

PC2323 Python Microservices Development with FastAPI



Goh Y. K. (gohyk@utar.edu.my)

05 Jun, 2022 (v 0.1)

Department of Mathematical and Actuarial Sciences,
Universiti Tunku Abdul Rahman

Section 1

Microservices and FastAPI

Introduction to Microservice

1 Introduction

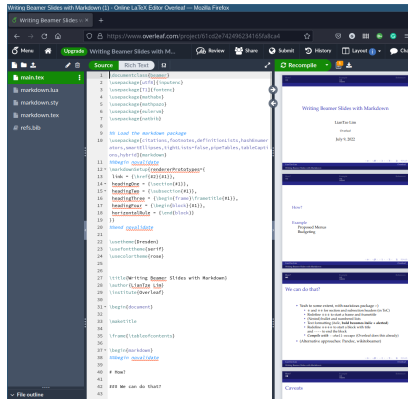
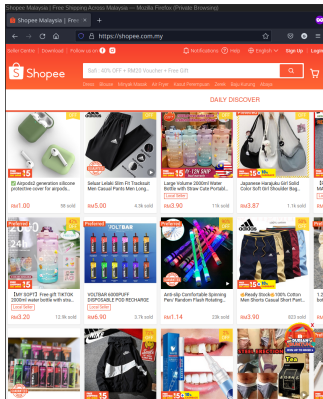
- ▷ what is web app vs web site
- ▷ monolithic architecture vs microservices

2 FastAPI framework

- ▷ features and advantage

Web Services

- Modern website are rarely just static web pages
 - dynamic content
 - application
 - url can be call directly without a browser (api!)



Microservices

Microservices architecture is a pattern for organising services provided by computer systems that can scale with demand.

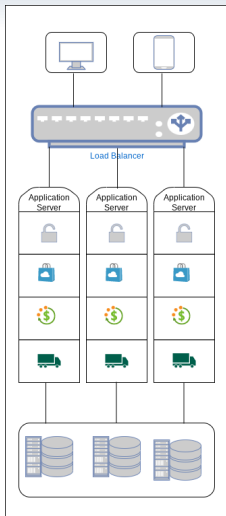
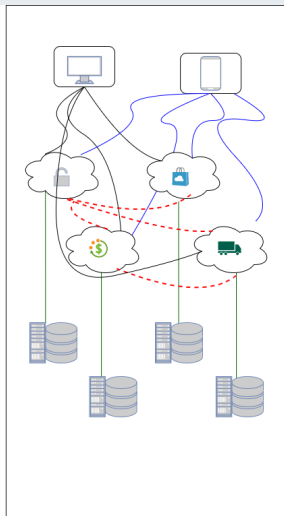
- opposite to monolith architecture
- advantages:
 - ▷ modularity: clean and clearer code for develop and tests
 - ▷ scalability: allow to scale according to demands of each services
 - ▷ distributed: possible different services run on different systems
 - ▷ heterogeneous: services via API calls and thus independent of OS
 - ▷ CI/CD: tools available for automated deployment

Microservices

Microservices architecture is a pattern for organising services provided by computer systems that can scale with demand.

- opposite to monolith architecture
- advantages:
 - ▷ modularity: clean and clearer code for develop and tests
 - ▷ scalability: allow to scale according to demands of each services
 - ▷ distributed: possible different services run on different systems
 - ▷ heterogeneous: services via API calls and thus independent of OS
 - ▷ CI/CD: tools available for automated deployment
- criticism:
 - ▷ complex failure tracing
 - ▷ possible increase in network latency due to inter-service calls
 - ▷ complex organization and responsibility allocations
 - ▷ harder to support as each services might built from different technologies
 - ▷ complicated to patch or update the underlying software
 - ▷ complex containers orchestration

Monolith vs Microservices

**MONOLITH****MICROSERVICES**

FastAPI as Microservice Framework

FastAPI is a relatively new microservice framework

- Initial release: 5 Dec 2018
- Developer: Sebastián Ramírez
- Website: <https://fastapi.tiangolo.com/>
- Adoption: 3rd most loved web framework in Stack Overflow 2021 Developer Survey



Why FastAPI

- **Fast to execute:**
 - ▷ Uses Starlette and Pydantic
 - ▷ on par with Node.js and Go
- **Fast to code:** speed up development by about 200% to 300%.
- **Fewer bugs:** Reduce about 40% of human (developer) induced errors.
- **Intuitive:**
 - ▷ Great editor support.
 - ▷ Completion everywhere.
- **Easy:** Designed to be easy to use and learn.
- **Short:** Minimize code duplication.
- **Robust:**
 - ▷ Get production-ready code.
 - ▷ With automatic interactive documentation.
- **Standards-based:**
 - ▷ Fully compatible with the open standards for APIs: OpenAPI, JSON Schema.

Modern Python and APIs

Modern Python features

- Type hints
- Rich editor support
- `async/await`
- ASGI servers
- Pydantic classes
- OpenAPI documentation
- `pytest`

Summary of the different Roadmaps

No.	Summary	No.	Summary
1.	minimal FastAPI	8.	pydantic data models
2.	setup git and ssh	9.	refactor and APIRouter
3.	end points and data	10.	deploy via nginx and gunicorn
4.	rendering html pages	11.	automate server deployment
5.	adding static files	12.	deploy via docker
6.	end points and external api	13.	deploy to Heroku
7.	async end points	14.	FastAPI with database

Section 2

First Simple APIs

First Simple APIs

Outline:

- setup minimal api (Roadmap #1)
- serve api from cloud (Roadmap #2)

Roadmap #1

Local computer

- setup virtual environment
- setup minimal FastAPI app
- start microservice

Tree view of the project

```
.  
├── roadmap_1/  
│   ├── main.py  
│   ├── requirements.txt  
│   └── venv/
```

Roadmap #1 - Setup

Choosing IDE

- Choose a good IDE that is “friendly” to Python project.
- PyCharm or vscode are good candidates
- Demo using PyCharm
 - ▷ auto-activate virtual environment
 - ▷ auto-import with Alt-Enter (you need to set it up)
 - ▷ pip install on the fly
 - ▷ built-in file manager and support tab editing

Create a project folder called PC2323/roadmap_1

- Remember, 1 project 1 folder
- cd into the folder
- create a virtual environment and activate

```
$ python -m venv venv
$ source ./venv/bin/activate # or in windows .\venv\Scripts\activate.bat
(venv) $
```

Roadmap #1 - minimal app

- Drag and drop the roadmap_1 folder into PyCharm:
 - ▷ now roadmap_1 is a project
 - ▷ make sure the PyCharm captured the correct Python interpreter
 - ▷ add in empty files main.py and requirements.txt in the project
 - ▷ click run to make sure all working
- Open main.py and add in

```
import fastapi
import uvicorn
```

