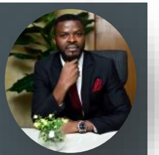
FULL-STACK NANODEGREE SESSION 6

AJIROGHENE SUNDAY



1. WHAT DOES **API** STANDS FOR

A	B	C	D	E
Application Interfaces	Application International Standard	Application Programing Interface	Application Program Interface	None of the above

2. THE FOLLOWING ARE FACTURES OF HTTP PROTOCOL EXCEPT ONE.

A	Connectionless
B	Stateless
C	Session/Caching
D	Media Independent
E	RESTful

Hypertext Transfer Protocol (HTTP) is a protocol that provides a standardized way for computers to communicate with each other. It has been the foundation for data communication over the internet since 1990 and is integral to understanding how client-server communication functions.

In this lesson, we'll discuss key features and elements of HTTP.

Features:

- **Connectionless:** When a request is sent, the client opens the connection; once a response is received, the client closes the connection. The client and server only maintain a connection during the response and request. Future responses are made on a new connection.
- **Stateless:** There is no dependency between successive requests.
- **Not Sessionless:** Utilizing headers and cookies, sessions can be created to allow each HTTP request to share the same context.
- **Media Independent:** Any type of data can be sent over HTTP as long as both the client and server know how to handle the data format. In our case, we'll use JSON.

3 WHAT IS THE DIFFERENCE BETWEEN URL VS URI (CHOOSE ALL THAT APPLIES)

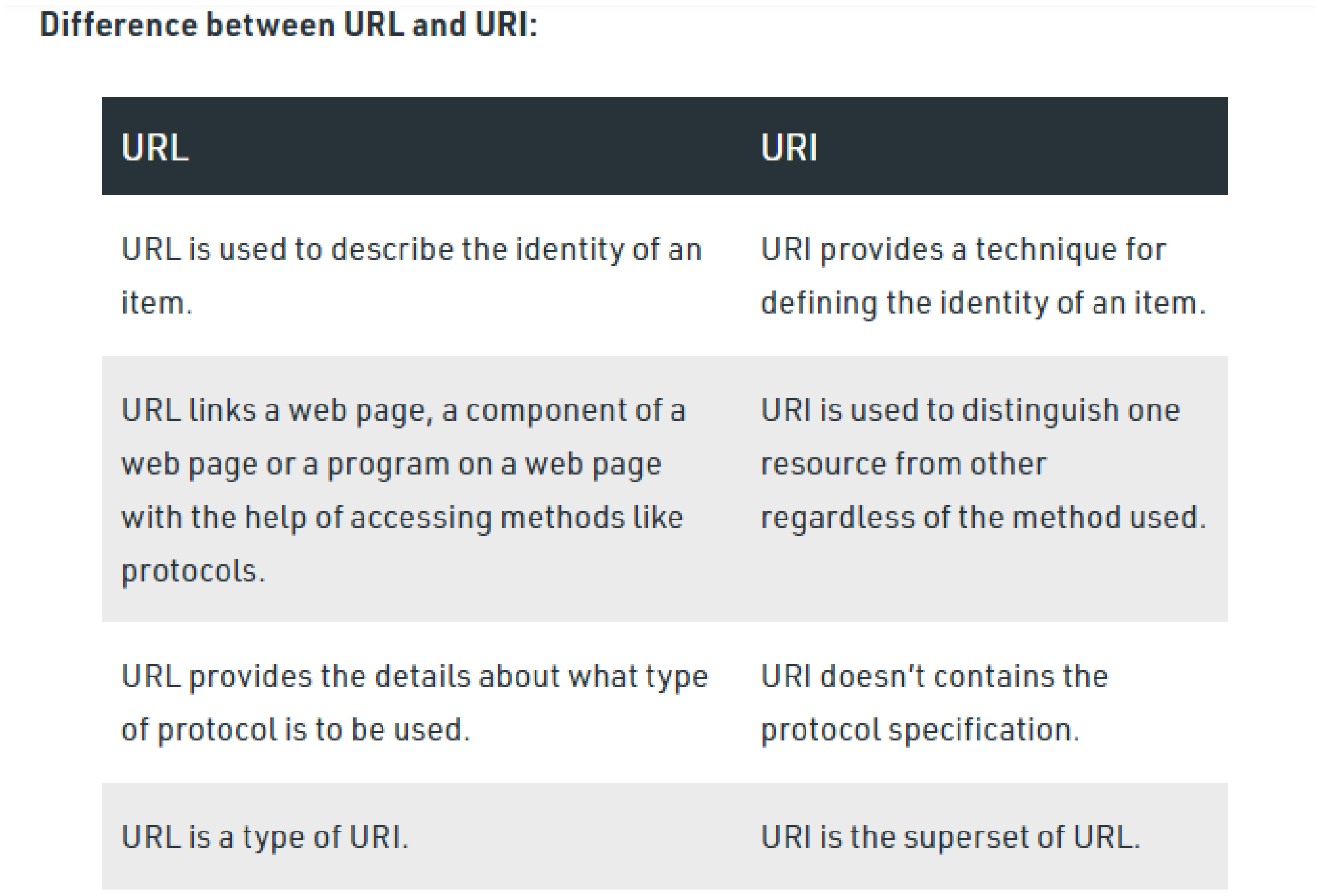
A. URL is domain specific while URI is Resource specific

B. URI identifies resource for which URL is a subset while URL locate a resource based on the resource address

C. URL is a type of URI while URI is the superset of URL.

D. URI can be accessed only by Java and Python while URL can be accessed by any programming language

E. URL is used to describe the identity of an item, while URI provides a technique for defining the identity of an item.



Difference between URL and URI:

URL	URI
URL is used to describe the identity of an item.	URI provides a technique for defining the identity of an item.
URL links a web page, a component of a web page or a program on a web page with the help of accessing methods like protocols.	URI is used to distinguish one resource from other regardless of the method used.
URL provides the details about what type of protocol is to be used.	URI doesn't contains the protocol specification.
URL is a type of URI.	URI is the superset of URL.

4 WHAT DOES IP STANDS FOR

A. International protocol

B. Internet Plagiarism

C. International Plagiarism

D. Internet Protocol

E. None of the Above

5 WHAT DOES TCP STANDS FOR

A	Transmission Controller protocol
B	Transmission Control protocol
C	Transported-data Collector protocol
D	A set of rule for data transmission
E	None of the above

6 BEST PRACTISE ABOUT ORGANISING API INCLUDES? (SELECT ALL THAT APPLIES)

A	Should be Intuitive and Organize by resource
B	None of the above
C	Use noun in path not verb
D	Keep a consistent theme
E	Don’t make them complex

Organizing API Endpoints

- Should be intuitive
- Organize by resource
- Use nouns in the path, not verbs

BAD:

- https://example.com/create-tasks
- https://example.com/send

GOOD:

- https://example.com/tasks
- https://example.com/messages

- Keep a consistent scheme
  - Plural nouns for collections
  - Use parameters to specify a specific item

BAD:

- https://example.com/user/task/

GOOD:

- https://example.com/users/1/tasks

- Don't make them too complex or lengthy
  - No longer than *collection/item/collection*

BAD:

- https://example.com/users/1/tasks/8/notes

GOOD:

- https://example.com/tasks/8/notes
- https://example.com/users/1/tasks

Table of contents

Introduction

In today’s lecture, we will cover

- API endpoints
- Cors
- Flask Error Handling
- API Testing

01

Recap

Application Programming Interface (API)

API stands for

Application **P**rogramming **I**nterface.

