"REST APIs with Flask and Python" by Jose Salvatierra

ThadT · Follow

52 min read · Feb 9, 2020





The following are some notes I took as I went through the very excellent Udemy course titled "REST APIs with Flask and Python" by Jose Salvatierra, who also happens to maintain a superb blog on Python called Teclado. If you find the below snippets easy to understand and has a structure which you like, I highly recommend going through Jose's course and he even provides a quick summary of the project he goes through in this e-book.

Again, the following are purely content I got from Jose's course, and is meant for my easy referencing in future. Feel free to jump around the sections to get an appreciation of how the course is structured, but I doubt you will be able to understand a great deal by reading these unstructured notes below. Do go through Jose's course instead! It is extremely worth your time, and highly rewarding from a learning point of view. :)

Python Basics

Functions and String Formatting

- Functions are things that perfom actions or calculates outputs (or both).
- String formatting allows you to easily replace the placeholder with any other parameter of choice.

```
greeting = 'Hello, {}'

with_name = greeting.format(name)

print(with_name)
```

- f strings allow you to use multiple placeholders in a string print statement
- The use of :.2f after the variable means format that input to 2 decimal places (syntax formatting)

```
print(f '{square_feet} square feet is {square_metres:.2f} square metres')
```

Getting user inputs

```
name = input('Enter your name')
```

Variable types

Lists: Mutable

```
l = ['Apple', 'Bee', 'Cat']
```

Tuples: Immutable, unlike a list (cannot add or remove elements)

```
t = ('Apple', 'Bee', 'Cat')
```

Sets: Canot have duplicate elements and has no order

```
s = {'Apple', 'Bee', 'Cat'}
```

- For lists and tuples you can use this index notation—*print(l[0])* which means print the first item in the list l
- l.append('Smith')—adding an element to the end of the list
- l.remove('Bob')— Removing an element from the list
- s.add('Smith')—also adding the element but cannot add multiple of the same element
- *local_friends = friends.difference(abroad)*—The difference operation takes 2 variables and returns only the elements that exists only in the friends variable, but not the abroad variable (whichever variable is stated first is very important)
- *friends = local.union(abroad)*— The union operation unites 2 sets and gives total of both

Booleans

- True or false variables
- print(s == s)—Comparison operator
- Other operators include <, >, !=, etc.

- IF statements can be used with boolean operators. Remember identation is important for IF statements
- ELSE statements can be used for multiple conditions

```
if day_ofweek ==" Monday" :
print("Have a great start to your week!")
elif day_ofweek :: "Tuesday":
print("It's Tuesday.)
else:
print("Full speed ahead!")
```

The in Keyword

```
movies_watched = {"The Matrix", "Green Book", "Her"}

print("Her" in movies_watched)
```

- This would return TRUE
- Note: Another useful operator is abs

```
elif abs(number - user_number)==1:

print ("You were off by one.")
```

Loops

- While loop — So long while condition is FALSE, loop continues

```
while user_input != "n":
```

user_number = int(input("Guess our number: "))

This is the same as

while TRUE:

user_input = input("Would you like to play? (Y/n)")

if user_input == "n":

break

- For loop — Allows you to iterate through a list

for friend in friends:

print(f"{friend} is my friend.")

List Comprehensions

Example 1:

numbers = [1, 3, 5]

*doubled = [num * 2 for num in numbers]*

Example 2:

friends = ["Rolf", "Sam", "Samantha", "Saurabh", "Jen"]

starts_s = [friend for friend in friends if friend.startswith("S")]

- The new lists are stored in different locations in python

Dictionaries

- Key-value pairs

friend_ages = {"Rolf": 24, "Adam": 30, "Anne": 27}

- List of dictionaries

friends = [

{"name": "Rolf", "age": 24},

{"name": "Adam", "age": 30},

{"name": "Anne", "age": 27},

]

Destructuring Variables

- Brackets are normally not necessary when defining a variable unless you want to explicitly declare it as a tuple

people = [("Bob", 42, "Mechanic"), ("James", 24, "Artist"), ("Harry", 32, "Lecturer")]

for name, age, profession in people:

print(f "Name: {name}, Age: {age}, Profession: {profession}")

- Using underscore ("_") as indication to ignore variable

person = ("Bob", 42, "Mechanic")

name, _, profession = person

- Using *tail — Note that the * indicates a placeholder for the rest of all the other values except the head

*head, *tail = [1, 2, 3, 4, 5]*

Functions

- Functions are like callable variables
- Defining functions

def hello()

print ("Hello!")

- Calling functions

hello()

- Do not reuse names

- Global vs Local variables → Do not use same names in both
- Cannot use functions before they are defined
- You put in parameters when defining functions: *def add(x, y)*
- You use arguments when calling the function: *add(5, 3)*
- Positional arguments — Which order you place the arguments matters; Vs. Keyword/Named arguments

E.g.

say_hello("Bob", "Smith") vs. *say_hello(surname = "Bob", name = "Smith")*

- Default parameters: *def add(x, y=8)*
- If you define a variable before defining a function which uses that variable as a parameter, even if you subsequently change the value of the variable, the function will stick to the originally defined variable value

default_y = 3

def add(x, y=default_y):

sum = x + y

print(sum)

default_y=4 → Not going to affect the function

- Functions return None → You need to return the value instead of printing it when defining the function (otherwise if you subsequently call the function and try to print the result, you will get None)

Lambda Functions

- Function without a name and used to return values (input > output), not to perform actions

lambda x, y: x + y

- You can give it a name if you want
- To use the lambda function:

print((lambda x, y: x + y)(5, 7))

- Lambda functions can be used similar to other types of functions:

doubled = [double(x) for x in sequence]

OR

doubled = map(double, sequence) → Normally this map function is used for consistency with other programming languages' syntax like JS

OR

*doubled = [(lambda x: x*2)(x) for x in sequence]* → But this is hard to read, so we use map instead

*doubled = list(map(lambda x: x*2, sequence))*

Dictionary Comprehensions

- Similar to list comprehensions, but result is a dictionary → Need to assign key value pairs

users = [(0, "Bob", "password"), (1, "Rolf", "bob123"), (2, "Jose", "longpassword")]

username_mapping = {user[1]: user for user in users}

print(username_mapping)

- Example above gets the username and associates with the entire user tuple for each user → Result is a dictionary with the keys as the usernames and values are the entire tuple
- Used in username, password mapping

_, username, password = username_mapping[username_input]

Note: First underscore is there because we don't care about the ID

Unpacking Arguments

- The below defines a function,which arguments are defined as a tuple of arguments

*def multiply(*args):*

print(args)

total = 1

for arg in args:

*total = total*arg*

return total

multiply (1, 2, 5)

- This is a way of collecting multiple arguments into a single variable when calling a function
- Can do the opposite too, and use the asterisk to de-structure arguments into multiple parameters (need to have same number of values as the defined function)

def add(x, y):

return x + y

nums = [3, 5]

*add(*nums)*

- If the nums are dictionaries

nums = {"x": 15, "y": 25}

print(add(x=nums["x"], y=nums["y"]))

→ This can be written as *print(add(**nums))* instead for easier reading

Below is an application of how this de-structuring of arguments can be used:

```
def multiply(*args):
    print(args)
    total = 1
    for arg in args:
        total = total * arg

    return total


def apply(*args, operator):
    if operator == "*":
        return multiply(*args)
    elif operator == "+":
        return sum(args)
    else:
        return "No valid operator provided to apply()."


print(apply(1, 3, 6, 7, operator="*"))
```

Unpacking Keyword Arguments (**kwargs)

- The 2 asterisks can be used in a function to collect named arguments into a dictionary, or conversely can be used in a function call to unpack a dictionary into keyword arguments

```
def named(**kwargs):
    print(kwargs)

named(name="Bob", age=25)
```

Collecting named arguments into a dictionary

The result of the above will be *{'name': 'Bob', 'age': 25}*

Unpacking a dictionary into keyword arguments

The result of the above will be *Bob 25*

Object-Oriented Programming

- Within classes you can define multiple methods (methods are what we call functions when inside of a class)
- *__init__(self)* → The "self" is just a variable, and can be named anything else but "self" is the convention used in Python. init is the method inside the Student class
- You call a class, the init method runs, and what you get back is the object you created and in the init method you have the opportunity to set and change its properties.
- Classes are always named with first character as CAPS

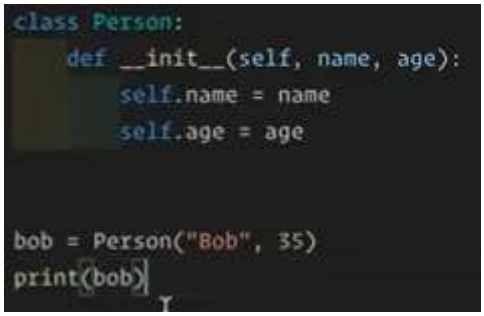
```python
class Student:
    def __init__(self):
        self.name = "Rolf"
        self.grades = (90, 90, 93, 78, 90)

    def average_grade(self):
        return sum(self.grades) / len(self.grades)


student = Student()
print(student.average_grade())
```

- The point of object oriented programming is to define other methods such as the average method inside the class. And these methods will take self (i.e. the object that was created initially with the init method) as a parameter.
- All methods inside a class need to have a parameter such as self but they can also take more arguments

```python
class Student:
    def __init__(self, name, grades):
        self.name = name
        self.grades = grades

    def average_grade(self):
        return sum(self.grades) / len(self.grades)


student = Student("Bob", (100, 100, 93, 78, 90))
student2 = Student("Rolf", (90, 90, 93, 78, 90))

print(student.name)
print(student.average_grade())
```

- Use *self.name = name* → Self.name which is the name property inside self will take the value of the name variable inside this method
- Magic Methods (__str__ & __repr__) → In Python methods with two underscores on each side are special methods also called magic methods

at times, because Python will call them for you in some situations → Mainly used for representation purposes (easier reading/ debugging)



- The result of the above is something like <__*main*__.*Person object at 0X106f9bc70*> which is Python printing out a string that represents the Bob object (a representation)

- When we use *__str__(self)*→ We are telling Python we want a string representation of this object → The result of the above is *Person Bob, 35 years old.*
- When we use *__repr__(self)* → The goal of the repr method is to be unambiguous, and if possible it should return a string that allows us to

recreate the original object very easily. → The result of the above is *<Person('Bob' , 35)>*

Class Methods and Static Methods

- **Instance Methods:** All functions inside the class that use the object as the first parameter, are called instance methods.
- Creating an object of type ClassTest = Creating an instance of ClassTest
- Used very often to produce an action that uses the data inside the object that you created earlier or to modify some sort of data inside self or the object

```
class ClassTest:
    def instance_method(self):
        print(f"Called instance_method of {self}")


test = ClassTest()
test.instance_method()
ClassTest.instance_method(test)
```

How to use an instance method

```
Called instance_method of <__main__.ClassTest object at 0x10c894c70>
Called instance_method of <__main__.ClassTest object at 0x10c894c70>
```

Result

- **Class Methods:** Instead of the instance, or the object, self, it actually takes a different parameter that we usually in Python, call cls (or whatever you like, this is a Python convention too)
- Class methods do not need to pass any parameters because we do not need an object/instance → Python will automatically pass ClassTest or cls

inside

- When you put *@classmethod* on top of the method definition, this will be the class itself
- *__main__* in the result just means you have called the class method (not very important)
- Used as factories

```
class ClassTest:
    def instance_method(self):
        print(f"Called instance_method of {self}")

    @classmethod
    def class_method(cls):
        print(f"Called class_method of {cls}")


ClassTest.class_method()
```

Using a class method

```
Called class_method of <class '__main__.ClassTest'>
```

Result

```python
class Book:
    TYPES = ("hardcover", "paperback")

    def __init__(self, name, book_type, weight):
        self.name = name
        self.book_type = book_type
        self.weight = weight

    def __repr__(self):
        return f"<Book {self.name}, {self.book_type}, weighing {self.weight}g>"

    @classmethod
    def hardcover(cls, name, page_weight):
        return cls(name, cls.TYPES[0], page_weight + 100)

    @classmethod
    def paperback(cls, name, page_weight):
        return cls(name, cls.TYPES[1], page_weight)


book = Book.hardcover("Harry Potter", 1500)
light = Book.paperback("Python 101", 600)
```

An example

- Example above shows how you can use a class method to create new objects using a class
- **Static Methods: Decorated with @staticmethod**
- No need to pass anything inside static methods (unlike instance method with self and class method with cls)
- Static methods are not really methods (i.e. do not have any information about the class/object), but just a function placed inside a class
- Used to just place a method inside a class (mainly for code organisation), very rarely used

```python
class ClassTest:
    def instance_method(self):
        print(f"Called instance_method of {self}")

    @classmethod
    def class_method(cls):
        print(f"Called class_method of {cls}")

    @staticmethod
    def static_method():
        print("Called static_method.")

ClassTest.static_method()
```

Using static methods

```
Called static_method.
```

Result

Class Inheritance

- Allows one class to take some methods and properties from another class

```python
class Device:
    def __init__(self, name, connected_by):
        self.name = name
        self.connected_by = connected_by
        self.connected = True

    def __str__(self):
        return f"Device {self.name!r} ({self.connected_by})"

    def disconnect(self):
        self.connected = False
        print("Disconnected.")
```

Defining a Device Class

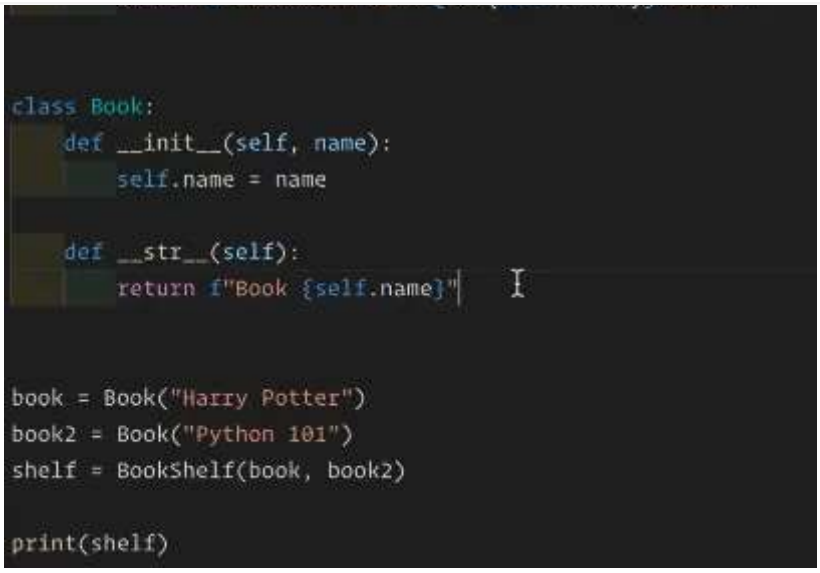
```python
class Printer(Device):
    def __init__(self, name, connected_by, capacity):
        super().__init__(name, connected_by)
        self.capacity = capacity
        self.remaining_pages = capacity

    def __str__(self):
        return f"{super().__str__()} ({self.remaining_pages} pages remaining)"

    def print(self, pages):
        if not self.connected:
            print("Your printer is not connected!")
            return
        print("Printing {pages} pages.")
        self.remaining_pages -= pages
```

Using an Inherited Class from Device Called Printer

- Allows you to use all the methods from the parent class → Use *super().__init___*
- So, if you call a method that is not in the Printer or the Device class, Python will then go to the Object class and see if it's there. And if it's not there, then you'll get an error.

Class Composition

- Composition is a counterpart to inheritance so you build out classes that use other classes
- Composition allows your classes to be simpler and reduces the complexity of your code overall



- Inheritance means that a book is a bookshelf (issues with this kind of application for inheritance due to plain logic and the technical fact that you do not use any of the parameters in the inherited class), composition means that a bookshelf has many books (more logical approach).

Type Hinting (For Python 3.5 and later)

```python
class Book:
    TYPES = ("hardcover", "paperback")

    def __init__(self, name: str, book_type: str, weight: int):
        self.name = name
        self.book_type = book_type
        self.weight = weight

    def __repr__(self) -> str:
        return f"<Book {self.name}, {self.book_type}, weighing {self.weight}g>"

    @classmethod
    def hardcover(cls, name: str, page_weight: int) -> "Book":
        return cls(name, cls.TYPES[0], page_weight + 100)

    @classmethod
    def paperback(cls, name: str, page_weight: int) -> "Book":
        return cls(name, cls.TYPES[1], page_weight)
```

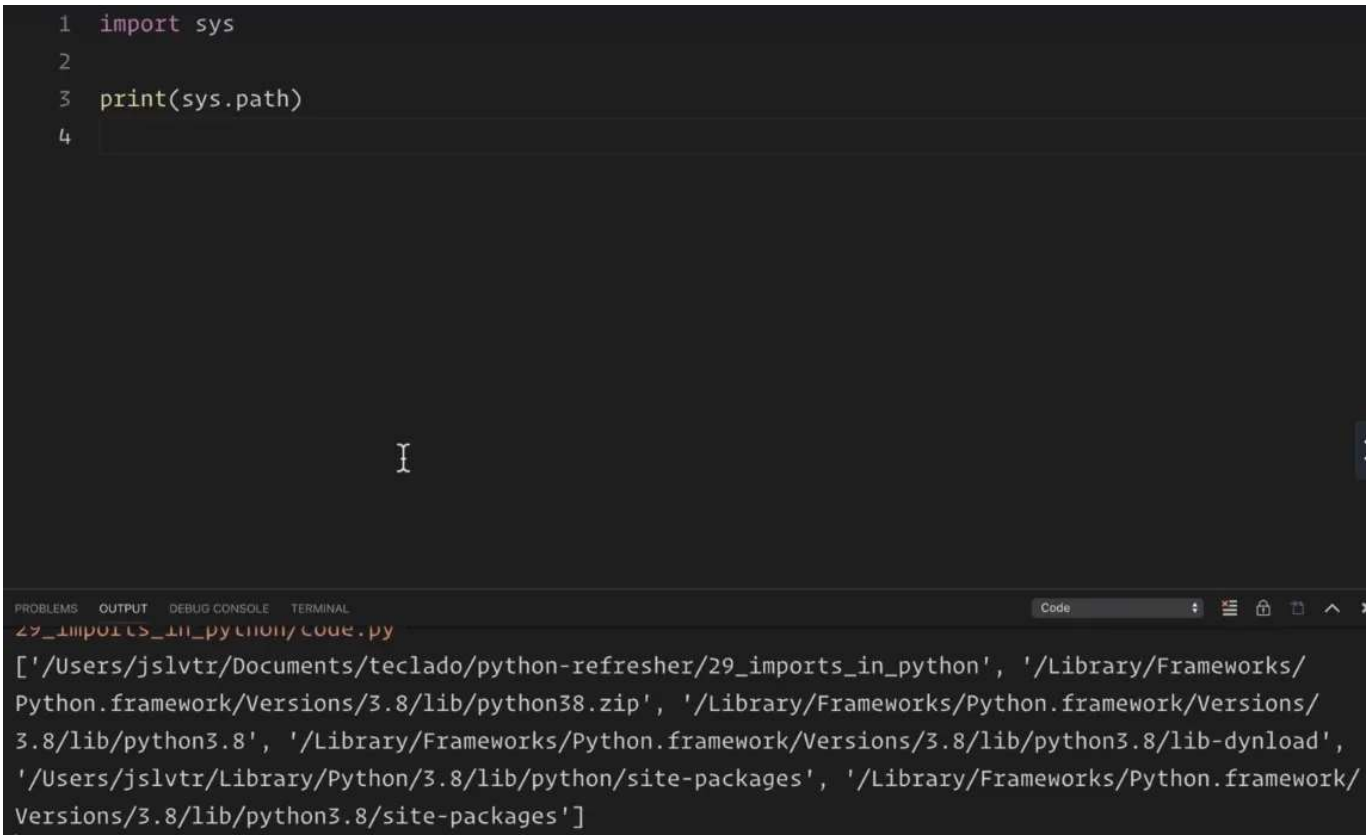
- *from typing import List*
- Type hinting helps you to get told if you pass in the wrong argument formats (examples below)
- *def __repr__(self) → str*
- *def hardcover(cls, name:str, page_weight: int) → "Book"*

Import

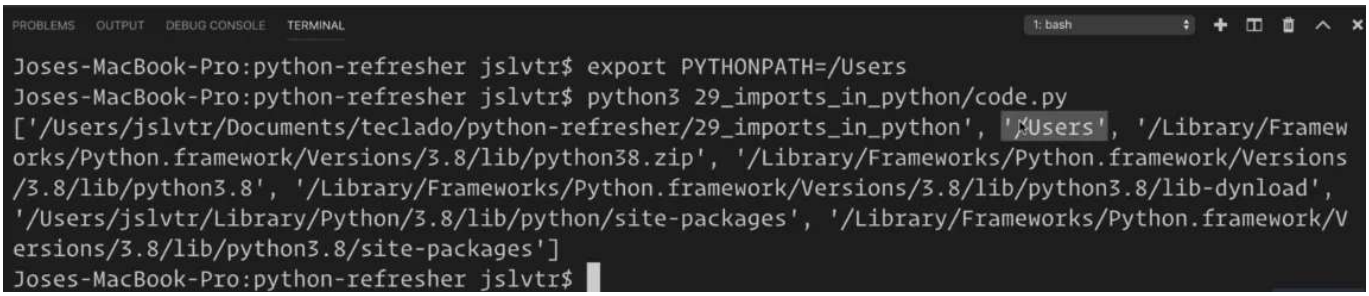
- Importing runs through a file and allows you to access the properties defined inside it
- Allows you to to import a specific thing from a folder → E.g. *from mymodule import divide*
- Or you can import the entire folder → E.g. *import mymodule* then after that do something like *mymodule.divide*
- import sys allows you to use the sys module that comes with Python, and that unlocks some certain system functionalities, and one of them is

sys.path.

- sys.path is the import paths where Python will look in order to find files to import.



- You get a whole bunch of different paths here inside a list and the first path that Python is going to look at is the first one in this list, and if it is not there, it will move to the next path and so on until the end where it will return error (ModuleNotFound, No module named sysasdf)
- First path is always the path of the file you ran
- The second path, if defined, is an environment variable called PYTHONPATH. In UNIX systems, you can do export PYTHONPATH, and then, give it a path. In Windows, instead of export, you'll have to use the keyword set to set an environment variable.



Exporting PYTHONPATH

- Python keeps track of the things that have already been imported, and if they have, you won't run through them again.

Relative Imports

- A relative import is one that can import from the current folder that the file is in. But it cannot import unless there is a folder name in the import path, in the module path.
- When you're trying to do a relative import, what Python's going to do is it's going to remove the file and it's just going to try to append whatever you import from into the rest of the import path.
- E.g. If we go to mylib (libs.mylib)and then we do from .operations import operator, Python will remove the file and try to put operations at the end (i.e. libs.operations) and import from there.

```
from .operations import operator

print("mylib.py:", __name__)
```

- Using double dots means it just skips the parent folder and goes up to the next one (need to be at least 2 levels deep)

```
print("operator.py: ", __name__)

from ..mylib import *
```

Errors in Python

- In Python, errors are often used for flow control, you're going to then be able to catch them in your code and handle them

```
def divide(dividend, divisor):
    if divisor == 0:
        raise ZeroDivisionError("Divisor cannot be 0.")

    return dividend / divisor


grades = []

print("Welcome to the average grade program.")
average = divide(sum(grades), len(grades))

print(f"The average grade is {average}.")
```

- Creating an exception object, which is another class defined inside Python
- Try-Except block allows you to handle the same types of errors in different places in your program (e.g. if zero division error occurs elsewhere you can have a different error message) → Ability to handle program context errors rather than just the mathematical error

```
students = [
    {"name": "Bob", "grades": [75, 90]},
    {"name": "Rolf", "grades": [50]},
    {"name": "Jen", "grades": [100, 90]},
]

try:
    for student in students:
        name = student["name"]
        grades = student["grades"]
        average = divide(sum(grades), len(grades))
        print(f"{name} averaged {average}.")
except ZeroDivisionError:
    print(f"ERROR: {name} has no grades!")
else:
    print("-- All student averages calculated --")
finally:
    print("-- End of student average calculation --")
```

- Python has many other built-in errors like TypeError, ValueError, RuntimeError, etc.
- Creating your own custom error class

```python
class TooManyPagesReadError(ValueError):
    pass


class Book:
    def __init__(self, name: str, page_count: int):
        self.name = name
        self.page_count = page_count
        self.pages_read = 0

    def __repr__(self):
        return (
            f"<Book {self.name}, read {self.pages_read} pages out of {self.page_count}>"
        )

    def read(self, pages: int):
        if self.pages_read + pages > self.page_count:
            raise TooManyPagesReadError(
                f"You tried to read {self.pages_read + pages} pages, but this book only
                has {self.page_count} pages."
            )
        self.pages_read += pages
        print(f"You have now read {self.pages_read} pages out of {self.page_count}.")
```

First Class Functions

- In first-class function, just means that functions are just variables. And you can pass them in as arguments to functions and use them in the same way you would use any other variable.

```python
def divide(dividend, divisor):
    if divisor == 0:
        raise ZeroDivisionError("Divisor cannot be 0.")

    return dividend / divisor


def calculate(*values, operator):
    return operator(*values)


result = calculate(20, 4, operator=divide)
print(result)
```

```python
def search(sequence, expected, finder):
    for elem in sequence:
        if finder(elem) == expected:
            return elem
    raise RuntimeError(f"Could not find an element with {expected}.")


friends = [
    {"name": "Rolf Smith", "age": 24},
    {"name": "Adam Wool", "age": 30},
    {"name": "Anne Pun", "age": 27},
]


def get_friend_name(friend):
    return friend["name"]


print(search(friends, "Rolf Smith", get_friend_name))
```

Another example of how first class functions can be used

- E.g. A search function that takes in

1. A sequence of things to search through
2. What you expect to find
3. A function that will be used to extract information from each item in the sequence, match it against the expected value, and see if we find what we wanted.