- Open requirements.txt and add in

```
fastapi
uvicorn
```

- Open Terminal and pip install

```
(venv) $ pip install -r requirements.txt
```


Roadmap #1 - serve the app

If running `main.py` returns no error, add the following:

```
import fastapi
import uvicorn

print("Hello fastapi")
api = fastapi.FastAPI()

@api.get('/api/endpoint')
def endpoint():
    return {"msg": "Hello everyone"}

uvicorn.run(api)
```

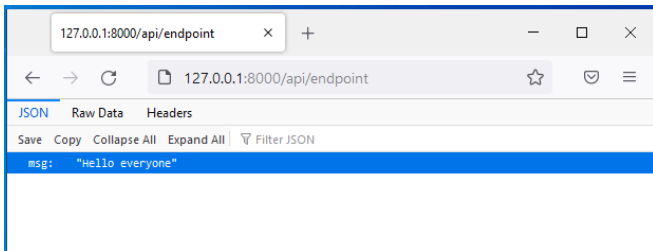
Now, run the app and should see:

```
(venv) python main.py
INFO:      Started server process [7308]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Roadmap #1 - browse the app

Point the browser at `http://127.0.0.1:8000/api/endpoint`

- use firefox for better handling of JSON



Roadmap #2

Git server

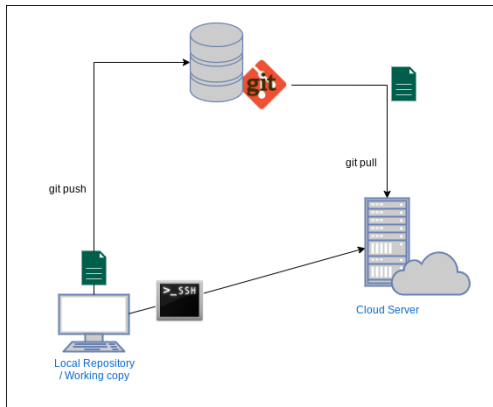
- setup a new git repo

Local computer

- git pull the repo
- edit .gitignore
- setup virtual env
- setup minimal FastAPI app
- git push

Cloud server

- ssh into the server
- git clone or git pull
- start microservice



Roadmap #2 - setup a new repo

Login and create a new git repo

- Demo with `github.com`
- Add `README.md` and `.gitignore` from Python project template

Search or jump to...

Pulls Issues Marketplace Explore

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?
[Import a repository.](#)

Owner ^{*} / Repository name ^{*}

yongkheng / new_pc2323 ✓

Great repository names are short and memorable. Need inspiration? How about [reimagined-octo-guacamole?](#)

Description (optional)

☒ Public
Anyone on the internet can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

☒ Add a README file
This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore
Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: Python

Roadmap #2

Git server

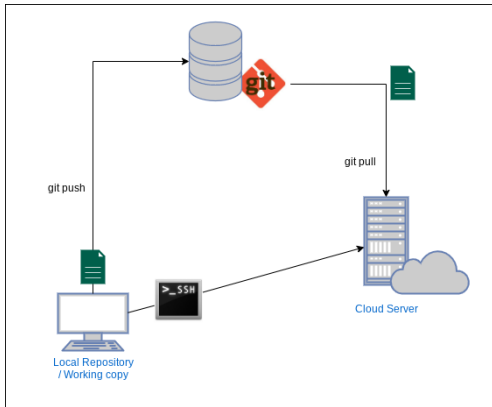
- setup a new git repo

Local computer

- `git pull` the repo
- edit `.gitignore`
- setup virtual env
- setup minimal FastAPI app
- `git push`

Cloud server

- `ssh` into the server
- `git clone` or `git pull`
- start microservice



Roadmap #2 - setup local repo

Take note of the git repo url, for example:

- https://github.com/yongkheng/new_pc2323.git

Change into PC2323 folder, and create a local repo

```
(venv) $ cd PC2323
(venv) $ git init -b main
(venv) $ ls
roadmap_1/
(venv) $ git remote add origin https://github.com/yongkheng/new_pc2323.git
(venv) $ git pull origin main
```

Edit `.gitignore` by adding the following line to the end of the file:

```
...
.idea/
```

This tells git to ignore the PyCharm auto-generated `.idea/` project directory.

Roadmap #2 - add files and push

Add files in roadmap_1 into the local repo and commit

```
(venv) $ git add -A
(venv) $ git commit -m "adding roadmap_1"
[main d4efe7a] add roadmap_1
3 files changed, 18 insertions(+)
create mode 100644 roadmap_1/main.py
create mode 100644 roadmap_1/requirements.txt
```

Make sure the files in venv and .idea are not in the list of changes.
Then, push

```
(venv) $ git push origin main
```

Now, the roadmap_1 folder is transferred to the remote git repo.

Roadmap #2

Git server

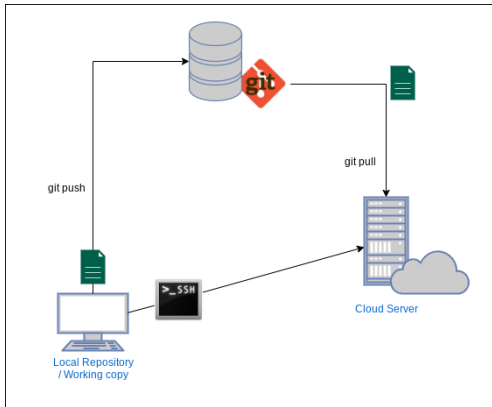
- setup a new git repo

Local computer

- git pull the repo
- edit .gitignore
- setup virtual env
- setup minimal FastAPI app
- git push

Cloud server

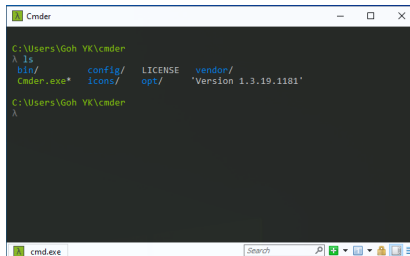
- ssh into the server
- git clone or git pull
- start microservice



Roadmap #2 - CLI

The default console tools in Windows are pretty bad. Get a good console emulator that comes with ssh, scp and basic Linux commands.

- Get cmdr - <https://github.com/cmdrdev/cmdr/releases>
- alternatives are:
 - ▷ WSL (Linux virtual machine)
 - ▷ MobaXterm (non-opensource)
- cmdr
 - ▷ Download the latest release
 - ▷ Extract to user home folder
 - ▷ Create a shortcut to the desktop or taskbar