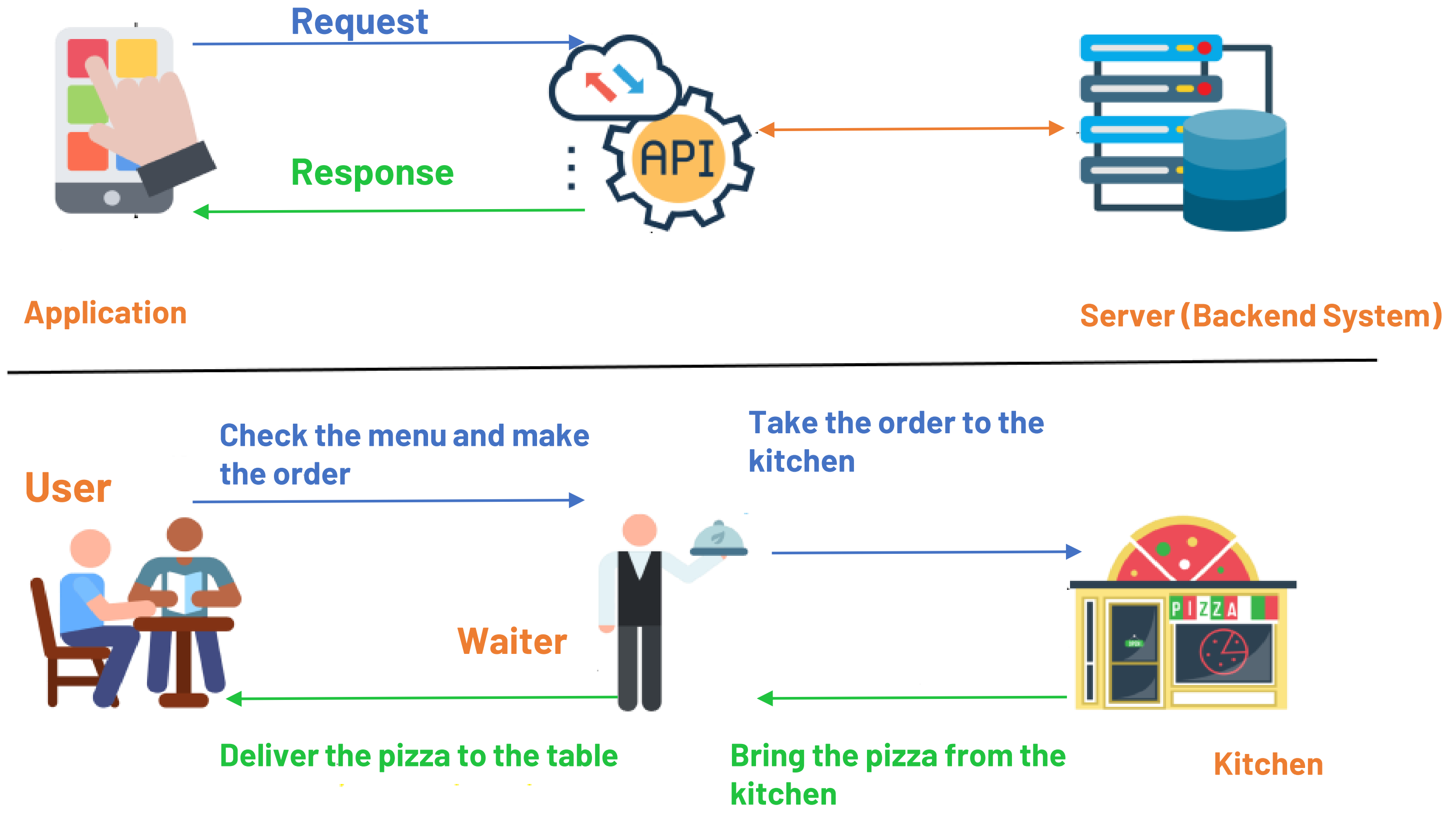
A Web API is an application programming interface for
the Web.

A Browser API can extend the functionality of a web browser.

A Server API can extend the functionality of a web server.

An API is a set of programming code that enables data transmission between one software product and another

How do APIs work?


Request
Response
API
Application
Server (Backend System)
Check the menu and make the order
Take the order to the kitchen
User
Waiter
P I Z Z A
Deliver the pizza to the table
Bring the pizza from the kitchen
Kitchen

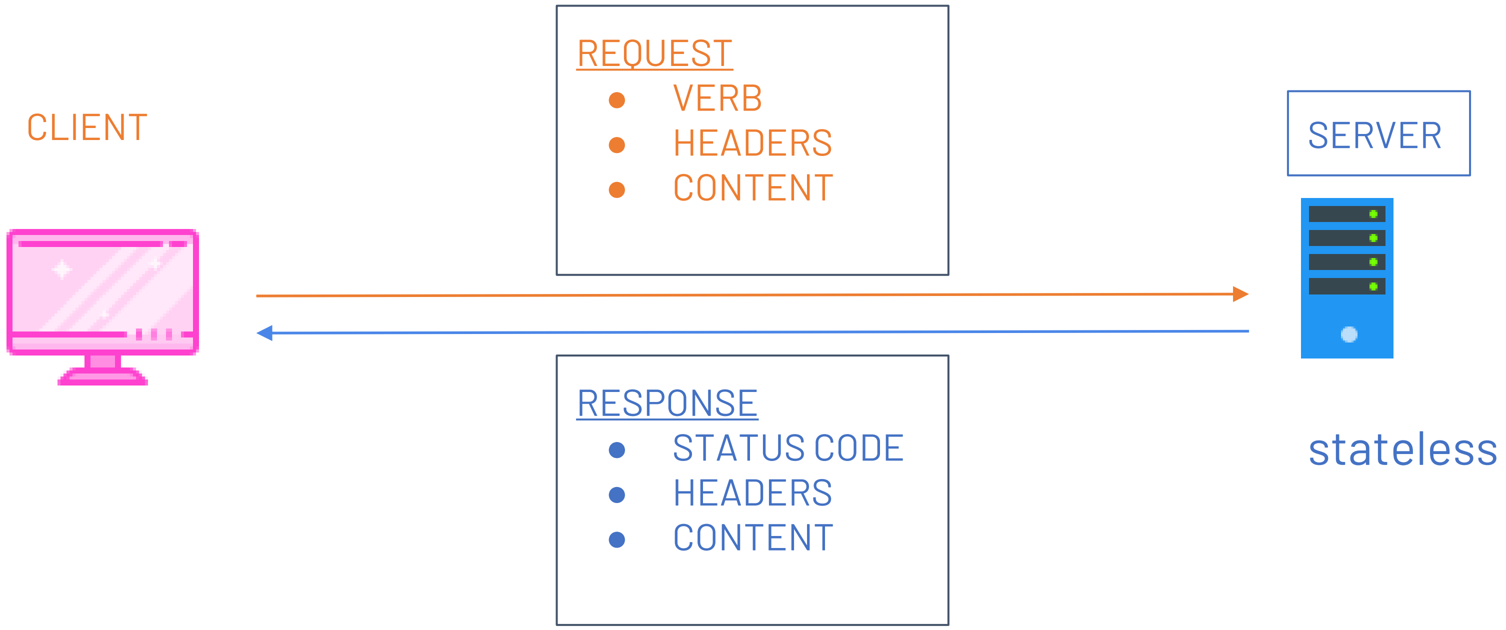
Why use APIs?

- To provide a standardized way of accessing data
- Share application without exposing the implementation to those who shouldn’t have access to it.
- To simplify how to access the application’s data and functionality.

Why we need APIs

- **Improved collaboration:** APIs enable integration so that platforms and apps can seamlessly communicate with one another
- **Easier innovation:** APIs offer flexibility, allowing companies to make connections with new business partners, offer new services to their existing market, and, ultimately, access new markets that can generate massive returns and drive digital transformation
- **Data monetization:** Many companies choose to offer APIs for free. However, if the API grants access to valuable digital assets, you can monetize it by selling access (this is referred to as the API economy).
- **Added security:** As noted above, APIs create an added layer of protection between your data and a server.

How does HTTP Work?



