14 seconds is not bad, can we do better? What if we sent all **10 API** requests at the same time?

It will improve the performance at the client end ( **Jupyter Notebook** ) but our **API** server is still sequential and will respond to the APIs call one sequentially.

To test this, we first need to create a function that sends this request using *aiohttp*

```
# Jupyter Notebook/ In[3]
URLS = []
for i in range ( 10 ):
    URLS . append ( "http://127.0.0.1:8000" )

async def test ( URL ):
    async with aiohttp . ClientSession () as session :
        async with session . post ( URL ) as resp :
            return await resp . text ()
```

Then let's execute the above function by executing it all at once by running the below

```
# Jupyter Notebook/ In[4]
start = time . time ()
loop = asyncio . get_event_loop ()
results = await asyncio . gather ( *[ loop . create_task ( test ( URL )) for URL
```

```
URLS ], return_exceptions = True )
print ( f "Time: { time . time () - start } " )
```

### Explanation :

**NOTE: This snippet will only run in Jupyter Notebook.** The reasons are a bit convoluted but if you try to run it as a program directly, it will throw an error stating no event loop exists.

The short explanation is that Jupyter Notebooks run in an async loop to begin with, and you cannot create an async event loop inside an async event loop. Which is why we used,

```
loop = asyncio . get_event_loop ()
```

To fetch the existing loop. Then we used `asyncio.gather()` to execute a loop of activities.

This snippet sent **10 API** calls simultaneously to our **API** , but considering our **A** is still synchronous, it will handle and response one at the time, but you will see a reduction at the client end.

For me this test took ~8.3 seconds

## 13.5 Test scenario 3

### Sequential API calls to asynchronous API endpoint

To make our program asynchronous, all we need to do is add the keyword **async** before the `index()` function definition as shown below

```
@ app . api_route ( "/" , methods = [ "GET" , "POST" ] )
async def index () :
    """
    Return the usage instructions that specifies
    1. which filters are available, and
    2. the method format
    """
    response = {
        "filters_available" : filters_available ,
        "usage" : { "http_method" : "POST" , "URL" : "/<filter_available>/" } ,
    }
    return jsonable_encoder ( response )
```



---

Go back to **Jupyter Notebook** and run the below snippet again

```
# Jupyter Notebook/ In[2]
URL = "http://127.0.0.1:8000"
start = time.time ()
results = []
for i in range ( 10 ):
    results . append ( requests . post ( URL ).content)
print ( f "Time: { time.time () - start } " )
```

This time, while our **API** is capable of performing asynchronously, but it gets only 1 request at a time, so the time taken to perform these will remain very similar.

My execution completed in ~13.4 seconds, very close to the 14 seconds in **Test Scenario 1**.

## 13.6 Test Scenario 4

### Asynchronous API calls to asynchronous API endpoint

Our **API** is already capable of handling tasks asynchronously. To test this let's re-run this snippet from **Test Scenario 2**

```
# Jupyter Notebook/ In[4]
start = time . time ()
loop = asyncio . get_event_loop ()
results = await asyncio . gather (*[ loop . create_task ( test ( URL )) for URL in URLS ], retu
True )
print ( f "Time: { time . time () - start } " )
```

Just as in **Test Scenario 2** , we are sending **10 API** calls simultaneously, but this time our **API** will execute these not in sequence, but asynchronously thus executing faster.

In my case it took, drum roll please, 5 seconds!!

## 13.7 Conclusion

We saw the execution time going down from 14 seconds down to 5 seconds, almost 64% improvement, even though this example was rather simplistic but serves the purpose to show the kind of performance improvements that can be designed with asynchronous programming.

## 13.8 Bonus Challenge

Make the entire filter **API** asynchronous with the following specifications

1. All functions are async with await at the right statements
2. The **API** should be able to take multiple images, apply the filter and return zipped file with all images

Send in your **GitHub** repository link to [student@CloudBytes.dev](mailto:student@CloudBytes.dev) with Subject "**Async Bonus Challenge**". The top 3 designs before 31<sup>st</sup> August will get \$200 Amazon gift vouchers.

# Chapter 14: Making a TODO API

**TODO** apps are a cornerstone of learning development, that is why you will see them everywhere. Internet is full of countless tutorials on how to develop them.