```
C:\Users\Goh YK\cmdr
^ ls
bin/          config/  LICENSE  vendor/
cmdr.exe*     icons/  opt/     'Version 1.3.19.1181'

C:\Users\Goh YK\cmdr
^
```

Roadmap #2 - ssh

A remote server was created for each participant in this course:

- the server are fresh new Ubuntu server
- the root password of the server is provided

To login the remote server, we can start a ssh session.

- Initially, the server is set to accept password.
- It is possible to setup passwordless login for convenience and for security reason.

Password login:

```
$ ssh root@<IP_ADDRESS_OF_THE_SERVER>  
root@<IP_ADDRESS_OF_THE_SERVER>'s password:
```

Roadmap #2 - ssh

Passwordless login:

- Generate a new ssh-key (avoid using your work id)

```
$ ssh-keygen -C "root" -f ~/.ssh/root_id
```

- copy the content of ~/.ssh/root_id.pub and add to ~/.ssh/authorized_keys in the remote server. In Linux, you can do

```
$ ssh-copy-id -i ~/.ssh/root_id root@<IP_ADDRESS_OF_THE_SERVER>
```

- Unfortunately, cmdr has no ssh-copy-id, hence copy manually.
- Now, ssh login by specifying the key:

```
$ ssh root@<IP_ADDRESS_OF_THE_SERVER> -i ~/.ssh/root_id
```

Roadmap #2 - git clone

Now, pull the FastAPI code from the git repo:

```
# git clone https://github.com/yongkheng/new_pc2323.git
```

If you already cloned the git repo and some of the files are updated, then you can pull down the updated files by:

```
# cd new_pc2323  
# git pull origin main
```

Follow the steps in roadmap_1 to start the microservice:

- cd into roadmap_1 folder
- create a virtual environment
- pip install the needed packages in the requirements.txt
- run the main.py script

Roadmap #2 - check the api

The microservice has started, but still cannot access outside the server.

- to check the microservice is running properly, we can use either `curl` or `httpie` to call the api

```
# apt install curl httpie
# curl http://127.0.0.1:8000/api/endpoint
{"msg": "Hello everyone"}
# http http://127.0.0.1:8000/api/endpoint
HTTP/1.1 200 OK
content-length: 24
content-type: application/json
date: Sat, 23 Jul 2022 09:16:28 GMT
server: uvicorn

{
  "msg": "Hello everyone"
}
```

- eventually we will allow the outside connections via reverse proxy and `gunicorn`

First Simple API - Conclusion

- roadmap_1
 - ▷ to setup a minimalist service
 - ▷ setup python and virtual environment
 - ▷ create the FastAPI script
 - ▷ run the service
 - ▷ access the service from a browser
- roadmap_2
 - ▷ to setup a minimal deployment route
 - ▷ setup a remote git repository
 - ▷ link up to the local git repo
 - ▷ adding the roadmap_1 project in the local repo
 - ▷ push the files to the remote git repo
 - ▷ setup ssh login to remote server
 - ▷ git clone the project repo
 - ▷ run the microservice

First Simple API - What's next

With the basic deployment concept is covered, we now can focus just on the fundamental on FastAPI without the complications of the handling of the remote server. We will look:

- RESTful API with HTTP verbs
- passing data to api
- requests and responses
- rendering HTML

Section 3

Fundamental of FastAPI

Fundamental of FastAPI

Outline:

- RESTful API with HTTP verbs
- passing data to api
- requests and responses
- rendering HTML

Roadmap #3

Local computer

- setup virtual environment
- copy content from roadmap_2 to roadmap_3
- start microservice

Tree view of the project

```
.  
├-- roadmap_3/  
|   |-- main.py  
|   |-- requirements.txt  
└-- venv/
```

Path and Path Operation Function

```
import fastapi
import uvicorn

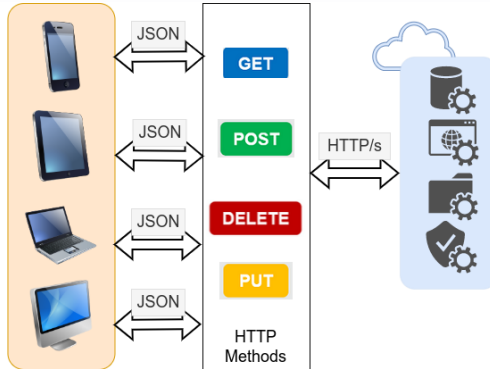
print("Hello fastapi")
api = fastapi.FastAPI()

@api.get('/api/endpoint')
def endpoint():
    return {"msg": "Hello everyone"}

uvicorn.run(api)
```

- `@api.get('/api/endpoint')` associates an operation to the immediate function:
 - ▷ `get()` is the **operation**
 - ▷ `/api/endpoint` is the **path** to the resources
 - ▷ `endpoint()` is the **path operation function**
- Other common operations are: `post`, `delete`, `put`
 - ▷ these are examples of http methods and RESTful API

RESTful API Architecture



REST API (Representational State Transfer API) principles:

- stateless, client-server, uniform interface, cacheable, layered system, code-on-demand

HTTP Methods

Frequently used HTTP methods:

Method	Meaning
GET	Read only request of resource
POST	Modify the requested resource
DELETE	Delete the requested resource
PUT	Put a resource on a specific location

Details:

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

Passing Data

roadmap_3:

- create a roadmap_3 folder and copy the requirements.txt file from roadmap_1 into the new folder
- create a main.py file with the following content

```
import fastapi
import uvicorn

api = fastapi.FastAPI()

@api.get("/api/pass_data1")
def pass_data1(x, y):
    print(type(x), type(y))
    return {"msg": x, "ans": x + y}

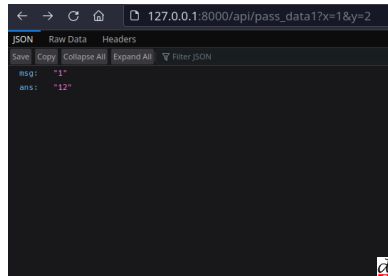
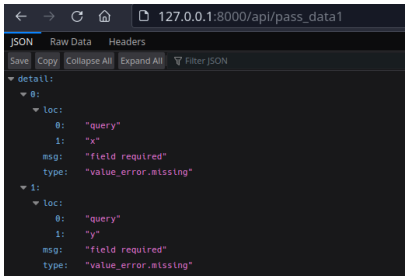
@api.get("/api/pass_data2/{x}/{y}")
def pass_data2(x, y):
    print(type(x), type(y))
    return {"msg": x, "ans": x + y}
```

Passing Data to Path Function

