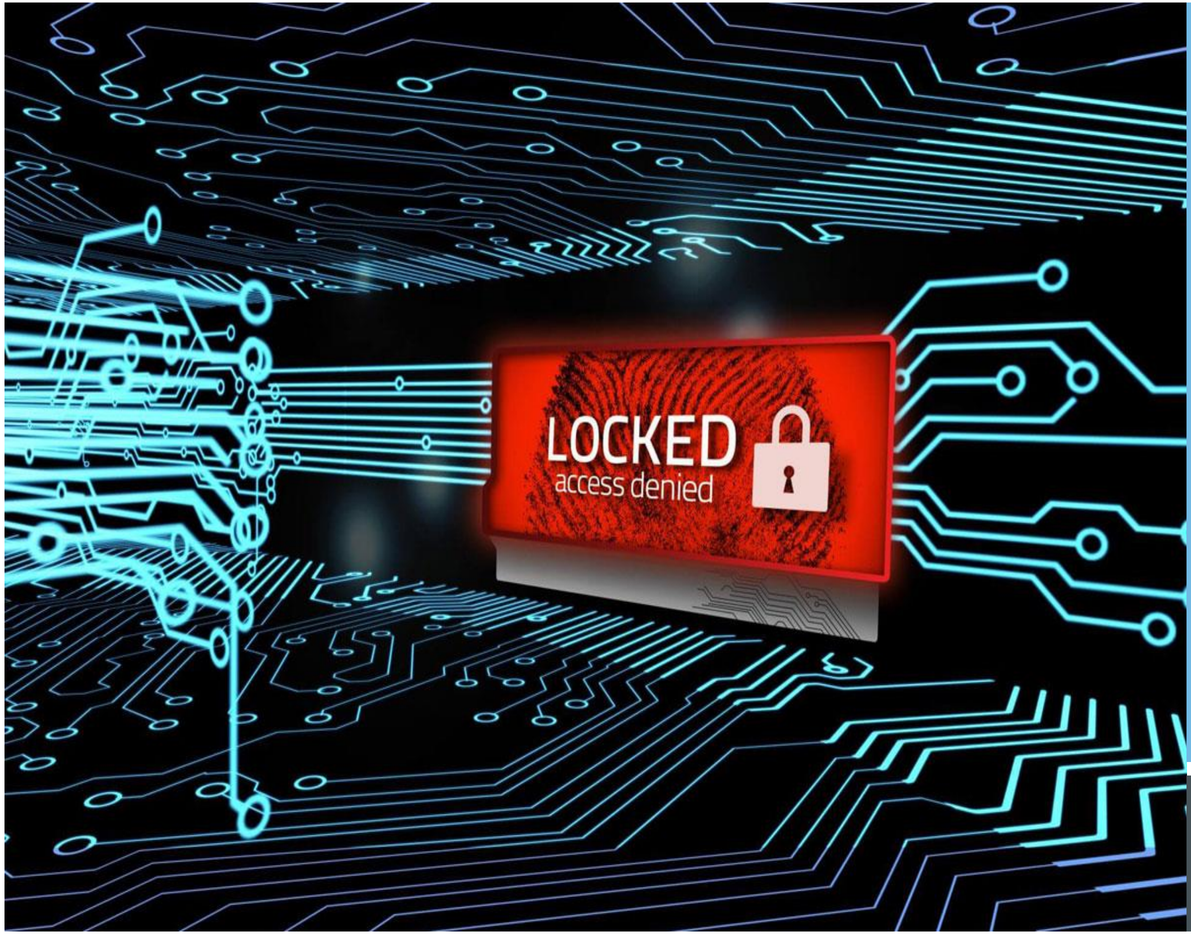
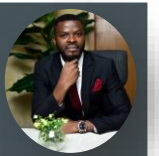


FULL-STACK NANODEGREE SESSION 7

AJIROGHENE SUNDAY



1. WHICH ENDPOINT(S) IS CORRECTLY FORMATTED TO GET ALL COMMENTS FOR A POST [ID: 9] OF A USER [ID:5] ?

A: /users/5/posts/9/comments

B: /user/5/post/9/comments

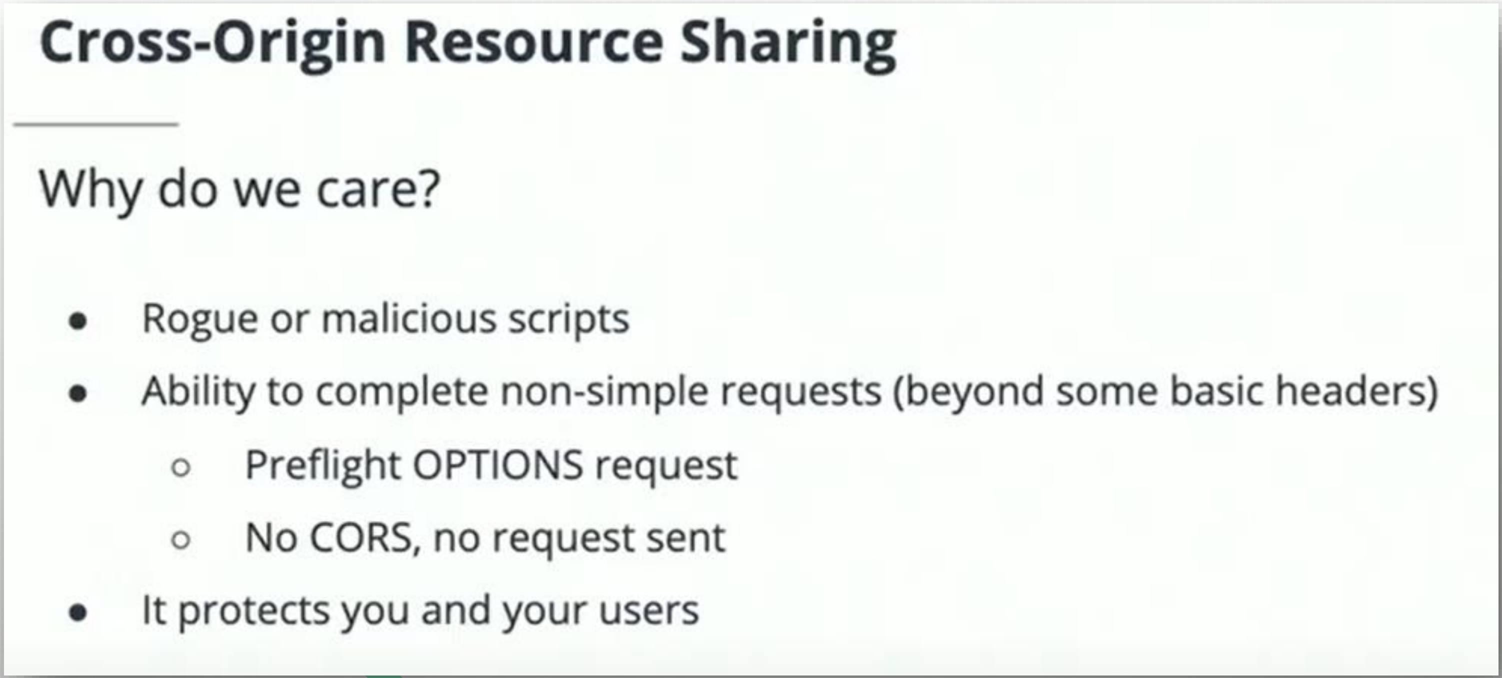
C: /users/5/posts & /posts/9/comments

D: /user/5/posts & /post/9/comments

E: None of the above

2. WHY SHOULD WE CARE ABOUT USING CORS

- A: Mitigate malicious JavaScript
- B: Ability to complete non-stop request
- C: Protect you and the user
- D: None of the above
- E: All of the above



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

3 WHICH OF THE FOLLOWING ENDPOINT-METHOD COMBINATIONS IS NOT ADVISED OR WOULD RAISE AN ERROR?

- A: POST /students/1
- B: DELETE /students
- C: GET /students/1
- D: UPDATE /students/1
- E: None of the above