In fact, developing a **TODO** app is considered landmark achievement for any developer. So unsurprisingly, we're going to build one too. The only difference being, our **TODO** app will be a simple **Web API** that you can interact with

## 14.1 API Specifications

Our simple **TODO** app should be able to do the following

1. *Create a new task, the task can contain*
  - a. **Serial Number** : a unique identifier for the task
  - b. **Summary** : Description of the task
  - c. **Priority** : With scale from 1 to 4, 1 being highest
2. *Ability to modify the task Summary, or Priority,*
3. *Ability to delete a task*

Finally, for sake of simplicity, we will store the tasks in a simple **JSON** file, however, this can easily be changes to a **Relational DB** ( **MySQL** , **Postgres** , etc) or **No-SQL** database ( **MongoDB** , etc.)

## 14.2 Initialise the environment

Follow the instructions from **Chapter 3** and use the below the starter kit for this exercise

```
https://github.com/CloudBytesDotDev/todo-api-fastapi.git
```

You need to do the following,

1. *Fork the repository to your account*
2. *Clone the repository to your PC*
3. *Create a Python virtual environment*
4. *Activate the environment*
5. *Install the dependencies*

## 14.3 Understanding the starter kit

**Repository Structure** : Our starter kit contains the following templates

```
.
├── main.py
├── data
│   └── tasks.json
├── model
│   ├── model.py
│   └── taskman.py
└── requirements.txt
```

## main.py

```
from fastapi import FastAPI
from datetime import datetime
from typing import Optional

from fastapi.encoders import jsonable_encoder
from model.model import Task, TaskList
import model.taskman as taskman

app = FastAPI()

@app.get("/api/tasks")
async def get_tasks():
    """ TODO
    Fetch the list of all tasks
    """
    return "TODO"

@app.get("/api/tasks/{id}")
async def get_task(id: int):
    """ TODO
    Fetch the task by id
    """
    return "TODO"

@app.post("/api/tasks/create")
async def create_task(task: Task):
    """ TODO
    1. Create a new task and
    2. Return the details of task
```

```

"""
    return "TODO"

@app.put("/api/tasks/{id}/update")
async def update_task(id: int, task: Task):
    """ TODO
    1. Update the task by id
    2. Return the updated task
    """
    return "TODO"

@app.delete("/api/tasks/{id}/delete")
async def delete_task(id: int):
    """ TODO
    1. Delete the task by id
    2. Return a confirmation of deletion
    """
    return "TODO"

```

Our main program contains 5 paths

1. “ `/api/tasks` ”: Will respond with a list of all tasks
2. “ `/api/tasks/{id}` ”: By specifying one can fetch a task by specifying the ID
3. “ `/api/tasks/create` ”: It creates a task and returns it along with ID
4. “ `/api/tasks/{id}/update` ”: Update a specific task by ID
5. “ `/api/tasks/{id}/delete` ”: Delete the task by ID

Additionally, in the methods `create_task` and `update_task`, you would notice an argument of the type “ `Task` ”. `Task` is a class that we will define that describes the data structure of a task, i.e. summary and priority.

## model/model.py

Our data will be organised in the following way

1. A tasklist that will contain an ID of the task along with task
2. A task that will contain summary and priority

The above data structure has been defined in as two classes that form the model of our data

```

from pydantic import BaseModel
from datetime import datetime
from typing import NewType, Optional

# Declare a new type of variable ID
ID = NewType("id", int)

class Task(BaseModel):
    """Definition of components of a task"""
    summary: str
    priority: int
    # due_date: Optional[datetime]

class TaskList(BaseModel):
    """Definition of the TaskList"""
    id: ID
    task: Task

```

But these aren't ordinary classes, and instead inherit the properties of *BaseModel* from a library called **Pydantic**. More on that later, but **TL;DR** version is **Pydantic** enforces that the data that you receive is validated as per the model schema, i.e. if you ask for an integer, you get an integer an integer.

### model/taskman.py

This is our task handler. We could have built this inside the routes themselves but the reason why they are split is because of top-down design paradigms where one can replace the handler which currently handles **JSON**, to a handler that handles either a **Relational DB** or a **NoSQL DB** without changing anything in the routes.