```
@api.get("/api/pass_data1")
def pass_data1(x, y):
    print(type(x), type(y))
    return {"msg": x, "ans": x + y}
```

Pass input arguments via path operation function:

- `pass_data1()` has 2 required parameters `x` and `y`
- to pass the parameter, append `?x=1&y=2` to the url

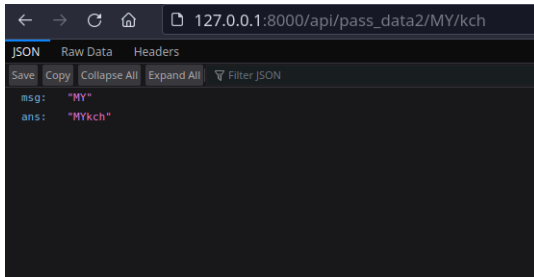


Passing Data to Path Function

```
@api.get("/api/pass_data2/{x}/{y}")
def pass_data2(x, y):
    print(type(x), type(y))
    return {"msg": x, "ans": x + y}
```

Another way of passing data to the path function is by using the path

- use {var_name} as a place-holder to capture the variable via url
- useful when request for web api to return items by id from database



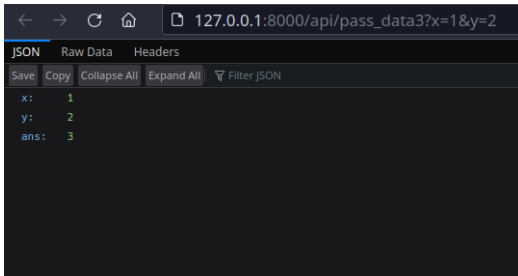
Passing Data - type hinting

The data send through the url requests are strings:

- FastAPI support auto type conversion via type hinting

```
@api.get("/api/pass_data3")
def pass_data3(x:int, y: int):
    ans = x + y
    return {"x": x, "y":y, "ans": ans}
```

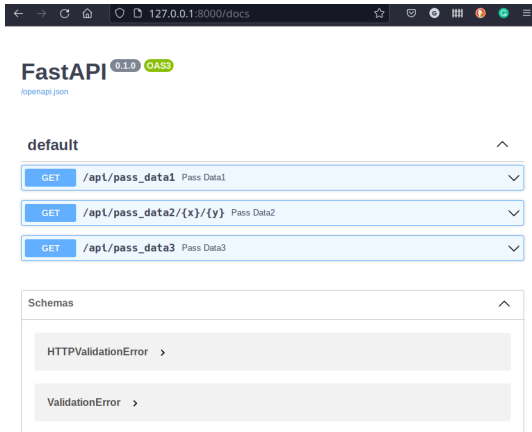
- now, x, y and ans are all integers



Requests and Responses

FastAPI comes with Swagger UI for visualise and interact with the API created.

- to access Swagger UI, open `http://127.0.0.1:8000/docs`



Requests and Responses

Try out the Swagger UI by submitting a GET request:

The screenshot displays a Swagger UI interface for a FastAPI application. It shows a GET request to the endpoint `http://127.0.0.1:8000/api/pass_data3?x=12&y=16` with the `accept` header set to `application/json`. The server response is a 200 status code with a JSON body: `{ "x": 12, "y": 16, "ans": 28 }`. The response headers include `content-length: 24`, `content-type: application/json`, `date: Fri, 29 Jul 2022 13:39:32 GMT`, and `server: uvicorn`. A 'Download' button is visible next to the response body.

```
Curl
curl -X 'GET' \
'http://127.0.0.1:8000/api/pass_data3?x=12&y=16' \
-H 'accept: application/json'

Request URL
http://127.0.0.1:8000/api/pass_data3?x=12&y=16

Server response
Code    Details
200
Response body
{
  "x": 12,
  "y": 16,
  "ans": 28
}
Download
Response headers
content-length: 24
content-type: application/json
date: Fri, 29 Jul 2022 13:39:32 GMT
server: uvicorn
```

- we can also use the CLI tools like `httpie` or `curl` to send requests without a browser.
- note the successful response gives status code of 200

Requests and Responses

Let's try a failed requests using httpie

- note the response status code 422 in the header:

```
$ http 'http://127.0.0.1:8000/api/pass_data3?x=12&y=a'
HTTP/1.1 422 Unprocessable Entity
content-length: 99
content-type: application/json
date: Fri, 29 Jul 2022 13:58:53 GMT
server: uvicorn
```

```
{
  "detail": [
    {
      "loc": [
        "query",
        "y"
      ],
      "msg": "value is not a valid integer",
      "type": "type_error.integer"
    }
  ]
}
```

HTTP Status Code

Status codes are feedbacks from server to the client:

- **1xx** Informational Response
- **2xx** Success
- **3xx** Redirection
- **4xx** Client Errors
- **5xx** Server Errors

Reference websites

- <https://httpstatuscodes.com>
- <https://http.cat/422>
- <https://http.dog/422>



Manual Response

The key of a good API is clear communication. The following is an example of bad response feedback (status 200!):

```
@api.get("/api/pass_data4")
def pass_data4(x:int, y: int):
    try:
        ans = x / y
    except ZeroDivisionError :
        return {"Error": "Divide by zero"}
    return {"x": x, "y":y, "ans": ans}
```

```
$ http "http://127.0.0.1:8000/api/pass_data4?x=1&y=0"
HTTP/1.1 200 OK
content-length: 27
content-type: application/json
date: Fri, 29 Jul 2022 15:56:35 GMT
server: uvicorn

{
  "Error": "Divided by zero"
}
```

Manual Response

Think another program that is trying to talk to this API, the wrong status code will lead to unexpected result. We want to return a proper response:

```
@api.get("/api/pass_data5")
def pass_data5(x:int, y: int):
    try:
        ans = x / y
    except ZeroDivisionError :
        return fastapi.Response(content='{"Error": "Divide by zero"}',
                                status_code=400)
    return {"x": x, "y":y, "ans": ans}
```

```
$ http "http://127.0.0.1:8000/api/pass_data5?x=1&y=0"
HTTP/1.1 400 Bad Request
content-length: 27
date: Fri, 29 Jul 2022 16:39:38 GMT
server: uvicorn

{"Error": "Divide by zero"}
```

Now the status code is 400 but the content-type field is missing, which means the content-type is not a JSON but a string.

Manual Response

Finally, we will use the `fastapi.responses.JSONResponse` as the specific response.