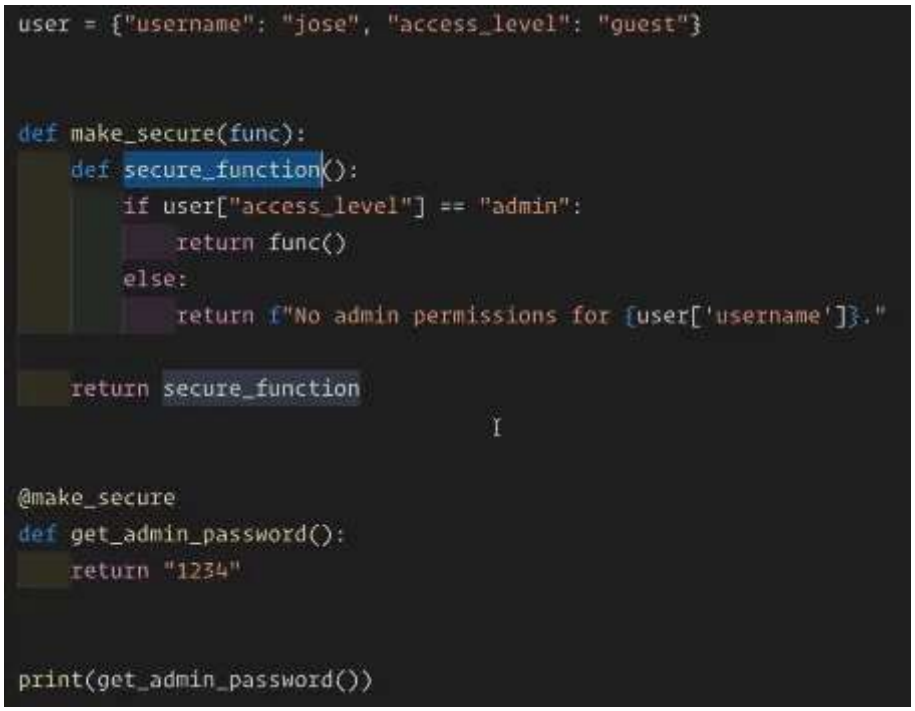
- So we will iterate over the sequence, then we will run the finder function on the element and see if it is equal to the expected value. And if it is, we will return the element otherwise we will raise a runtime error

Simple Decorators in Python

- Allow us to very easily modify functions

```python
user = {"username": "jose", "access_level": "guest"}


def get_admin_password():
    return "1234"


def make_secure(func):
    def secure_function():
        if user["access_level"] == "admin":
            return func()
        else:
            return f"No admin permissions"

    return secure_function


get_admin_password = make_secure(get_admin_password)

print(get_admin_password())
```

- *get_admin_password* is the function we want to secure, and this is passed to the *make_secure* function, which in turns defines another function (function inside a function), and this function (*secure_function*), when called, will check the user's access level and return calling the original function (*get_admin_password*).
- This simple decorator will create a function and replace the original function with this secure one. So that you can no longer call get_admin_password without having the admin access level.
- The @ syntax for decorators can be used instead of the *get_admin_password = make_secure(get_admin_password)*



```python
user = {"username": "jose", "access_level": "guest"}


def make_secure(func):
    def secure_function():
        if user["access_level"] == "admin":
            return func()
        else:
            return f"No admin permissions for {user['username']}."

    return secure_function


@make_secure
def get_admin_password():
    return "1234"


print(get_admin_password())
```

- Problem: When you replace this function, the name of the function changes. The get admin password still exists as a name inside Python but it is no longer registered as a function internally. Now the secure function is registered as a function internally, and there are some libraries that will actually use that internal name.
- Solution: Use another decorator by importing functools, which is short for function tools (a built in module in Python) and then we're going to decorate our secure function.

```python
import functools

user = {"username": "jose", "access_level": "guest"}


def make_secure(func):
    @functools.wraps(func)
    def secure_function():
        if user["access_level"] == "admin":
            return func()
        else:
            return f"No admin permissions for {user['username']}."

    return secure_function


@make_secure
def get_admin_password():
    return "1234"


print(get_admin_password.__name__)
```

- What this does is essentially it keeps the name and the documentation of this function if there are any
- Decorating functions with parameters because what you usually do with decorators, is you make them take unlimited numbers of arguments and keyword arguments (*args, **kwargs) so you can cater for everything (if you introduce a parameter to one of the functions, you need to put it into the other functions as well — which couples the two functions [not ideal]).

```python
def make_secure(func):
    @functools.wraps(func)
    def secure_function(*args, **kwargs):
        if user["access_level"] == "admin":
            return func(*args, **kwargs)
        else:
            return f"No admin permissions for {user['username']}."

    return secure_function


@make_secure
def get_password(panel):
    if panel == "admin":
        return "1234"
    elif panel == "billing":
        return "super_secure_password"


print(get_password("billing"))
```

- Adding parameters to decorators → Create a function (*make_secure*) that is used to create a decorator
- As make_secure now requires an access_level parameter, the benefit of that is now inside here, we can check that access level (i.e. @make_secure("admin")). So we can say that if the user's access level is equal to the access level that we want to secure for, then we have access to the function

```python
import functools

user = {"username": "jose", "access_level": "guest"}


def make_secure(access_level):
    def decorator(func):
        @functools.wraps(func)
        def secure_function(*args, **kwargs):
            if user["access_level"] == access_level:
                return func(*args, **kwargs)
            else:
                return f"No {access_level} permissions for {user['username']}."

        return secure_function

    return decorator


@make_secure("admin")
def get_admin_password():
    return "admin: 1234"
```

Mutability in Python

- Most things in Python are mutable except for tuples, strings, integers, floats, and booleans
- If you want to create an immutable class and immutable objects, just don't add any methods in them that can change the object's properties.
- Note: Never make a parameter equal to a mutable value by default → The function parameters evaluate when the function is defined, not when the function is called.

```python
from typing import List


class Student:
    def __init__(self, name: str, grades: List[int] = []):  # This is bad!
        self.name = name
        self.grades = grades

    def take_exam(self, result: int):
        self.grades.append(result)


bob = Student("Bob")
rolf = Student("Rolf")
bob.take_exam(90)
print(bob.grades)
print(rolf.grades)
```

- When you create two students, *self.grades* in both of them are names to the same list that was created when the function was created, not when the function was called. So as long as your using the default parameters, all your students they will share grades (not ideal).
- Solution: Make *grades: List[int]* equal to none and then you can make this *self.grades* to grades or empty list → If this is none then none or empty list will evaluate to empty list

```python
class Student:
    def __init__(self, name: str, grades: List[int] = None):  # This is bad!
        self.name = name
        self.grades = grades or []
```

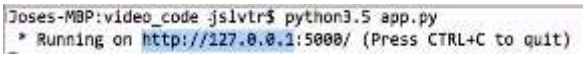
REST APIs

APIs

- An API is a program that takes in some data and gives back some other data, usually after processing it

- You can interact with APIs by using the *requests* library in Python
- Installing Flask (a Python library for creating REST APIs) with *pip3.5 install flask*
- *from flask import Flask* → Note that classes always start with uppercase and packages always in lowercase
- *app = Flask(__name__)* → A special python variable which essentially gives each file a unique name
- @app.route('/') → Defining the request using a decorator pointing to the endpoint

A very simple API



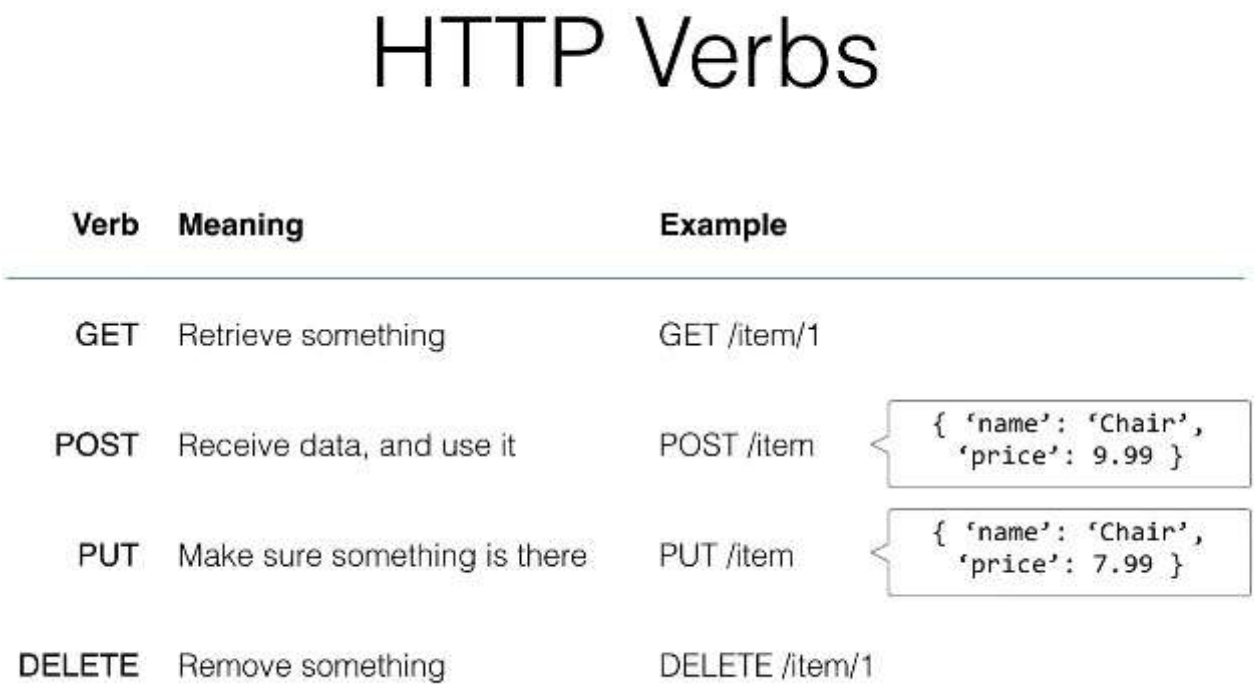
Console result

- Note: 127.0.0.1 is a special IP address that is reserved for your computer, specifically.

HTTP Verbs

- HTTP is a protocol that enables communications between 2 internet connected elements

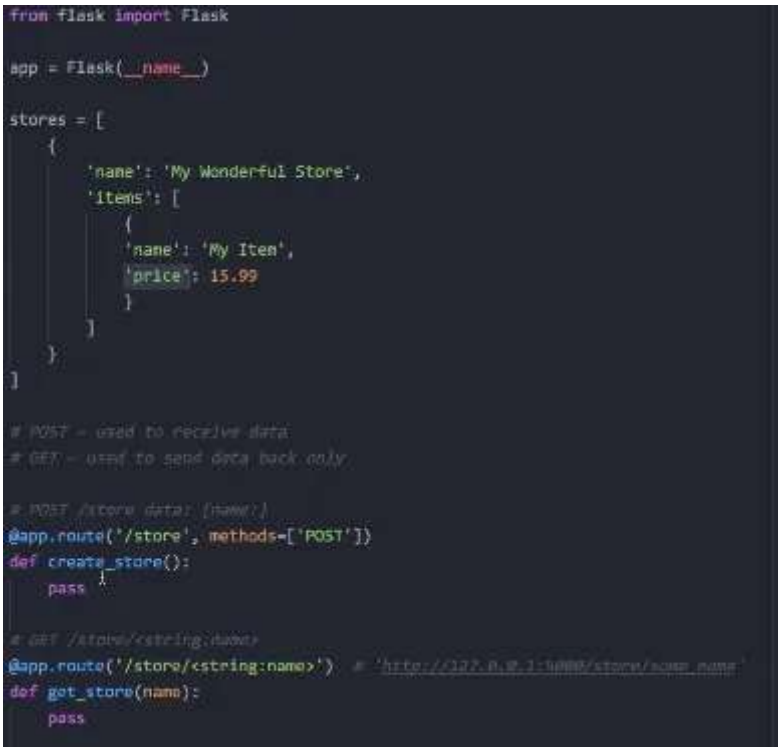
- Web server request example → Get/http/1.1, and the host is www.google.com
- The server can give you errors if the path (/) is not found, or HTTP is not supported, or if server is unavailable
- The server can also return HTML code/text/nothing
- There are many different HTTP verbs



REST Principles

- REST is a way of thinking about how a web server responds to your requests and how a web server behaves in general
- It doesn't respond with just data. It responds with something called resources.

- All of these four requests have the same endpoint */item/chair*. That's because they are all accessing the same resource.
- REST is also stateless → One request cannot depend on any other requests. So the server only knows about the current request and not about any previous requests.
- E.g. I log in to Twitter and the server responds with some data. After that, the server doesn't "remember" if I am logged on or not, it needs to retrieve from somewhere. That's why we need to put in a token in our subsequent interactions so that the server knows I have logged in previously.
- Creating endpoints: By default, when you use *@app.route*, that is a GET request, therefore for all other verbs you need to define it explicitly



```
from flask import Flask

app = Flask(__name__)

stores = [
    {
        'name': 'My Wonderful Store',
        'items': [
            {
            'name': 'My Item',
            'price': 15.99
            }
        ]
    }
]

# POST - used to receive data
# GET - used to send data back only

# POST /store data: {name:}
@app.route('/store', methods=['POST'])
def create_store():
    pass

# GET /store/<string:name>
@app.route('/store/<string:name>')  # 'http://127.0.0.1:5000/store/some_name'
def get_store(name):
    pass
```

- Remember to create a list of dictionaries (i.e. a list of stores with various parameters) to allow interactions
- Note: We are coding for the behavior of the web server, so when we **receive** a POST request from a browser, we need to create a store

JSON and a Simple GET Request

- JSON is something like a dictionary (but it is not) or set of key value pairs → JSON is actually text/a long string suitable for sending over the internet, which Javascript has to read and convert to a suitable format like a dictionary for Python to interpret
- Flask has a method called *jsonify* which takes in a dictionary and converts it to JSON (remember to import)
- Return *jsonify(stores)* converts the stores variable into JSON. However, JSON is a dictionary and our stores variable is not a dictionary, but a list.

Problem is, JSON cannot be a list. → Solution: We make it a dictionary by introducing ({'stores': stores}) to the original *jsonify(stores)*

- When you run this, you get a JSON representation of our stores, each of which is a dictionary with one key called stores
- Note: JSON always uses double quotes and never single quotes, and it always has to start with a dictionary, so you cannot return a list only

Implementing the Other Endpoints

Creating a new store

- Must first import *request*

- *request_data= request.get_json()* → This request is the request that was made to this endpoint
- We will then create a new store (a dictionary) with a name (*request_data['name']*) because that's gonna be the JSON.
- New store will be appended to the empty list
- Important to *return jsonify(new_store)* because if not, we will try to return a dictionary and that will fail because we have to return a string.

GET

```
# GET /store/<string:name>
@app.route('/store/<string:name>')  # 'http://127.0.0.1:5000/store/some_name'
def get_store(name):
    for store in stores:
        if store['name'] == name:
            return jsonify(store)
    return jsonify({'message': 'store not found'})
```

- To retrieve a specific store, go over the stores, finding the one that matches this name, and return that one. If none matches, return an error message.

```
# GET /store/<string:name>/item
@app.route('/store/<string:name>/item')
def get_items_in_store(name):
    for store in stores:
        if store['name'] == name:
            return jsonify({'items': store['items']})
    return jsonify({'message': 'store not found'})
```

- Retrieval of specific item in store utilizes same sequence

PUT

- Iterate through stores, if the store name matches our name, then we're going to create a new item and that item is going to be one with a name and a price which is going to be the request.

```
# POST /store/<string:name>/item {name:, price:}
@app.route('/store/<string:name>/item', methods=['POST'])
def create_item_in_store(name):
    request_data = request.get_json()
    for store in stores:
        if store['name'] == name:
            new_item = {
                'name': request_data['name'],
                'price': request_data['price']
            }
            store['items'].append(new_item)
            return jsonify(new_item)
    return jsonify({'message': 'store not found'})
```

- PUT is idempotent (i.e. even the thing that you are doing with this request has already happened, you can still do the request again and it won't fail)

Calling API Using JavaScript (Not Common, More For Knowledge)

```
<html>
<head>
<script type="text/javascript">
    function httpGetAsync(theUrl, callback) {
        var xmlHttp = new XMLHttpRequest();
        xmlHttp.onreadystatechange = function() {
            if (xmlHttp.readyState == 4 && xmlHttp.status == 200)
                callback(xmlHttp.responseText);
        }
        xmlHttp.open("GET", theUrl, true); // true for asynchronous
        xmlHttp.send(null);
    }
</script>
</head>
<body>

<div id="myElement">
    Hello, world!
</div>

</body>
</html>
```

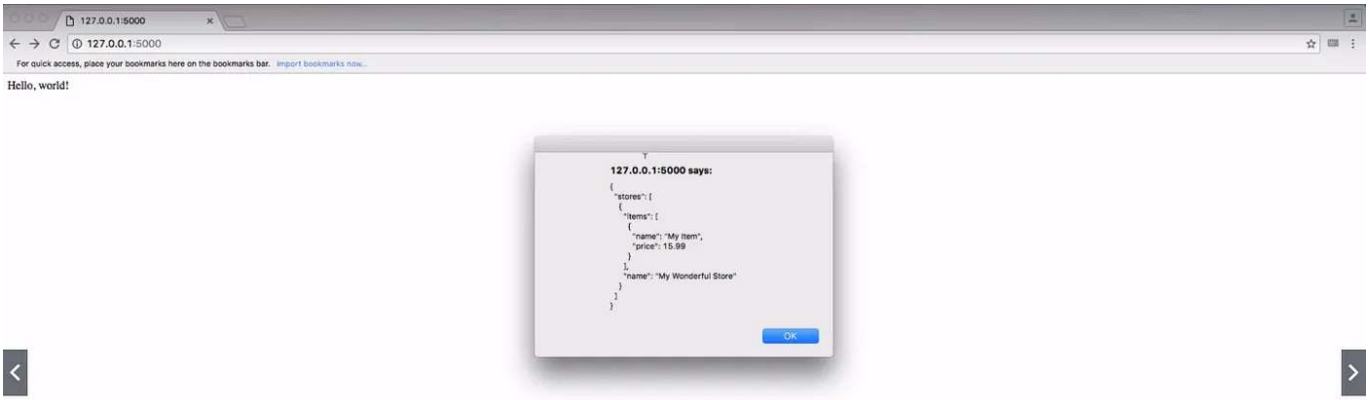
- HTML file above, with a JS script in the head tag
- This function calls a URL using a GET request, and then when it's done, it calls something else that we define
- Rendering HTML code from within your Flask application (if you want to create web apps from Flask)

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

app = Flask(__name__)

stores = [
    {
        'name': 'My Wonderful Store',
        'items': [
            {
            'name': 'My Item',
            'price': 15.99
            }
        ]
    }
]

@app.route('/')
def home():
    return render_template('index.html')
```

- Import *render_template,* then create another endpoint, such as '*home*', that is going to return a render_template of 'index.html' → When you run app.py, you access the root endpoint
- Calling the JS function which creates the GET request → Inside the function, say 'httpGetAsync', pass the URL that we want to call, and finally a callback function.

```html
<html>
<head>
<script type="text/javascript">
    function httpGetAsync(theUrl, callback) {
        var xmlHttp = new XMLHttpRequest();
        xmlHttp.onreadystatechange = function() {
            if (xmlHttp.readyState == 4 && xmlHttp.status == 200)
                callback(xmlHttp.responseText);
        }
        xmlHttp.open("GET", theUrl, true); // true for asynchronous
        xmlHttp.send(null);
    }

    httpGetAsync('http://127.0.0.1:5000/store', function(response) {
        alert(response);
    })
</script>
</head>
<body>