Or if we wanted to change the routes in future you don't have to change anything in task handler.

```

from model.model import Task, TaskList
import json
from pydantic import parse_file_as
from typing import List, Optional

filepath = "data/tasks.json"

```

```

async def data_to_json ( data : List ):
    """
    TODO
    1. Take input data, of a list of tasks
    2. Write the data into a json file (tasks.json)
    """
    pass

async def get_tasks ( id : Optional [ int ] = 0 ):
    """
    TODO
    1. Fetch all tasks if no argument (id) provided
    2. Else fetch the task by id provided
    """
    return "response"

async def create_task ( new_task : Task ):
    """
    TODO :
    1. Create a new task and add it to the list of tasks
    2. Write the updated tasklist to file
    """
    return "id"

async def delete_task ( id ):
    """
    TODO :
    1. Delete the task by id provided
    """
    return "id"

async def update_task ( id : int , new_task : Task ):
    """
    TODO
    1. Update the task by id based on new task details
    2. write the updated tasklist to file
    """
    return "id"

```

---

These mirror the actions that will be performed on the list of tasks.

### **data/tasks.json**

This file contains a few sample tasks. The tasks are stored as a **JSON** list similar to the below

```
[
  {
    "id" : 1 ,
    "task" : {
      "summary" : "This is a summary of task" ,
      "priority" : 1
    }
  },
  {
    "id" : 3 ,
    "task" : {
      "summary" : "new update" ,
      "priority" : 2
    }
  }
]
```

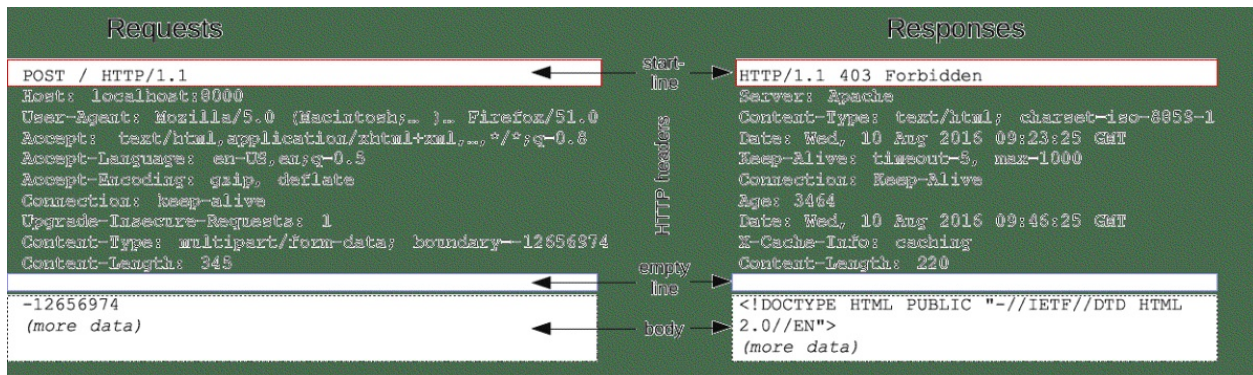
## **14.4 Request Body and Data Validation**

For a moment let's go back to **Chapter 1** and recall that the response that we got when we sent the **HTTP** request to **CoinDesk** . The first line of the response header looked something like this

```
HTTP/ 1.1 200 OK
```

This start-line tells us the **HTTP** standards being used to **encode** the **HTTP** request packet being sent, in this case v1.1 is being used. Additionally, HTTP messages can also contain header (top of the message in our example) and body (the **JSON** response we got).





Content in headers and body technically can be almost anything, but there are standards defined elsewhere that limits what kind of headers you can use. In our filter **API**, we sent a “ *mime-type* ”, we could have called it “ *wolverine-type* ” for a purpose but the receiver of the response, typically a browser, will interpret “ *image/jpeg* ” as an image of JPEG format and display it in browser. If we really changed the “ *mime-type* ” to “ *wolverine-type* ” the browser wouldn’t know what to do with that and ask you to save the file instead of displaying it.

The conclusion is there are only so many keywords that are used in headers as a convention, but the body is a different story altogether.

The body of a **HTTP** message is like a blank slate. But again, as a convention the world follows some loose guidelines such as, as we explained previously, most data exchanges between front-end and back-end are typically in **JSON**.



But what is included in these **JSON** data? Could be anything.

So, coming back to our **TODO API** , it could be sent a **JSON** string that contains unnecessary information such as with keys that are misspelled. E.g. the **API** is supposed to receive the summary as string and priority as integer, but what if someone decided to send the priority by writing the literal, e.g. “one” instead. We would need to spend **CPU** cycles to figure out that the data that we received is not in the format we expected it to be.

This brings the need for data validation. What if we could force the user to send us a **JSON** that must contain “ **summary** ” that is string, and a “ **priority** ” that is integer and nothing else?

That is exactly purpose of **Pydantic** .