```
@api.get("/api/pass_data6")
def pass_data6(x:int, y: int):
    try:
        ans = x / y
    except ZeroDivisionError :
        return fastapi.responses.JSONResponse(
            content={"Error": "Divide by zero"}, status_code=400)
    return {"x": x, "y":y, "ans": ans}
```

```
$ http "http://127.0.0.1:8000/api/pass_data6?x=1&y=0"
HTTP/1.1 400 Bad Request
content-length: 26
content-type: application/json
date: Fri, 29 Jul 2022 16:47:41 GMT
server: uvicorn
```

```
{
  "Error": "Divide by zero"
}
```


Rendering HTML

Finally, if we would like to display a HTML page, we can use `HTMLResponse`:

```
@api.get("/")
def index():
    body = '''
    <html> <body style='padding: 10px;'>
    <h1>Landing Page</h1>
    <div>
    API hook: <a href="api/pass_data1?x=3&y=5"> api/pass_data1?x=3&y=5 </a>
    </div>
    </body></html>
    '''
    return fastapi.responses.HTMLResponse(content=body)
```



Landing Page

API hook: api/pass_data1?x=3&y=5

- In next section, we will use Jinja2 to create dynamic pages.

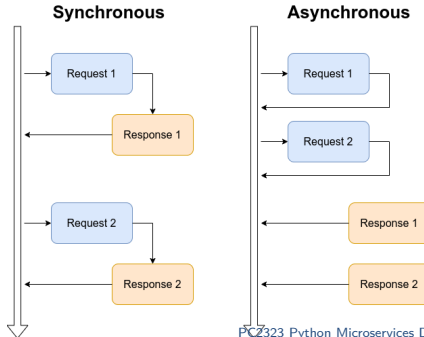
Section 4

Modern Python Language Features

Asynchronous Programming

Asynchronous programming:

- coding that allows program to divert and handle other tasks while waiting for the response of existing tasks.
- the key component for asynchronous program is **coroutine**:
 - ▷ coroutine is a function that can be paused and resume at later point.
 - ▷ in Python, coroutine can be created using `async def`.
 - ▷ then, in the `async` function use, `await` to mark the diversion.





async/await

Asynchronous Program

```
import asyncio
import time

async def get_urls(arg):
    await asyncio.sleep(arg)
    print("retrieved all urls")

async def write_msg(arg):
    await asyncio.sleep(arg)
    print("write all to database")

async def main():
    tic = time.time()
    await asyncio.gather(
        get_urls(3),
        write_msg(2),
    )
    toc = time.time()
    print(f"Elapsed: {toc-tic} s")

asyncio.run(main())
```

Synchronous Program

```
import time

def get_urls(arg):
    time.sleep(arg)
    print("retrieved all urls")

def write_msg(arg):
    time.sleep(arg)
    print("write all to database")

def main():
    tic = time.time()
    get_urls(3)
    write_msg(2)
    toc = time.time()
    print(f"Elapsed: {toc-tic} s")

main()
```

FastAPI

Asynchronous FastAPI

FastAPI supports `async/await` out of the box.

- when your path function is awaiting for coroutines, it can be made `async` by declaring `async def`

```
@api.get('some_path/')
async def read_results():
    results = await coroutine_to_retrieve_results()
    return results
```

- if no coroutine involves in path function, then it is synchronous

```
@api.get('some_path/')
def read_results():
    results = function_to_retrieve_results()
    return results
```

- Note that you will need a `async` web server gateway to leverage on FastAPI `async` capabilities. A good candidate is `uvicorn`.

WSGI vs ASGI

WSGI (Web Server Gateway Interface) servers allow server side running of Python app

- WSGI is the core of microservices
- Traditional web server like apache does not run Python directly
- `mod_python` was used prior to WSGI but lack of maintenance
- examples of WSGI: Gunicorn, uWSGI, FastWSGI, uvicorn

Advantage of WSGI:

- **flexible**
 - ▷ able to swap out the web stack components without modifying the application stack
 - ▷ for example: exchange uWSGI to Gunicorn
- **scaling**
 - ▷ segregation of application stack and web stack

ASGI is WSGI with async capabilities:

- examples
 - ▷ uvicorn - support HTTP/1.1 and WebSocket
 - ▷ hypercorn - support HTTP/2
 - ▷ daphne - support HTTP/2

Type Hints

Using type hint is similar to declaration in other programming languages:

- Type hint is optional to Python
- Type hint helps to “teach” IDE to be smart
- Also tells “variable x is an object of type Item”

Type hint:

- introduced since Python 3.5
- completely ignored by Python interpreter
- use external tools like `mypy` to do type checking

Syntax:

```
my_var : int = 10

def square( input_var : float ) -> float:
    return input_var ** 2
```

Type Hint Applied

Without type hints:

```
rolling_sum = None

def accumulate(items):
    gain = 0

    # ... is items iterable?

    return gain
```

- `TypeError` if increment with `rolling_sum += 1`.
- `items` is plural, is it iterable?

With type hints:

```
rolling_sum : Optional[int] = None

def accumulate(items: Iterable[Item]) -> int:
    gain = 0

    # ... is items iterable?

    return gain
```

- `rolling_sum += 1` is OK.
- clearly `items` is an iterable container containing the item objects.

Type Checking with mypy

The following code has a bug at the last line.

```
from typing import Union

def square(x : Union[int, float]) -> Union[int, float]:
    return x*x

print(square(12345))
print(square(1.2345))
print(square('12345'))
```

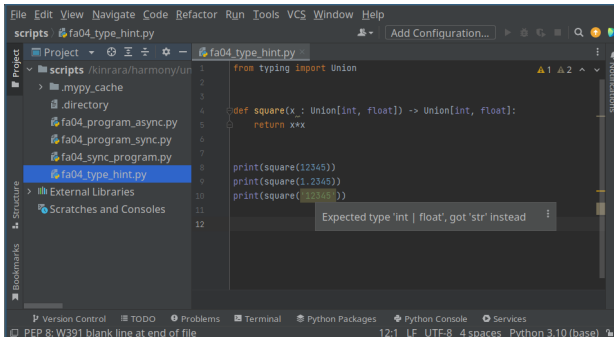
Type checking using mypy helps to identify the bug.