<div id="myElement">
    Hello, world!
</div>

</body>
</html>
```

- The function callback also receives the response content, which is some JSON string. And what we're going to do in this function, is we're just going to throw an alert with the response.

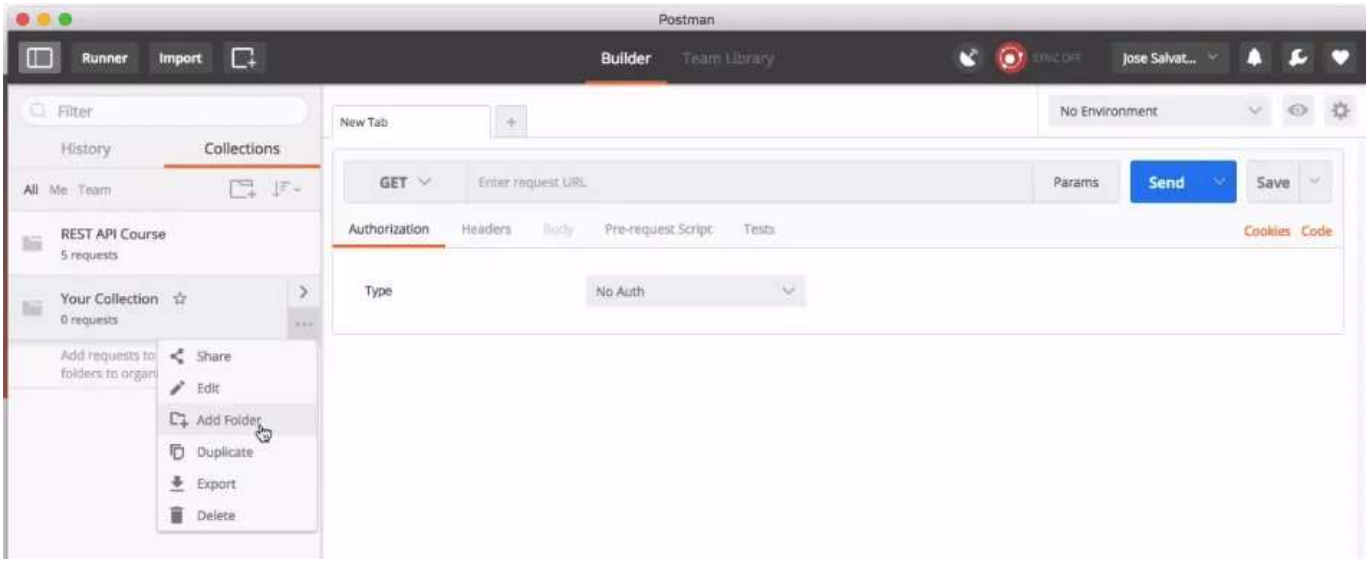


The alert with a JSON content

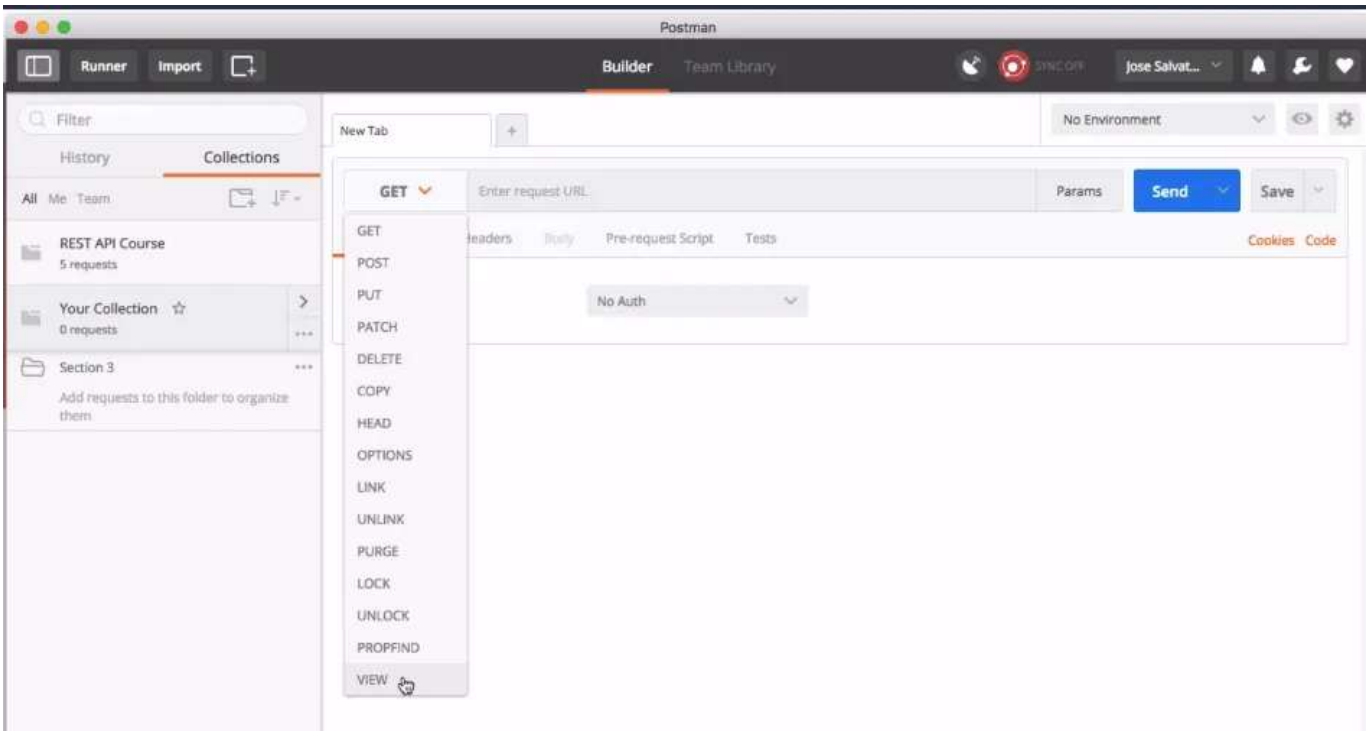
- This is how JavaScript , which is always running on the browser, has to call your API to retrieve data from it, so that it can then show it on the website.

Using Postman for API Testing

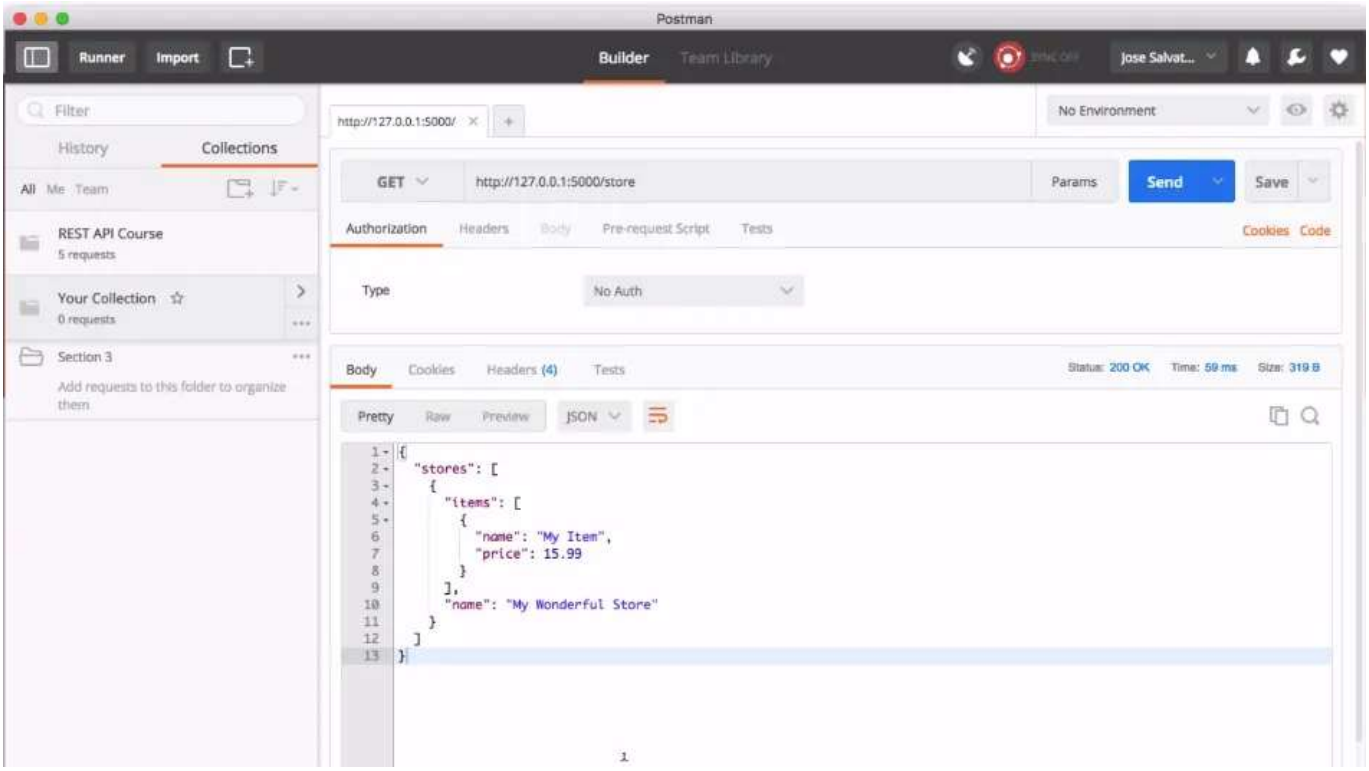
- Professional tool used to test APIs
- Accessing endpoints



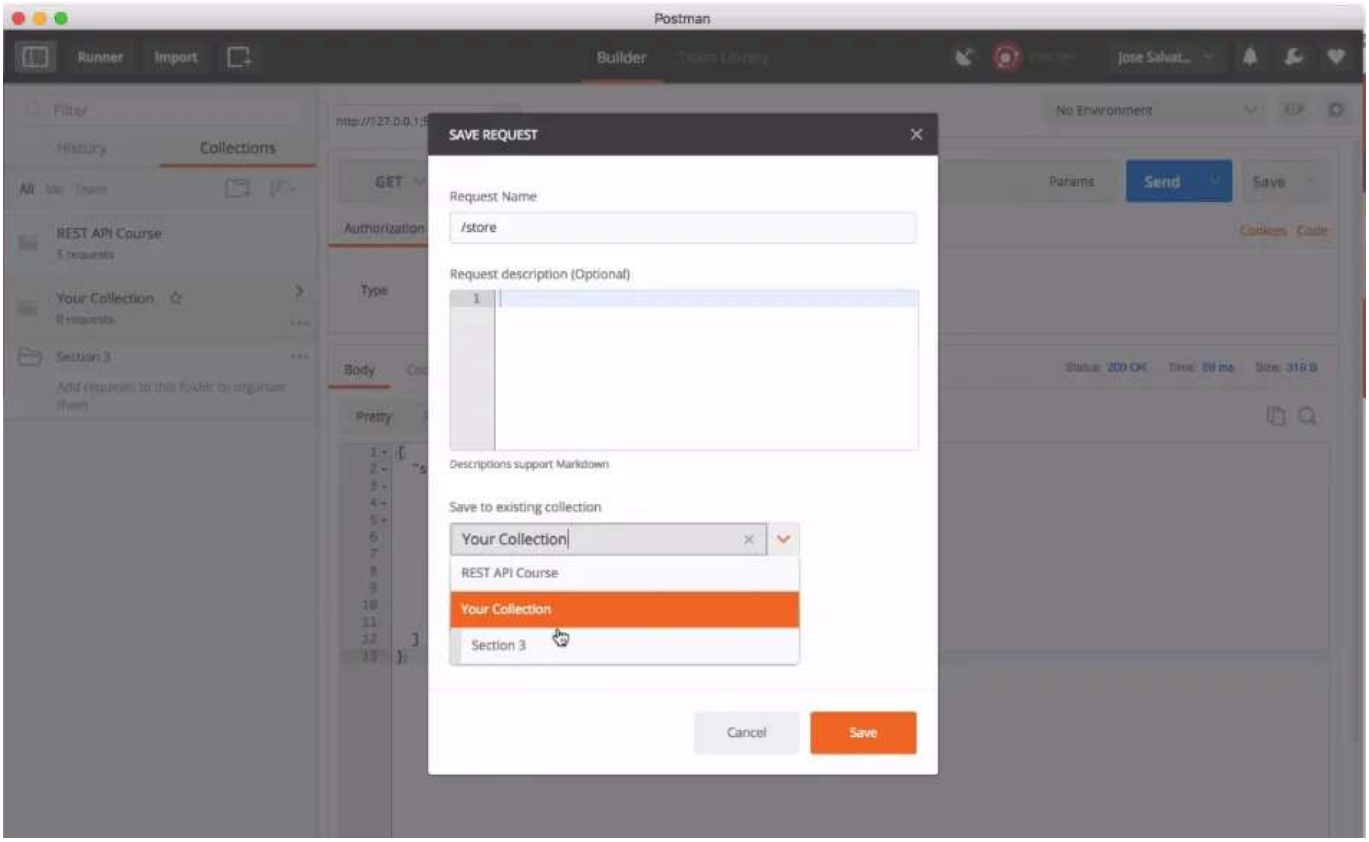
Main console for creating folders on left bar



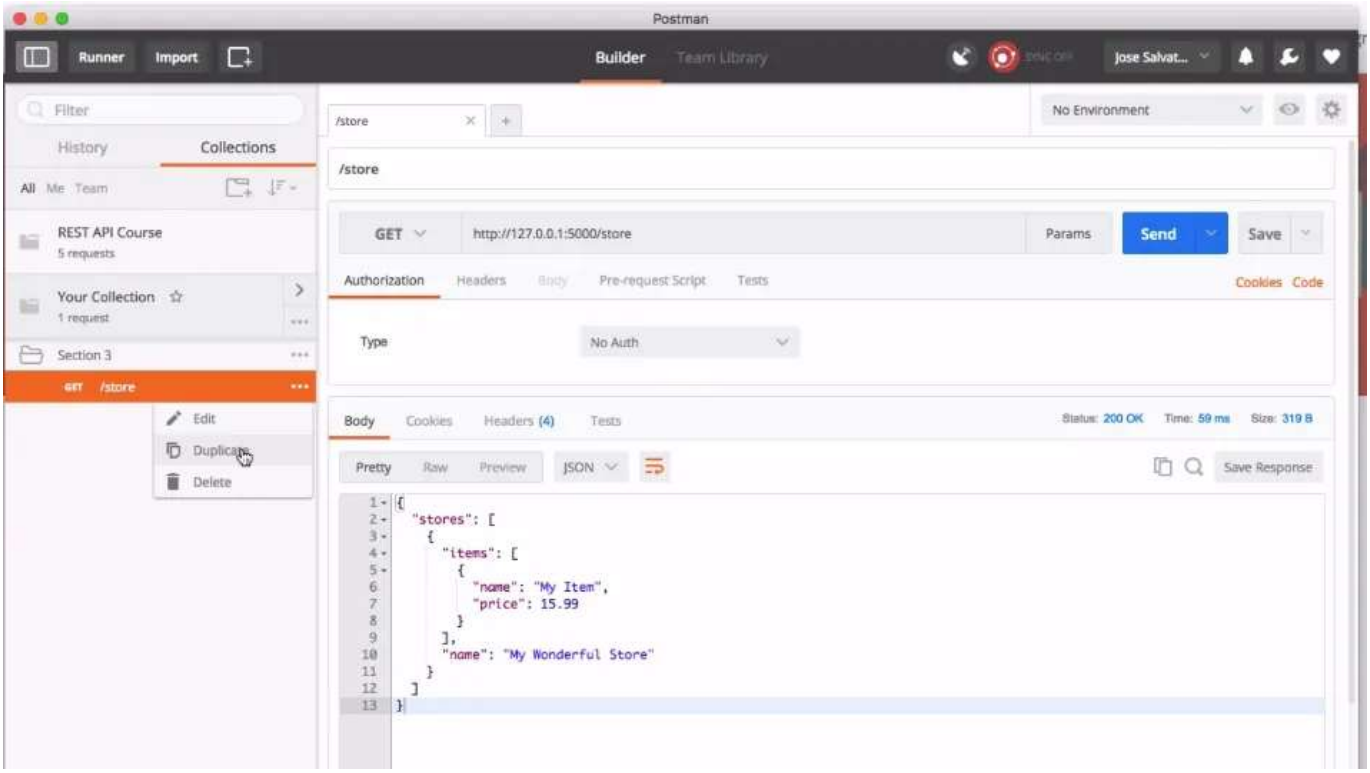
Actions bar with HTTP verbs



Firing first API call

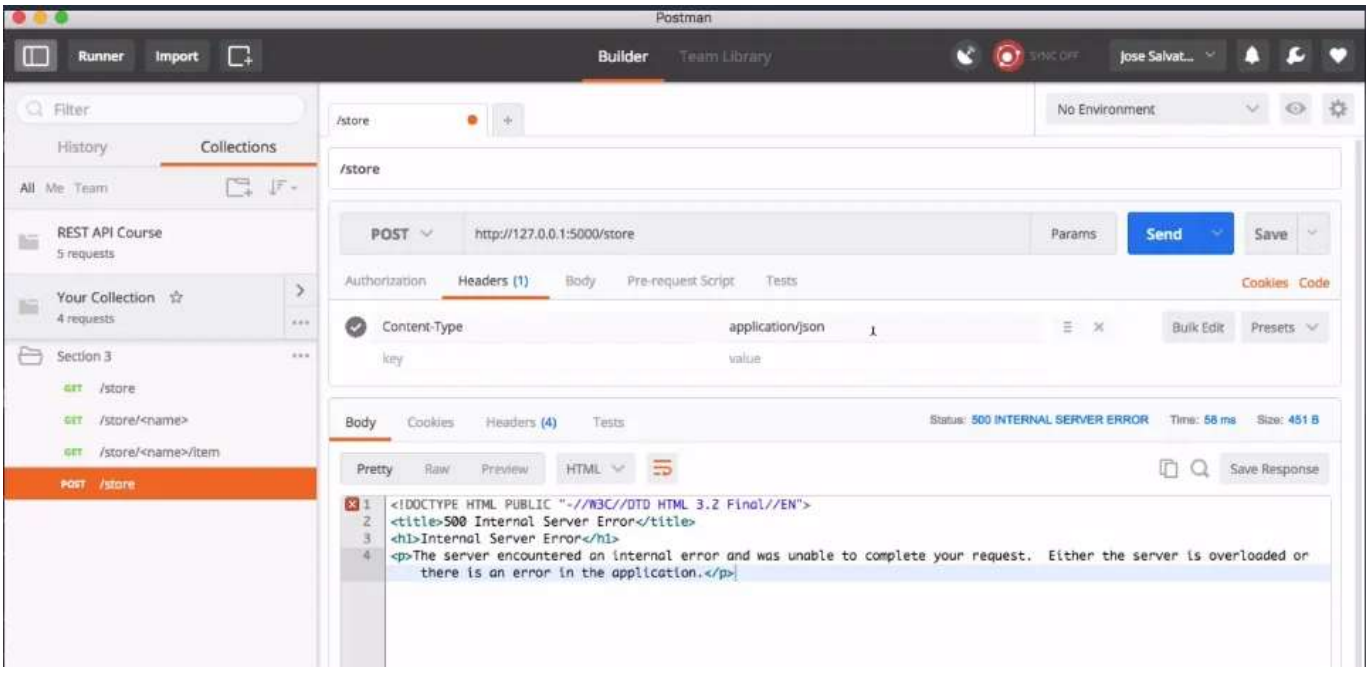


Saving requests



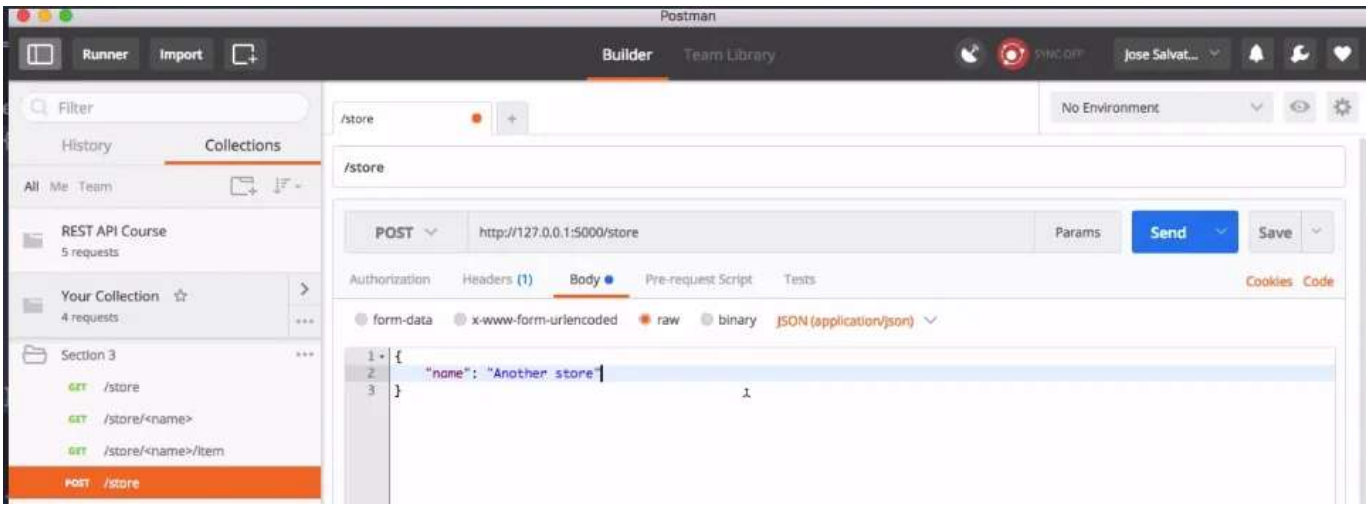
Duplicating requests

- Creating



Manipulating headers

- So the first thing that flask is going to do when it receives a request is look at the headers to try and understand sort of what this request is.
- In the headers we can say things like what sort of data, what type of data we're sending. So this is a set of key value pairs and the first header is going to be Content-Type and the value is going to be application/json.



Putting JSON inputs into the body

- Note: Whenever we restart the server, we lose the applications memory, and therefore the stores variable (whatever you juts posted) gets deleted. Better solution is to save to a database/file

Virtual envs and setting up Flask-RESTful

- Flask-RESTful is an extension to the Flask library, but even though it reduces the flexibility of Flask, it makes us adhere to the REST standard

pip3.5 freeze gives you the libraries you have installed and the versions you've got these libraries on

- If we now create another application and Flask has been updated, we can choose two options

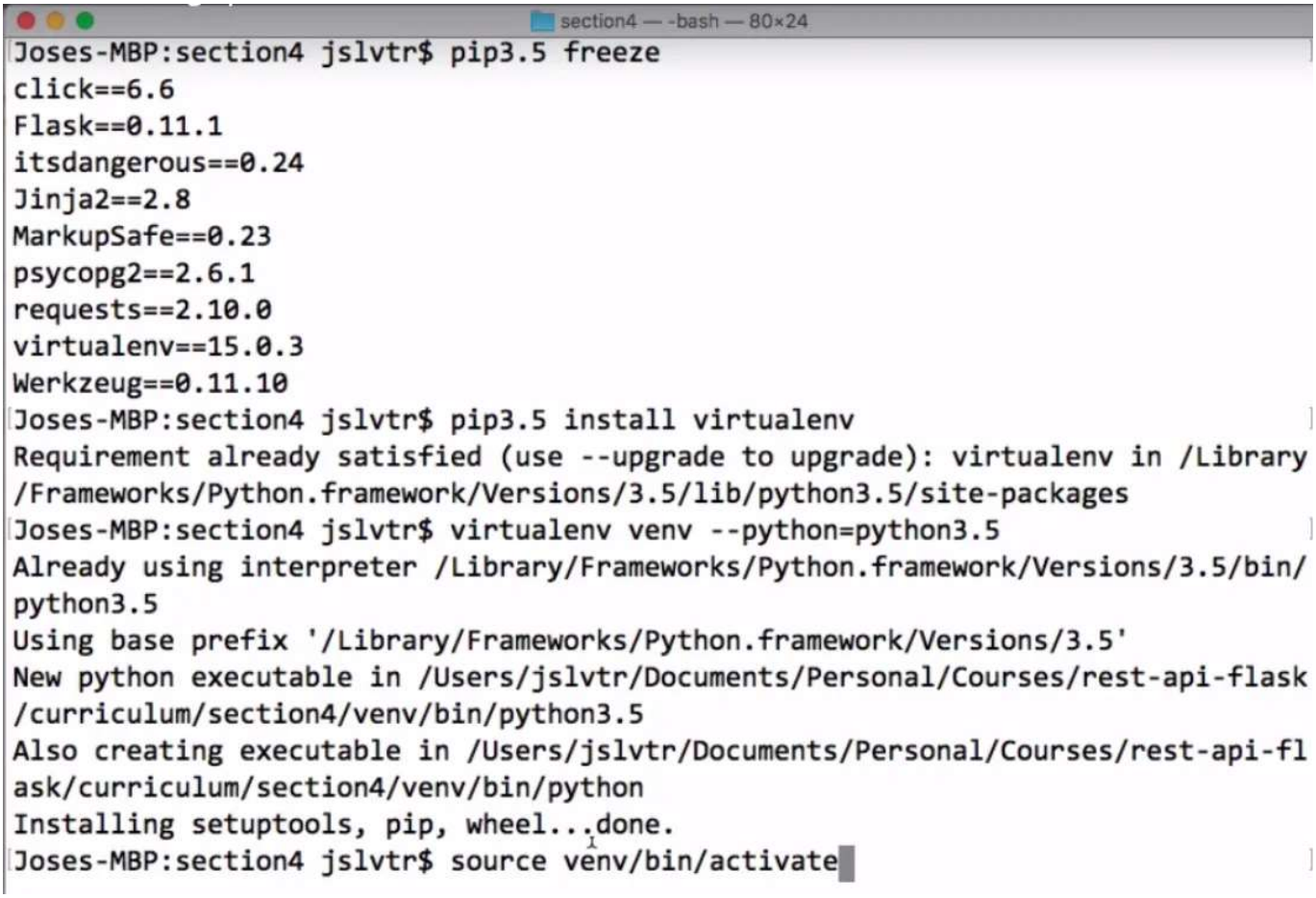
1. Update Flask → But may break old applications
2. Let the old applications use the old versions, but start new projects using the latest version (make sure projects don't share libraries) → i.e. Use virtual environments that mimic reinstalling Python with no libraries.

- Installing virtual environments: *pip3.5 instal virtualenv*

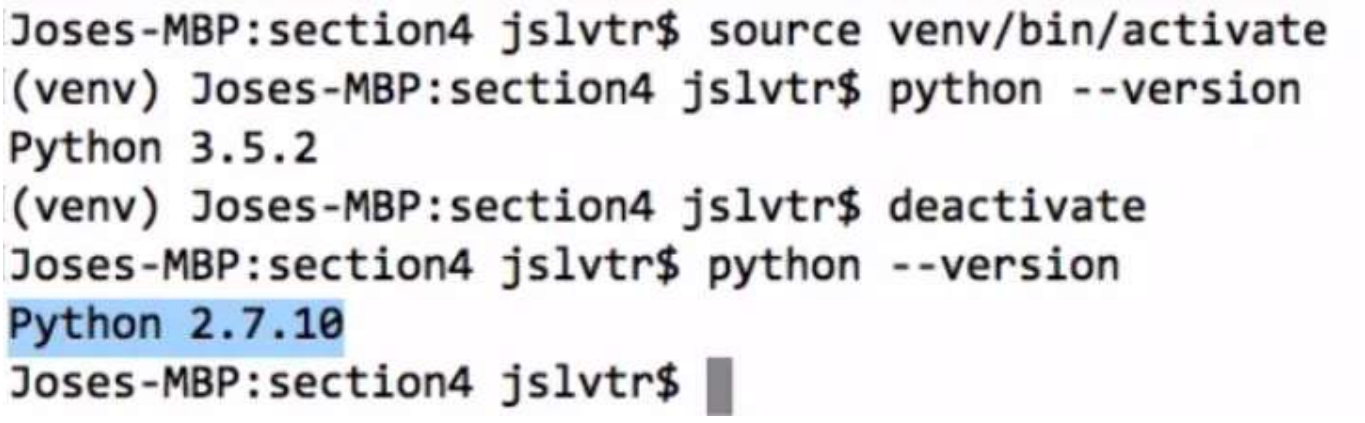
```
Joses-MBP:section4 jslvtr$ pip3.5 install virtualenv
Requirement already satisfied (use --upgrade to upgrade): virtualenv in /Library
/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages
Joses-MBP:section4 jslvtr$ virtualenv venv --python=python3.5
```

- The above is calling the virtualenv programme that we've installed
- and what's going to happen is it's going to create a folder inside our current folder called venv, and in it it's going to put a fresh Python installation.

- Then, put *source venv/bin/activate* (for Windows use *./venv/Scripts/activate.bat*)



- Thereafter, the prompt is going to change to show (venv) in front
- To stop the virtual environment, just use *deactivate*



- Install Flask-RESTful → *pip install Flask-RESTful*

Flask REST-ful APIs

- A resource represents a thing that our API can return and create etc.
- Resources are usually mapped into database tables as well
- *API* is also imported from *flask_restful* and that's just going to allow us to very easily add these resources to it
- The API works with resources and every resource has to be a class

```
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class Student(Resource):
    def get(self, name):
        return {'student': name}

api.add_resource(Student, '/student/<string:name>')  # http://127.0.0.1:5000/student/Rolf

app.run(port=5000)
```

Sample of a Flask RESTful API structure

- Test-first design is useful in helping you in thinking about what request your API needs to force you to identify what need there is for each of these requests.

Creating a Simple Flask REST-ful App

```python
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

items = []

class Item(Resource):
    def get(self, name):
        for item in items:
            if item['name'] == name:
                return item
        return {'item': None}, 404

    def post(self, name):
        item = {'name': name, 'price': 12.00}
        items.append(item)
        return item, 201

api.add_resource(Item, '/item/<string:name>')

app.run(port=5000)
```

- Above are simple implementations of POST and GET in Flask REST-ful where POST appends the newly created item into the items list and GET looks into this list and retrieves an item
- We also no longer need to do JSONify when talking with Flask REST-ful because Flask REST-ful does it for us, so we can just return dictionaries
- Error handling → Since all Python methods return none by default, what we have to do is change that to *return {'item': None}, 404*
- Status Codes

1. 200 → Server returns some data and everything is OK
2. 201 → Created
3. 202 → Accepted
4. 404 → Not Found

```python
from flask import Flask, request
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

items = []

class Item(Resource):
    def get(self, name):
        for item in items:
            if item['name'] == name:
                return item
        return {'item': None}, 404

    def post(self, name):
        data = request.get_json()
        item = {'name': name, 'price': data['price']}
        items.append(item)
        return item, 201


class ItemList(Resource):
    def get(self):
        return {'items': items}

api.add_resource(Item, '/item/<string:name>')
api.add_resource(ItemList, '/items')

app.run(port=5000, debug=True)
```

- Flask has a way of showing good error messages with a nice html page → Just include *debug = True* before or after *port = 5000*
- Getting a JSON payload from a request (sent from Postman for example) → *import request* then in the class definition put in the method *data = request.get_json()*

1. You can pass into this method *force* = *True* → Means you do not need the content-type header. It will just look in the content and it will format it even if the content type header is not set to be application/JSON
2. Or you can pass into this method *silent* = *True* → It doesn't give an error, and just basically returns none

```
class Item(Resource):
    def get(self, name):
        for item in items:
            if item['name'] == name:
                return item
        return {'item': None}, 404

    def post(self, name):
        data = request.get_json(force=True)
        item = {'name': name, 'price': 12.00}
        items.append(item)
        return item, 201
```

Using force = True

```
def post(self, name):
    data = request.get_json(silent=True)
    item = {'name': name, 'price': 12.00}
    items.append(item)
    return item, 201
```

Using silent = True

- This means if the request does not attach a JSON payload or the request does not have the proper content-type header, this will give an error

Improving Code and Error Control

```
from flask import Flask, request
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

items = []

class Item(Resource):
    def get(self, name):
        item = next(filter(lambda x: x['name'] == name, items), None)
        return {'item': item}, 200 if item else 404

    def post(self, name):
        if next(filter(lambda x: x['name'] == name, items), None):
            return {'message': "An item with name '{}' already exists.".format(name)}, 400

        data = request.get_json()
        item = {'name': name, 'price': data['price']}
        items.append(item)
        return item, 201


class ItemList(Resource):
    def get(self):
        return {'items': items}

api.add_resource(Item, '/item/<string:name>')
api.add_resource(ItemList, '/items')

app.run(port=5000, debug=True)
```

- GET Lambda function is a filter function that takes two arguments. First a filtering function and then the list of items that we're filtering.
- We're going to go through each item, execute this function and see if the item's name x matches the name which is the parameter. And if it does, then we're going to return it.
- Potentially many items generated, and so to ensure only 1 item is returned, we use *next* which gives us the first item found by this filter function

- Note: We can call next again if there are more items and that would give us the second item and then the third item and so on. If no items, it may raise an error, so we add in *None* as a default value
- POST lambda function ensures that if we found an item matching this name and that item is not none (i.e. there is an item that matches the name), we don't want to create a new item.

Authentication

- *pip install Flask-JWT*
- JWT stands for JSON Web Token which is an obfuscation of data (i.e. we're going to be encoding some data and that's a JSON Web Token)
- User sends username and a password, and the client returns a JWT. When the client has the JWT, they can send it to us with any request they make and when they do that it's going to tell us that they have previously authenticated and are logged in.
- Use *app.secret_key* to decrypt the encrypted message

```
users = [
    {
        'id': 1,
        'username': 'bob',
        'password': 'asdf'
    }
]

username_mapping = { 'bob': {
        'id': 1,
        'username': 'bob',
        'password': 'asdf'
    }
}

userid_mapping = { 1: {
        'id': 1,
        'username': 'bob',
        'password': 'asdf'
    }
}

def authenticate(username, password):
    user = username_mapping.get(username, None)
    if user and user.password == password:
        return user

def identity(payload):
    user_id = payload['identity']
    return userid_mapping.get(user_id, None)
```

- In-memory table of our registered users, then create user name mapping
- Authenticate function → It's the function that given a user name and a password is going to select the correct user name from our list [*username_mapping.get(username)*] + Set default value as None
- Identity function → Takes in a payload (i.e. contents of the JWT Token) and then we're going to extract the user ID from that payload. Once we

have the user ID, we can retrieve the specific user that matches this payload by just doing *return userid_mapping.get(user_id)* or None as a default.