### 14.4.1 Pydantic

**Pydantic** is a must-know **Python** library for people who juggle data around either for building **APIs** or for data science.

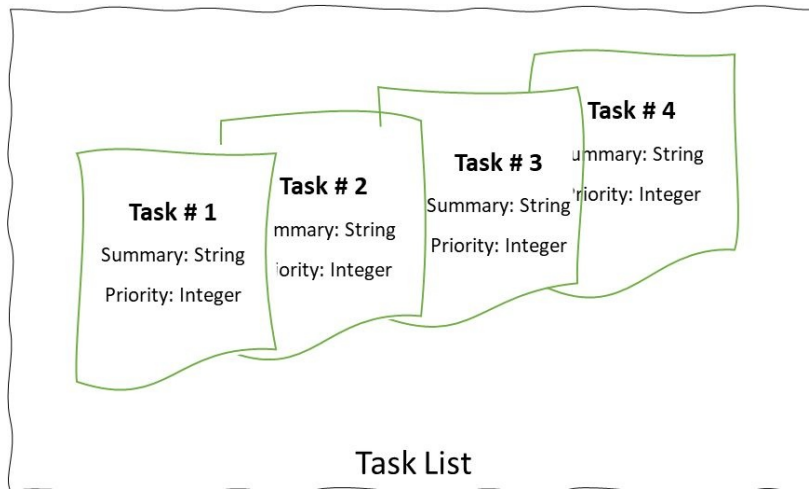
**Pydantic** solves two problems

1. *As a developer, how do I know what kind of data is supposed to be sent to an **API** ?*
2. *How do I prevent typos in keys or use the correct keys in a **JSON** or dictionary during development?*

**Pydantic** is a parsing and validation library that does both respectively. How does it do that is explained in the next section.

## 14.5 Data Model for **TODO**

The intent is our data model to look like below



That is, we are defining a “*task*” as a data structure that contains “*summary*” and “*priority*”.

We declare this in our model as

```
class Task ( BaseModel ):
    """Definition of components of a task"""
    summary : str
    priority : int
```

Then we create a list of tasks that contains a task “*id*” and the task details. We define this as

```
class TaskList ( BaseModel ):
    """Definition of the TaskList"""
    id : ID
    task : Task
```

Finally, in this case we will use an integer as the ID, so we declare the ID as a special type using the “*typing*” library, so our model becomes

### model/model.py

```
from pydantic import BaseModel
from typing import NewType, Optional

# Declare a new type of variable ID
ID = NewType ( "id", int )
```

```

class Task ( BaseModel ):
    """
    Definition of components of a task
    """
    summary : str
    priority : int

class TaskList ( BaseModel ):
    """
    Definition of the TaskList
    """
    id : ID
    task : Task

```

## 14.6 Task manager

Following best practices, we will split any non-API data handling and response work to a sub-module, in this case our task manager or taskman will perform the actual work required. The taskman will look like below

### model/taskman.py

```

from model.model import Task, TaskList
import json
from pydantic import parse_file_as
from typing import List, Optional

filepath = "data/tasks.json"

async def data_to_json ( data : List ):
    """ TODO
    1. Take input data, of a list of tasks
    2. Write the data into a json file (tasks.json)
    """
    data = json.dumps ( data )
    with open ( filepath , "w" ) as file :
        file . write ( data )

async def get_tasks ( id : Optional [ int ] = 0 ):

```

```

""" TODO
1. Fetch all tasks if no argument (id) provided
2. Else fetch the task by id provided
"""

tasks = parse_file_as ( List [ TaskList ], "data/tasks.json" )
data = { task . id : task . dict () for task in tasks }
response = data if id == 0 else data [ id ]
return response

async def create_task ( new_task : Task ):
    """ TODO
    1. Create a new task and add it to the list of tasks
    2. Write the updated tasklist to file
    """

    tasks = parse_file_as ( List [ TaskList ], "data/tasks.json" )
    id = max ( [ task . id for task in tasks ] ) + 1
    tasks . append ( TaskList ( id = id , task = new_task ))
    data = [ task . dict () for task in tasks ]
    await data_to_json ( data )
    return id

async def delete_task ( id ):
    """ TODO
    1. Delete the task by id provided
    """

    tasks = parse_file_as ( List [ TaskList ], "data/tasks.json" )
    tasks = [ task for task in tasks if task . id != id ]
    data = [ task . dict () for task in tasks ]
    await data_to_json ( data )
    return id

async def update_task ( id : int , new_task : Task ):
    """ TODO
    1. Update the task by id based on new task details
    2. write the updated tasklist to file
    """

    tasks = parse_file_as ( List [ TaskList ], "data/tasks.json" )
    data = [ task . dict () for task in tasks ]
    for task in data :

```

```

if task [ "id" ] == id :
    task [ "task" ] = new_task . dict ()

await data_to_json ( data )
return id

```

## Explanation

Our taskman contain 6 methods

First, `data_to_json()` that takes a list of Tasks, converts it to **JSON** and then writes it to a file.