4 HOW MANY EXCEPT STATEMENTS CAN A TRY-EXCEPT BLOCK HAVE?

A. Zero

B. More than Zero

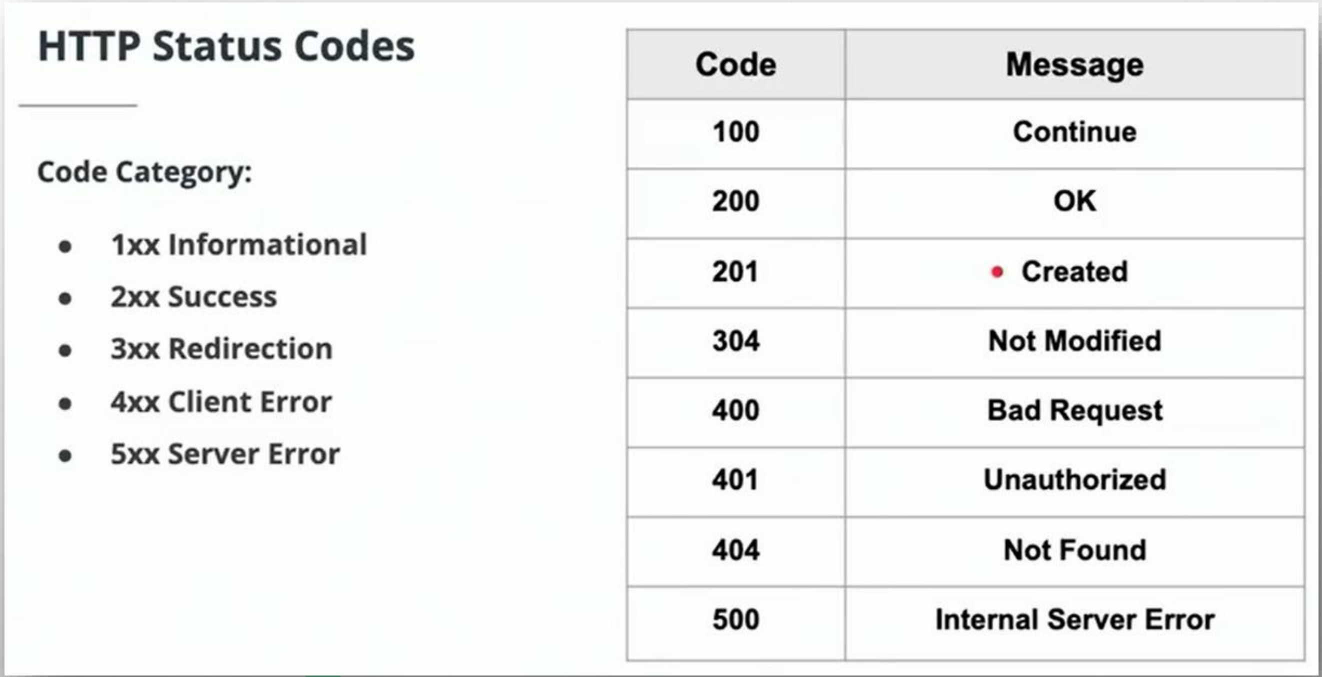
C. One

D. More than One

E. None of the Above

5 WHICH STATUS CODE CLASS/CLASSES IS/ARE USED FOR ERROR PURPOSES

1xx	A
2xx	B
3xx	C
4xx	D
5xx	E



HTTP Status Codes

Code Category:

- 1xx Informational
- 2xx Success
- 3xx Redirection
- 4xx Client Error
- 5xx Server Error

Code	Message
100	Continue
200	OK
201	Created
304	Not Modified
400	Bad Request
401	Unauthorized
404	Not Found
500	Internal Server Error

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

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

7 Why do we do API testing and What is the difference between *setUp()* and *setUpClass()* in the Python unittest framework?

THEORY ON API TESTING - QUIZ

WHY TEST AN API?

- ✓ VERIFY SUCCESS BEHAVIOR
- ✓ VERIFY ERROR HANDLING

SETUP()

Method called to prepare the individual test.Runs before every test.

SETUPCLASS()

A class method called before tests in an individual class are run.Runs before all the tests and must be decorated as a classmethod()

TEARDOWN()

Method called immediately after the test method has been called and the result recorded.

TEARDOWNCLASS()

A class method called after tests in an individual class have run. Teardownclass is called with the class as the only argument and must be decorated as a classmethod()

8 Why do we use API documentation and Why do we use project documentation?

THEORY ON API TESTING - QUIZ

The Importance Of A README File

- Describes the purpose of the project
- Guides new developers on how to set up the project
- API references
- Authors

TABLE OF CONTENTS

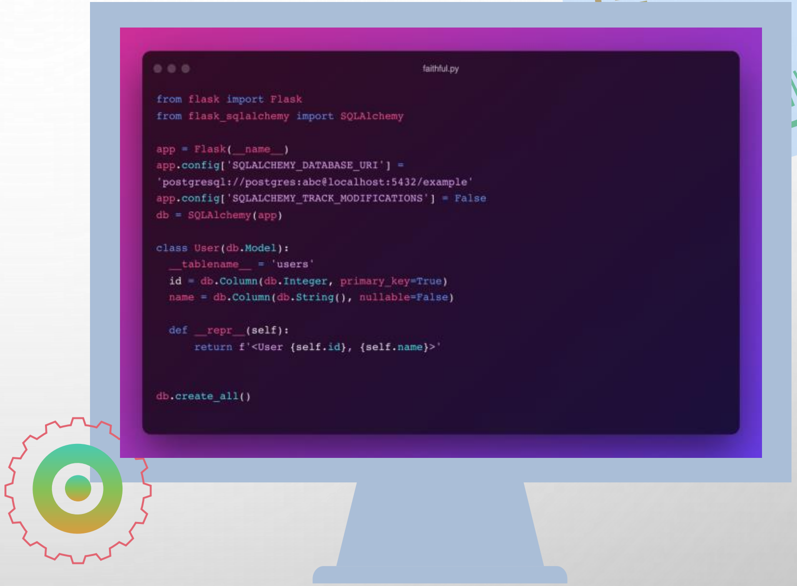
01

RECAP

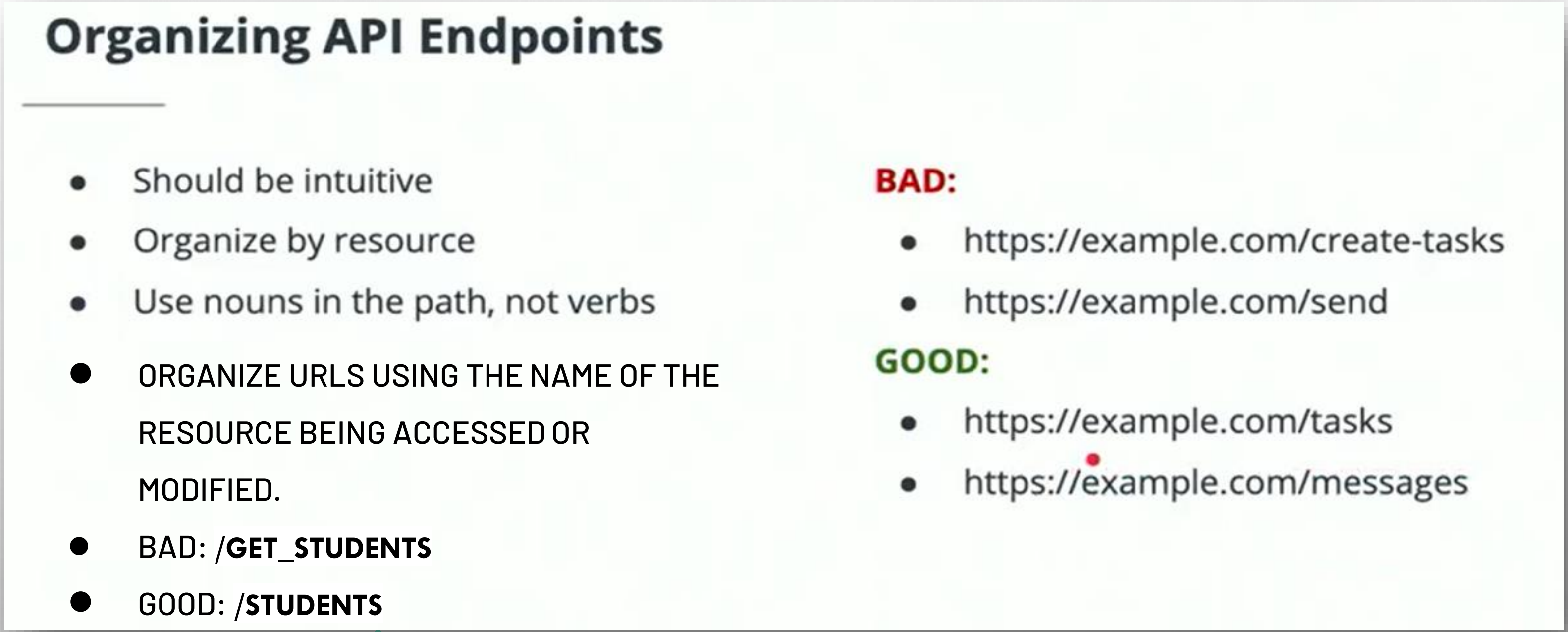
PRACTICAL

Pagination

Virtual Environment


faithful.py

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'postgresql://postgres:abc@localhost:5432/example'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

class User(db.Model):
  __tablename__ = 'users'
  id = db.Column(db.Integer, primary_key=True)
  name = db.Column(db.String(), nullable=False)

  def __repr__(self):
      return f'<User {self.id}, {self.name}>'