```
from user import User

users = [
    User(1, 'bob', 'asdf')
]

username_mapping = {u.username: u for u in users}
userid_mapping = {u.id: u for u in users}

def authenticate(username, password):
    user = username_mapping.get(username, None)
    if user and user.password == password:
        return user

def identity(payload):
    user_id = payload['identity']
    return userid_mapping.get(user_id, None)
```

- Creating a user object
- Instead of assigning values, u for u is users, we're assigning key value pairs

```
from werkzeug.security import safe_str_cmp
from user import User

users = [
    User(1, 'bob', 'asdf')
]

username_mapping = {u.username: u for u in users}
userid_mapping = {u.id: u for u in users}

def authenticate(username, password):
    user = username_mapping.get(username, None)
    if user and safe_str_cmp(user.password, password):
        return user

def identity(payload):
    user_id = payload['identity']
    return userid_mapping.get(user_id, None)
```

- Flask comes with a library called werkseug, which aids comparison strings.

- safe_str_cmp (i.e. safe string compare) returns true if the strings are the same
- JWT creates a new endpoint which is /auth. When we call /auth we send it a username and a password and the JWT extension gets that username and password and sends it over to the authenticate function that takes in a username and a password.
- We are then going to find the correct user object using that username and we're going to compare its password to the one that we receive through the auth endpoint. If they match we're going to return the user and that becomes sort of the identity. The auth endpoint then returns a JWT token

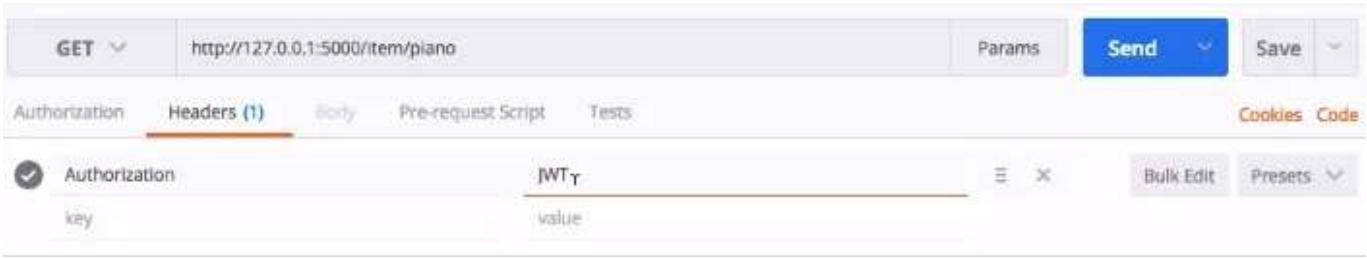
```python
from flask import Flask, request
from flask_restful import Resource, Api
from flask_jwt import JWT, jwt_required

from security import authenticate, identity

app = Flask(__name__)
app.secret_key = 'jose'
api = Api(app)

jwt = JWT(app, authenticate, identity)  # /auth

items = []
```

JWT extension

```python
from werkzeug.security import safe_str_cmp
from user import User

users = [
    User(1, 'bob', 'asdf')
]

username_mapping = {u.username: u for u in users}
userid_mapping = {u.id: u for u in users}

def authenticate(username, password):
    user = username_mapping.get(username, None)
    if user and safe_str_cmp(user.password, password):
        return user

def identity(payload):
    user_id = payload['identity']
    return userid_mapping.get(user_id, None)
```

Authenticate and Identity function

- The above will mean the user was authenticated, the JWT token is valid, and all is good



Put into header Authorization + JWT — **entire JWT token**

- DELETE

```
def delete(self, name):
    global items
    items = list(filter(lambda x: x['name'] != name, items))
```

- Note that for the above there is a need to declare items as a global variable because there is a local *items* that exists only in the method created above
- PUT

```
def put(self, name):
    data = request.get_json()
    item = next(filter(lambda x: x['name'] == name, items), None)
    if item is None:
        item = {'name': name, 'price': data['price']}
        items.append(item)
    else:
        item.update(data)
    return item
```

- Put can be used as an itempotent action — which means that you can call the same put request multiple times and the output or what it causes should never change

- What is happening is that we are getting data from the request and then essentially create the item or update the item if it already exists.
- Remember for verification you can put *@jwt required* on any of these verbs and they will require a jwt token and authorization header to be executed.

Request Parsing

- For PUT, if we were updating the item, we were going to use the entire payload. If the payload were to include a name, the item's name would change alongside this update.
- Flask RESTful uses *reqparse* to make sure that only some elements can be passed in through the JSON payload

```
def put(self, name):
    parser = reqparse.RequestParser()
    parser.add_argument('price',
        type=float,
        required=True,
        help="This field cannot be left blank!"
    )
    data = parser.parse_args()

    item = next(filter(lambda x: x['name'] == name, items), None)
    if item is None:
        item = {'name': name, 'price': data['price']}
        items.append(item)
    else:
        item.update(data)
    return item
```

Storing Resources in a Database

- SQLite library allows us to connect to a SQLite database, and to run SQL queries

- Cursor allows us to to select things and executing queries like inserting into database, selecting from database, and storing
- CREATE TABLE essentially creates the schema
- INSERT INTO inserts the user into the database

```python
import sqlite3

connection = sqlite3.connect('data.db')

cursor = connection.cursor()

create_table = "CREATE TABLE users (id int, username text, password text)"
cursor.execute(create_table)

user = (1, 'jose', 'asdf')
insert_query = "INSERT INTO users VALUES (?, ?, ?)"
cursor.execute(insert_query, user)

users = [
    (2, 'rolf', 'asdf'),
    (3, 'anne', 'xyz')
]
cursor.executemany(insert_query, users)

select_query = "SELECT * FROM users"
for row in cursor.execute(select_query):
    print(row)

connection.commit()

connection.close()
```

Some SQLite Commands

- Store users in SQLite and when the user calls the auth endpoint, to retrieve their username and password from the database. Then, compare that username and password to the username and password they're sending in the request, and if match, send back a JWT token.

```python
from werkzeug.security import safe_str_cmp
from user import User

users = [
    User(1, 'bob', 'asdf')
]

username_mapping = {u.username: u for u in users}
userid_mapping = {u.id: u for u in users}

def authenticate(username, password):
    user = username_mapping.get(username, None)
    if user and safe_str_cmp(user.password, password):
        return user

def identity(payload):
    user_id = payload['identity']
    return userid_mapping.get(user_id, None)
```

Authentication function

```python
import sqlite3

class User:
    def __init__(self, _id, username, password):
        self.id = _id
        self.username = username
        self.password = password

    @classmethod
    def find_by_username(cls, username):
        connection = sqlite3.connect('data.db')
        cursor = connection.cursor()

        query = "SELECT * FROM users WHERE username=?"
        result = cursor.execute(query, (username,))
        row = result.fetchone()
        if row:
            user = cls(*row)
        else:
            user = None

        connection.close()
        return user
```

Retrieving users by username

- Above: *result = cursor.execute(query, (username,))* → Parameters always have to be in the form of a tuple → This way of expressing (username,) is

to make a single value tuple → The bracket for the solitary username parameter is to indicate to the program that username should be executed 1st

- Adding users

```
import sqlite3

connection = sqlite3.connect('data.db')
cursor = connection.cursor()

create_table = "CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, username text, password text)"
cursor.execute(create_table)

connection.commit()

connection.close()
```

- INTEGER PRIMARY KEY instead of int if you want an auto incremental column

```
import sqlite3
from flask_restful import Resource

class User:=

class UserRegister(Resource):
    def post(self):
        connection = sqlite3.connect('data.db')
        cursor = connection.cursor()

        query = "INSERT INTO users VALUES (NULL, ?, ?)"
        cursor.execute(query, (data['username'], data['password']))

        connection.commit()
        connection.close()

        return {"message": "User created successfully."}, 201
```

- Create a new class called UserRegister, and this is going to be a Resource, just so we can add it to the API using flask_restful,

- Since id is auto-incrementing, insert NULL into the query, while the other 2 parameters are question marks (?)

```
class UserRegister(Resource):

    parser = reqparse.RequestParser()
    parser.add_argument('username',
        type=str,
        required=True,
        help="This field cannot be blank."
    )
    parser.add_argument('password',
        type=str,
        required=True,
        help="This field cannot be blank."
    )

    def post(self):
        data = UserRegister.parser.parse_args()

        connection = sqlite3.connect('data.db')
        cursor = connection.cursor()

        query = "INSERT INTO users VALUES (NULL, ?, ?)"
        cursor.execute(query, (data['username'], data['password']))

        connection.commit()
        connection.close()

        return {"message": "User created successfully."}, 201
```

- Create a parser which will parse through the JSON of the request and make sure the username and password are there
- Preventing duplicated users:

```python
def post(self):
    data = UserRegister.parser.parse_args()

    if User.find_by_username(data['username']):
        return {"message": "A user with that username already exists"}, 400

    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()

    query = "INSERT INTO users VALUES (NULL, ?, ?)"
    cursor.execute(query, (data['username'], data['password']))

    connection.commit()
    connection.close()

    return {"message": "User created successfully."}, 201
```

Retrieving Items From Database

- Creating a GET method

```python
import sqlite3
from flask_restful import Resource, reqparse
from flask_jwt import jwt_required

class Item(Resource):
    parser = reqparse.RequestParser()
    parser.add_argument('price',
        type=float,
        required=True,
        help="This field cannot be left blank!"
    )

    @jwt_required()
    def get(self, name):
        connection = sqlite3.connect('data.db')
        cursor = connection.cursor()

        query = "SELECT * FROM items WHERE name=?"
        result = cursor.execute(query, (name,))
        row = result.fetchone()
        connection.close()

        if row:
            return {'item': {'name': row[0], 'price': row[1]}}
        return {'message': 'Item not found'}, 404

    def post(self, name):
        if next(filter(lambda x: x['name'] == name, items), None):
            return {'message': "An item with name '{}' already exists.".format(name)}, 400

        data = Item.parser.parse_args()
```

- Creating an items table and inserting some data into it

```python
import sqlite3

connection = sqlite3.connect('data.db')
cursor = connection.cursor()

create_table = "CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, username text, password text)"
cursor.execute(create_table)

create_table = "CREATE TABLE IF NOT EXISTS items (name text, price real)"
cursor.execute(create_table)

cursor.execute("INSERT INTO items VALUES ('test', 10.99)")

connection.commit()

connection.close()
```

- Note: if __name__=='__main__' → This syntax is used for preventing the flask app from running if we were importing something from the app (normally Python will just run the app when you indicate it under import)

```
from flask import Flask
from flask_restful import Api
from flask_jwt import JWT

from security import authenticate, identity
from user import UserRegister
from item import Item, ItemList

app = Flask(__name__)
app.secret_key = 'jose'
api = Api(app)

jwt = JWT(app, authenticate, identity)  # /auth

api.add_resource(Item, '/item/<string:name>')
api.add_resource(ItemList, '/items')
api.add_resource(UserRegister, '/register')

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

- This circumvents that because whenever you run a Python file. Python assigns a special name to the file we run and that name is always __main__.
- If the name is main, we know that we have run this file, and therefore we want to start the flask app. If it's not main, then that means that we have imported this file from elsewhere and therefore we don't want to run the flask app.

Writing Item Resources to Database

```python
def get(self, name):
    item = self.find_by_name(name)
    if item:
        return item
    return {'message': 'Item not found'}, 404

@classmethod
def find_by_name(cls, name):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()

    query = "SELECT * FROM items WHERE name=?"
    result = cursor.execute(query, (name,))
    row = result.fetchone()
    connection.close()

    if row:
        return {'item': {'name': row[0], 'price': row[1]}}

def post(self, name):
    if self.find_by_name(name):
        return {'message': "An item with name '{}' already exists.".format(name)}, 400

    data = Item.parser.parse_args()

    item = {'name': name, 'price': data['price']}
    items.append(item)
    return item, 201

def delete(self, name):
    global items
    items = list(filter(lambda x: x['name'] != name, items))
```

- Create separate method called find_by_name → We extracted the useful code, essentially the code that doesn't deal with returning things. We've extracted that into a separate method which means that we can now call it from our post method.
- Replace the append item code with INSERT (from in-memory to database storage)

```python
def post(self, name):
    if self.find_by_name(name):
        return {'message': "An item with name '{}' already exists.".format(name)}, 400

    data = Item.parser.parse_args()

    item = {'name': name, 'price': data['price']}

    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()

    query = "INSERT INTO items VALUES (?, ?)"
    cursor.execute(query, (item['name'], item['price']))

    connection.commit()
    connection.close()

    return item, 201
```

- DELETE from database

```python
def delete(self, name):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()

    query = "DELETE FROM items WHERE name=?"
    cursor.execute(query, (name,))

    connection.commit()
    connection.close()
```

- Refactoring insertion of items through exception handling

```python
def post(self, name):
    if self.find_by_name(name):
        return {'message': "An item with name '{}' already exists.".format(name)}, 400

    data = Item.parser.parse_args()

    item = {'name': name, 'price': data['price']}

    try:
        self.insert(item)
    except:
        return {"message": "An error occurred inserting the item."}, 500

    return item, 201
```

- PUT → Insert item or update item. First, find the item and if it exists, update it. Otherwise, insert it.

```
def put(self, name):
    data = Item.parser.parse_args()

    item = self.find_by_name(name)
    updated_item = {'name': name, 'price': data['price']}

    if item is None:
        try:
            self.insert(updated_item)
        except:
            return {"message": "An error occurred inserting the item."}, 500
    else:
        try:
            self.update(updated_item)
        except:
            return {"message": "An error occurred updating the item."}, 500
    return updated_item

@classmethod
def update(cls, item):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()

    query = "UPDATE items SET price=? WHERE name=?"
```

- GET → Retrieving the item list. GET method will return every row from our database as a dictionary in JSON format

```
class ItemList(Resource):
    def get(self):
        connection = sqlite3.connect('data.db')
        cursor = connection.cursor()

        query = "SELECT * FROM items"
        result = cursor.execute(query)
        items = []
        for row in result:
            items.append({'name': row[0], 'price': row[1]})

        connection.close()

        return {'items': items}
```

Note: CRUD APIs refer to create, read, update, delete

Refer here for advanced configuration of Flask-JWT.

Simplifying Storage with Flask-SQLAlchemy

- pip install Flask-SQLAlchemy
- To transform created folders into packages, create a new file within the folders and call it *__init__.py* → Tells Python that it can look inside these folders for Python files

- A model is our internal representation of an entity whereas a resource is the external representation of an entity (what the API clients interact with). So, the model essentially is a helper that allows our program or gives us more flexibility in our program without polluting the resource (which is what client interact with)

```
def post(self, name):
    if self.find_by_name(name):
        return {'message': "An item with name '{}' already exists.".format(name)}

    data = Item.parser.parse_args()

    item = {'name': name, 'price': data['price']}

    try:
        self.insert(item)
    except:
        return {"message": "An error occurred inserting the item."}, 500

    return item, 201

@classmethod
def insert(cls, item):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()

    query = "INSERT INTO items VALUES (?, ?)"
    cursor.execute(query, (item['name'], item['price']))

    connection.commit()
    connection.close()
```

Resources are defined by the HTTP verbs and should not be polluted by methods like the insert method in this screen

- Note: Methods within a class should be separated by a new line but classes within a file should be separated by two new lines as a Python coding standard.
- **Importing ItemModel and applying it to our original code:** After changing and shifting the methods that are non-essential to the resources folder to the model folder, we need to go find all the places where we used self.find_by_name and change that to use the item model
- We need to change all the method parameters to self

```
def insert(self):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()

    query = "INSERT INTO items VALUES (?, ?)"
    cursor.execute(query, (self.name, self.price))

    connection.commit()
    connection.close()

def update(self):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()

    query = "UPDATE items SET price=? WHERE name=?"
    cursor.execute(query, (self.price'], item['name']))
```

- find_by_name method is still a class method which returns an object of ItemModel as opposed to a dictionary. Since its a class method, it has *cls* which is a reference to the class. *cls* calls the ItemModel in its method, row[0] Was the name, which goes into the name argument, row[1] Was the price , which goes into the price argument, and that gives us a new ItemModel object.

```
@classmethod
def find_by_name(cls, name):
    connection = sqlite3.connect('data.db')
    cursor = connection.cursor()

    query = "SELECT * FROM items WHERE name=?"
    result = cursor.execute(query, (name,))
    row = result.fetchone()
    connection.close()
```

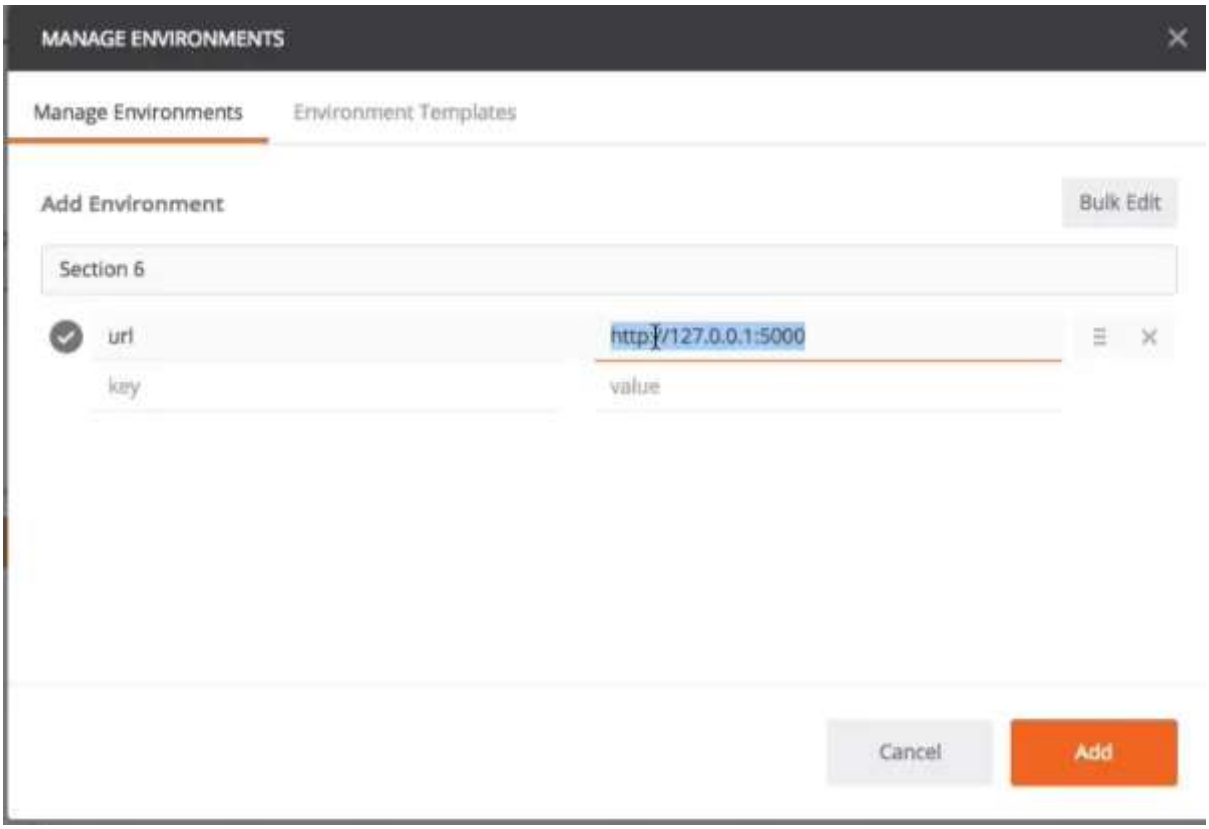
- If you see below, the same changes were made to *ItemModel.find_by_name* which now returns an item object as opposed to a dictionary. We have to return *items.json*

```
def put(self, name):
    data = Item.parser.parse_args()

    item = ItemModel.find_by_name(name)
    updated_item = ItemModel(name, data['price'])

    if item is None:
        try:
            updated_item.insert()
        except:
            return {"message": "An error occurred inserting the item."}, 500
    else:
        try:
            updated_item.update()
        except:
            return {"message": "An error occurred updating the item."}, 500
    return updated_item.json()
```

Advanced Postman Usage

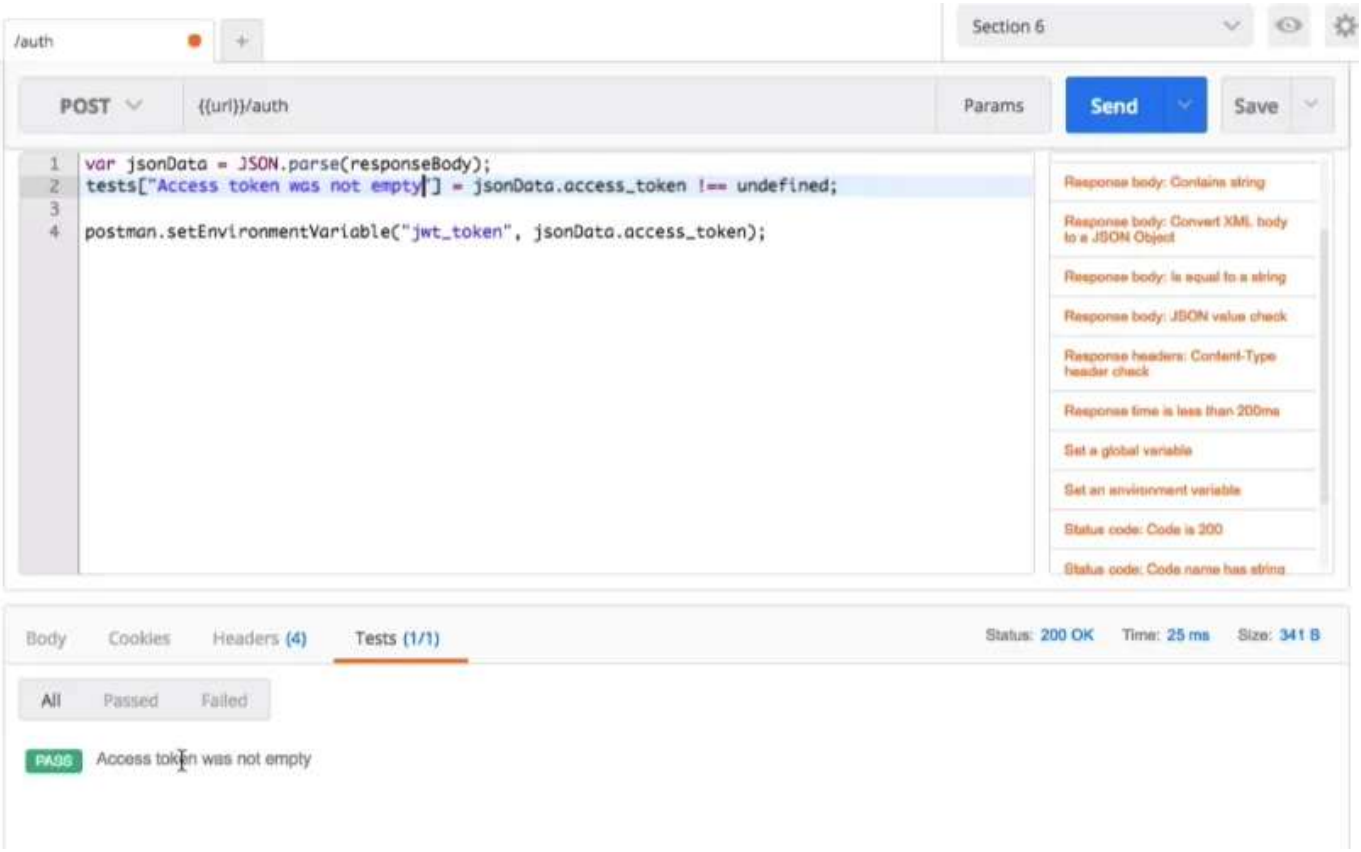
- Instead of keying in the full URL, you can put in *{{url}}/items*

- Go to settings icon on top right > Manage Environment > Define url in the console



- Long string of JWT token can also be replace with JWT{{jwt_token}}

- Go to auth endpoint and look under Tests tab → These tests are JavaScript code. JavaScript code that runs after the request has been processed by the server and sent back to us. What that means is that when the server sends us the JWT Token in JSON format, these tests here can access the value of the JWT Token and they can do things like verify it is not empty.