Second, `get_tasks()` that accepts an argument `id` which is an integer that identifies task number. By default, it is set to be 0. If the id remains 0, the function will return the entire list of tasks, if not, it will send only the task id that is specified.

To do that, the method first reads the `tasks.json` file that contains the list of tasks, and parses it as the object of the class `TaskList` that we have defined, converts the data to a dictionary and then selects the appropriate response based on the value of **id variable** .

Third, `create_task()` that accepts an argument of type “ `Task` ”. Remember `Task` is a class that we defined in our model that specifies the data structure each task should have. Once a task is received as a class object, the method first reads the `tasks.json` file that contains the list of tasks, and parses it as the object of the class `TaskList` that we have defined, then adds the new task to the list of tasks, converts the data to a dictionary and uses `data_to_json()` method to write it back to the file.

Fourth, `delete_task()` , as the name suggest delete the task as per id provided. We achieve this by simply first parsing the `json.task` file as `TaskList` , then we copy all tasks except the one whose id matches the one we received. After that we again convert this list of tasks into a dictionary and write it to a file

Fifth, `update_task()` , updates the task as specified by the id, with the new task.

Now our taskman should be able to

1. Create a new task
2. Return a particular task
3. Return all tasks
4. Update a particular task

5. *And delete a particular task*

## 14.7 Building the API

Let's look at the headers first, apart from the modules from **FastAPI** library, we also import the Task specification and taskman model

```
from fastapi import FastAPI

from fastapi.encoders import jsonable_encoder
from model.model import Task
import model.taskman as taskman
```

We use the methods we just defined in our taskman to perform **CRUD** operations on a low-fi json database.

### 14.7.1 Create a task

We have already defined the logic for creating new tasks, all we need to do is

1. *Force the user to send a **JSON** that is in the format of “ **Tas k** ”. We will test this later*
2. *Use the task data and call `create_task()` method from taskman to create a new task*

```
@app.post("/api/tasks/create")
async def create_task(task: Task):
    """ TODO
    1. Create a new task and
    2. Return the details of task
    """
    id = await taskman.create_task(task)
    return await taskman.get_tasks(id)
```

### 14.7.2 Viewing a task

To view a particular task, we define it as a path variable.

```
@app.get("/api/tasks/{id}")
async def get_task(id: int):
    """ TODO
    Fetch the task by id
    """
```

```
return await taskman.get_tasks(id)
```

So, visiting the URL “ /api/tasks/ 3 ” will show us the task #3.

### 14.7.3 View all tasks

The api/tasks route when called will send the list of all tasks that are currently available. To do this, all we need to do now is call the `get_tasks()` method from taskman.

```
@app.get("/api/tasks")
async def get_tasks():
    """ TODO
    Fetch the list of all tasks
    """
    return await taskman.get_tasks()
```

### 14.7.4 Update a task

Similar to the creation of new task, we ask the API consumer to provide us updated task in Task schema and use that to update the existing task

```
@app.put("/api/tasks/{id}/update")
async def update_task(id: int, task: Task):
    """ TODO
    1. Update the task by id
    2. Return the updated task
    """
    await taskman.update_task(id, task)
    return await taskman.get_tasks(id)
```

### 14.7.5 Delete a task

We ask the API consumer for the task id that needs to be deleted and use the taskman to delete the task.

```
@app.delete("/api/tasks/{id}/delete")
async def delete_task(id: int):
    """ TODO
    1. Delete the task by id
```



```

2. Return a confirmation of deletion
"""
    id = await taskman.delete_task(id)
    response = {id: "Task successfully deleted"}
    return jsonable_encoder(response)