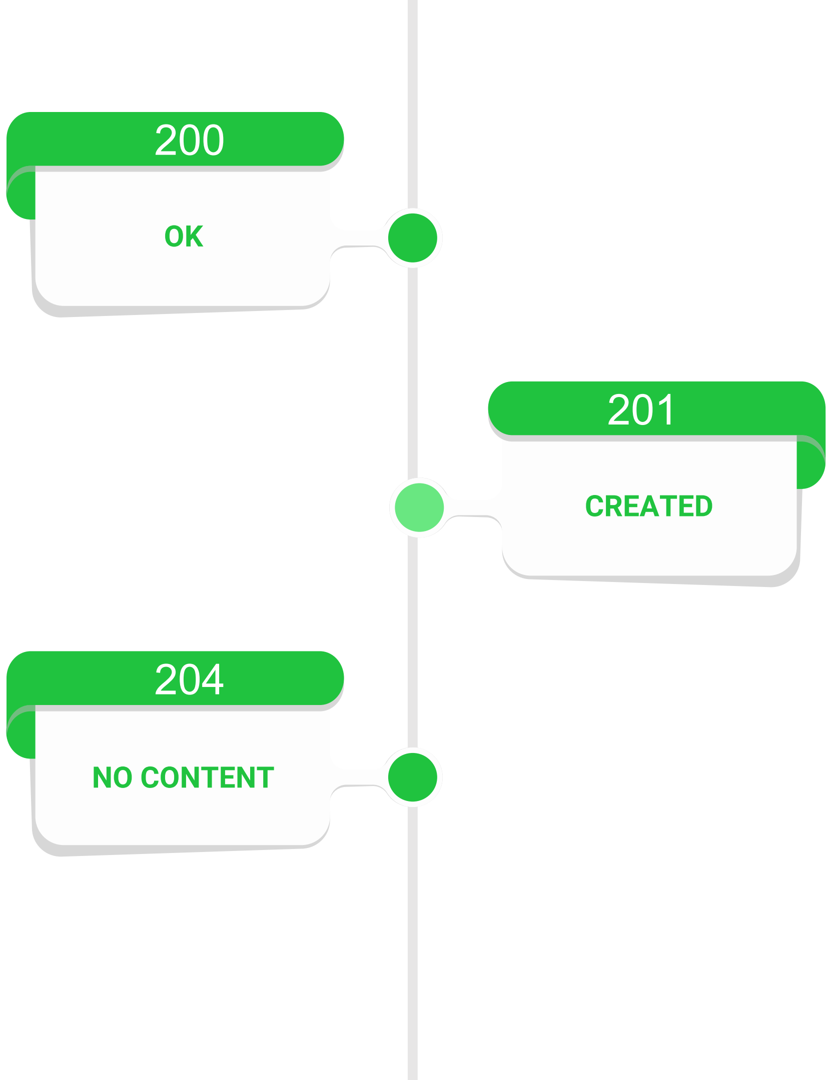
HTTP Methods (VERBS)

CRUD

		ACTIONS TO PERFORM		
Create	→	Create Resource	→	POST
Read	→	Request Resource	→	GET
Update	→	Update Resource	→	PUT
		Update Partial Resource	→	PATCH
Delete	→	Delete Resource	→	DELETE

Successful Responses

Range: 200 - 299



Client Error Responses

Range: 400 - 499

400	BAD REQUEST
401	UNAUTHORIZED
403	FORBIDDEN
404	NOT FOUND
405	METHOD NOT ALLOWED
409	CONFLICT
422	UNPROCESSABLE ENTITY

Server Error Responses

Range: 500 - 599

500

INTERNAL SERVER ERROR

503

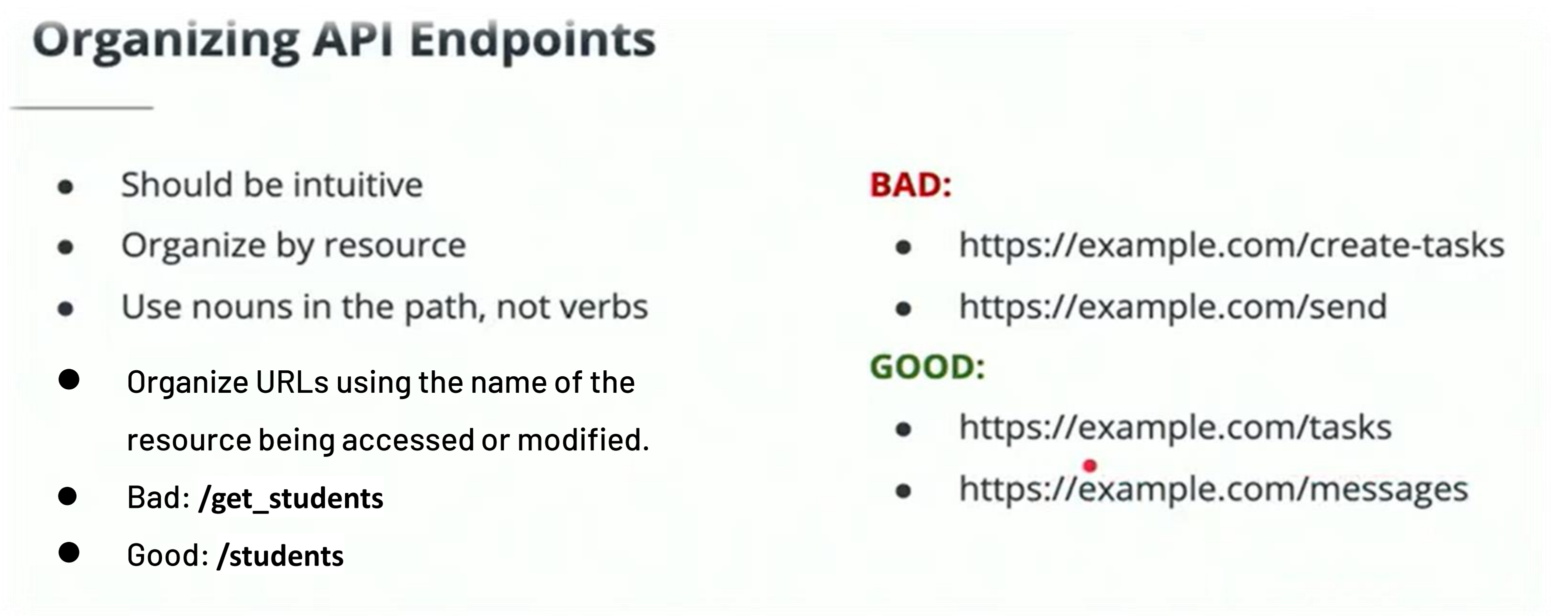
SERVER UNAVAILABLE

504

GATEWAY TIMEOUT

02

ORGANIZING API ENDPOINTS



Organizing API Endpoints

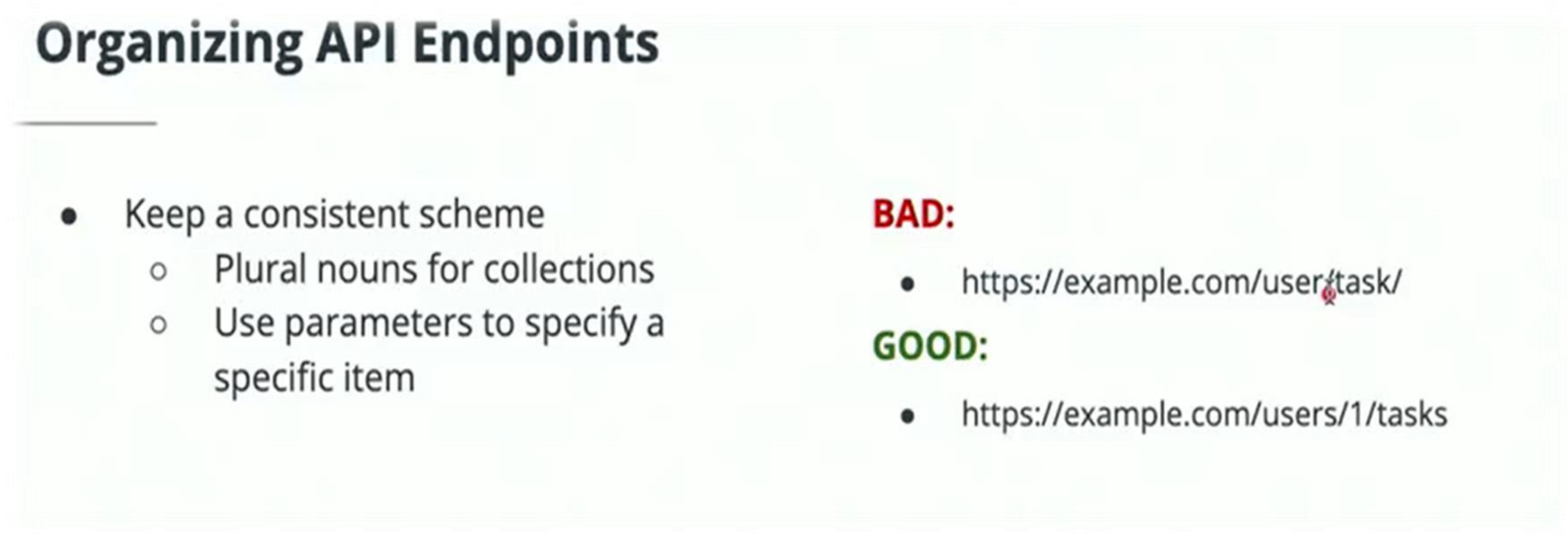
- Should be intuitive
- Organize by resource
- Use nouns in the path, not verbs
- Organize URLs using the name of the resource being accessed or modified.
- Bad: **/get_students**
- Good: **/students**

BAD:

- https://example.com/create-tasks
- https://example.com/send

GOOD:

- https://example.com/tasks
- https://example.com/messages



Organizing API Endpoints

- Keep a consistent scheme
  - Plural nouns for collections
  - Use parameters to specify a specific item