```
$ mypy fa04_type_hint.py
fa04_type_hint.py:8: error: Argument 1 to "square" has incompatible
type "str"; expected "Union[int, float]"
Found 1 error in 1 file (checked 1 source file)
```

Type Hinting with IDE

Modern python IDE like pycharm uses type hint to annotate the code:

- identify type errors
- better suggestion on auto-complete in variable pop-up menu
- **less manual typing, less errors!**



In FastAPI, type hint is used in **auto-conversion** of data type.

Model Validation

Submitting a data to a network API is complex:

- data submitted in JSON, API uses a custom class **model**.
- there will be **conversions** and **validations**
- display error messages when conversions failed etc

The custom class that used in a API is called a **model**

- a model defines the content and data access layer (input to the API)
- when data submitted to API, usually are key-value pairs of strings (JSON)
- the value strings will need to **convert** to correct data types by the model
- the data will also need to **validate** to ensure correct format
- if there is any error happening during the conversion or validation, the custom class will also need to handle the appropriate error messages.

Model Validation - the hard way

```
from datetime import datetime
from typing import List

post_data = {
    'post_id': '123',
    'author': 'John Smith',
    'msg': 'Hello World',
    'reviews': [5, 4, '3']
}

class Post:
    def __init__(self, post_id: int, author: str, msg: str,
                  reviews: List[int]):
        self.post_id = post_id
        self.author = author
        self.msg = msg
        self.reviews = reviews

    def __str__(self):
        return str(self.__dict__)

post = Post(**post_data)
print(post)
```

Model Validation - the hard way

Output of the script

```
{'post_id': '123', 'author': 'John Smith', 'msg': 'Hello World',  
 'reviews': [5, 4, '3']}
```

- `post_id` and `reviews` are having wrong data types
- although type hints are used, but Python interpreter just ignore the hints
- a more complete script that also handling error messages is in `fa04_model_validation_complete.py`
- writing the complete script for the custom model is not trivial

To simplify the coding, use **pydantic**.

Model Validation - the Pydantic way

```
from pydantic import BaseModel

from datetime import datetime
from typing import List

post_data = {
    'post_id': '123',
    'author': 'John Smith',
    'msg': 'Hello World',
    'reviews': [5, 4, '3']
}

class Post(BaseModel):
    post_id: int
    author: str
    msg: str
    reviews: List[int]

post1 = Post(**post_data)
post2 = Post(**post_data)
print(post1)
print(post1 == post2, post1 is post2)
```

Here pydantic helps type conversion, validation and error handling.

Templating with Jinja2

Next modern Python feature is templating tools.

- the most basic usage of templating is using `str.format()`
- however, is lack of flexibility of logic expressions and looping
- FastAPI uses `jinja2` to handle
 - ▷ string substitution
 - ▷ looping and conditional expression
 - ▷ render template from file
 - ▷ environmental variables and configurations

A simple `jinja2` example:

```
from jinja2 import Template
template = Template("Hello {{ name }}! Nice to see you!")
print(template.render(name="John Smith"))
```

Jinja2 - example template

jinja2 delimiters:

- {% ... %} for **Statements**
- {{ ... }} for **Expressions to print** to the template output
- {# ... #} for **Comments** not included in the template output

```
<!DOCTYPE html>
<html lang="en">
<head> <title>My Webpage</title> </head>
<body>
  <ul id="navigation">
    {% for item in navigation %}
      <li><a href="{ item.href }">{{ item.caption }}</a></li>
    {% endfor %}
  </ul>

  <h1>My Webpage</h1>
  {{ a_variable }}

  {# a comment #}
</body>
</html>
```


Jinja2 - rendering template

The following code takes a JSON dictionary and renders according the html template in the previous slide.

```
from jinja2 import Environment, FileSystemLoader

data={
    'navigation': [
        {'href': '/index.html', 'caption': 'Home'},
        {'href': '/about.html', 'caption': 'About'},
        {'href': '/contact.html', 'caption': 'Contact'},
    ],
    'a_variable': 'A short message',
}

env = Environment(
    loader = FileSystemLoader(['./', '/other/path'], encoding='utf8')
)
template = env.get_template('fa04_jinja2_template.j2')
with open('rendered.html', 'w', encoding='utf8') as f:
    rendered = template.render(data)
    f.write(rendered)
```

Section 5

Build a RESTful Service

Build a RESTful Service

Outline:

- serving the landing page (roadmap #4)
- adding static files (roadmap #5)
- calling external API (roadmap #6)
- making API async (roadmap #7)
- using pydantic model in FastAPI (roadmap #8)
- refactor with APIRouter (roadmap #9)

Roadmap #4

Local computer¹

- 1 setup virtual environment
- 2 setup FastAPI app
- 3 **adding and rendering landing page**

```
.  
├-- roadmap_4/  
│   ├── index.html  
│   ├── layout.html  
│   ├── main.py  
│   ├── requirements.txt  
│   └-- venv/
```

¹Step 1 and 2 are covered previously, we will focus on Step 3 serving the landing page.

Roadmap #4 - render html in FastAPI

- setup virtual environment
- copy the provided index.html into roadmap_4 folder.
- create the main.py

```
import fastapi
import uvicorn
from starlette.requests import Request
from starlette.templating import Jinja2Templates

api = fastapi.FastAPI()
templates = Jinja2Templates('.')

@api.get('/')
def index(request: Request):
    return templates.TemplateResponse('index.html', {'request': request})

if __name__ == "__main__":
    uvicorn.run('main:api', port=8000, host='127.0.0.1', reload=True)
```

Roadmap #4 - dependencies

```
from starlette.requests import Request
from starlette.templating import Jinja2Templates
```

- the starlette package comes when installing FastAPI, however
 - ▷ StaticFiles requires aiofiles
- hence, add jinja2 into requirements.txt

```
fastapi
uvicorn
jinja2
aiofiles
```

- finally, run the main.py script.
 - ▷ the reload=True flag allows the app to reload when upon saved changes
 - ▷ the reload=True only work in **import mode** (i.e. main:api string)

Roadmap #4 - jinja2 templates

The landing page will be rendered from `index.html`:

- note `index.html` is not a complete html

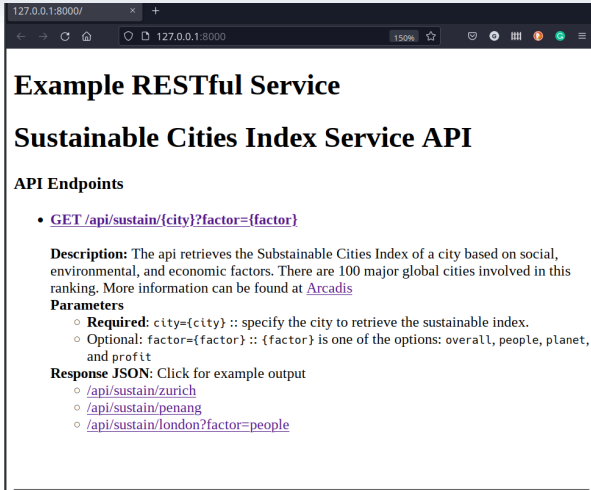
```
{% extends "layout.html" %}
{% block content %}

<h1>Weather Service API</h1>
...