- Above shows some of the JS tests that come with Postman, where you can check the contents of the JSON response and respond in different ways
- If you click the eye icon on the top right you get to see the defined shortcuts like the url and the jwt_token

SQLAlchemy

- First, create new file that will host the SQLAlchemy object > from flask_sqlalchemy import SQLAlchemy > Initialise a variable called db to be a SQLAlchemy.
- The SQLAlhemy object will link to the flask app and look at the objects we instruct it to > Then it will map these objects to rows in a database (i.e. save object's properties into database)

```
import sqlite3
from db import db

class UserModel(db.Model):
    def __init__(self, _id, username, password):
        self.id = _id
        self.username = username
        self.password = password
```

- You need to import the db and then get the classes to extend db.model → This tells the SQLAlchemy entity that these classes here are things that we are going to be saving to a database and retrieving from a database. So it's going to create that mapping between the database and these objects.
- Next, we tell SQLAlchemy the table name where these models are going to be stored (i.e. the table, as well as the columns/properties the models will have)

```
import sqlite3
from db import db

class UserModel(db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80))
    password = db.Column(db.String(80))

    def __init__(self, _id, username, password):
        self.id = _id
        self.username = username
        self.password = password
```

- Then we specify a configuration property which is *app.config*

```python
from flask import Flask
from flask_restful import Api
from flask_jwt import JWT

from security import authenticate, identity
from resources.user import UserRegister
from resources.item import Item, ItemList

app = Flask(__name__)
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.secret_key = 'jose'
api = Api(app)

jwt = JWT(app, authenticate, identity)  # /auth

api.add_resource(Item, '/item/<string:name>')
api.add_resource(ItemList, '/items')
api.add_resource(UserRegister, '/register')

if __name__ == '__main__':
    from db import db
    db.init_app(app)
    app.run(port=5000, debug=True)
```

- In order to know when an object had changed but not been saved to the database, the extension flask SQLAlchemy was tracking every change that we made to the SQLAlchemy session, and that took some resources. Now we're turning it off because SQLAlchemy itself, the main library, has its own modification tracker → Declaring it *False* turns off the flask SQLAlchemy modification tracker.
- Note: It is very important to ensure the table has the correct format to use SQLAlchemy (e.g. if our item model has an *id* column we have to make sure that this column exists in the table)

```python
@classmethod
def find_by_name(cls, name):
    return cls.query.filter_by(name=name).first()
```

- SQLAlchemy handles all of the connection to the database, cursor creation, even the queries.

- If we use SQLAlchemy to find data it doesn't find a row, it automatically converts that row to an object if it can.
- For example in above, the *query* and the *filter_by* methods are from SQLAlchemy, and the statement above returns an ItemModel object that has self.name and self.price
- Adding object into database → The session in this instance is a collection of objects that we're going to write to the database. This can be inserting or updating (upserting)

```
def insert(self):
    db.session.add(self)
    db.session.commit()
```

- Delete from database

```
def delete_from_db(self):
    db.session.delete(self)
    db.session.commit()
```

```
def delete(self, name):
    item = Item.find_by_name(name)
    if item:
        item.delete_from_db()

    return {'message': 'Item deleted'}
```

- Changing the PUT method, leveraging on SQLAlchemy

```
def put(self, name):
    data = Item.parser.parse_args()

    item = ItemModel.find_by_name(name)

    if item is None:
        item = ItemModel(name, data['price'])
    else:
        item.price = data['price']

    item.save_to_db()

    return item.json()
```

- Telling SQLAlchemy where to find the data.db file at app.config → sqllite3:///data.db says is that the SQLAlchemy database is found at the root folder of our project.
- Note: It is not only compatible with SQLite, but also MySQL, PosgreSQL, etc.

```
from flask import Flask
from flask_restful import Api
from flask_jwt import JWT

from security import authenticate, identity
from resources.user import UserRegister
from resources.item import Item, ItemList

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///data.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.secret_key = 'jose'
api = Api(app)

jwt = JWT(app, authenticate, identity)  # /auth

api.add_resource(Item, '/item/<string:name>')
api.add_resource(ItemList, '/items')
api.add_resource(UserRegister, '/register')

if __name__ == '__main__':
    from db import db
    db.init_app(app)
    app.run(port=5000, debug=True)
```

- Returning the user that matches the username passing as an argument. The first item returned, SQLAlchemy then converts to a UserModel object

```
@classmethod
def find_by_username(cls, username):
    return cls.query.filter_by(username=username).first()
```

```
@classmethod
def find_by_id(cls, _id):
    return cls.query.filter_by(id=_id).first()
```

- The above are 2 APIs (not REST)/endpoints/methods that are an interface for other parts of our program
- Below shows how we can easily display the ItemList resource with SQLAlchemy

```
class ItemList(Resource):
    def get(self):
        return {'items': [item.json() for item in ItemModel.query.all()]}
```

- Creating tables using SQLAlchemy using the decorator *@app.before_first_request* which affects the method below it and it's going to run that method before the first request into this app → This creates the file *data.db* and all of the tables in the file

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///data.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.secret_key = 'jose'
api = Api(app)

@app.before_first_request
def create_tables():
    db.create_all()
```

- Creating a new model → StoreModel

```python
from db import db

class StoreModel(db.Model):
    __tablename__ = 'stores'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80))

    items = db.relationship('ItemModel')

    def __init__(self, name):
        self.name = name

    def json(self):
        return {'name': self.name, 'items': [item.json() for item in self.items]}

    @classmethod
    def find_by_name(cls, name):
        return cls.query.filter_by(name=name).first()

    def save_to_db(self):
        db.session.add(self)
        db.session.commit()

    def delete_from_db(self):
        db.session.delete(self)
        db.session.commit()
```

- In ItemModel, we will add in another column to the table → Add *store_id* as a foreign key which allows for easy referencing (i.e. which items belong to the same store and vice versa), and it also prevents accidental deletion of stores
- Remember to also replicate what we have in our item list GET endpoint → *def json(self)*
- In SQLAlchemy, we can define the relationship instead of doing a usual SQL JOIN

```python
class ItemModel(db.Model):
    __tablename__ = 'items'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80))
    price = db.Column(db.Float(precision=2))

    store_id = db.Column(db.Integer, db.ForeignKey('stores.id'))
    store = db.relationship('StoreModel')
```

- To define a back reference, just put *items = db.relationship('ItemModel')*
- Whenever we create a store model, we're going to go and create an object for each item → This may be computationally expensive → To overcome this, put *lazy = 'dynamic'* so that self.items no longer is a list of items, but a query builder that has the ability to look into the items table, and then we can use .all to retrieve all of the items in that table
- Therefore every time you call the JSON method, you have to go into the table, making the speed slower (ie. creation of store is faster because you do not need to create all the item objects, but trade-off with a slower speed of calling the JSON method)

```python
items = db.relationship('ItemModel', lazy='dynamic')
```

- Creating store resource (POST, PUT, DELETE)

```python
from flask_restful import Resource
from models.store import StoreModel

class Store(Resource):
    def get(self, name):
        store = StoreModel.find_by_name(name)
        if store:
            return store.json()
        return {'message': 'Store not found'}, 404

    def post(self, name):
        if StoreModel.find_by_name(name):
            return {'message': "A store with name '{}' already exists.".format(name)}, 400

        store = StoreModel(name)
        try:
            store.save_to_db()
        except:
            return {'message': 'An error occurred while creating the store.'}, 500

        return store.json(), 201

    def delete(self, name):
        store = StoreModel.find_by_name(name)
        if store:
            store.delete_from_db()

        return {'message': 'Store deleted'}
```

Version Control With Git

- Git is a set of layers and each layer has a function. All you do with Git is you move files between these layers, and the layers provide error control, redundancy, and more things

- Git is the thing that allows you to move files between these layers and allows you to commit and push. GitHub is essentially just a server that hosts your Git repositories so that you have a Git repository in your computer, a local repository, and also a remote repository; a repository that is hosted in GitHub.
- Optional to push commits to remote repository (for redundancy)
- The layers are physical (inside a hidden .git folder)

The Git Workflow

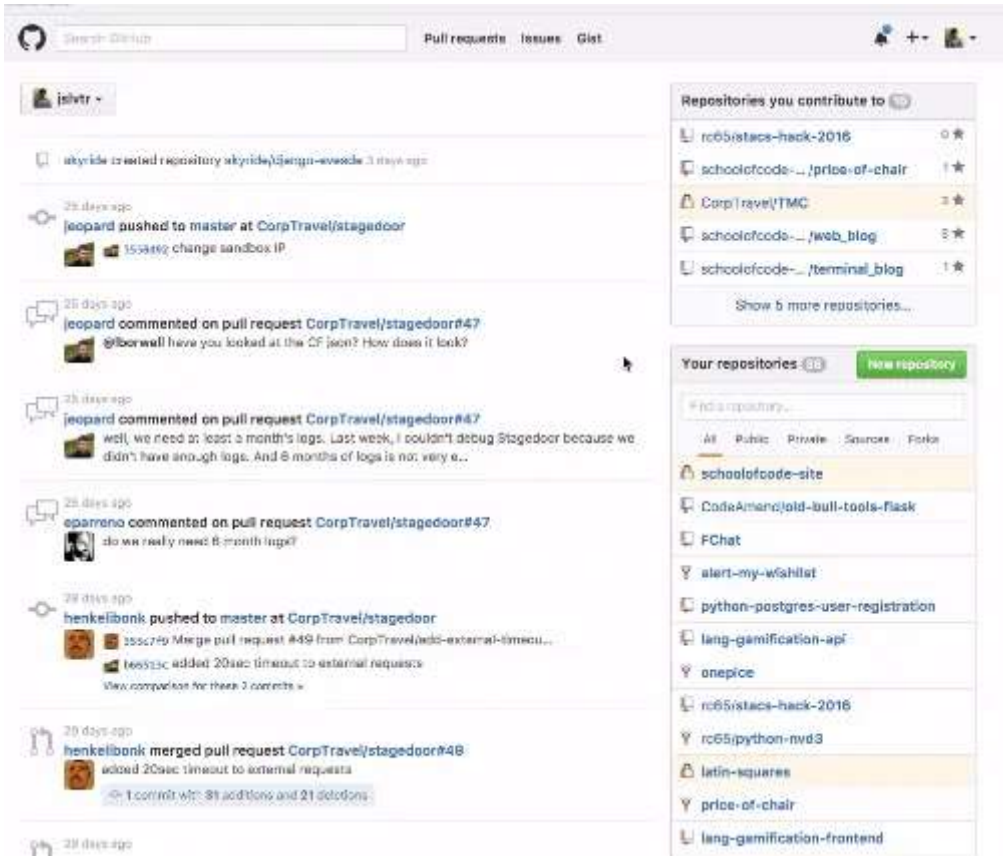
- git init → Git will start being able to tell what files I have changed and what files I have created.
- git status → Git status tells us a few things. It tells us that we are on branch master, that this is the first commit, and also that there's nothing to commit, that we can create or copy files in and then we can use git add to track those files

```
Joses-MBP:section7 jslvtr$ mkdir code
Joses-MBP:section7 jslvtr$ cd code/
Joses-MBP:code jslvtr$ git init
Initialized empty Git repository in /Users/jslvtr/Documents/Personal/Courses/rest-api-flask/curriculum/section7/co
de/.git/
Joses-MBP:code jslvtr$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
Joses-MBP:code jslvtr$ touch app.py
Joses-MBP:code jslvtr$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        app.py

nothing added to commit but untracked files present (use "git add" to track)
Joses-MBP:code jslvtr$ git add app.py
Joses-MBP:code jslvtr$
```

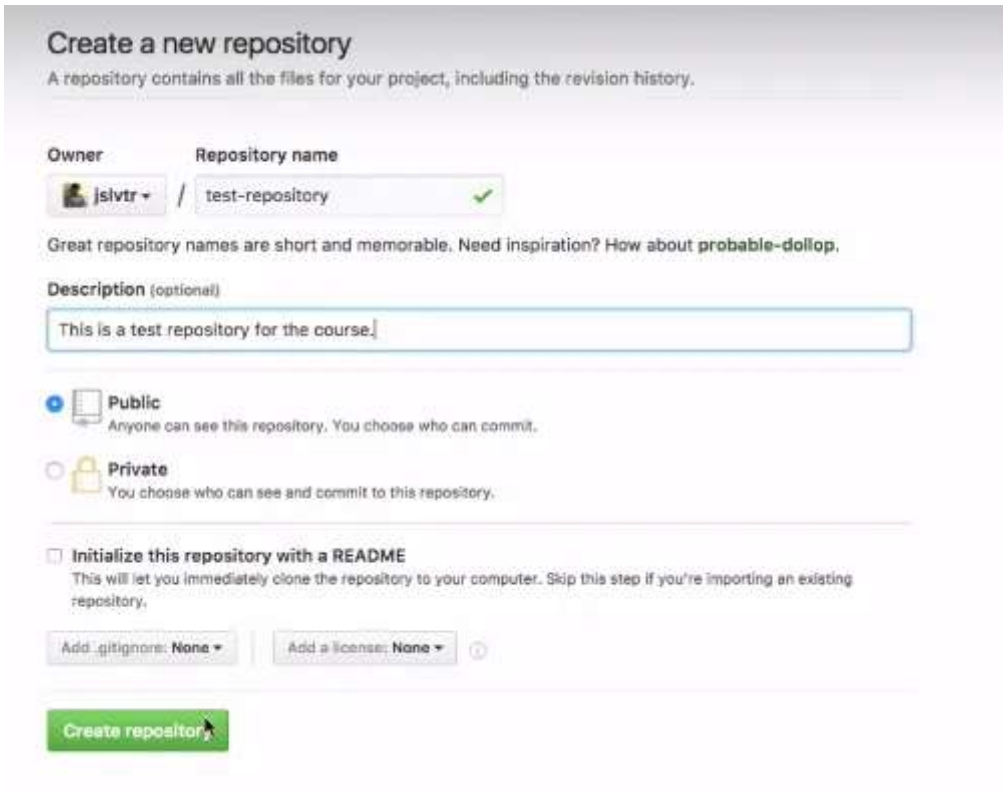
- We can always use *git rm — cached* to unstage a file and remove it from the staging area if we don't want to include it in the next commit
- git commit -m → Adding comments with " "

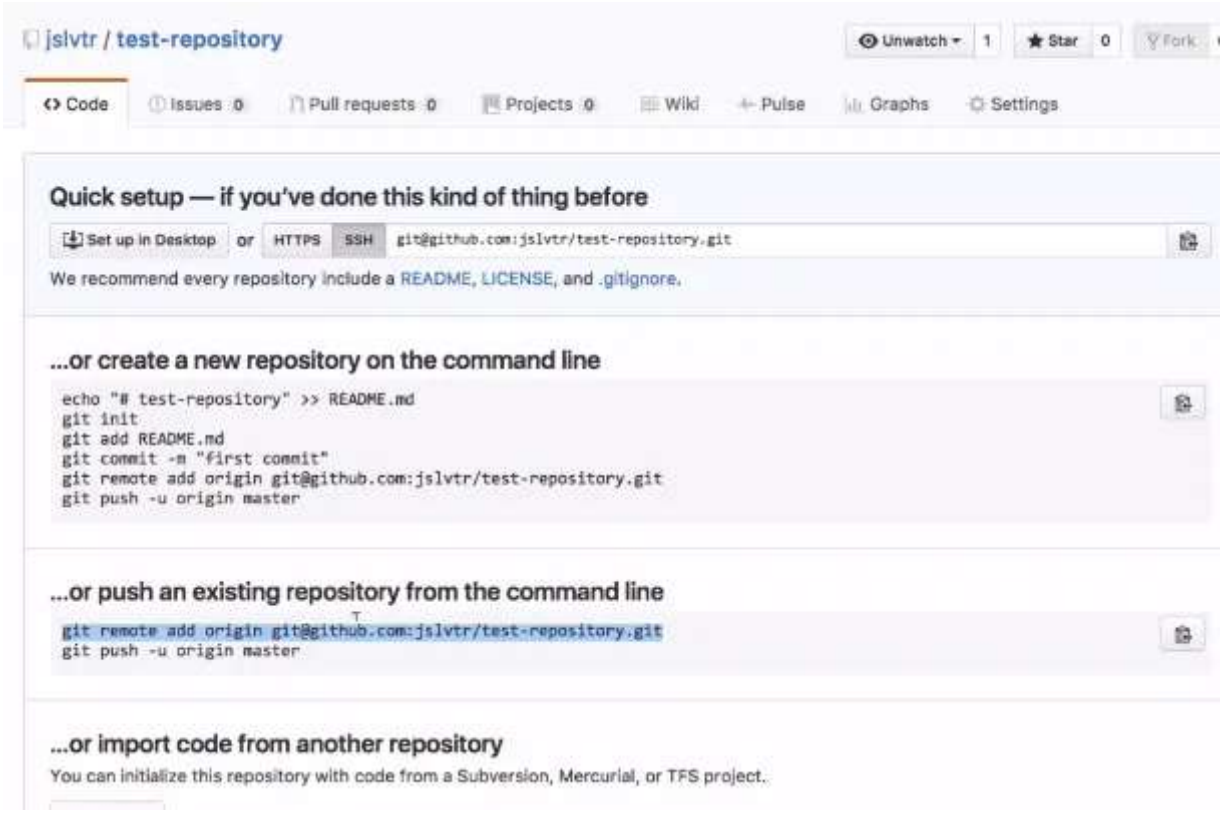
```
Joses-MBP:code jslvtr$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   app.py

Joses-MBP:code jslvtr$ git commit -m "Created a simple Flask application"
[master (root-commit) 0bf23b7] Created a simple Flask application
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 app.py
Joses-MBP:code jslvtr$
```

Adding Files to GitHub



- Top right → List of repositories
- Create a new repository





- Download repository → Initialise a new repository locally

- Use repository as remote host → Push existing repository from command line means to add this remote repository as a remote repository to our local repository

```
Joses-MBP:code jslvtr$ git status
On branch master
nothing to commit, working directory clean
Joses-MBP:code jslvtr$ git remote add origin https://github.com/jslvtr/test-repository.git
Joses-MBP:code jslvtr$ git push
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin master

Joses-MBP:code jslvtr$ git push --set-upstream origin master
Username for 'https://github.com': jslvtr
Password for 'https://jslvtr@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 231 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/jslvtr/test-repository.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
Joses-MBP:code jslvtr$
```

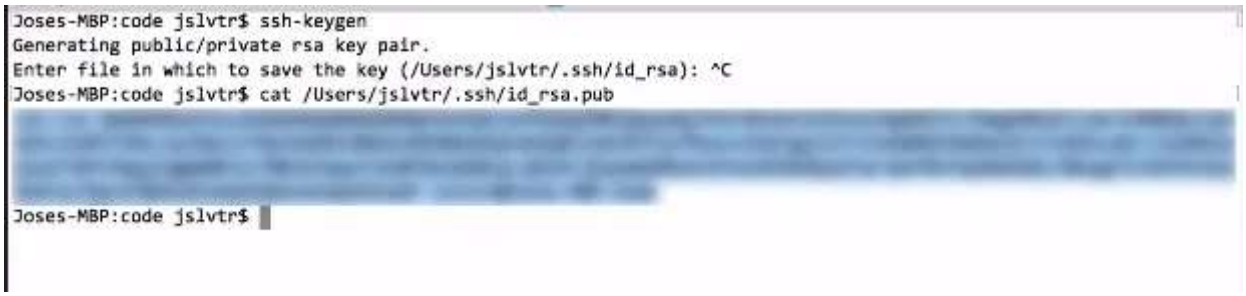
- git remote add origin → Adds remote repository
- git push → To push our local code into the remote repository. Automatically see that origin exists and it will get your master branch with all the commits that the master branch has; it will push them and put them in the remote repository
- git pull → download changes from the remote repository into your local repository

Using SSH Keys for Security

- An SSH key is a way of encrypting information. Whenever you generate an SSH key there are two keys generated, a public key and a private key.

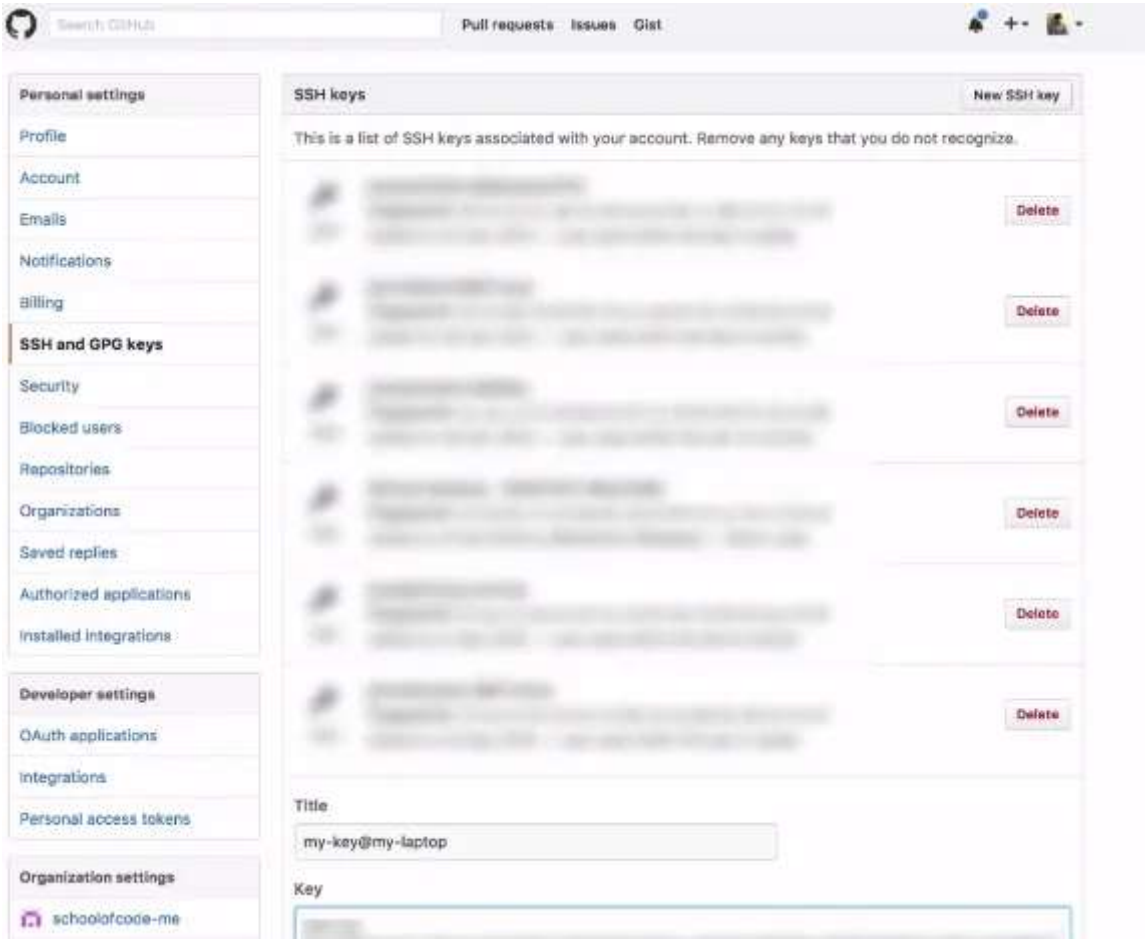
Anybody that has a public key can decrypt information. If you have the public key you can decrypt. Those who have the private key can encrypt.

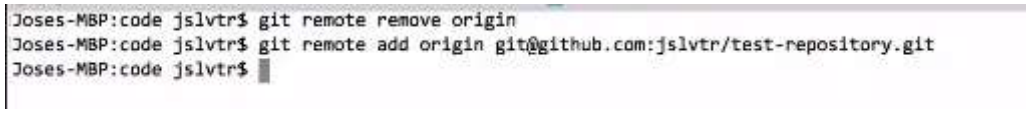
- You can share your public key with Github, then whenever you send them data that is encrypted using your private key they know that it's coming from you if they can decrypt it because no other public key can decrypt your private key.
- Generating SSH key pairs locally → ssh-keygen
- You should generate a key for every device you've got but the device itself should generate it



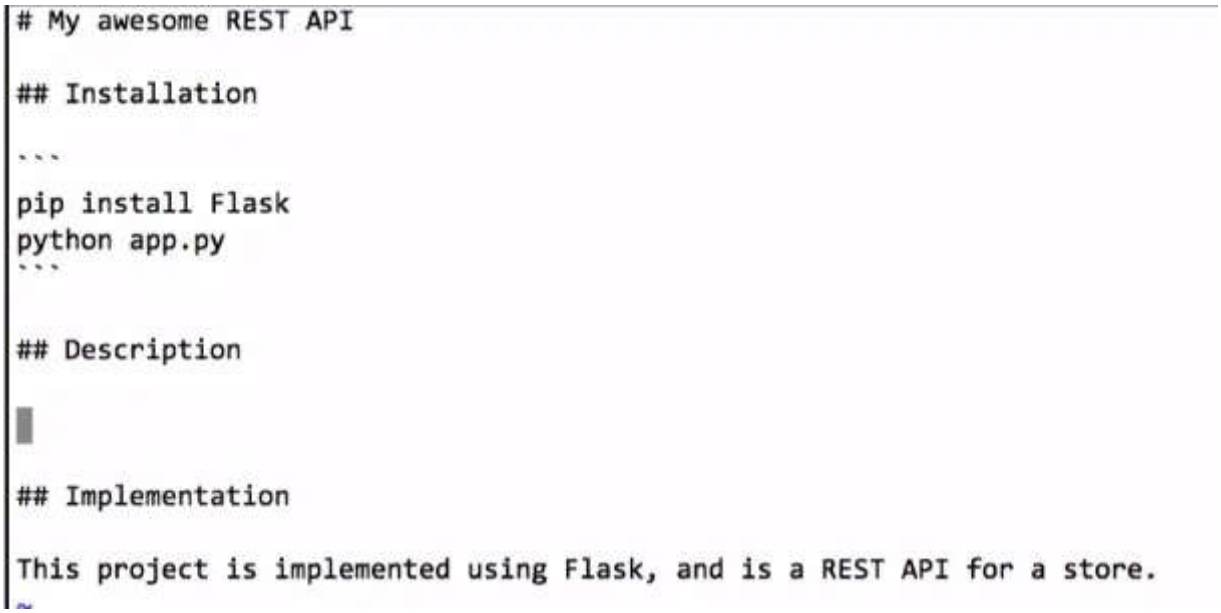
Public key is masked here

- Sharing your public key with GitHub → Go to profile, edit your profile, SSH and GPG keys, add a new SSH key → When that's done, then you should be able to, when you create or edit a repository, modify the origin if you want





- If you are putting your code out there for jobs and for other people to see its always very useful to have a Read Me file → vim README.md → This initialises a text editor in your console called VIM
- If you want to insert text into a file you've opened with VIM, you can press the I key, that's the I key, and that will put you into insert mode and then you can insert your description of your project here



- Utilises markdown → E.g. # is for a header/title, ## is subtitles
- Once you have written your description of your project and everything, just press the ESC key on your keyboard and then type :WQ and then VIM will write the file to disc and quit the programme.

Heroku

- Heroku is a web service that runs (hosts) your code and allows other people to interact with it (includes Flask apps)
- A dyno is Heroku's version of a server (virtual machine). So when you run your application, your application doesn't have control of the computer in which it is running. It only has control of the virtual system that has been created specifically for that dyno.
- Heroku is also limited also in running other things simultaneously
- UWSGI is a way of serving your flask application (another Python library)

- Heroku app can also have configuration properties (e.g. what database to use, where to log)
- Can also run multiple dynos (faster and more available)
- Secure → Automatically enables SSL (encrypted communication between client and server)
- Getting the code into GitHub

```
Joses-MBP:code jslvtr$ ls
app.py          db.py           models          resources       security.py
Joses-MBP:code jslvtr$ ls models/
__init__.py     item.py         store.py        user.py
Joses-MBP:code jslvtr$ ls resources/
__init__.py     item.py         store.py        user.py
Joses-MBP:code jslvtr$ atom .
Joses-MBP:code jslvtr$
```

Below are the files that we do not need to import to GitHub (.gitignore) → pycache is the cache of all compiled versions of Python code; pyc is the

compiled Python code → This is a hidden file (starts with a full stop) so Git doesn't add it by default → You can manually add it using *git add .gitignore*

Then do git init in the terminal to initialise the repository.