BAD:

- https://example.com/user/task/

GOOD:

- https://example.com/users/1/tasks

Organizing API Endpoints

- Don't make them too complex or lengthy
  - No longer than *collection/item/collection*

BAD:

- https://example.com/users/1/tasks/8/notes

GOOD:

- https://example.com/tasks/8/notes
- https://example.com/users/1/tasks

Limit the number of URL segments

- As a guideline, do not exceed 3 segments in the URL.
- Good: **/students/1/projects**
- **Bad: /students/1/projects/2/tasks**

Use the HTTP Method to define the action

- Recall that HTTP Methods map to CRUD operations.
- Retrieve Students: **GET /students**
- Create Student: **POST /students**
- Modify Student: **PATCH /students/1**
- Delete Student: **DELETE /students**

Examples of Flask Endpoint Definitions

- Retrieve Students: **@app.route('/students', methods=['GET'])**
- Create Student: **@app.route('/students', methods=['POST'])**
- Modify Student: **@app.route('/students/<int:id>', methods=[‘PUT'])**
- Modify Student: **@app.route('/students/<int:id>', methods=['PATCH'])**
- Delete Student: **@app.route('/students/<int:id>', methods=['DELETE'])**

Methods & Endpoints Review

Resource	GET	POST	PATCH	DELETE
/tasks	Get all tasks	Create a new task	Partial update of all tasks	Delete all tasks
/tasks/1	Get the details of task 1	Error!	Partial update of task 1	Delete task 1
/tasks/1/notes	Get all the notes for task 1	Create a new note for task 1	Partial update of all notes of task 1	Delete all notes of task 1

03

CORS

CORS - Security

It takes 20 years to build a reputation and few minutes of cyber-incident to ruin it.”
– Stephane Nappo

CORS - Cross-Origin Resources Sharing

CORS is the process of sharing resource(s) from between different origins / addresses

Origin - Address

Same-origin policy

Web Applications are **not** allowed by default to share resources with another application on a different origin or address for security reasons.

Cross-Origin Resource Sharing

CORS

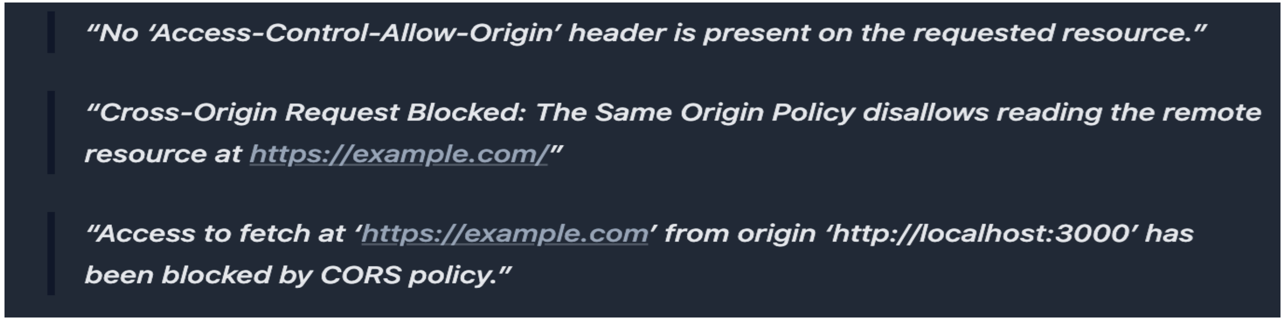
- Security and the Same-Origin Policy
- Block requests from rogue JavaScript

CORS

Cross-Origin Implies:

- Different domains: https://udacity.com and https://github.com
- Different subdomains: https://business.udacity.com and https://status.udacity.com
- Different ports: http://localhost:3000 and http://localhost:5000
- Different protocols: http://example.com and https://example.com

Samples of CORS error message



> *"No 'Access-Control-Allow-Origin' header is present on the requested resource."*

> *"Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://example.com/"*

> *"Access to fetch at 'https://example.com' from origin 'http://localhost:3000' has been blocked by CORS policy."*

Cross-Origin Resource Sharing

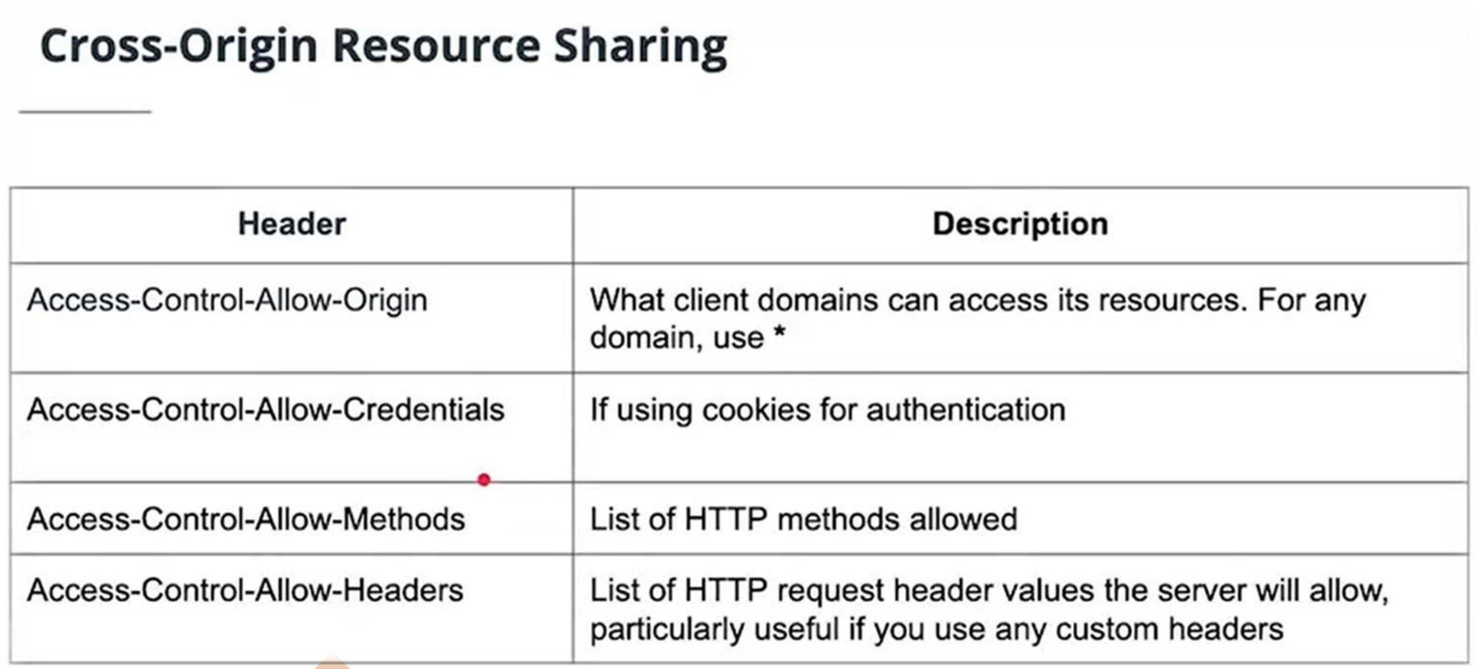
CORS

- Triggered when making requests from...
  - Different domains
  - Different subdomains (example.com and api.example.com)
  - Different ports (example.com and example.com:1234)
  - Different protocols (http://example.com and https://example.com)

Cross-Origin Resource Sharing

Why do we care?

- Rogue or malicious scripts
- Ability to complete non-simple requests (beyond some basic headers)
    - Preflight OPTIONS request
    - No CORS, no request sent
- It protects you and your users



Cross-Origin Resource Sharing

Header	Description
Access-Control-Allow-Origin	What client domains can access its resources. For any domain, use *
Access-Control-Allow-Credentials	If using cookies for authentication
Access-Control-Allow-Methods	List of HTTP methods allowed
Access-Control-Allow-Headers	List of HTTP request header values the server will allow, particularly useful if you use any custom headers

CORS

CORS headers are used by the server to allow applications of other origins to access our application.