{% endblock %}
```

- `{% extends "layout.html" %}` injects remaining html codes from `layout.html`
- `{% block content %}` in `index.html` and `layout.html` indicate the location of injection
- the separation of two html files allow
 - ▷ `layout.html` - focus on the consistent style of website
 - ▷ `index.html` - focus on the content of the page

Roadmap #4 - rendered landing page



At the moment, the links to the apis are yet implemented.

Roadmap #5

Local computer

- 1 setup virtual environment
- 2 setup FastAPI app
- 3 adding and rendering landing page
- 4 **adding static files**

```
.  
└-- roadmap_5/  
    |-- index.html  
    |-- layout.html  
    |-- main.py  
    |-- requirements.txt  
    |-- static/  
        |-- base.css  
        |-- bootstrap.min.css  
        |-- favicon.ico  
        |-- font-awesome.min.css  
        |-- style.css  
        |-- weather.png  
    |-- venv/
```

FastAPI

Roadmap #5 - render html in FastAPI

- duplicate roadmap_4 to roadmap_5 folder
- copy the provided files in the static folder into roadmap_5 folder.
- modify main.py

```
import fastapi
import uvicorn
from starlette.requests import Request
from starlette.templating import Jinja2Templates
from starlette.staticfiles import StaticFiles

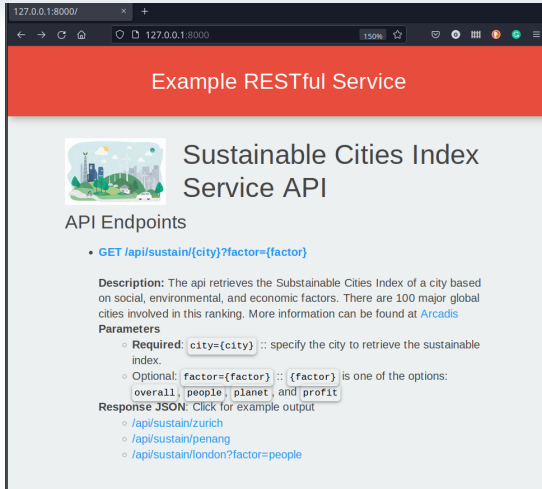
api = fastapi.FastAPI()
templates = Jinja2Templates('.')

api.mount('/static', StaticFiles(directory='static'), name='static')

@api.get('/')
def index(request: Request):
    return templates.TemplateResponse('index.html', {'request': request})

if __name__ == "__main__":
    uvicorn.run('main:api', port=8000, host='127.0.0.1', reload=True)
```

Roadmap #5 - rendered landing page



Now the web page is with styling.

Roadmap #6

Local computer

...

- 4 adding static files
- 5 add new endpoint operation function
- 6 consuming external api service

```
.  
|-- roadmap_6/  
|   |-- static/  
|   |-- venv/  
|   |-- README.md  
|   |-- greencities.csv  
|   |-- index.html  
|   |-- layout.html  
|   |-- main.py  
|   |-- requirements.txt
```

Roadmap #6

Adding new endpoint in main.py:

```
...  
  
@api.get('/api/sustain/{city}')  
def greencity(city: str, factor: Optional[str] = 'overall') -> dict:  
    msg = {'city': city, 'info': None}  
    report = _get_external_report(city, factor)  
    if report:  
        msg = {  
            'country': report.get('country'),  
            'city': city,  
            'factor': int(report.get(factor))  
        }  
    return msg
```

- the endpoint still needs `_get_external_report()` function.

Roadmap #6

Adding the function to communicate with external api.

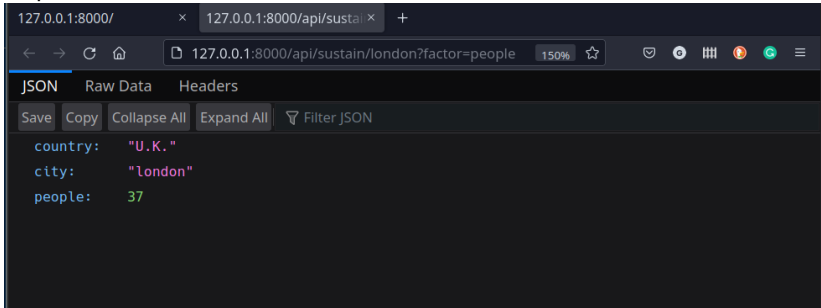
```
import pandas as pd

...
def _get_external_report(city: str, factor: Optional[str]="overall"):
    df = pd.read_csv("greencities.csv")
    df.index = df.city.apply(lambda x: x.lower().replace(' ', '_'))
    resp = {}
    if city in df.index:
        resp = {
            'country': df.loc[city]['Country'],
            'people': df.loc[city]['People'],
            'planet': df.loc[city]['Planet'],
            'profit': df.loc[city]['Profit'],
            'overall': df.loc[city]['Overall']
        }
    return resp
```

- now the links in the landing page are working.

Roadmap #6

click on the last link in the landing page and we will get the JSON response:



- next, we want to make the endpoint async.

Roadmap #7

Local computer

...

- 4 adding static files
- 5 (revisit) making endpoint operation function async
- 6 (revisit) awaiting external api service

```
.  
├-- roadmap_7/  
│  ├── static/  
│  ├── venv/  
│  ├── README.md  
│  ├── greencities.csv  
│  ├── index.html  
│  ├── layout.html  
│  ├── main.py  
└-- requirements.txt
```


Roadmap #7 - async endpoint

- modify greencity() to async

```
...  
  
@api.get('/api/sustain/{city}')
```

```
async def greencity(city: str, factor: Optional[str] = 'overall') -> dict:  
    msg = {'city': city, 'info': None}  
    report = await _get_external_report(city, factor)  
    if report:  
        msg = {  
            'country': report.get('country'),  
            'city': city,  
            'factor': int(report.get(factor))  
        }  
    return msg
```

- but await requires _get_external_report() to be a coroutine.

Roadmap #7 - coroutine client

```
async def _get_external_report(city: str, factor: Optional[str]="overall"):
    df = pd.read_csv("greencities.csv")
    df.index = df.city.apply(lambda x: x.lower().replace(' ', '_'))
    resp = {}
    if city in df.index:
        resp = {
            'country': df.loc[city]['Country'],
            'people': df.loc[city]['People'],
            'planet': df.loc[city]['Planet'],
            'profit': df.loc[city]['Profit'],
            'overall': df.loc[city]['Overall']
        }
    return resp
```

- if the external service is an api, an async client would be needed.
 - ▷ example async http client is httpx

That's it. Now the endpoint is asynchronous. Next is to simplify the code using pydantic.

Roadmap #8

Although the api is now working, it is best practice to convert the input parameters into a pydantic class.

- pydantic allows auto data validation and conversion
- less errors on incompatible parameters types
- shorter code, hence less bugs

In this part, we will modify `main.py` to use pydantic model.

Roadmap #8 - pydantic model