db.create_all()
```



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

- ORGANIZE URLS USING THE NAME OF THE RESOURCE BEING ACCESSED OR MODIFIED.
- BAD: /**GET_STUDENTS**
- GOOD: /**STUDENTS**

Organizing API Endpoints

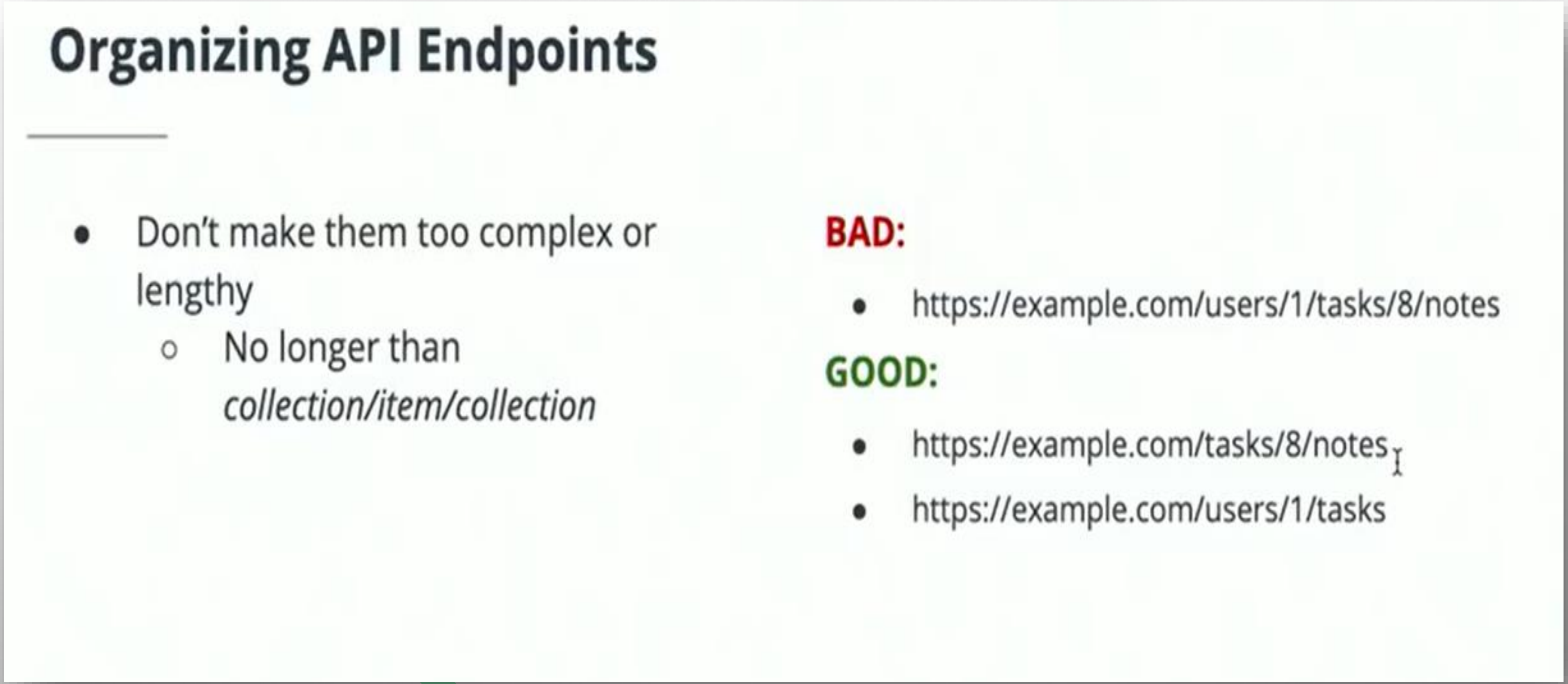
- Keep a consistent scheme
  - Plural nouns for collections
  - Use parameters to specify a specific item

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

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

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

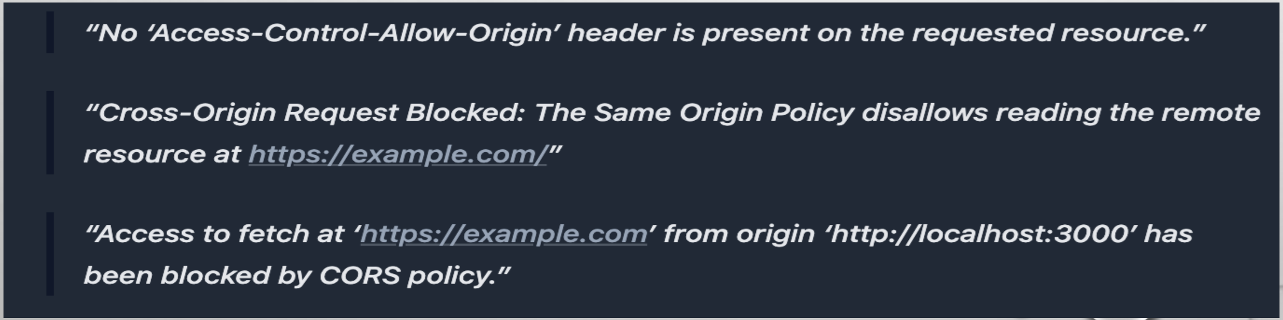
Web Applications are **not** allowed by default to share resources with another application on a different origin or address for security reasons.

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."*

02

PYTHON DECORATORS

PYTHON DECORATORS

A DECORATOR IS A DESIGN PATTERN IN PYTHON THAT ALLOWS A USER TO ADD NEW FUNCTIONALITY TO AN EXISTING OBJECT OR FUNCTION WITHOUT MODIFYING ITS STRUCTURE.

FUNCTIONS ARE OBJECTS.

FUNCTIONS WHICH TAKE OTHER FUNCTIONS AS PARAMETERS OR PERFORM OPERATIONS ON OTHER FUNCTIONS ARE CALLED **HIGHER ORDER FUNCTIONS**

PYTHON DECORATORS ARE FUNCTIONS OR CLASSES IN PYTHON THAT TAKES ANOTHER FUNCTION AS A PARAMETER OR RETURN A FUNCTION

PYTHON DECORATORS

This is also called **metaprogramming** because a part of the program tries to modify another part of the program at compile time.

UNDERSTANDING HOW DECORATORS WORKS

FACTS

- Everything in python (even classes), are objects.
- Names that we define are simply identifiers bound to these objects.
- Functions are no exceptions, they are objects too (with attributes).
- Various different names can be bound to the same function object.

EXAMPLE

```
def first(msg):
    print(msg)


first("hello")

second = first
second("hello")

OUTPUT

hello
hello
```

Functions can be passed as arguments to another function. Such functions that take other functions as arguments are also called **higher order functions**. Here is an example of such a function.

```
def inc(x):
    return x + 1

def dec(x):
    return x - 1



def operate(func, x):
     result = func(x)
     return result

Invoking the Function
>>> operate(inc, 3)
    4
>>> operate(dec, 3)
    2
```

Furthermore, a function can return another function. Below *is_returned()* is a nested function which is defined and returned each time we call *is_called()*

```
def is_called():
    def is_returned():
        print("Hello")
    return is_returned


new = is_called()

# Outputs "Hello"
new()
```

LET’S TALK ABOUT NON-LOCAL VARIABLE & CLOSURE

Nested functions can access variables of the enclosing scope. In Python such non-local variables are read-only by default, however in order to modify them, we declare them explicitly as non-local (using nonlocal keyword). Following is an example of a nested function accessing a non-local variable.

```python
def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function
        print(msg)

    printer()

# We execute the function
# Output: Hello
print_msg("Hello")
```

What would happen if the last line of the function print_msg() returned the printer() function instead of calling it? This means the function was defined as follows:

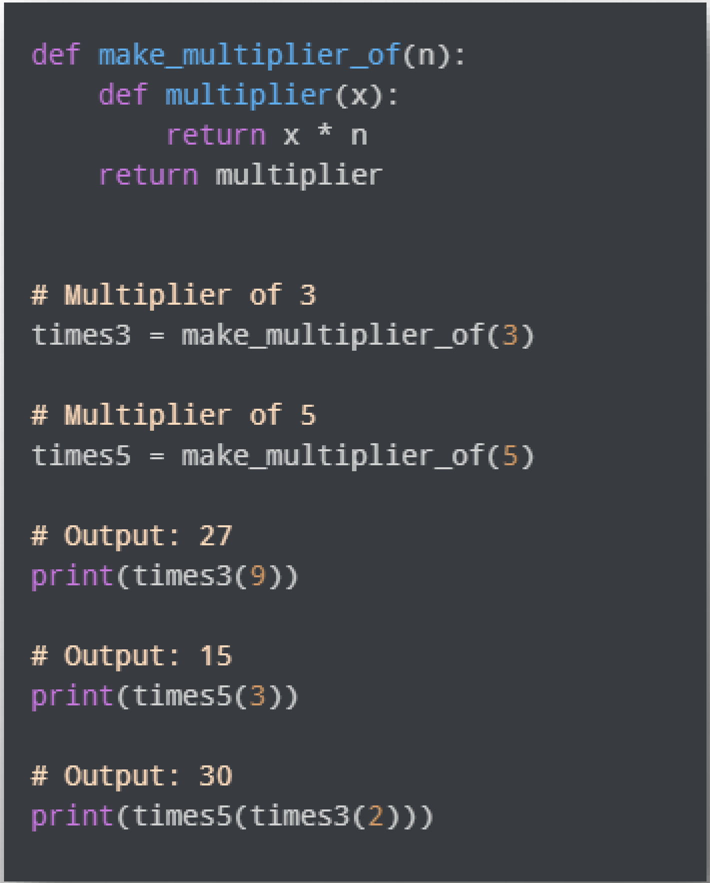
```python
def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function
        print(msg)