- **Access-Control-Allow-Origin** - http://udacity.com or *

  What client domains can access its resources. For any domain use *

- **Access-Control-Allow-Credentials**

  Only if using cookies for authentication - in which case its value must be true

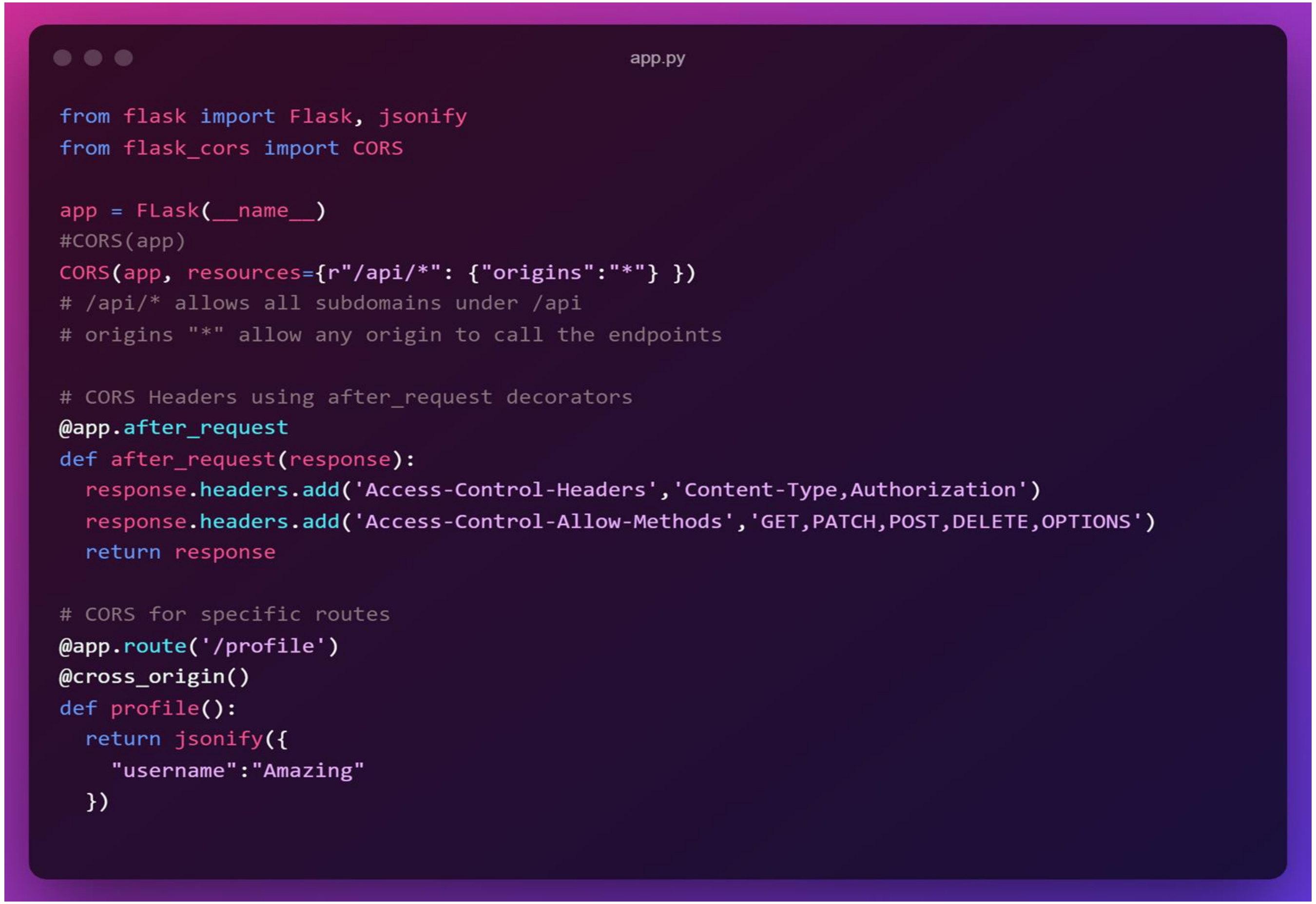
- **Access-Control-Allow-Methods**

  List of HTTP request types allowed

- **Access-Control-Allow-Headers**

  List of http request header values the server will allow, particularly useful if you use any custom headers

Flask-CORS is the extension for handling CORS on a flask application.



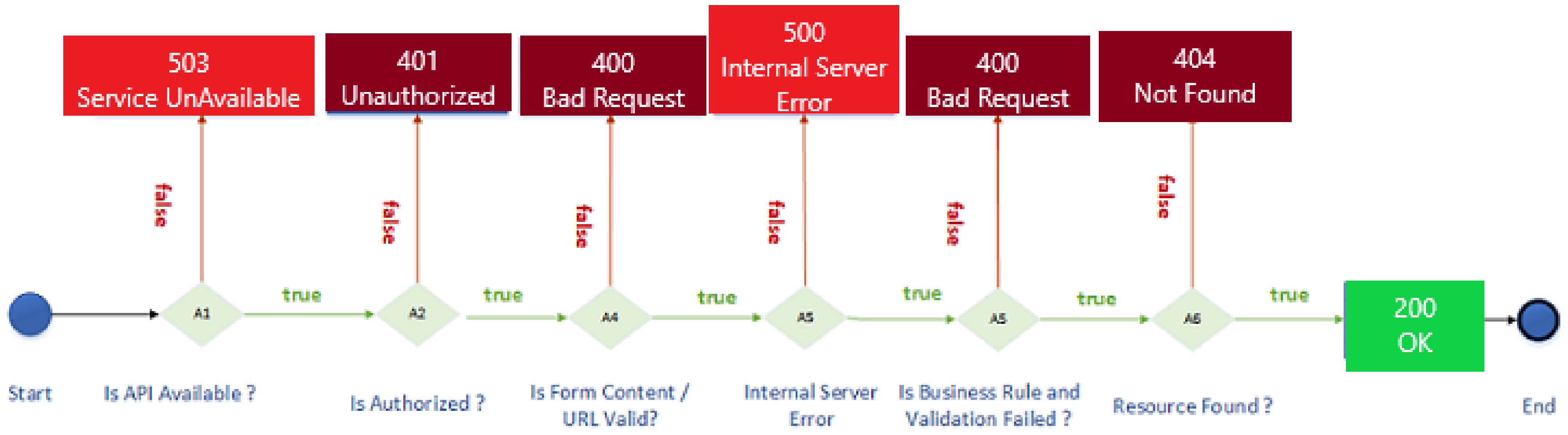
app.py

```python
from flask import Flask, jsonify
from flask_cors import CORS

app = FLask(__name__)
#CORS(app)
CORS(app, resources={r"/api/*": {"origins":"*"} })
# /api/* allows all subdomains under /api
# origins "*" allow any origin to call the endpoints

# CORS Headers using after_request decorators
@app.after_request
def after_request(response):
  response.headers.add('Access-Control-Headers','Content-Type,Authorization')
  response.headers.add('Access-Control-Allow-Methods','GET,PATCH,POST,DELETE,OPTIONS')
  return response

# CORS for specific routes
@app.route('/profile')
@cross_origin()
def profile():
  return jsonify({
    "username":"Amazing"
  })
```

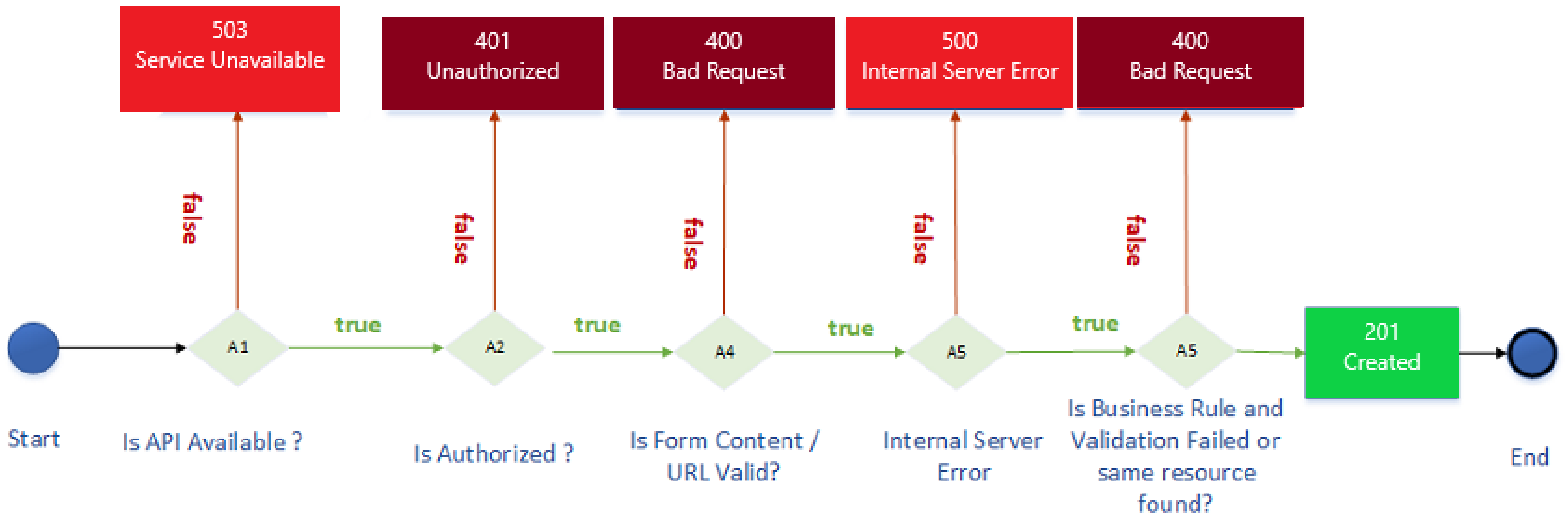
04

FLASK ERROR HANDLING (Practical)