```

## 14.8 Final API design

So, the final program will look like the below

main.py

```

from fastapi import FastAPI

from fastapi.encoders import jsonable_encoder
from model.model import Task
import model.taskman as taskman

app = FastAPI()

@app.get("/api/tasks")
async def get_tasks():
    """ TODO
    Fetch the list of all tasks
    """
    return await taskman.get_tasks()

@app.get("/api/tasks/{id}")
async def get_task(id: int):
    """ TODO
    Fetch the task by id
    """
    return await taskman.get_tasks(id)

@app.post("/api/tasks/create")
async def create_task(task: Task):
    """ TODO
    1. Create a new task and
    2. Return the details of task
    """

```

```

id = await taskman.create_task(task)
return await taskman.get_tasks(id)

@app.put("/api/tasks/{id}/update")
async def update_task(id: int, task: Task):
    """ TODO
    1. Update the task by id
    2. Return the updated task
    """
    await taskman.update_task(id, task)
    return await taskman.get_tasks(id)

@app.delete("/api/tasks/{id}/delete")
async def delete_task(id: int):
    """ TODO
    1. Delete the task by id
    2. Return a confirmation of deletion
    """
    id = await taskman.delete_task(id)
    response = {id: "Task successfully deleted"}
    return jsonable_encoder(response)

```

## 14.9 Testing the API

By now you know how to test the **API**, but what I want to show you is the power of **FastAPI** and **Pydantic** using **Python** type-hinting.

Let's fire up the **Swagger UI** by running the app and then opening the **URL**

```
http://127.0.0.1:8000/docs
```

This will bring you to the **API** documentation where you can see all the 5 paths that we have defined as well as the **HTTP** method they are using.

default		^
GET	/api/tasks Get Tasks	v
GET	/api/tasks/{id} Get Task	v
POST	/api/tasks/create Create Task	v
PUT	/api/tasks/{id}/update Update Task	v
DELETE	/api/tasks/{id}/delete Delete Task	v

Let's try to create a new task by clicking on the *Create Task* section, then click on 'Try it out'. This will bring up a message box that says, "Request body". So, our API documentation tells us that this path is expecting a request body with a very particular format, i.e.

```
{
  "summary" : "string" ,
  "priority" : 0
}
```

In this **JSON** formatted response body you are allowed to change the values as long as they are the data type suggested, in this case string for summary, integer for priority. If you were to either change the keys or the data type of values, e.g. change "summary" to "summaries", the server will reject your request without invoking the program.

```
INFO: 127.0.0.1:40382 - "POST /api/tasks/create HTTP/1.1" 422 Unprocessable Entity
```

This *Error 422* is generated by the server because our server recognised that the API was expecting a particular format of data and since it did not match it, it simply returned a *422 Error* stating that an *Unprocessable Entity* was supplied.

Our program did not throw any error because our program never executed this API request.

Why is this impressive? Because it reduces the development time significantly as I do not need to catch every possible error that could be encountered by incorrect data since the data being received is being validated before execution saving CPU cycles as well.

But that in no way is not an invitation to write code without error handling!

## 14.10 Where to from here?

What we learnt so far has been introduction to how **APIs** are created, but this should provide the foundational knowledge for you to learn more advanced concepts that goes into developing **APIs**. But the good thing is you can use these foundations and layer over additional blocks to make you **API** production ready.

E.g. you can easily configure user management and authentication on top of **API** or add error handling to ensure the app doesn't break.



# Au Revoir

Did this book help you in some way? If so, I'd love to hear about it. Honest reviews help readers find the right book for their needs.

Is there any other topics you'd like me to cover? If you didn't get what you came looking for, tweet to me at [@rehanhaider](https://twitter.com/rehanhaider) . I will as always be happy to provide guidance and constructive feedback will make this book better for future readers.

# About the Author

Rehan Haider loves breaking things apart and reassembling them back (mostly) successfully. He wrote his first program in Visual Basic as a pre-teen when Wikipedia didn't exist, only reference available was the F1 documentation and his own reverse engineering acumen.

Rehan has a 4-year Bachelor's Degree in Computer Science, a Bachelor's Degree in Mathematics (Hons.), and an MBA in IT & Systems Management.

At present, Rehan is an Enterprise Solution Architect working for a global consultancy MNC and has already designed solutions for over 250 clients ranging, most of them belonging to Fortune 500, consulting on Cloud, DevOps, and Enterprise applications, and IT Systems.

.  
He is a former teacher who taught high school and college mathematics specialising Multivariate Calculus, Operations Research and Optimisation Theory, along with C & Python programming.

You can Tweet memes at him at <https://twitter.com/rehanhaider>

# Acknowledgements

Icons used in this were made by Pixel perfect from [www.flaticon.com](http://www.flaticon.com)