    return printer  # returns the nested function


# Now let's try calling this function.
# Output: Hello
another = print_msg("Hello")
another()
```

WHEN DO WE HAVE CLOSURE ?

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope.

The Criteria That Must Be Met To Create Closure In Python Are Summarized In The Following Points.

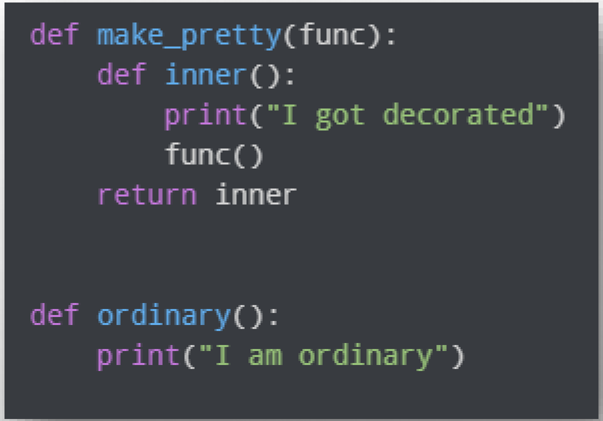
- ✓ We Must Have A Nested Function (Function Inside A Function).
- ✓ The Nested Function Must Refer To A Value Defined In The Enclosing Function.
- ✓ The Enclosing Function Must Return The Nested Function.



```python
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier


# Multiplier of 3
times3 = make_multiplier_of(3)

# Multiplier of 5
times5 = make_multiplier_of(5)

# Output: 27
print(times3(9))

# Output: 15
print(times5(3))

# Output: 30
print(times5(times3(2)))
```

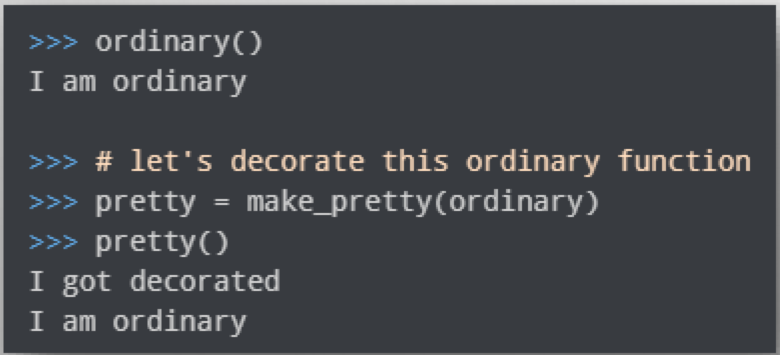
DECORATOR

Basically, a decorator takes in a function, adds some functionality and returns it.



```python
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner


def ordinary():
    print("I am ordinary")
```

When you run the following codes in shell,



```
>>> ordinary()
I am ordinary

>>> # let's decorate this ordinary function
>>> pretty = make_pretty(ordinary)
>>> pretty()
I got decorated
I am ordinary
```

The function `ordinary()` got decorated and the returned function was given the name *pretty*.

We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift.

The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it.

DECORATOR

This is a common construct and for this reason, Python has a syntax to simplify this.
We can use the **@ symbol** along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

It’s equivalent to

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

This is just a syntactic sugar to implement decorators.

The previous decorators are simple and only worked with functions that do not have any parameters.
What if we had functions that took in parameters like:

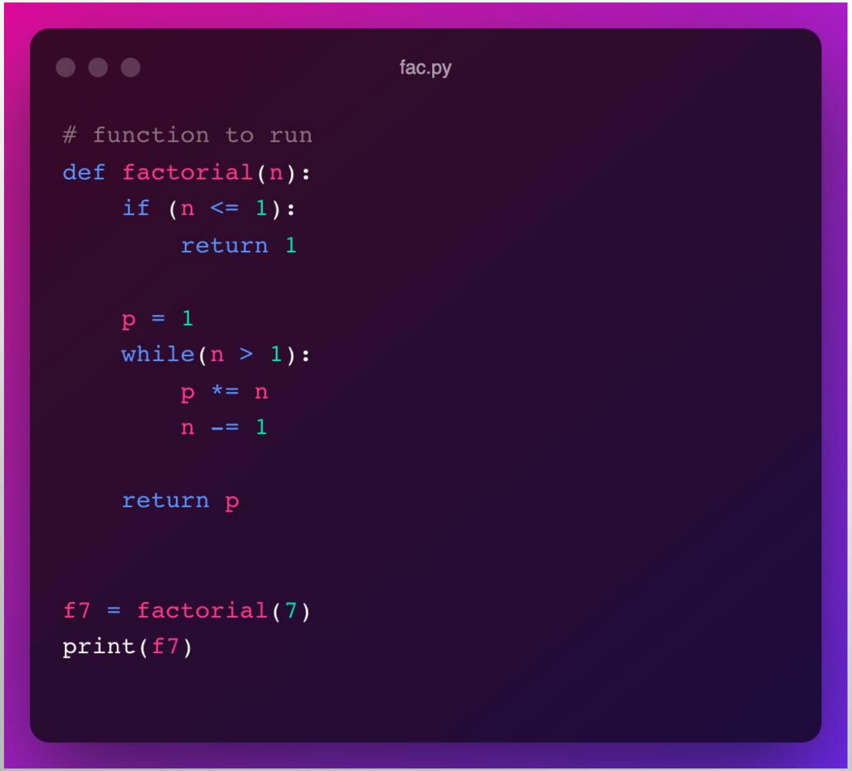
```
def divide(a, b):
   return a/b

>>> divide(2,5)
0.4
>>> divide(2,0)

Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

We’ll make a decorator that checks for the case that cause error:

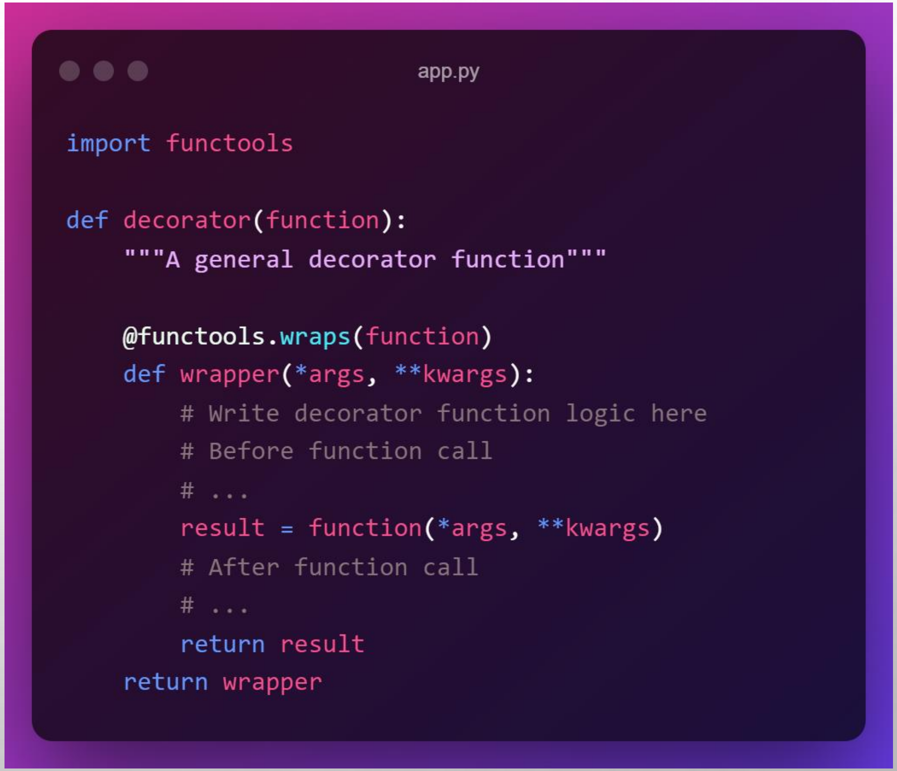
```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner


@smart_divide
def divide(a, b):
    print(a/b)
```