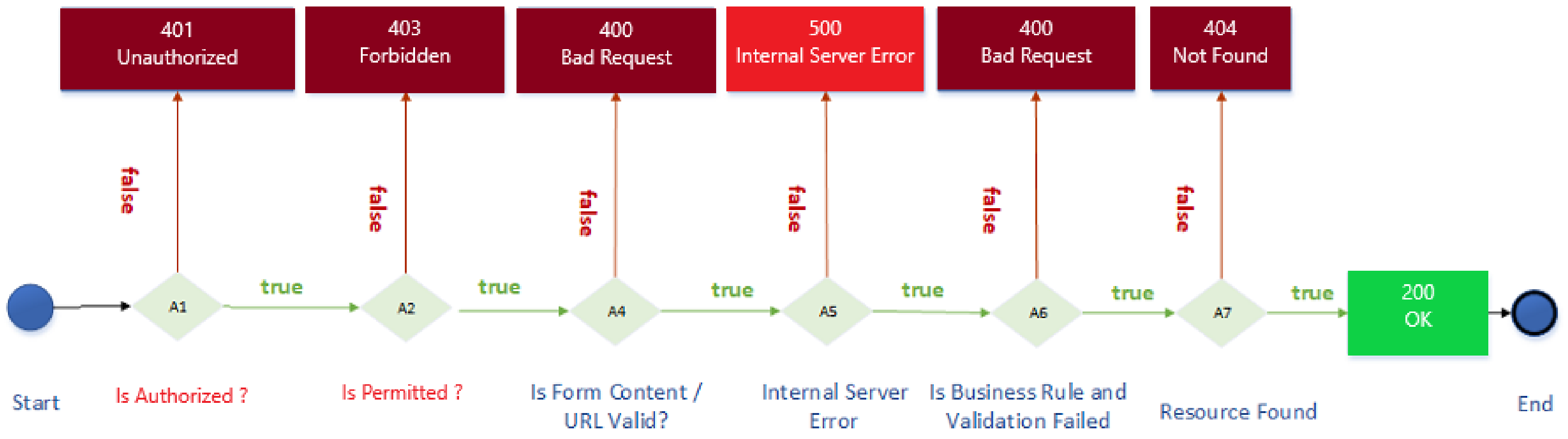
GET RESOURCE SEQUENCE



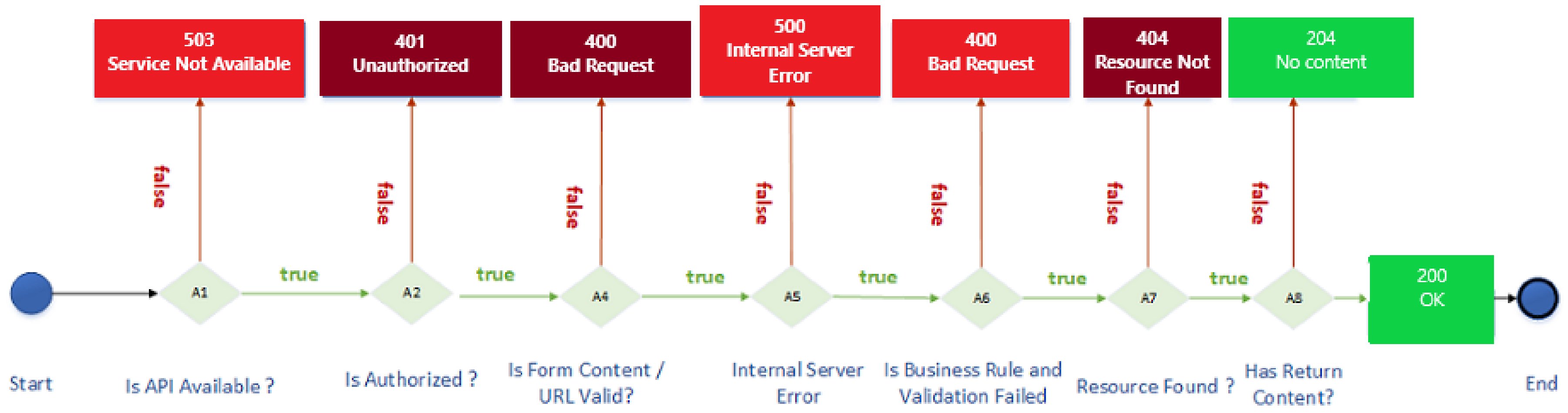
POST RESOURCE SEQUENCE



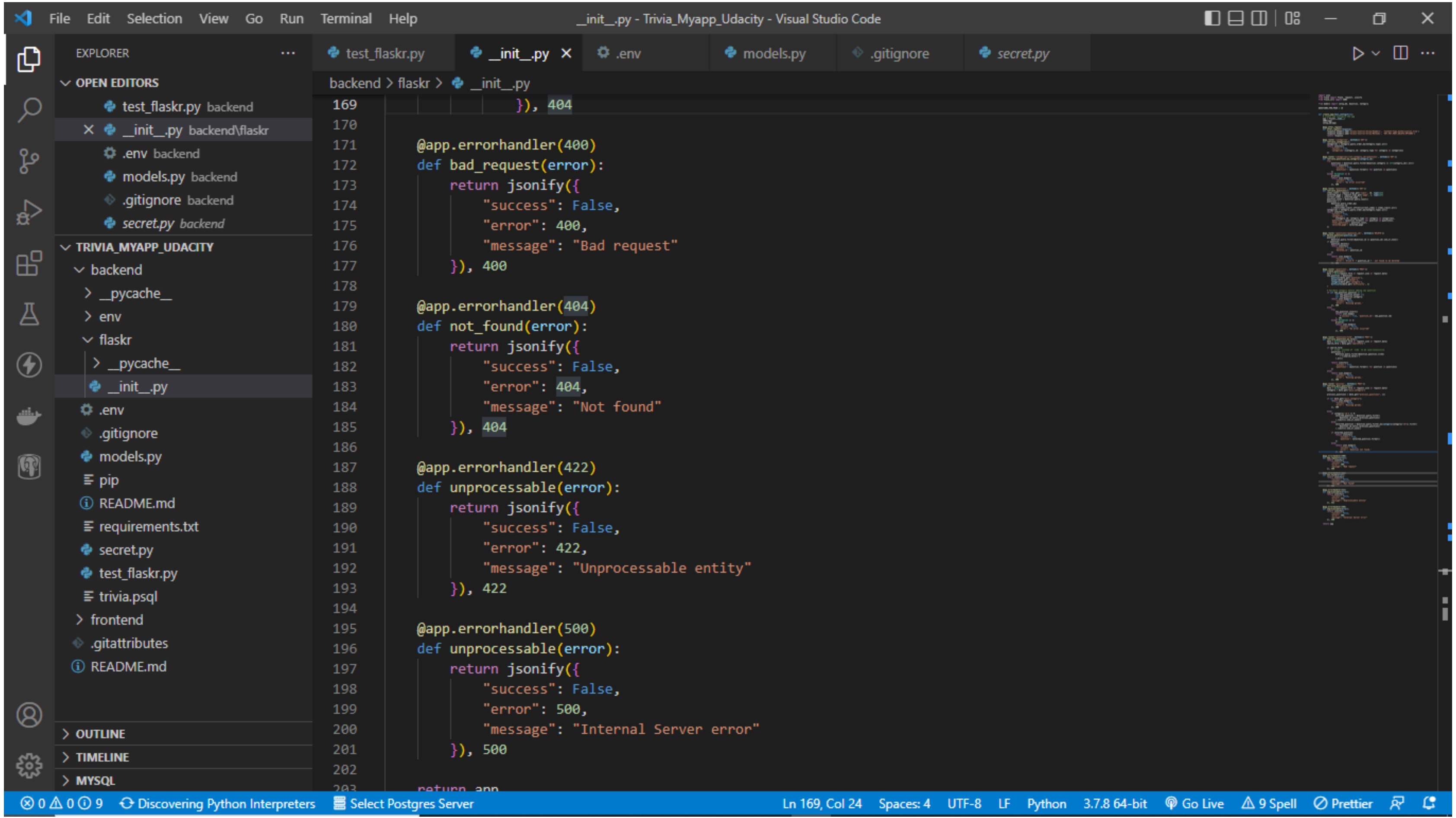
PUT/PATCH RESOURCE SEQUENCE



DELETE RESOURCE SEQUENCE



Error Handling



```
            }), 404

    @app.errorhandler(400)
    def bad_request(error):
        return jsonify({
            "success": False,
            "error": 400,
            "message": "Bad request"
        }), 400

    @app.errorhandler(404)
    def not_found(error):
        return jsonify({
            "success": False,
            "error": 404,
            "message": "Not found"
        }), 404

    @app.errorhandler(422)
    def unprocessable(error):
        return jsonify({
            "success": False,
            "error": 422,
            "message": "Unprocessable entity"
        }), 422

    @app.errorhandler(500)
    def unprocessable(error):
        return jsonify({
            "success": False,
            "error": 500,
            "message": "Internal Server error"
        }), 500

    return app
```

05

API TESTING
(Unit Testing 101)

API TESTING

As with any code you write, you want to test your API to ensure that requests are processed as you expect, the responses sent are correct, and the operations performed on the database are correct and persist.