```
Joses-MBP:code jslvtr$ git add .gitignore
Joses-MBP:code jslvtr$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   app.py
        new file:   db.py
        new file:   models/__init__.py
        new file:   models/item.py
        new file:   models/store.py
        new file:   models/user.py
        new file:   resources/__init__.py
        new file:   resources/item.py
        new file:   resources/store.py
        new file:   resources/user.py
        new file:   security.py
```

First time Git knows about these files, so all labelled as new. Subsequent changes will all be tracked (i.e. modified). This is when we do a *git commit*. Thereafter you can include a commit message

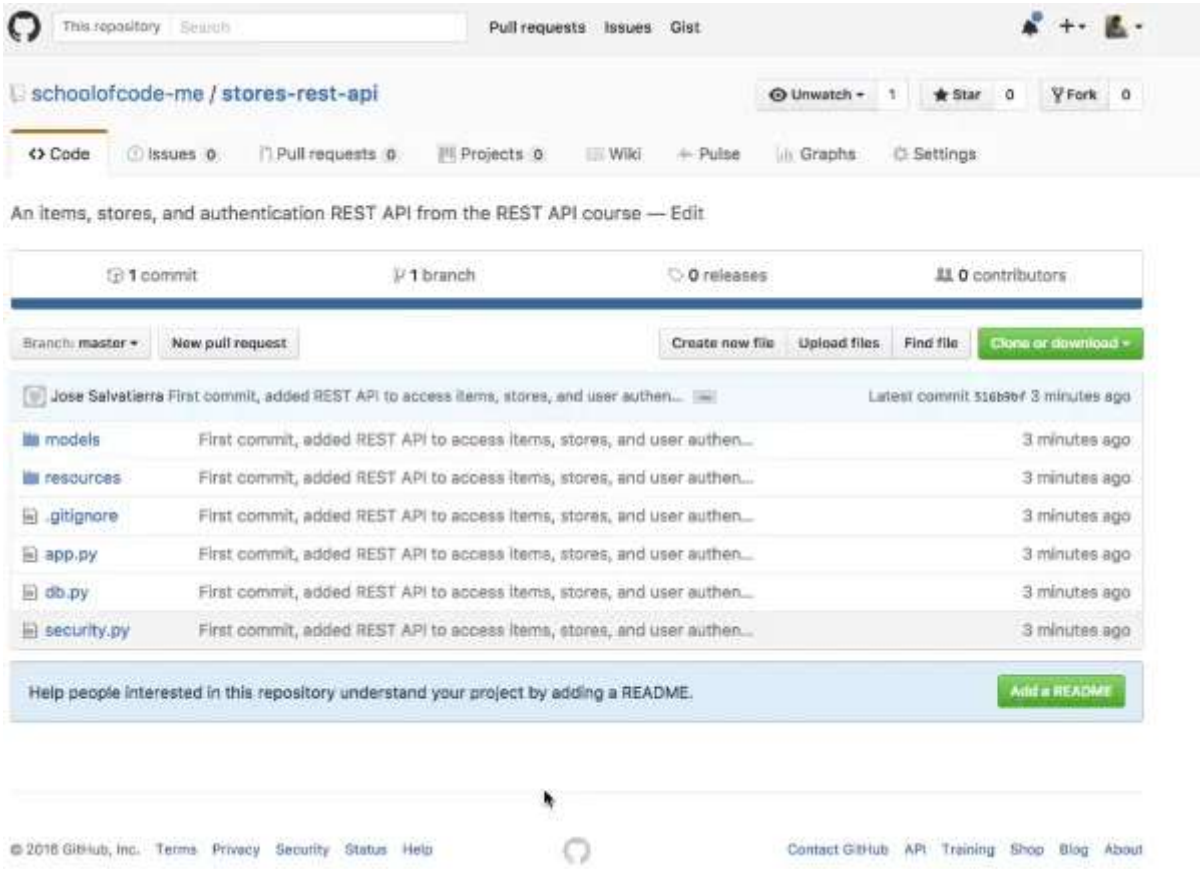
```
Joses-MBP:code jslvtr$ git commit -m "First commit, added REST API to access items, stores, and user authenticatio
n."
[master (root-commit) 516b9bf] First commit, added REST API to access items, stores, and user authentication.
 12 files changed, 256 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 app.py
 create mode 100644 db.py
 create mode 100644 models/__init__.py
 create mode 100644 models/item.py
 create mode 100644 models/store.py
 create mode 100644 models/user.py
 create mode 100644 resources/__init__.py
 create mode 100644 resources/item.py
 create mode 100644 resources/store.py
 create mode 100644 resources/user.py
 create mode 100644 security.py
```

Then, you can use *git remote add origin* to add the remote repository

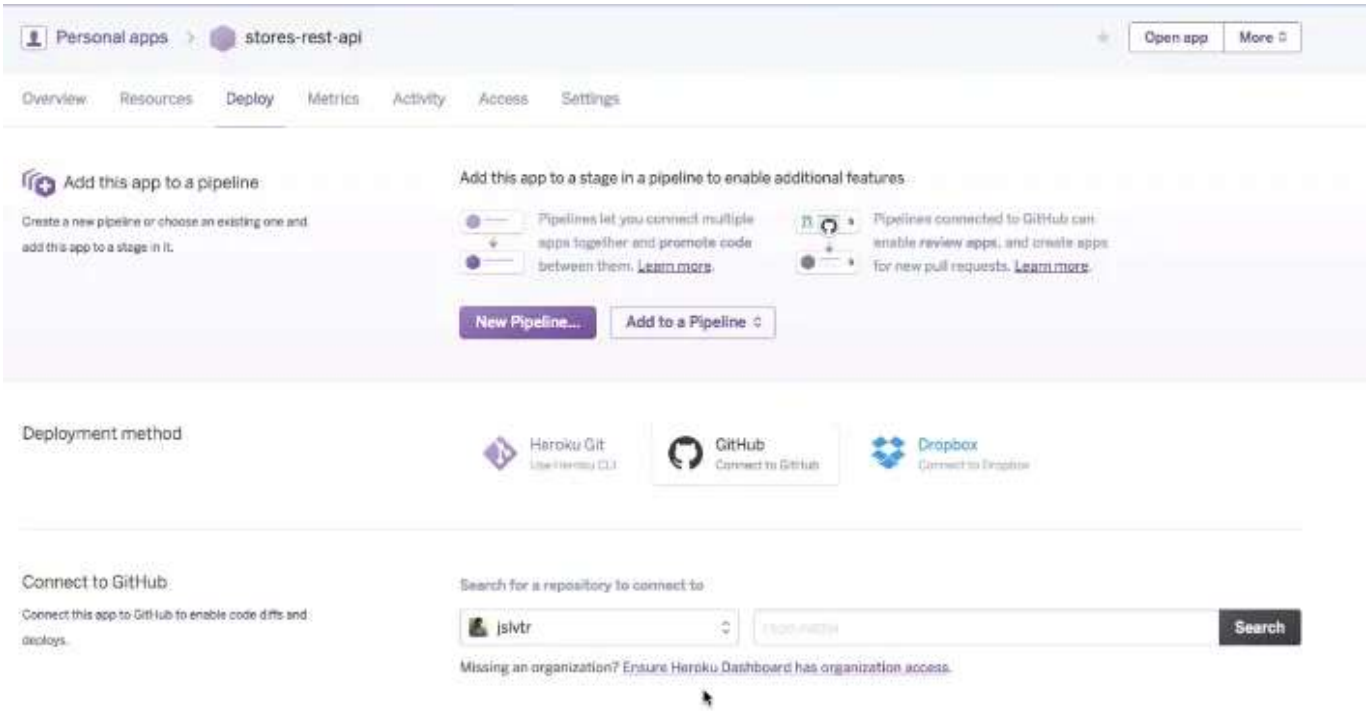
```
Joses-MBP:code jslvtr$ git remote add origin git@github.com:schoolofcode-me/stores-rest-api.git
Joses-MBP:code jslvtr$ git push -u origin master
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (15/15), 3.29 KiB | 0 bytes/s, done.
Total 15 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To github.com:schoolofcode-me/stores-rest-api.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Thereafter you can use *git push* and define the address of the repository and the branch

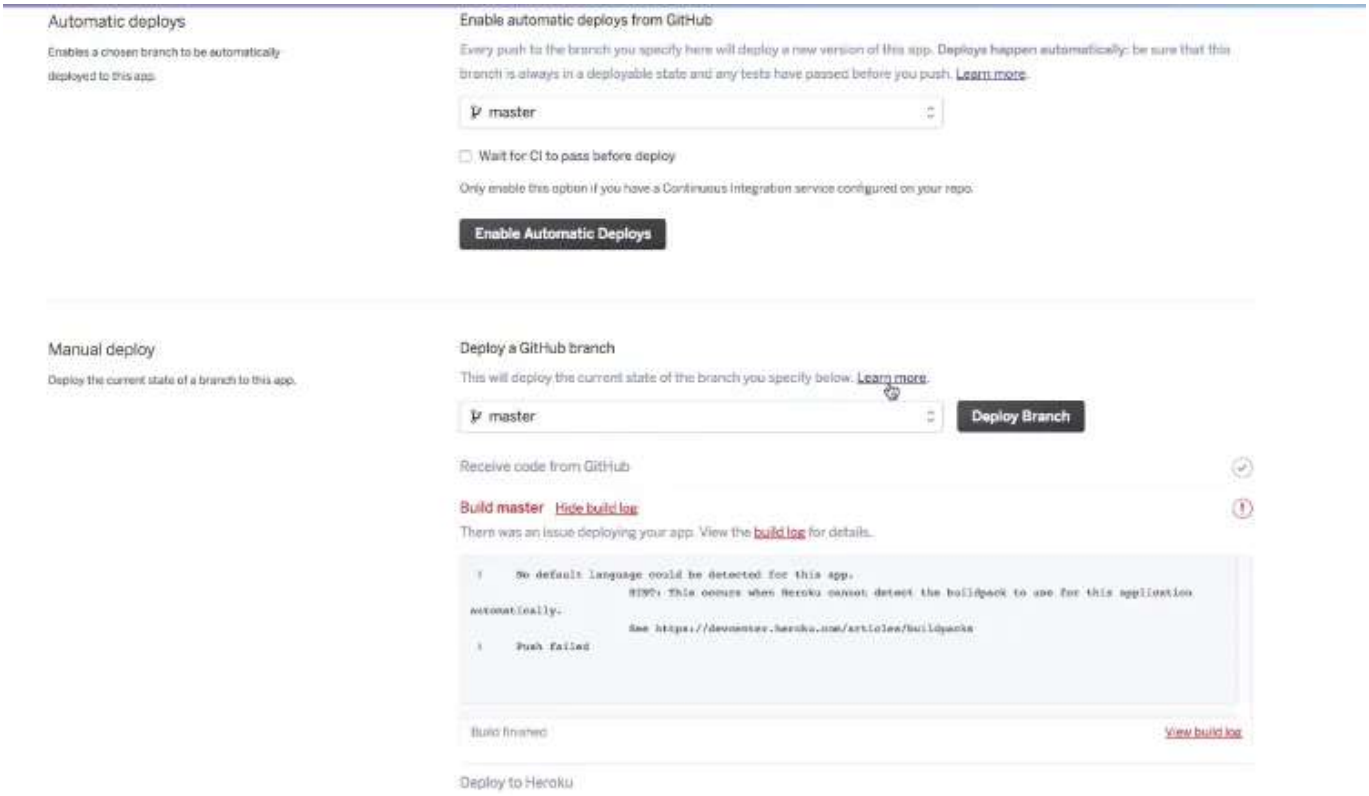
- Note: You can use fork (top right icon in GitHub) to copy the entire repository to your account → Can make changes without affecting the original repository



- Heroku Dashboard → Create new app

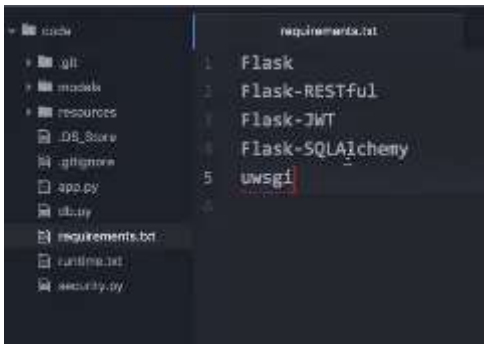


- Then you are able to connect to GitHub → Search for repository to connect to
- Automatic vs. Manual deploys → An automatic deploy means that whenever we push a commit to the remote branch, that will automatically appear in Heroku after a couple of minutes. The manual deploy allows us to just deploy the current branch into Heroku



- Defining runtime.txt → Tell Heroku we are running Python 3.5.2

- Defining requirements.txt → Python has a standard way of telling other systems what libraries we're using and that's in a file called requirements.txt



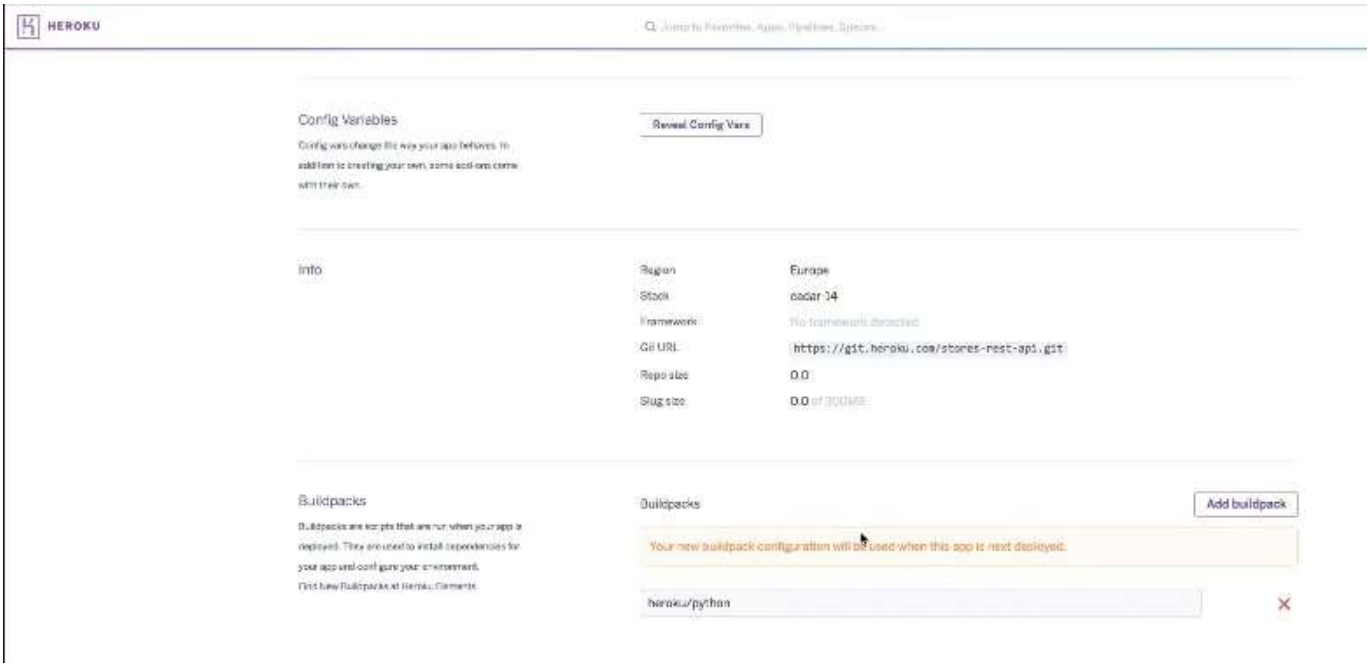
- uwsgi.ini → Set configuration parameters for the UWSGI process to run our app (port, whether we are using master process when running UWSGI which controls slave processes, die-on-term [which if true, we're gonna kill the UWSGI process just to free up the resources], the module, memory-report)

- Procfile → Define what dyno we want to use in Heroku (e.g. whether it connects to a HTTP port, going to be listening on that port to receive incoming requests, web type? → But no need if we are using UWSGI [see below image])

- Above: We are running the UWSGI process with this file, which is the configuration file that we've created. And that is then going to load up the Python app and run that. So, when we run the UWSGI process, that is going to listen in to this module, it's going to run that, and that's going to start our Flask application.
- Then add and commit these files, then *git push*

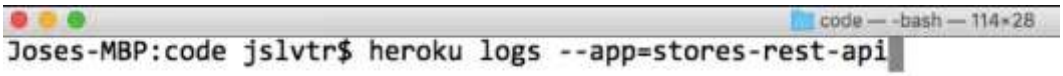
```
Joses-MBP:code jslvtr$ git add *
Joses-MBP:code jslvtr$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   Procfile
        new file:   requirements.txt
        new file:   runtime.txt
        new file:   uwsgi.ini

Joses-MBP:code jslvtr$ git commit -m "Adding Heroku required files for running."
[master ce79d69] Adding Heroku required files for running.
 4 files changed, 13 insertions(+)
 create mode 100644 Procfile
 create mode 100644 requirements.txt
 create mode 100644 runtime.txt
 create mode 100644 uwsgi.ini
Joses-MBP:code jslvtr$ git push
```

- Then go back to Heroku > Settings > Buildpacks section > Select Python > Save



- Then you deploy master branch and wait for all the libraries defined in requirements.txt to be installed and the app to be compressed and deployed to the internet
- Logging in Heroku

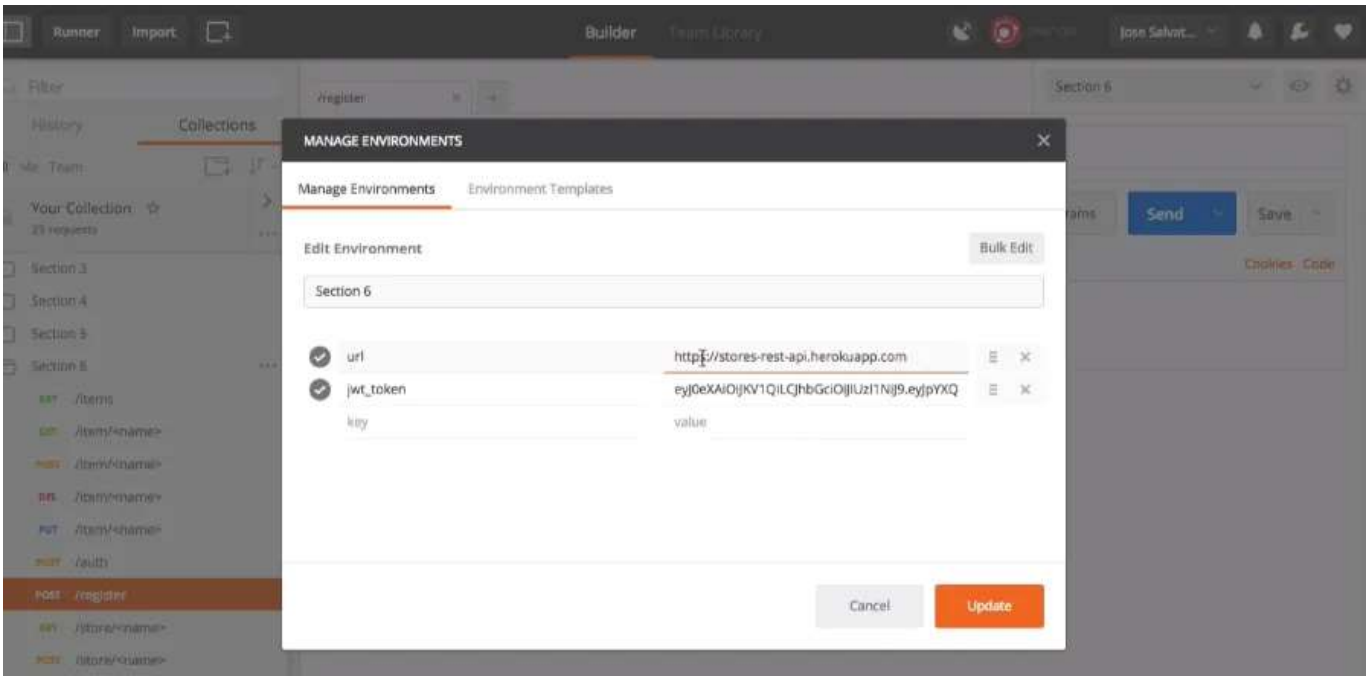
Download Heroku command line > Restart terminal > Do Heroku login (email and PW prompt) > Logged in to Heroku Command Line interface > Then type heroku logs — app > Then you will see the logs of your app