PYTHON DECORATORS - EXAMPLE

An example of an external library we can improve using decorators


fac.py

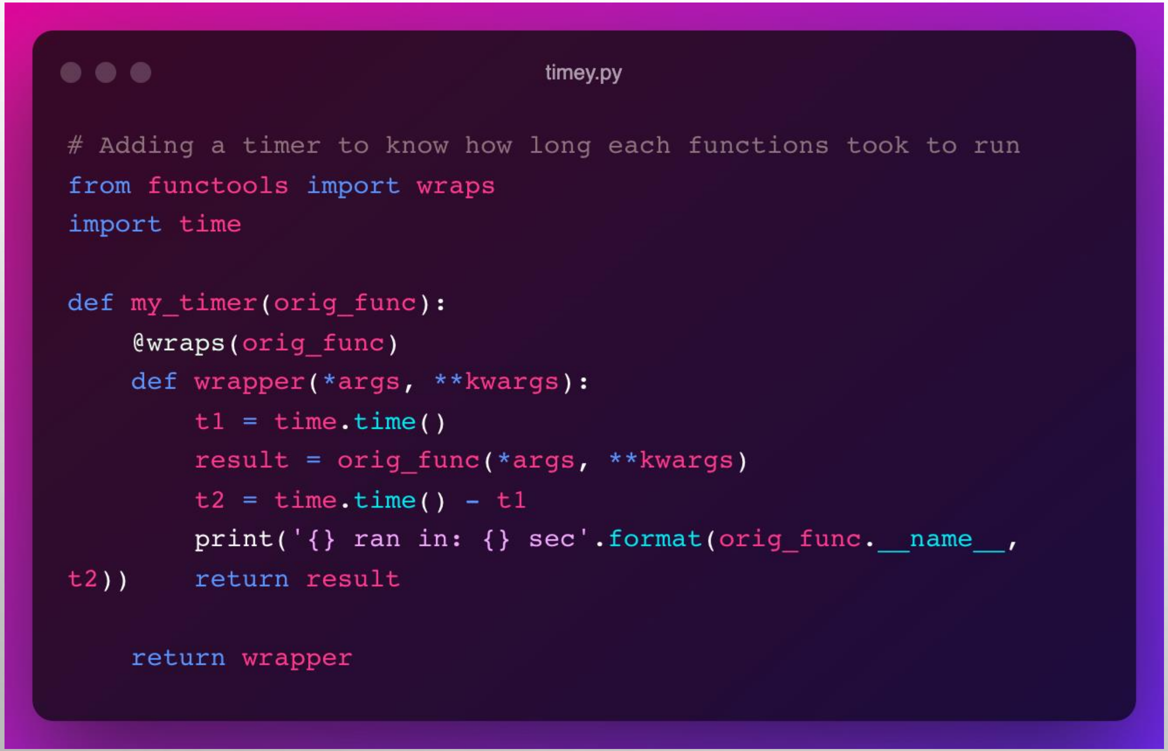
```python
# function to run
def factorial(n):
    if (n <= 1):
        return 1

    p = 1
    while(n > 1):
        p *= n
        n -= 1

    return p


f7 = factorial(7)
print(f7)
```


app.py

```python
import functools

def decorator(function):
    """A general decorator function"""

    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        # Write decorator function logic here
        # Before function call
        # ...
        result = function(*args, **kwargs)
        # After function call
        # ...
        return result
    return wrapper
```

PYTHON DECORATORS - EXAMPLE

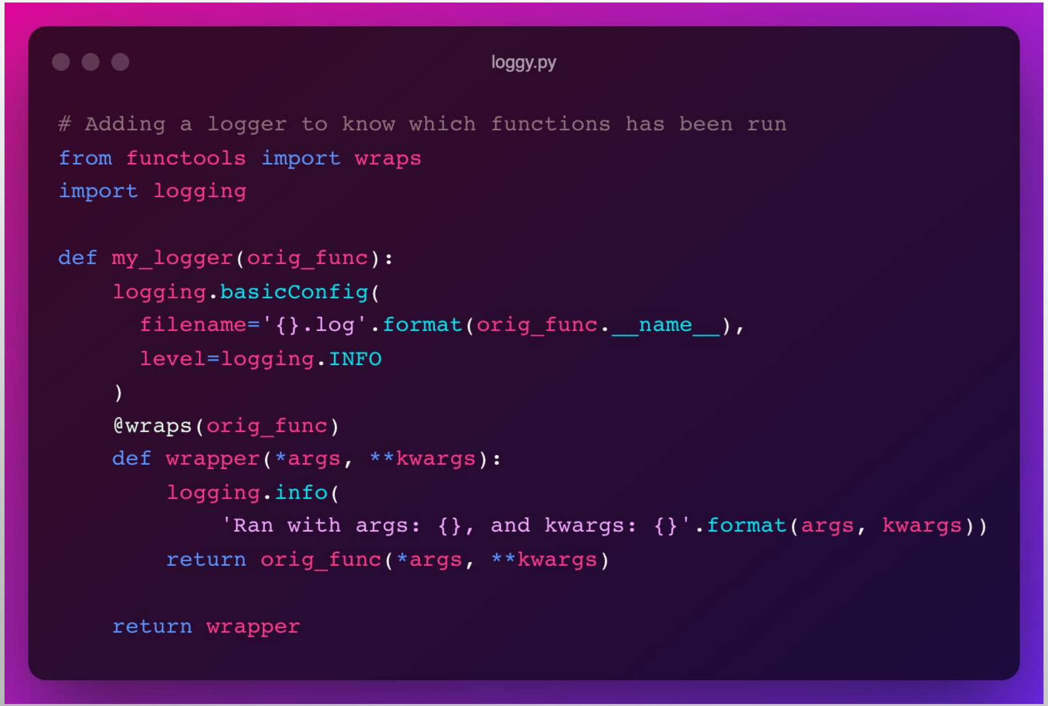
A timing function to keep track of how long it takes the function to run



```
# Adding a timer to know how long each functions took to run
from functools import wraps
import time

def my_timer(orig_func):
    @wraps(orig_func)
    def wrapper(*args, **kwargs):
        t1 = time.time()
        result = orig_func(*args, **kwargs)
        t2 = time.time() - t1
        print('{} ran in: {} sec'.format(orig_func.__name__,
t2))    return result

    return wrapper
```

PYTHON DECORATORS - EXAMPLE

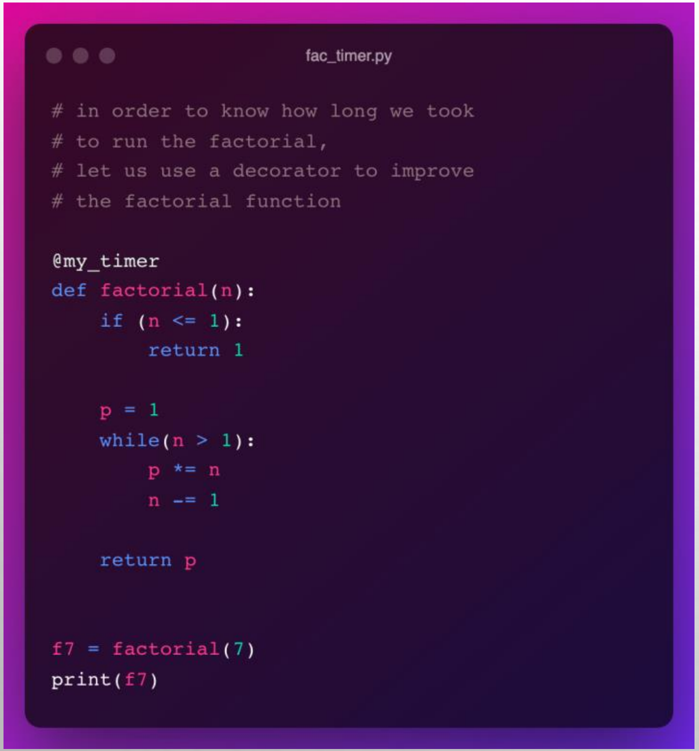
A logging function to log the function that has been ran, in order to keep track of which function we have run



```python
# Adding a logger to know which functions has been run
from functools import wraps
import logging

def my_logger(orig_func):
    logging.basicConfig(
      filename='{}.log'.format(orig_func.__name__),
      level=logging.INFO
    )
    @wraps(orig_func)
    def wrapper(*args, **kwargs):
        logging.info(
            'Ran with args: {}, and kwargs: {}'.format(args, kwargs))
        return orig_func(*args, **kwargs)

    return wrapper
```