In this lesson we'll cover:

- Purpose and Benefits of API Testing
- Testing a Flask Api with Unittest
- Test-Driven Development for APIs

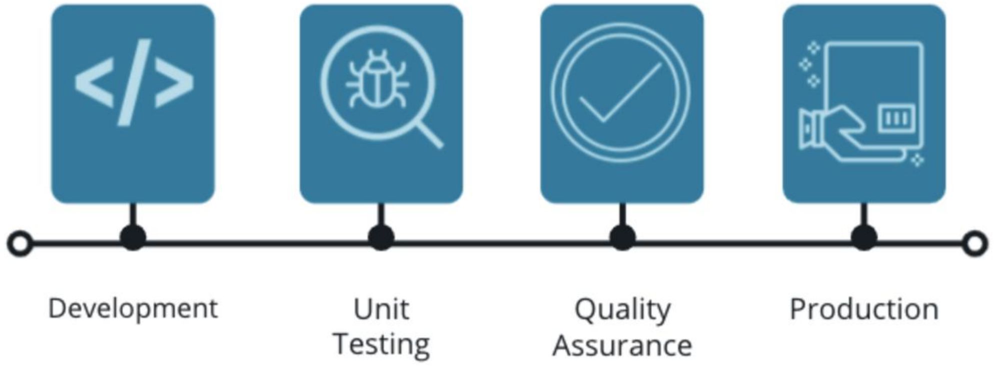
WHY API TESTING

As with all tests, writing unit tests for your API verifies the behavior. For APIs, test should be written:

- To confirm expected request handling behavior
- To confirm success-response structure is correct
- To confirm expected errors are handled appropriately
- To confirm CRUD operations persist

In addition to verifying behavior, having a thorough test suite ensures that when you update your API, you can easily test all previous functionality.

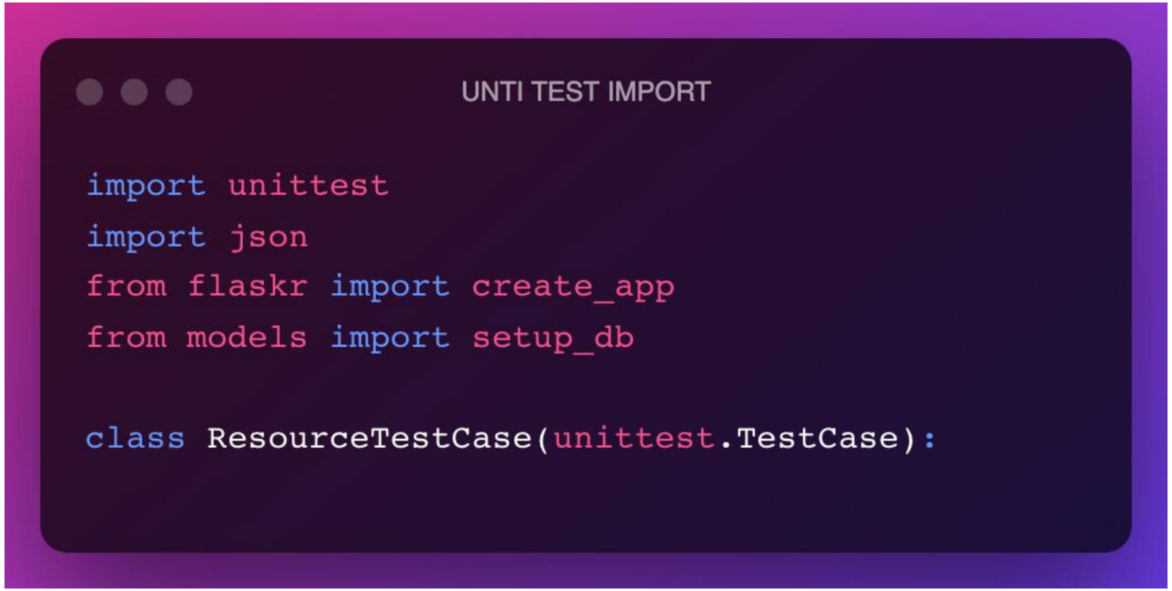
ORDER OF DEVELOPMENT



App Development Timeline

TESTING IN FLASK

Before you get started writing your own testing code, we'll review the essential elements of writing a test suite using unittest in Python3. We'll break down the code into its different components and provide a little more explanation of each



```python
import unittest
import json
from flaskr import create_app
from models import setup_db

class ResourceTestCase(unittest.TestCase):
```