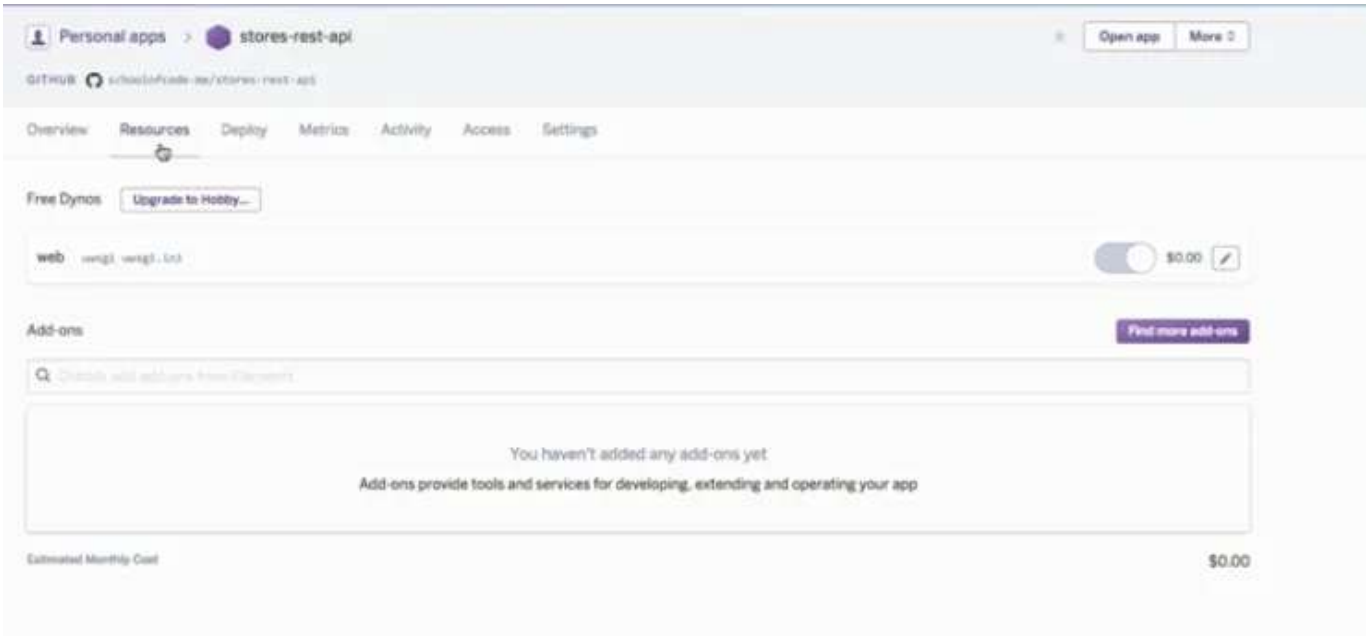

```
2016-11-14T11:33:33.966875+00:00 app[web.1]: *** uWSGI is running in multiple interpreter mode ***
2016-11-14T11:33:33.966891+00:00 app[web.1]: spawned uWSGI master process (pid: 4)
2016-11-14T11:33:33.968296+00:00 app[web.1]: spawned uWSGI worker 1 (pid: 7, cores: 1)
2016-11-14T11:33:34.417979+00:00 heroku[web.1]: State changed from starting to up
2016-11-14T11:33:35.570060+00:00 heroku[router]: at=info method=GET path="/" host=stores-rest-api.herokuapp.com re
quest_id=3e5e1953-6950-47b5-9e15-9531ab829d3a fwd="80.234.172.233" dyno=web.1 connect=0ms service=29ms status=500
bytes=375
2016-11-14T11:33:35.565778+00:00 app[web.1]: [2016-11-14 11:33:35,563] ERROR in app: Exception on / [GET]
2016-11-14T11:33:35.565790+00:00 app[web.1]:   File "/app/.heroku/python/lib/python3.5/site-packages/flask/app.py"
, line 1988, in wsgi_app
2016-11-14T11:33:35.565791+00:00 app[web.1]:     response = self.full_dispatch_request()
2016-11-14T11:33:35.565789+00:00 app[web.1]: Traceback (most recent call last):
2016-11-14T11:33:35.565792+00:00 app[web.1]:   File "/app/.heroku/python/lib/python3.5/site-packages/flask/app.py"
, line 1634, in full_dispatch_request
2016-11-14T11:33:35.565792+00:00 app[web.1]:     self.try_trigger_before_first_request_functions()
2016-11-14T11:33:35.565793+00:00 app[web.1]:   File "/app/.heroku/python/lib/python3.5/site-packages/flask/app.py"
, line 1660, in try_trigger_before_first_request_functions
2016-11-14T11:33:35.565794+00:00 app[web.1]:     func()
2016-11-14T11:33:35.565795+00:00 app[web.1]:   File "./app.py", line 18, in create_tables
2016-11-14T11:33:35.565796+00:00 app[web.1]:     db.create_all()
2016-11-14T11:33:35.565797+00:00 app[web.1]: NameError: name 'db' is not defined
2016-11-14T11:33:35.586356+00:00 app[web.1]: {address space usage: 232374272 bytes/221MB} {rss usage: 30556160 byt
es/29MB} [pid: 7|app: 0|req: 1/1] 10.79.145.42 () {50 vars in 941 bytes} [Mon Nov 14 11:33:35 2016] GET / => gener
ated 291 bytes in 28 msecs (HTTP/1.1 500) 2 headers in 84 bytes (1 switches on core 0)
```

Heroku Logs

- To retest all of this newly deployed app using the endpoints from earlier sections, remember to go to Postman's "Manage Environments" and change the URL



- Adding PostgreSQL to Heroku app



Add PostgreSQL as add-on > Resources section > Find more add ons > Install Heroku Postgres

Config Vars > Database URL → Use this URL in our application's environment variable

import os > os.environ.get DATABASE_URL → This is the name of the variable Heroku has created for us, the environment variable

```
import os

from flask import Flask
from flask_restful import Api
from flask_jwt import JWT

from security import authenticate, identity
from resources.user import UserRegister
from resources.item import Item, ItemList
from resources.store import Store, StoreList

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get('DATABASE_URL', 'sqlite:///data.db')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.secret_key = 'jose'
api = Api(app)
```

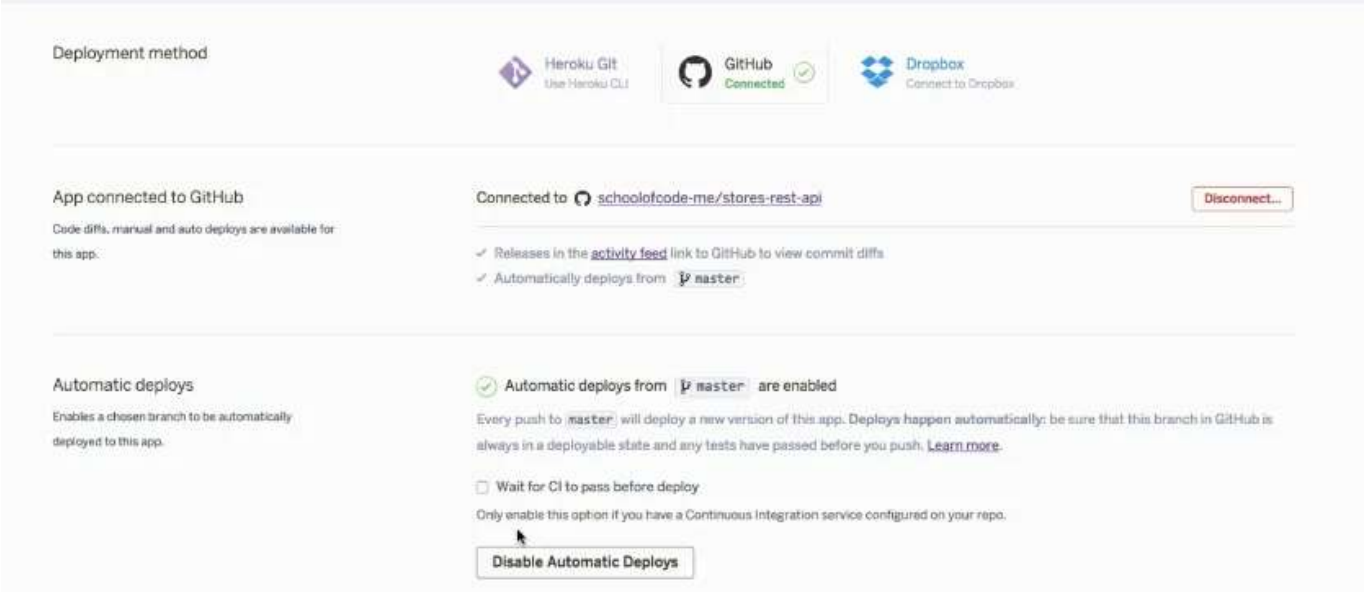
Go to requirements.txt and add *psycopg2* → A popular Python library used to interact with Postgres

Then do a git commit -am and git push

```
Joses-MBP:code jslvtr$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   app.py
        modified:   requirements.txt

no changes added to commit (use "git add" and/or "git commit -a")
Joses-MBP:code jslvtr$ git commit -am "Added PostgreSQL environment variable and psycopg2."
[master 7ec0684] Added PostgreSQL environment variable and psycopg2.
 2 files changed, 4 insertions(+), 1 deletion(-)
Joses-MBP:code jslvtr$ git push -f
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 480 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:schoolofcode-me/stores-rest-api.git
 + 93488c7...7ec0684 master -> master (forced update)
```

To look at status of the database: Then go the Chrome > Deploy tab > Overview section of our app > Heroku Postgres

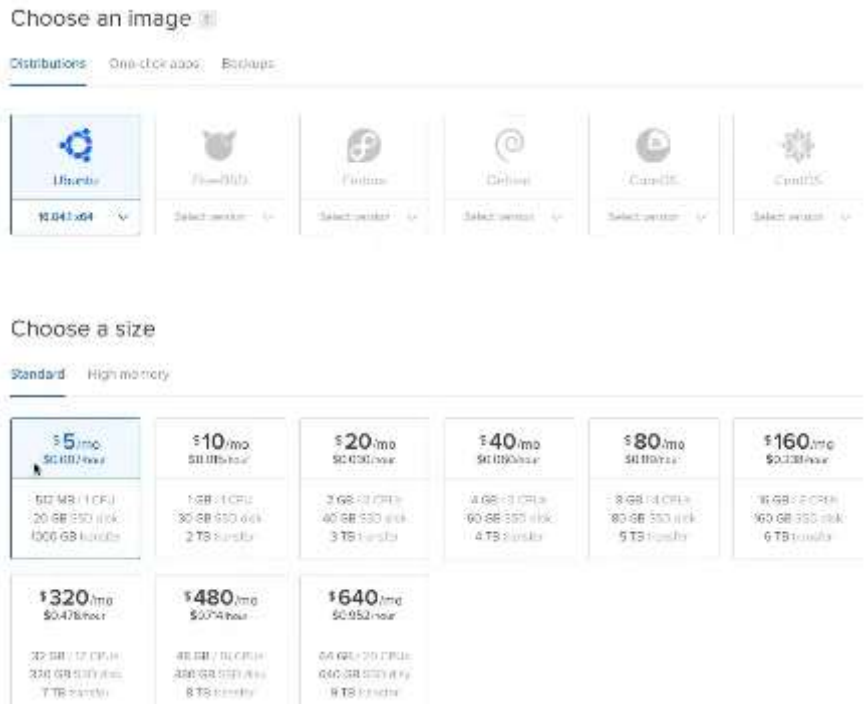
- Automated deployments



Go to Git's dashboard > Deploy > Select branch > Enable automatic deploys → This means that whenever a commit arrives at master, it's going to deploy to GitHub (you can see this in the Activity tab) > After deployment, go back to terminal and do *git branch -d* to delete the branch to prevent working on the wrong branch in future and to keep the master branch clean (no untested code gets committed to master)

Deploying to Own Server

- Allows full control over users, speed, security, performance and deployment
- Setting up DigitalOcean



In DigitalOcean, droplets are the servers that you own. Click on Droplets > Choose image (i.e. what OS droplet is going to run) > Choose data centre region > Additional options (e.g. private networking, backups, internet protocol like IPv6, SSH keys)

SSH keys won't generate a password, and users can only connect with the computer that owns the SSH key

After creation of droplets > Go to "More" > Access Console > Login is root and password is from your email > You will see that you are the root user in the command prompt on Ubuntu (Unix server)

- Installing PostgreSQL

```
Joses-MBP:~ jslvtr$ ssh root@188.226.129.140
root@188.226.129.140's password:
Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-47-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

21 packages can be updated.
8 updates are security updates.


Last login: Thu Nov 24 10:11:00 2016
root@rest-api-course-trial:~# apt-get update
Hit:1 http://security.ubuntu.com/ubuntu xenial-security InRelease
Hit:2 http://ams2.mirrors.digitalocean.com/ubuntu xenial InRelease
Hit:3 http://ams2.mirrors.digitalocean.com/ubuntu xenial-updates InRelease
Hit:4 http://ams2.mirrors.digitalocean.com/ubuntu xenial-backports InRelease
Reading package lists... Done
root@rest-api-course-trial:~# apt-get install postgresql postgresql
```

apt-get update → Tell apt, the Ubuntu package manager, to find out if there are any updates available for anything we've got installed in the server (upgrade will not be performed, just listing all the potential upgrades)

apt-get install → Tell apt to install specific packages

When we install PostgreSQL, it adds another user to our server (the owner of PostgreSQL) → To swap to this user, type *sudo -i -u postgres*

We've created a PostgreSQL database called, postgres, and when we run the psql command, we connect to the database that is called the same as our user.

Can use *\conninfo* to get some information about the connection

Can user *\q* to exit the psql process, leave database, and goes back to Unix terminal

Can key in *exit* to log out of the postgres user and go back to being the root user

```
postgres=# \conninfo
You are connected to database "postgres" as user "postgres" via socket in "/var/run/postgresql" at port "5432".
postgres=# \q
postgres@rest-api-course-trial:~$ exit
logout
root@rest-api-course-trial:~#
```

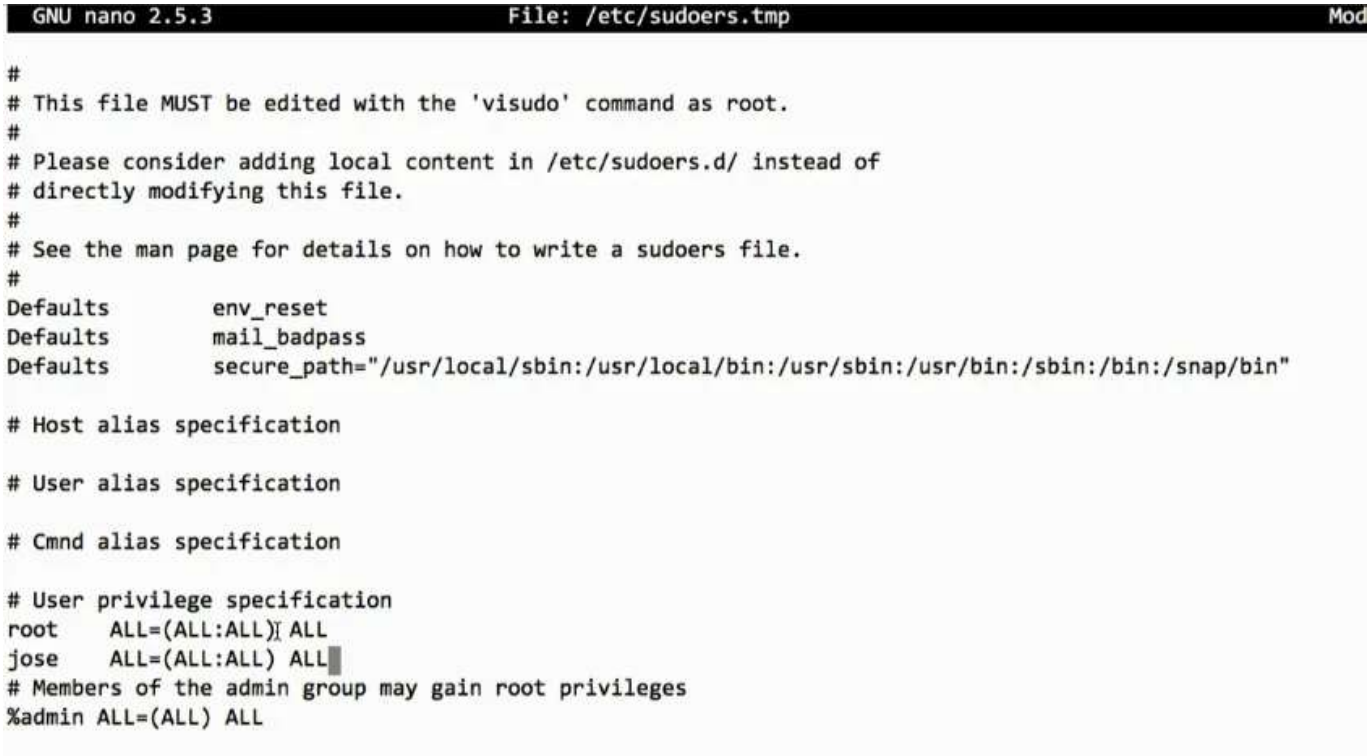
- Creating another user in Unix

```
root@rest-api-course-trial:~# adduser jose
Adding user `jose' ...
Adding new group `jose' (1000) ...
Adding new user `jose' (1000) with group `jose' ...
Creating home directory `/home/jose' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for jose
Enter the new value, or press ENTER for the default
        Full Name []: Jose Salvatierra
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n]
root@rest-api-course-trial:~#
```

adduser jose [this creates a new user called jose, and also a new group called jose] → A group is a collection of users and you can set group permissions

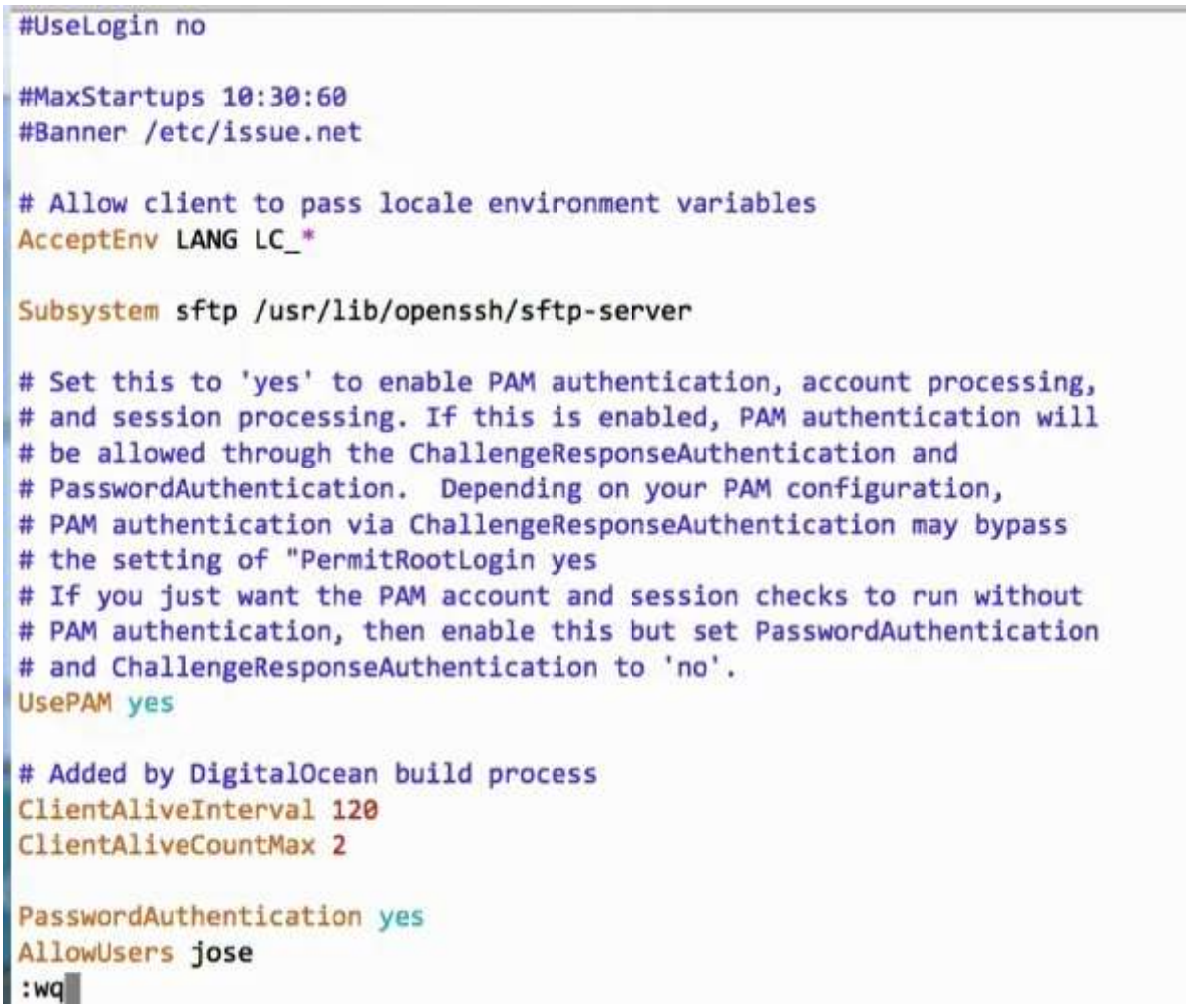
visudo → Opens up a file which you can modify to allow jose acess to being root sometimes

Type *Jose ALL=(ALL:ALL) ALL* > Save file to disk by running CTRL+O > CTRL + X to exit



After this we should be able to log in to the server directly as jose instead of root user using *vi /etc/ssh/sshd_config* > This opens up another text file where you should look for a liner called "PermitRootLogin yes" → This means we are permitting people to log in as root user (not secure) > Therefore press "L" key to go into insert mode before deleting the *yes* and replace with a *no* > Press escape to leave insert mode > Go to bottom of file and press "L" key again to go into insert mode > Write "AllowUsers jose" → This tells SSH

(connection protocol) to allow jose to connect with this server> Press escape > Press :wq to write and quit > Write *service sshd reload*



After exiting, go to terminal and log in as jose > Since jose has access to being root, type *sudo su* to log in as root user > Press exit to bring back to computer

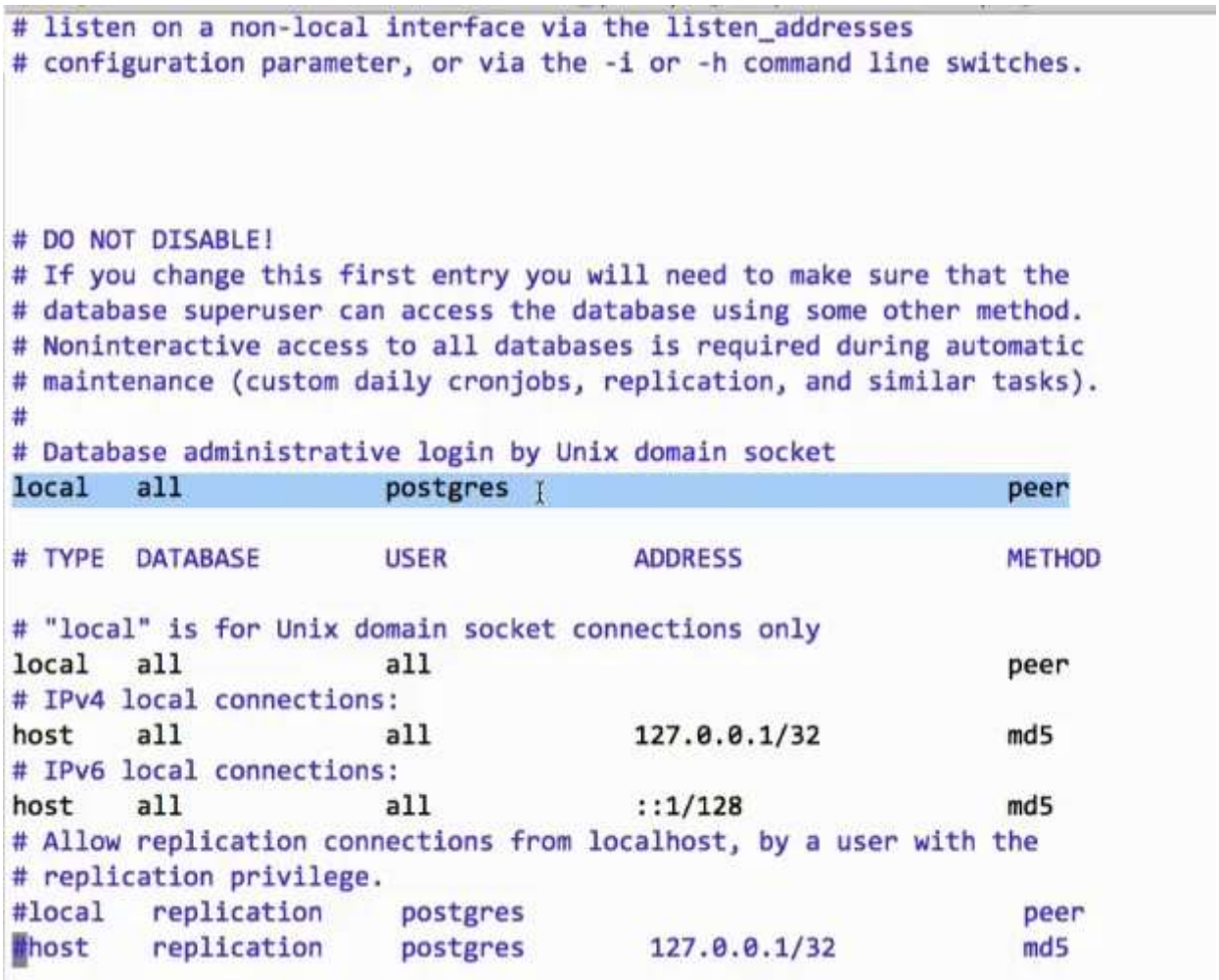
```
Joses-MBP:~ jslvtr$ ssh jose@95.85.15.100
jose@95.85.15.100's password:
Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-47-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

21 packages can be updated.
8 updates are security updates.



The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

jose@rest-api-course-trial:~$ sudo su
[sudo] password for jose:
root@rest-api-course-trial:/home/jose# exit
exit
jose@rest-api-course-trial:~$ exit
logout
Connection to 95.85.15.100 closed.
Joses-MBP:~ jslvtr$
```

- Setting up user with PostgreSQL permissions

```
Last login: Thu Nov 24 11:08:22 2016 from 80.234.172.233
jose@rest-api-course-trial:~$ sudo su
[sudo] password for jose:
root@rest-api-course-trial:/home/jose# sudo -i -u postgres
postgres@rest-api-course-trial:~$ createuser jose -P
Enter password for new role:
Enter it again:
postgres@rest-api-course-trial:~$ createdb jose
postgres@rest-api-course-trial:~$ exit
logout
root@rest-api-course-trial:/home/jose# exit
exit
jose@rest-api-course-trial:~$ psql
psql (9.5.5)
Type "help" for help.
```

Log in to server > Input password > Become root user > Become postgres user > Create PostgreSQL user (must have same name as Unix user we created → *createuser jose -P*) > *-P* allows us to set password for this user > Create a database for that user (*createdb*) > Press exit to go back to user > Type *psql* to connect to jose database > You can do *dropdb* to delete a database (need to be postgres user)

Run *vi /etc/postgresql/9.5/main/pg_hba.conf (* PostgreSQL's user login security configuration file) as root



There are four lines that are not comments. The first one means if we connect from within our server to any database as the postgres user, the peer means we will have access to everything without questions asked. The next

one, if we are locally, and we are any user asking for any database, we're going to have access. The third one means when we are connecting from a different machine or through the internet/IP so if somebody requests to join or connect to the database and that request is coming from 127.0.0.1 (host, i.e. request is coming from the internet as opposed to from the server/local), no matter who they are and what database they're connecting to, we're going to use the md5 method (i.e. asking for a password). Final one just means instead of using IPB4, it's using IPV6.