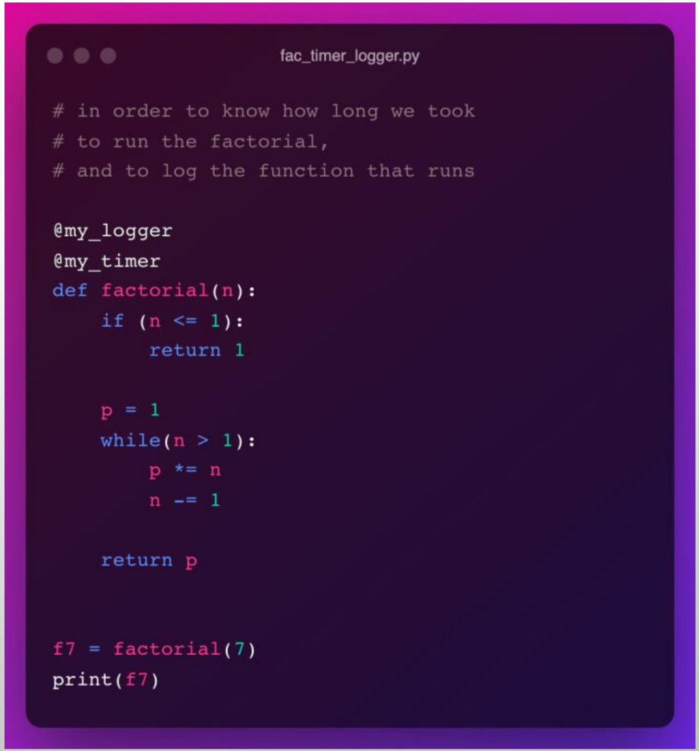
PYTHON DECORATORS - EXAMPLE

Running the factorial functions using the decorators



fac_timer.py

```python
# in order to know how long we took
# to run the factorial,
# let us use a decorator to improve
# the factorial function

@my_timer
def factorial(n):
    if (n <= 1):
        return 1

    p = 1
    while(n > 1):
        p *= n
        n -= 1

    return p


f7 = factorial(7)
print(f7)
```



fac_timer_logger.py

```python
# in order to know how long we took
# to run the factorial,
# and to log the function that runs

@my_logger
@my_timer
def factorial(n):
    if (n <= 1):
        return 1

    p = 1
    while(n > 1):
        p *= n
        n -= 1

    return p


f7 = factorial(7)
print(f7)
```

03

WEB SECURITY

PLAYING WITH FIRE

- DEALING WITH THE WEB MEANS THAT YOUR DATA IS EXPOSED TO VIRTUALLY BILLIONS OF PEOPLE
- WE CAN'T "ASSUME" THAT NONE OF THEM ARE "BAD ACTORS" MEANING TO HARM OUR DATA
- ASSUME THAT YOUR ENVIRONMENT IS ALREADY COMPROMISED AND APPLY A HEALTHY LEVEL OF MISTRUST TO ANY USER OR DEVICE ATTEMPTING TO ACCESS DATA OR SERVICES.

“DATA IS THE TARGET OF ALL THREAT ACTORS”

BORN FROM NECESSITY

TO DEAL WITH THAT THREAT, TWO PHILOSOPHIES WERE BORN:

- **AUTHENTICATION**
  - ANSWERS THE QUESTION OF [WHO ARE YOU?]
  - VERIFIES THE IDENTITY OF THE USER
  - WORKS THROUGH AUTHENTICATION METHODS
- **AUTHORIZING**
  - ANSWERS THE QUESTION OF [WHAT ARE YOU ALLOWED TO DO?]
  - DETERMINES THEIR ACCESS RIGHTS
  - WORKS THROUGH SETTINGS THAT ARE IMPLEMENTED AND MAINTAINED BY THE ORGANIZATION

AUTHENTICATION METHODS

- **USERNAME/PASSWORD**
  - A ROUTINE LOG IN PROCESS THAT REQUIRES A USERNAME AND PASSWORD COMBINATION TO ACCESS A GIVEN SYSTEM WHICH VALIDATES THE PROVIDED CREDENTIALS. MOST OFTEN USED AS A LAST OPTION WHEN COMMUNICATING BETWEEN A SERVER AND DESKTOP OR REMOTE DEVICE

AUTHENTICATION METHODS

- **SINGLE SIGN-ON** (SSO)
  - USERS ONLY HAVE TO LOG IN TO ONE APPLICATION AND IN DOING SO, GAIN ACCESS TO MANY OTHER APPLICATIONS. OFTEN MORE CONVENIENT FOR USERS, AS IT REMOVES THE OBLIGATION TO RETAIN MULTIPLE SETS OF CREDENTIALS AND CREATES A MORE SEAMLESS EXPERIENCE DURING OPERATIVE SESSIONS.

AUTHENTICATION METHODS

- **MULTI-FACTOR AUTHENTICATION**
  - USERS MORE THAN ONE AUTHENTICATION FACTOR TO VERIFY A USER’S IDENTITY.
- **PASSWORD-LESS**
  - MEANS VERIFYING A USER’S IDENTITY WITHOUT USING A PASSWORD. AN EXAMPLE OF HOW THIS CAN BE ACHIEVED:
    - BIOMETRICS
    - MAGIC LINKS.
    - POSSESSION FACTORS. AUTHENTICATION VIA SOMETHING A USER OWNS OR CARRIES WITH THEM

AUTHENTICATION METHODS

- **BIOMETRIC AUTHENTICATION**
  - A SECURITY PROCESS THAT RELIES ON THE UNIQUE BIOLOGICAL CHARACTERISTICS OF INDIVIDUALS TO VERIFY WHO THEY SAY THEY ARE.
  - EXAMPLES INCLUDE FINGERPRINTS, RETINA SCANS, FACIAL RECOGNITION, IRIS RECOGNITION AND EAR AUTHENTICATION AMONG OTHERS .

WHAT IS A JWT?

- JSON WEB TOKEN OR JWT, IS A STANDARD FOR SAFELY PASSING SECURITY INFORMATION (SPECIFICALLY CLAIMS) BETWEEN APPLICATIONS IN A SIMPLE, OPTIONALLY VALIDATED AND/OR ENCRYPTED, FORMAT.
- THE STANDARD IS SUPPORTED BY ALL MAJOR WEB FRAMEWORKS. (FLASK, DJANGO, EXPRESS ...)

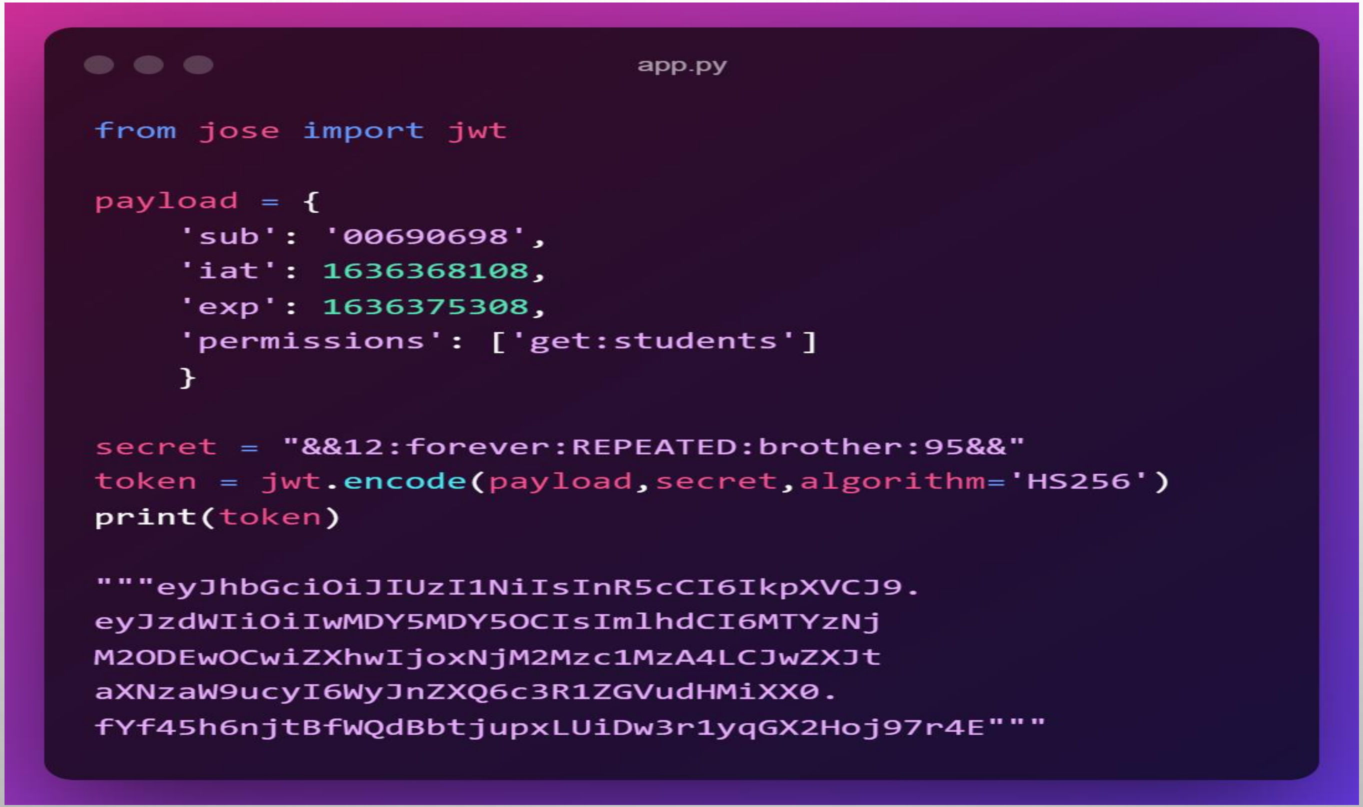
WHAT IS A CLAIM?