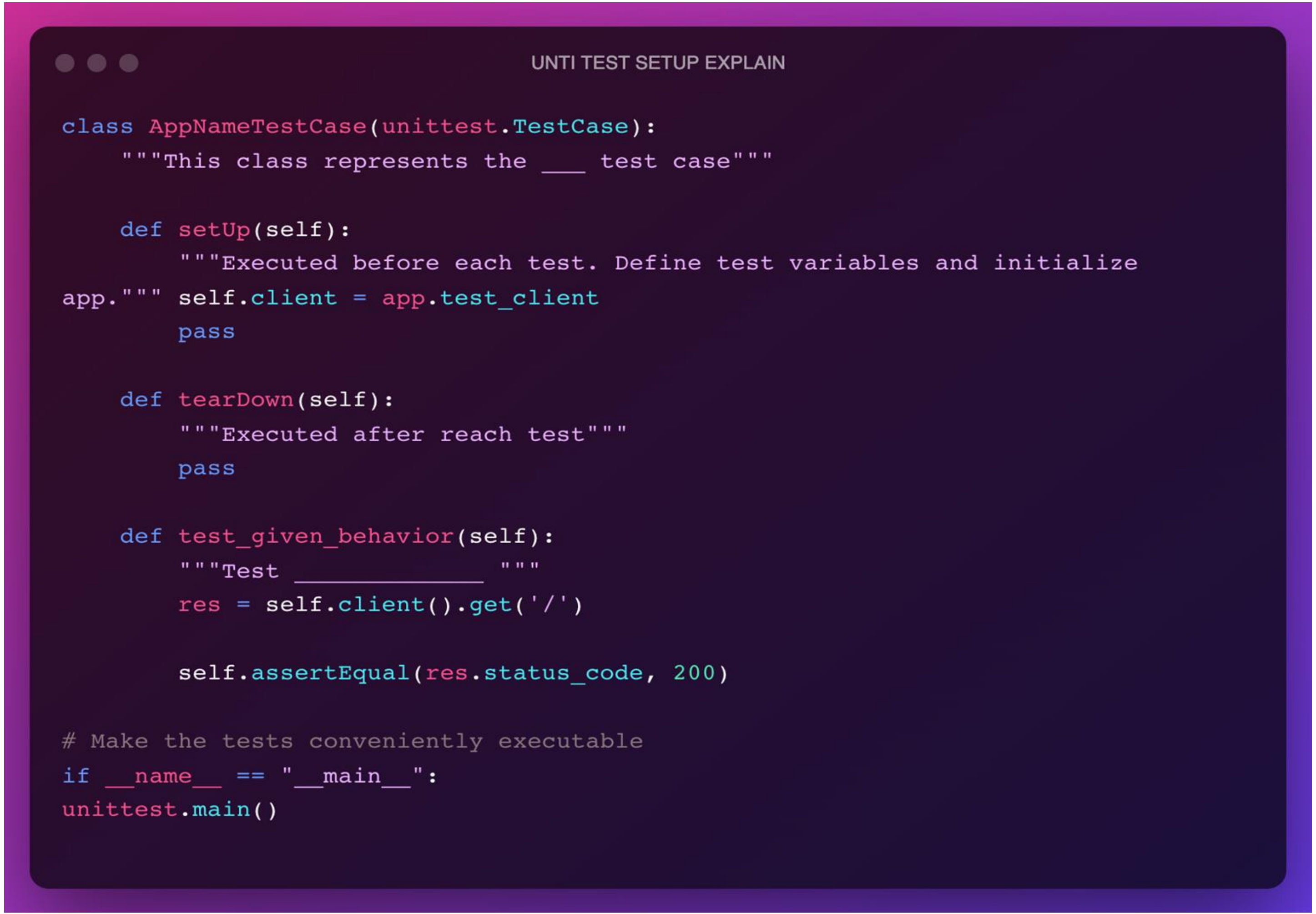
UNITTEST STRUCTURE

1. **Define the test case class** for the application (or section of the application, for larger applications).
2. **Define and implement the `setUp` function.** It will be executed before each test and is where you should initialize the app and test client, as well as any other context your tests will need. The Flask library provides a test client for the application, accessed as shown below.
3. **Define the `tearDown` method**, which is implemented after each test. It will run as long as `setUp` executes successfully, regardless of test success.

UNITTEST STRUCTURE

4. **Define your tests.** All should begin with `"test_"` and include a doc string about the purpose of the test. In defining the tests, you will need to:
   - Get the response by having the client make a request
   - Use `self.assertEqual` to check the status code and all other relevant operations.
5. **Run the test suite**, by running `python test_file_name.py` from the command line.

TESTING IN FLASK

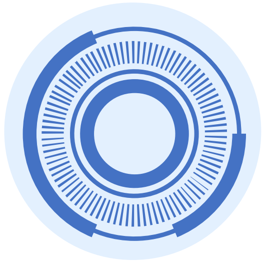
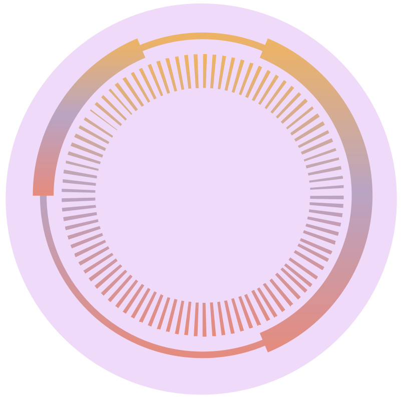


```
class AppNameTestCase(unittest.TestCase):
    """This class represents the ___ test case"""

    def setUp(self):
        """Executed before each test. Define test variables and initialize
app.""" self.client = app.test_client
        pass

    def tearDown(self):
        """Executed after reach test"""
        pass

    def test_given_behavior(self):
        """Test _____________ """
        res = self.client().get('/')

        self.assertEqual(res.status_code, 200)

# Make the tests conveniently executable
if __name__ == "__main__":
unittest.main()
```

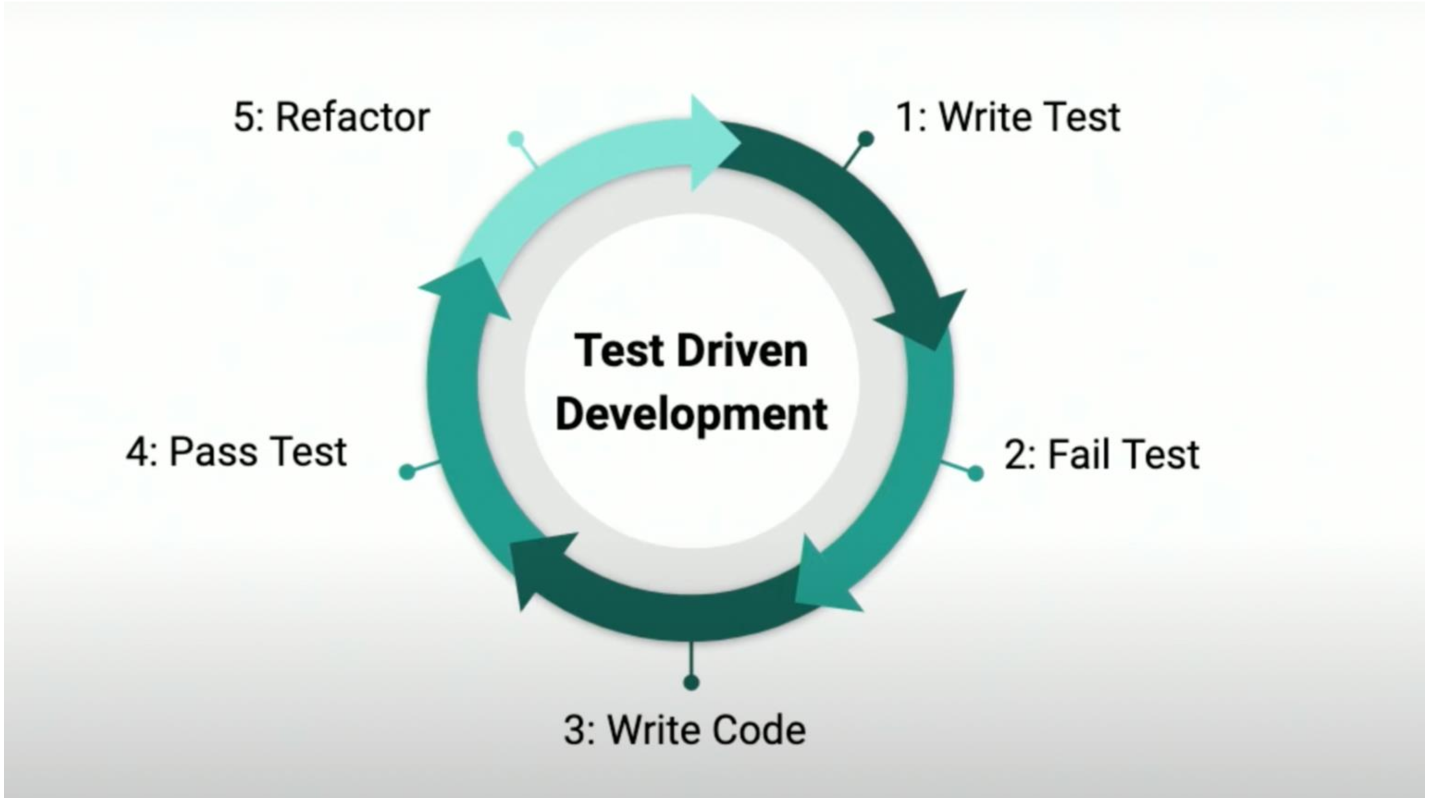
TEST-DRIVEN DEVELOPMENT (TDD)

Test-Driven Development (or **TDD**) is a software development paradigm used very commonly in production. It is based on a short, rapid development cycle in which tests are written before the executable code and constantly iterated on.

1. Write test for specific application behavior.
2. Run the tests and watch them fail.
3. Write code to execute the required behavior.
4. Test the code and rewrite as necessary to pass the test
5. Refactor your code.
6. Repeat - write your next test.

TDD CYCLE


5: Refactor
1: Write Test
Test Driven Development
4: Pass Test
2: Fail Test
3: Write Code

QUESTIONS

API & DOCUMENTATION

PRE STUDY PLAN

API & Route

- RestFul APIs Fundamental: https://www.techtarget.com/searchapparchitecture/definition/RESTful-API
- **RESTful-API video Tutorial** : https://youtu.be/rtWH70_MMHM
- Restful Api with Python Flask : https://youtu.be/Sf-7zXBB_mg

Flask

- Flask Fundamental: https://www.tutorialspoint.com/flask/index.htm
- **Flask video Tutorial** : https://youtu.be/Z1RJmh_OqeA

Testing Fundamentals and Application

- https://youtu.be/iQVvpnRYl-w;
- https://youtu.be/dTvJwxrM1VY
- A simple Introduction to Test Driven Development with Python

THANK YOU