For server security purposes → Press "I" to go into insert mode > Change 2nd option from peer to md5 > Press escape > Press :wq (write and quit)

- Setting up Nginx → Reverse proxy, i.e. a gateway between our application and external users, accepting requests and deciding what to do with these requests. Also communicates with uwsgi to enable multi-threaded operation of apps. Can also allow you to run multiple Flask apps simultaneously in your server while redirecting incoming requests to different apps

```
jose@rest-api-course-trial:~$ sudo apt-get update
Get:1 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Hit:2 http://ams2.mirrors.digitalocean.com/ubuntu xenial InRelease
Hit:3 http://ams2.mirrors.digitalocean.com/ubuntu xenial-updates InRelease
Hit:4 http://ams2.mirrors.digitalocean.com/ubuntu xenial-backports InRelease
Fetched 102 kB in 1s (96.5 kB/s)
Reading package lists... Done
jose@rest-api-course-trial:~$ sudo apt-get install nginx
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  fontconfig-config fonts-dejavu-core libfontconfig1 libgd3 libjbig0 libjpeg-turbo8 libjpeg8 libtiff5 libvpx3
  libxpm4 nginx-common nginx-core
Suggested packages:
  libgd-tools fcgiwrap nginx-doc
The following NEW packages will be installed:
  fontconfig-config fonts-dejavu-core libfontconfig1 libgd3 libjbig0 libjpeg-turbo8 libjpeg8 libtiff5 libvpx3
  libxpm4 nginx nginx-common nginx-core
0 upgraded, 13 newly installed, 0 to remove and 23 not upgraded.
```

We will set up Nginx to allow requests to go straight to application:

Log in to server > *sudo apt-get update* (apt-get communicates with Ubuntu package manager to change stuff like version number, etc. → run as root) > *sudo apt-get install nginx* > Let Nginx have access through firewall (*sudo ufw status*) → ufw = Ubuntu Firewall > If status is inactive, key *sudo ufw enable* > *sudo ufw allow 'Nginx HTTP'* > Check if Nginx is running using *systemctl status nginx*

```
jose@rest-api-course-trial:~$ sudo ufw status
Status: inactive
jose@rest-api-course-trial:~$ sudo ufw enable
Command may disrupt existing ssh connections. Proceed with operation (y|n)? y
Firewall is active and enabled on system startup
jose@rest-api-course-trial:~$ sudo ufw allow 'Nginx HTTP'
Rule added
Rule added (v6)
jose@rest-api-course-trial:~$ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
Nginx HTTP                 ALLOW       Anywhere
Nginx HTTP (v6)            ALLOW       Anywhere (v6)

jose@rest-api-course-trial:~$ systemctl status nginx
● nginx.service - A high performance web server and a reverse proxy server
   Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2016-11-24 11:22:54 UTC; 1min 30s ago
 Main PID: 9271 (nginx)
   CGroup: /system.slice/nginx.service
           ├─9271 nginx: master process /usr/sbin/nginx -g daemon on; master_process on
           └─9272 nginx: worker process
```

Ensuring Nginx can access the app:

```
server {
listen 80;
real_ip_header X-Forwarded-For;
set_real_ip_from 127.0.0.1;
server_name localhost;

location / {
include uwsgi_params;
uwsgi_pass unix:/var/www/html/items-rest/socket.sock;
uwsgi_modifier1 30;
}

error_page 404 /404.html;
location = /404.html {
root /usr/share/nginx/html;
}

error_page 500 502 503 504 /50x.html;
location = /50x.html {
root /usr/share/nginx/html;
}
```

s*udo vi /etc/nginx/sites-available/items-rest.conf* > Press "I" to go into insert mode > Start writing Nginx config > 1st part is server, where server is listening on port 80 (default HTTP port) > *Real_ip_header X-Forwarded-For* (forward the ip address of the requester over to our Flask app) > *set_real_ip_from 127.0.0.1* (tell app that request really is coming from 127.0.0.1, the local host) > *server_name localhost* (Flask app is running on local host) > Under *location*, we will define that whenever somebody accesses root location of the server, redirect them to our Flask app > *include uwsgi_params* (allows Nginx to communicate with the uwsgi programme) > *uwsgi_pass unix:/var/www/html/items-rest/socket.sock* is going to pass to this file all the necessary parameters, and this file will be the connection point between our Flask app and Nginx > *uwsgi_modifier1 30* (passing uwsgi parameter that tells threads when to die if they become blocked) > Define error pages

```
jose@rest-api-course-trial:~$ systemctl status nginx
● nginx.service - A high performance web server and a reverse proxy server
   Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2016-11-24 11:22:54 UTC; 1min 30s ago
 Main PID: 9271 (nginx)
   CGroup: /system.slice/nginx.service
           ├─9271 nginx: master process /usr/sbin/nginx -g daemon on; master_process on
           └─9272 nginx: worker process
jose@rest-api-course-trial:~$ sudo vi /etc/nginx/sites-available/items-rest.conf
jose@rest-api-course-trial:~$ sudo ln -s /etc/nginx/sites-available/items-rest.conf /etc/nginx/sites-enabled/
jose@rest-api-course-trial:~$ sudo mkdir /var/www/html/items-rest
jose@rest-api-course-trial:~$ sudo chown jose:jose /var/www/html/items-rest
jose@rest-api-course-trial:~$ cd /var/www/html/items-rest/
jose@rest-api-course-trial:/var/www/html/items-rest$ git clone https://github.com/schoolofcode-me/stores-rest-api
.git .
Cloning into '.'...
remote: Counting objects: 36, done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 36 (delta 11), reused 36 (delta 11), pack-reused 0
Unpacking objects: 100% (36/36), done.
Checking connectivity... done.
jose@rest-api-course-trial:/var/www/html/items-rest$ mkdir log
jose@rest-api-course-trial:/var/www/html/items-rest$ ls
app.py  log     Procfile   requirements.txt  run.py       security.py
db.py   models  readme.md  resources         runtime.txt  uwsgi.ini
jose@rest-api-course-trial:/var/www/html/items-rest$ sudo apt-get install python-pip python3-dev libpq-dev
Reading package lists... 0%
```

Above: *sudo ln -s /etc/nginx/sites-available/items-rest.conf /etc/nginx/sites-enabled* → create a link, a soft link between the config file we just created and the sites enabled folder, where Nginx reads the config properties > *sudo mkdir /var/www/html/items-rest* (we used this folder in the config file and this is where our application is going to live) > *sudo chown jose:jose /var/www/html/items-rest* (change owner of this directory to jose user and the jose group as opposed to the root user and the root group)> git clone our Python app through the hit hub repository and with the stores-rest.api > Create directory called log (*mkdir log*) > Install a bunch of Python related things that we need (pip, uwsgi, psycoPG, libpq-dev) > pip install virtualenv > Create a virtual environment within the same folder > Go into the virtual environment (*venv/bin/activate*) > i*nstall -r requirements.txt* (get all of the requirements from the requirements.txt file, and install them in this virtual environment)

- Setting up uWSGI to run Flask app

```
jose@rest-api-course-trial:~$ cd /var/www/html/items-rest/
jose@rest-api-course-trial:/var/www/html/items-rest$ ls
app.py  log     Procfile   requirements.txt  run.py       security.py  venv
db.py   models  readme.md  resources         runtime.txt  uwsgi.ini
jose@rest-api-course-trial:/var/www/html/items-rest$ sudo vi /etc/systemd/system/uwsgi_items_rest.service
[sudo] password for jose:
```

Create an open to service (a service is a descriptor of a programme, and you can, essentially, set the service to run and restart, set environment variables, etc.) > *sudo vi /etc/systemd*/system*/UWSGI_items_rest.service* (open up text editor called vi and create text file to be edited is /etc/systemd for system daemon → a programme that runs and looks at things inside this folder, /system/UWSGI_items_rest.service)

```
[Unit]
Description=uWSGI items rest

[Service]
Environment=DATABASE_URL=postgres://jose:1234@localhost:5432/jose
ExecStart=/var/www/html/items-rest/venv/bin/uwsgi --master --emperor /var/www/html/items-rest/uwsgi.ini --die-on-
term --uid jose --gid jose --logto /var/www/html/items-rest/log/emperor.log
Restart=always
KillSignal=SIGQUIT
Type=notify
NotifyAccess=all

[Install]
WantedBy=multi-user.target
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

Key "I" to go into insert mode > Type [Unit] → This is a section of a service which allows us to define some properties of the service > Define description of service (useful especially for logging) > Define service properties > Environment (Format: username, colon, password, @ server, colon, port) > Type ExecStart (this is what programme this service runs) →

start with programme name (*/var/www/html/items-rest/venv/bin/UWSGI*), use *master* process (a way that UWSGI has of keeping track of other processes in case we use more than one), use *emperor* (a controller as well), define configuration file for uWSGI (*/var/www/html/items-rest/UWSGI.ini*), *die-on-term* (when we terminate the uWSGI process, the Python code doesn't keep running in the background without doing anything), *uid* (user id), *gid* (group id), log folder (*logto/var/www/html/items-rest/log/emperor.log*) > *Restart=always* (if the process crashes or stopped, then the assessment controller is going to restart the process so it's not dead for a long period of time) > KillSignal=SIGQUIT (There are many different ways to terminate a process in Unix, but SEGQUIT is one of them, and that doesn't force quit. So it will shut down gracefully without losing data, things like that) > Notification parameters > Install → *WantedBy=mult-user.target* (allows us to start the service when the server boots up) > Call on *wq*

```
[uwsgi]
base = /var/www/html/items-rest
app = run
module = %(app)

home = %(base)/venv
pythonpath = %(base)

socket = %(base)/socket.sock

chmod-socket = 777

processes = 8

threads = 8

harakiri = 15

callable = app

logto = /var/www/html/items-rest/log/%n.log
```

Change uwsgi.ini file > Delete everything using d + SHIFT + G > Press "I" to go into insert mode>Define section [uwsgi] > Find base folder where uWSGI programme exists > Define file in which to find the app > Define module we are importing> Define home of the uWSGI process > Define Python path > Define socket file > Change socket permissions to 777 (i.e. anybody can access this socket file) > Define number of processes and threads we want > Define harakiri (If one of these threads gets blocked or has an error and it shuts down, then after 15 seconds, we're going to destroy the thread and create another one) > Define callable > Define place to log Python app > *:wq*

```
jose@rest-api-course-trial:~$ cd /var/www/html/items-rest/
jose@rest-api-course-trial:/var/www/html/items-rest$ ls
app.py  log     Procfile   requirements.txt  run.py       security.py  venv
db.py   models  readme.md  resources         runtime.txt  uwsgi.ini
jose@rest-api-course-trial:/var/www/html/items-rest$ sudo vi /etc/systemd/system/uwsgi_items_rest.service
[sudo] password for jose:
jose@rest-api-course-trial:/var/www/html/items-rest$ vi uwsgi.ini
jose@rest-api-course-trial:/var/www/html/items-rest$ sudo systemctl start uwsgi_items_rest
[sudo] password for jose:
jose@rest-api-course-trial:/var/www/html/items-rest$ vi log/uwsgi.log
```

Starting up our Flask app (*sudo systemctl start uwsgi_items_rest*) → This is going into the service, run the XX start command, which is going to run uWSGI, look at the uWSGI.ini file, run the Flask app and create the socket file, which is going to communicate with engine.exe to route the requests to our Flask app> Then look at uWSGI log folder to see if app is working properly or not

To test the app, make sure you do the following:

- Make sure to delete the configuration property of nginx → or else whenever you try to access your API, the first thing you get is 404 not found, because the default configuration property for nginx will be the first one that gets loaded

```
jose@rest-api-course-trial:/var/www/html/items-rest$ sudo rm /etc/nginx/sites-enabled/default
```

- Reload the configuration for nginx and then restart nginx

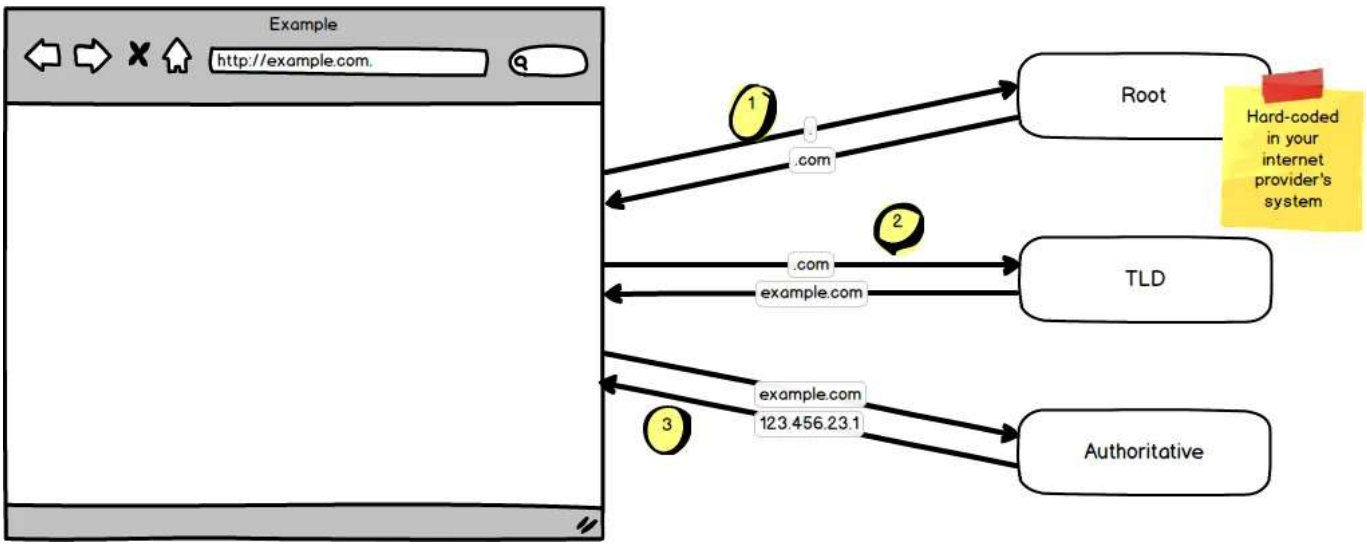
```
jose@rest-api-course-trial:/var/www/html/items-rest$ sudo systemctl reload nginx
jose@rest-api-course-trial:/var/www/html/items-rest$ sudo systemctl restart nginx
jose@rest-api-course-trial:/var/www/html/items-rest$ sudo systemctl start uwsgi_items_rest
jose@rest-api-course-trial:/var/www/html/items-rest$
```

Security in APIs

In order to prevent people from intercepting the data on the internet and being able to read it, we must use Secure Sockets Layer. This sits on top of HTTP and encrypts all communication between a server and a client.

- What is a domain? — A memorable string that internet connected devices can use instead of the IP address they represent. We've got to first purchase the domain name, and then tell the Domain Name System what IP address the domain should point to.
- What is DNS? — The Domain Name System allows all devices to retrieve the IP address corresponding to a domain.

http://example.com.



There are three (and sometimes four) steps in the Domain Name System.

1. The root servers;
2. The TLD servers;
3. The Authoritative server(s).

When your browser requests `http://example.com`, two things can happen. Either the browser has the IP address stored in memory (e.g. because you've accessed it recently), or it has to go to the DNS to find out the IP address (DNS query).

The browser asks the Operating System to perform a series of requests. First, it goes to the hard-coded root servers. These can be hard-coded in your internet provider or your operating system.

- The root servers know what the `.com` DNS servers are (these are TLD name servers);

- The TLD servers know what the `example.com` DNS servers are (these are Authoritative nameservers);
- The Authoritative nameservers know what the IP address of `example.com` is.

Content Delivery Network (CDN) like Cloudflare also lets us use them as Authoritative nameservers. CDNs sit in front of our servers, so anyone accessing our domain will in fact access the CDN first. This is helpful in terms of boosting access speeds, and also improves our security through prevention of denial-of-service attacks.

- Types of DNS records

The **Address** (A) record maps a domain name to an IPv4 address. For example, `rest-api-course-trial.com` to `95.85.15.100`. An IPv4 address is the normal IP address that we know and love. This is what DigitalOcean gives us as a location of our server.

The **IPv6 Address** (AAAA) record maps a domain name to an IPv6 address. For example, `rest-api-course-trial.com` to `2001:0db8:85a3:0000:0000:8a2e:0370:7334`. An IPv6 address behaves in just the same way as IPv4. However, as you can see it is longer and has a slightly different structure. IPv6 was created because we are *running out* of IPv4 addresses. There are too many devices that IPv4 can no longer provide a unique address to each device. This is where the longer IPv6 comes in.

A **Canonical name** (CNAME) record is used to create an alias for a domain name. For example, we can create an alias for `rest-api-course-trial.com` that

is `www.rest-api-course-trial.com`. That way, our users will be able to access either of those two domains when requesting our page.

The **Mail Exchange** (MX) record is used to specify mail exchange servers for a domain name. A mail exchange server is one that processes or forwards mail. This type of record is used when creating e-mail addresses that use our domain name (e.g. `jose@schoolofcode.me`.

The **Text** (TXT) record is used to store a string against a domain name. These are often used to verify domain ownership (e.g. if you can set a TXT record to have the value `x`, that means you own the domain because you were able to perform that change).

- HTTPS — You need SSL certificate key and pem files, which the server needs to enable HTTPS.
- After you get the SSL certificate in your server, tell NGINX to use it by modifying the configuration file.
- Sometimes, users will access the `http` version of our site—accidentally or because they have it bookmarked. We can tell nginx to redirect users to the `https` version of the site when this happens.
- Below are a couple resources explain SSL well:

1. http://robertheaton.com/2014/03/27/how-does-https-actually-work/
2. https://www.digicert.com/ssl/
3. https://www.websecurity.symantec.com/security-topics/how-does-ssl-handshake-work

flask-JWT-extended

- I did not take down much notes for this section. However, the official Flask-JWT-Extended doumentation is a great place to learn more!
- Claims in flask-JWT-extended are just pieces of data we can choose to attach to the JWT payload. Claims are used to add some extra data that will allow us to do something when the JWT comes back to us.
- jwt_optional is going to let you add into any endpoint that the jwt can or cannot be present. Then inside the endpoint, you can choose what to do if it is present or not. This can be useful if you want to allow your endpoints to return some data if the user is logged in and a different data if they are logged out.
- Token refreshing → Fresh token is generated upon user authentication (i.e. enter username and password). A non-fresh token is a token you've received by refreshing a previous token.

Note for above, if we set fresh to "true" that means that all tokens this user would create, both by logging in and by refreshing would be fresh.

```python
@jwt_required
def get(self, name):
    item = ItemModel.find_by_name(name)
    if item:
        return item.json()
    return {'message': 'Item not found'}, 404

@fresh_jwt_required
def post(self, name):
    if ItemModel.find_by_name(name):
        return {'message': "An item with name '{}' already exists.".format(name)}, 400
```

Using decorators for fresh and non-fresh tokens requirements

- flask-JWT-extended allows us to create blacklists. A blacklist allows us to have a list of things that we don't want to allow access to.

```python
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///data.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['PROPAGATE_EXCEPTIONS'] = True
app.config['JWT_BLACKLIST_TOKEN_CHECKS': str
app.config['JWT_BLACKLIST_TOKEN_CHECKS'] = ['access', 'refresh']
app.secret_key = 'jose'  # app.config['JWT_SECRET_KEY']
api = Api(app)
```

```python
@jwt.token_in_blacklist_loader
def check_if_token_in_blacklist(decrypted_token):
    return decrypted_token['identity'] in BLACKLIST
```

- Blacklists can also enable user log outs. By blacklisting the JWT they sent us, they will have to get a new JWT by logging in again. All we have to do is get a unique identifier for the access token (jti in jwt standards and language. So the jti, it stands for jwt ID).

```python
class UserLogout(Resource):
    @jwt_required
    def post(self):
        jti = get_raw_jwt()['jti']  # jti is "JWT ID", a unique identifier for a JWT.
        BLACKLIST.add(jti)
        return {'message': 'Successfully logged out.'}, 200
```

Rest Api Python

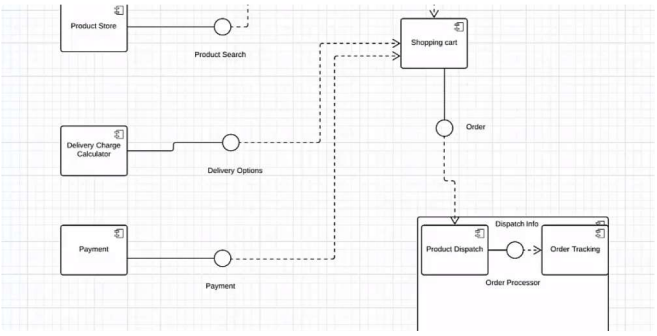
Written by ThadT

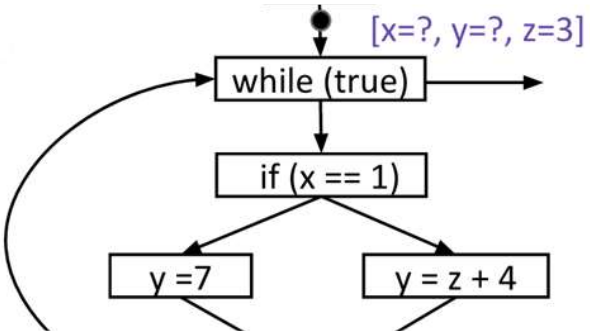
Follow

82 Followers

Technologically philosophizing

More from ThadT





ThadT

Enterprise Software Architecture

This is a short course on Udemy (Software Architecture for the Enterprise Architect)...

13 min read · Mar 19, 2020

15 1

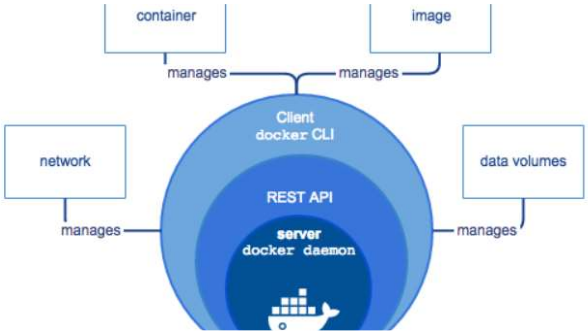
ThadT

Software Analysis (Basic Introduction)

Software analysis consists of methods with specialized tools for software diagnosis and...

12 min read · Feb 9, 2020

10



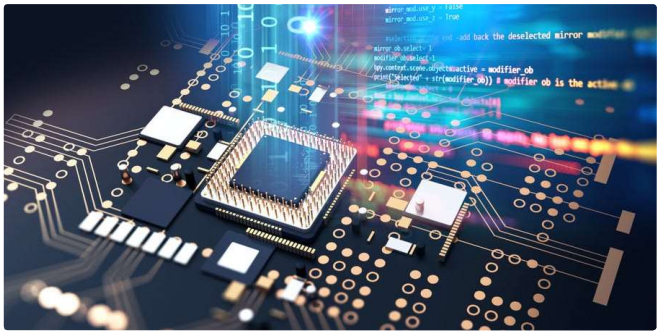
ThadT

A High Level Overview of Docker

What is Docker?

9 min read · Feb 28, 2020

2



ThadT in Analytics Vidhya

A Primer To Data Engineering

Data engineering is a truly intimidating field to those who are looking to venture into it....

12 min read · Jan 10, 2020

1

See all from ThadT

Recommended from Medium

Ayush Sharma

Creating Your Own API in Python: A Beginner's Guide

Introduction: In today's interconnected world, Application Programming Interfaces (APIs)...

3 min read · Jul 16

16

Sandyjtech

Building a Full Stack App with Flask, React, MySQL

Creating a full-stack app is a daunting task and I remember feeling overwhelmed and lo...

6 min read · Sep 1

14

Lists

Coding & Development

11 stories · 313 saves

Predictive Modeling w/ Python

20 stories · 687 saves

Practical Guides to Machine Learning

10 stories · 785 saves

ChatGPT

23 stories · 311 saves

Ashley Alex Jacob

Flask API Folder Guide 2023

Having a consistent folder structure is key to success for various projects. After spending ...

12 min read · Jul 29

10 1

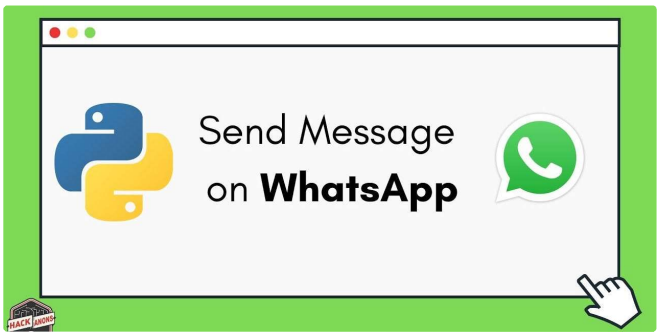
Brian

Logging in Flask: Introduction and Practical Example

Logging in Flask is essential for debugging, monitoring, and tracking the behavior of you...

3 min read · Sep 27

5



SarahDev

Automate WhatsApp Messages With Python using Pywhatkit...

Automating WhatsApp messages using Python and the Pywhatkit module is a fun an...

· 2 min read · Aug 4

83

Pooya Oladazimi

Flask App Postgres Database Initialization: Step-by-Step Guide...

Most applications we develop need Relational Databaes(s) to store different types of...

6 min read · Aug 5

18

See more recommendations