- A CLAIM IS A DEFINITION OR ASSERTION MADE ABOUT A CERTAIN PARTY OR OBJECT [TYPICALLY A USER].
  - EXAMPLES: ROLE, PERMISSION …
- SOME OF THESE CLAIMS AND THEIR MEANING ARE DEFINED AS PART OF THE JWT SPEC.
- STANDARDS CLAIMS ALLOW FRAMEWORKS TO BE ABLE TO CHECK STANDARD FIELDS SUCH AS EXPIRY (OR VALIDITY) AUTOMATICALLY WITHOUT REQUIRING ADDITIONAL CODE FROM THE DEVELOPER.
- EXAMPLES:
  - ISS (ISSUER): ISSUER OF THE JWT
  - AUD (AUDIENCE): RECIPIENT FOR WHICH THE JWT IS INTENDED
  - EXP (EXPIRATION TIME): TIME AFTER WHICH THE JWT EXPIRES

WHY WE USE JWTS

- THEY ARE SIMPLE,COMPACT AND USABLE.
- WE WILL USE THEM FOR:
  - AUTHENTICATION: TO IDENTIFY THE USER.
  - AUTHORIZATION: TO EVALUATE THE USER'S PERMISSIONS.
- THEY PROVIDE US WITH A WAY TO IMPLEMENT STATELESS SESSIONS.

SNIPPET: ENCODING A JWT

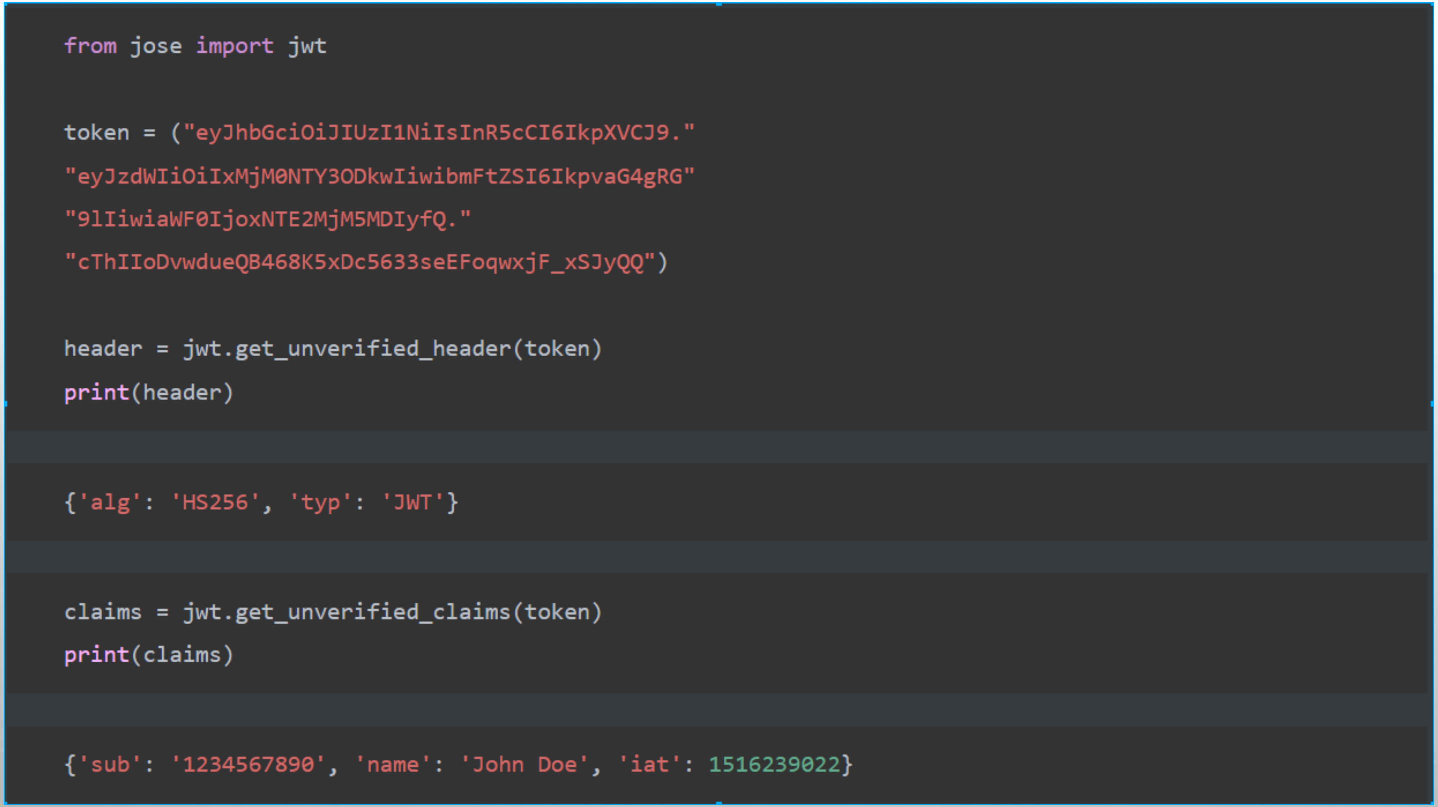

app.py

```python
from jose import jwt

payload = {
    'sub': '00690698',
    'iat': 1636368108,
    'exp': 1636375308,
    'permissions': ['get:students']
    }

secret = "&&12:forever:REPEATED:brother:95&&"
token = jwt.encode(payload,secret,algorithm='HS256')
print(token)

"""eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIwMDY5MDY5OCIsImlhdCI6MTYzNj
M2ODEwOCwiZXhwIjoxNjM2Mzc1MzA4LCJwZXJt
aXNzaW9ucyI6WyJnZXQ6c3R1ZGVudHMiXX0.
fYf45h6njtBfWQdBbtjupxLUiDw3r1yqGX2Hoj97r4E"""
```

DISSECTING A JWT

- A JWT IS MADE OF THE FOLLOWING PARTS:
  - HEADER: HOLDS METADATA SUCH AS ENCRYPTION TYPE
  - PAYLOAD: HOLDS USER CLAIMS
  - SIGNATURE: HOLDS THE DIGITAL SIGNATURE OF THE TOKEN

SNIPPET: DISSECTING A JWT



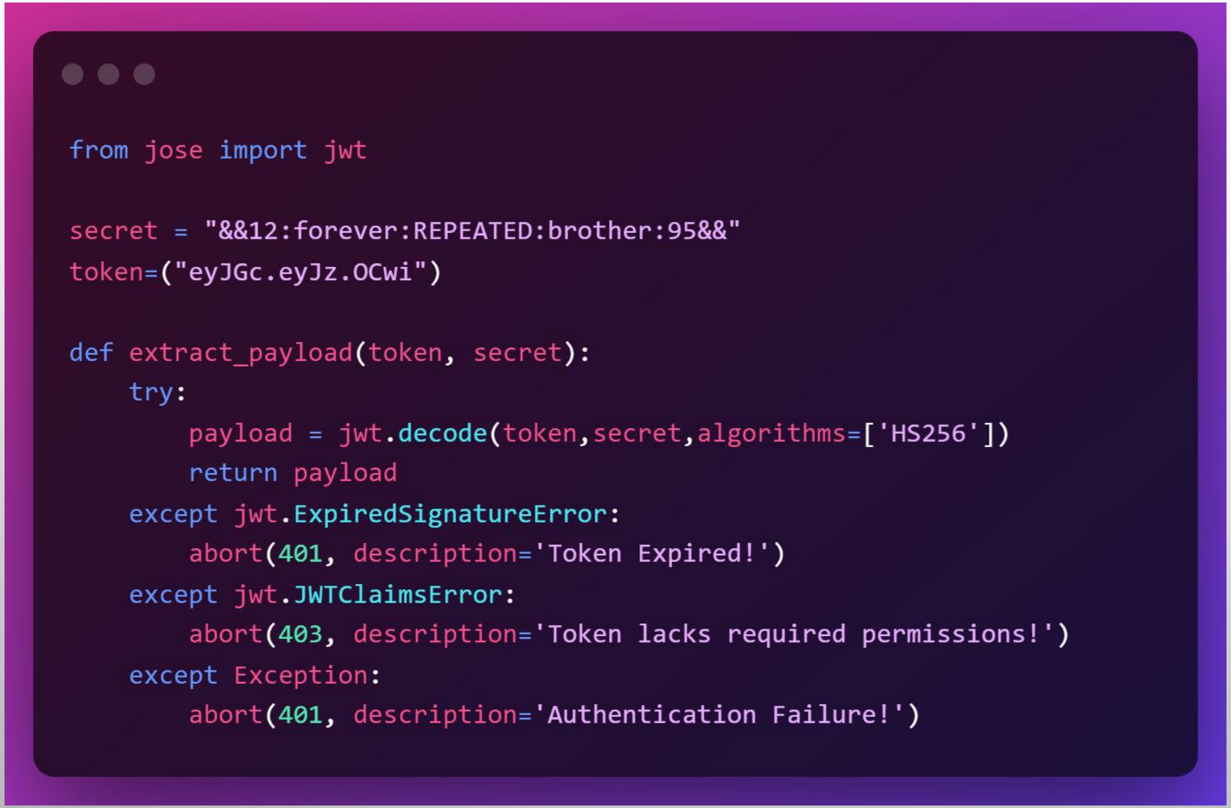
```
from jose import jwt

token = ("eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9."
"eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG"
"9lIiwiaWF0IjoxNTE2MjM5MDIyfQ."
"cThIIoDvwdueQB468K5xDc5633seEFoqwxjF_xSJyQQ")

header = jwt.get_unverified_header(token)
print(header)
```

```
{'alg': 'HS256', 'typ': 'JWT'}
```

```
claims = jwt.get_unverified_claims(token)
print(claims)
```

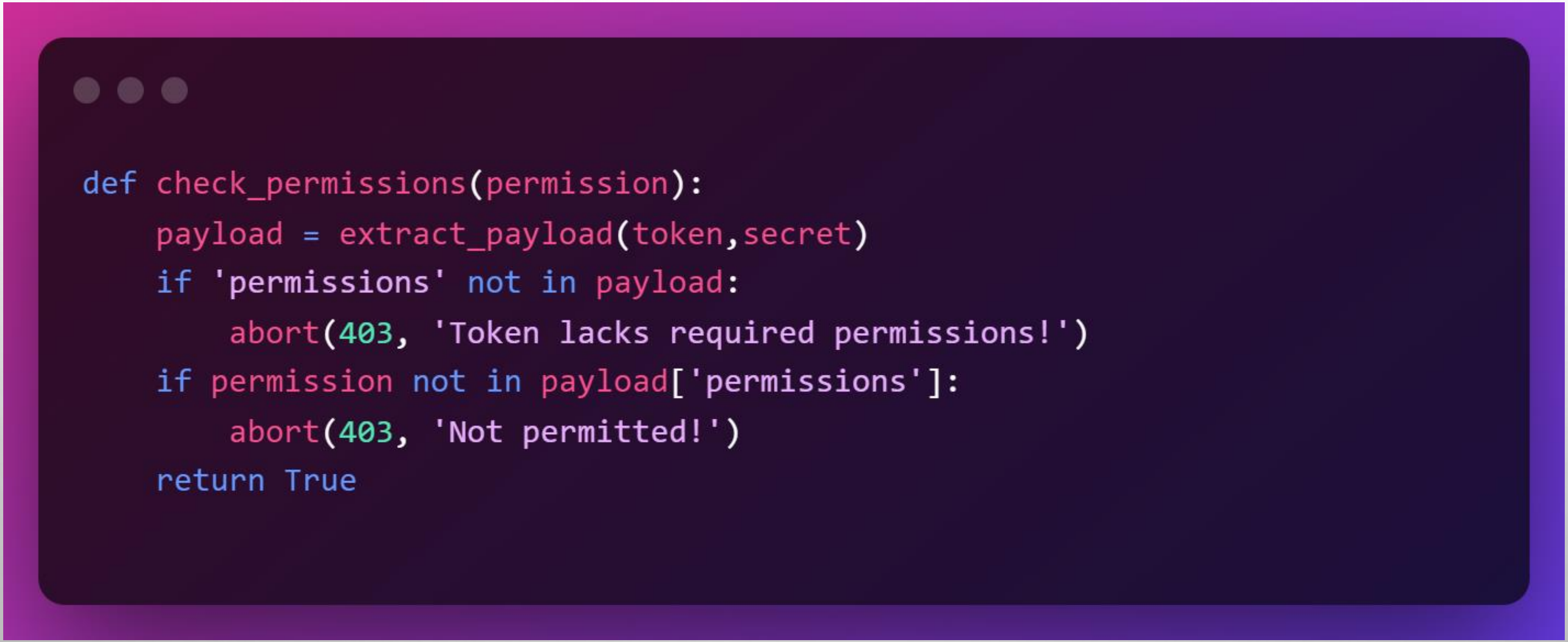
```
{'sub': '1234567890', 'name': 'John Doe', 'iat': 1516239022}
```

SNIPPET: VALIDATE JWT



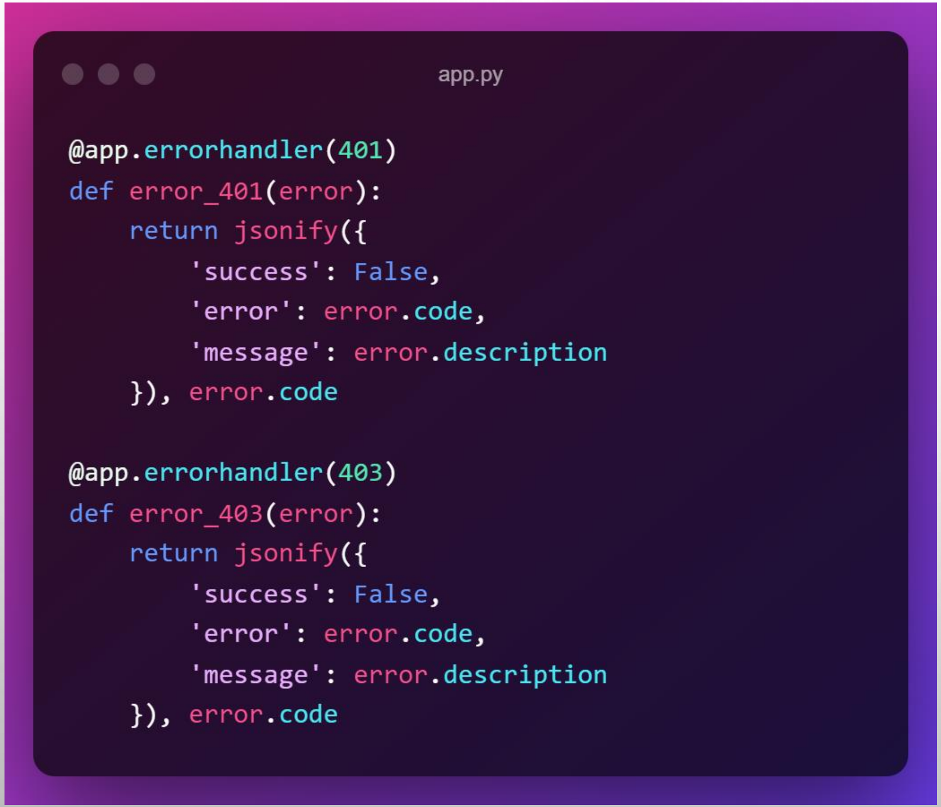
```python
from jose import jwt

secret = "&&12:forever:REPEATED:brother:95&&"
token=("eyJGc.eyJz.OCwi")

def extract_payload(token, secret):
    try:
        payload = jwt.decode(token,secret,algorithms=['HS256'])
        return payload
    except jwt.ExpiredSignatureError:
        abort(401, description='Token Expired!')
    except jwt.JWTClaimsError:
        abort(403, description='Token lacks required permissions!')
    except Exception:
        abort(401, description='Authentication Failure!')
```

SNIPPET: VALIDATE JWT



```python
def check_permissions(permission):
    payload = extract_payload(token,secret)
    if 'permissions' not in payload:
        abort(403, 'Token lacks required permissions!')
    if permission not in payload['permissions']:
        abort(403, 'Not permitted!')
    return True
```

SNIPPET: CUSTOM ERROR HANDLERS



```python
@app.errorhandler(401)
def error_401(error):
    return jsonify({
        'success': False,
        'error': error.code,
        'message': error.description
    }), error.code

@app.errorhandler(403)
def error_403(error):
    return jsonify({
        'success': False,
        'error': error.code,
        'message': error.description
    }), error.code
```

05 WHAT NEXT

WHAT NEXT

STUDY THE FOLLOWING LESSONS FROM "IDENTITY ACCESS MANAGEMENT" PART:

- PASSWORDS

QUESTIONS

THANK YOU

ADDITIONAL RESOURCES

- PREMIER OF PYTHON DECORATORS
- YOUTUBE VIDEOS ON DECORATORS FOR PYTHON BEGINNERS THEORY AND DEMO
- EXAMPLE OF TOKEN-BASED AUTHENTICATION WITH FLASK USING FLASK_JWT
- WHAT ARE JSON WEB TOKENS?