```
from pydantic import BaseModel
```

```
...
```

```
class Query(BaseModel):  
    city: str  
    factor: Optional[str] = 'overall'
```

```
@api.get('/api/sustain/{city}')  
async def greencity(query: Query=fastapi.Depends()) -> dict:  
    msg = {'city': query.city, 'info': None}  
    report = await _get_external_report(query.city, query.factor)  
    if report:  
        msg = {  
            'country': report.get('country'),  
            'city': query.city,  
            'query.factor': int(report.get(query.factor))  
        }  
    return msg
```

Next, we will re-organise the code.

Roadmap #9

In our FastAPI app there is only one landing page and one api. In most cases, you will have more than just one api or pages.

- Organise the FastAPI app according to their functionalities
- Use `fastapi.APIRouter` to avoid circular dependencies

Suggested folder structures:

```
.  
├── roadmap_9/  
│   ├── api/  
│   ├── infrastructure/  
│   ├── models/  
│   ├── static/  
│   ├── templates/  
│   ├── venv/  
│   ├── views/  
│   ├── main.py  
│   ├── requirements.txt  
│   └── settings.json
```

Roadmap #9 - templates/

Create the templates folder to store the html pages

- move index.html and layout.html into templates/ folder
- edit main.py to tell jinja2 to get the templates from here.

```
...  
  
templates = Jinja2Templates('templates')  
  
...
```

Roadmap #9 - models/

models/ stores all the pydantic models created for the project

- create a file called query.py
- copy the definition of Query class to this file
- delete the definition of Query class in main.py

Your models/query.py should look like this:

```
from typing import Optional
from pydantic import BaseModel

class Query(BaseModel):
    city: str
    nextday: Optional[int]=0
    units: Optional[str]='metric'
```

Roadmap #9 - infrastructure/

infrastructure/ keeps all external services used by the project. In our case, `_get_external_report()`

- create a file called `greencity.py`, and move 'greencities.csv' to infrastructure/
- move `_get_external_report()` from `main.py` to the file
- remember to import the necessary libraries

```
from typing import Optional
import pandas as pd

async def _get_external_report(city: str, factor: Optional[str] = "overall"):
    df = pd.read_csv("greencities.csv")
    df.index = df.city.apply(lambda x: x.lower().replace(' ', '_'))
    resp = {}
    if city in df.index:
        resp = {
            'country': df.loc[city]['Country'],
            'people': df.loc[city]['People'],
            'planet': df.loc[city]['Planet'],
            'profit': df.loc[city]['Profit'],
            'overall': df.loc[city]['Overall']
        }
    return resp
```


Roadmap #9 - api/

api/ stores all apis that the project offers. In our case, greencity().

- create a file called greencity_api.py
- move the definition of greencity() from main.py into the file
- import _get_external_report() from greencity.py

```
import fastapi
from starlette.responses import JSONResponse
from infrastructure.greencity import _get_external_report
from models.query import Query

@api.get('/api/sustain/{city}')
async def greencity(query: Query = fastapi.Depends()) -> JSONResponse:
    msg = {'city': query.city, 'info': None}
    report = await _get_external_report(query.city, query.factor)
    if report:
        msg = {
            'country': report.get('country'),
            'city': query.city,
            query.factor: int(report.get(query.factor))
        }
    return JSONResponse(msg, status_code=200)
```

Roadmap #9 - views/

If we think api is consumed by other computer programs, then views are consumed by web browsers.

- create a file called `index.py`
- move `index()` from `main.py` to this file

```
from starlette.requests import Request
from starlette.templating import Jinja2Templates

templates = Jinja2Templates('templates')

@api.get('/')
def index(request: Request):
    return templates.TemplateResponse('index.html', {'request': request})
```

Roadmap #9 - `main.py`

After refactoring, now `main.py` is quite empty. However, when running the `main.py`, we see that the browser complains “Not Found”.

- “Not Found” because there is no endpoints in `main.py`
- we can import the endpoints in `api.greencity_api` and `views.index`, but this still will not work
 - ▷ in `api.greencity_api` and `views.index` need `api` defined in `main.py`
 - ▷ in `main.py` we need the endpoints from the two files
 - ▷ we got a **circular referencing** problem

To overcome this, we use the `FastAPI.APIRouter` to link up the endpoints.

Roadmap #9 - adding APIRouter

In `api/greencity_api.py`:

```
from fastapi import Depends, APIRouter

router = APIRouter()
@router.get('/api/sustain/{city}')
async def greencity(query: Query=Depends()) -> JSONResponse:

...
```

In `views/index.py`:

```
from fastapi import APIRouter

router = APIRouter()
@router.get('/')
def index(request: Request):

...
```

Roadmap #9 - main.py

Now, import the routers into the main.py:

```
import fastapi
import uvicorn
from starlette.staticfiles import StaticFiles

from api import greencity_api
from views import index

api = fastapi.FastAPI()
api.mount('/static', StaticFiles(directory='static'), name='static')
api.include_router(greencity_api.router)
api.include_router(index.router)

if __name__ == "__main__":
    uvicorn.run('main:api', port=8000, host='127.0.0.1', reload=True)
```

Now the FastAPI app is up.

Roadmap #9 - better structures

When the project get more complex, the number of routers and other configurations will grow. Better to put these in a function.

```
import fastapi
import uvicorn
from starlette.staticfiles import StaticFiles

from api import greencity_api
from views import index

api = fastapi.FastAPI()

def configure():
    api.mount('/static', StaticFiles(directory='static'), name='static')
    api.include_router(greencity_api.router)
    api.include_router(index.router)

if __name__ == "__main__":
    configure()
    uvicorn.run('main:api', port=8000, host='127.0.0.1', reload=True)
else:
    configure()
```

Roadmap #9 - keeping secret

Sometimes, the external services used in the project require authentication key. This api key could be store in a json file.

- create a `settings.json` to store the real api key
- you don't want to push this `settings.json` to git repo
 - ▷ add it to `.gitignore`
 - ▷ visit www.shhgit.com to see how many people carelessly “publish” their secret api key to the world
- create a `settings_dummy.json` as dummy file which reminds what to put in the `settings.json`.

Since our project does not requires any api key for external service, we will just use a dummy api key:

```
{  
  "api_key": "Put your api key here",  
  "warning": "do not upload this to your public repo",  
  "setting": "dev"  
}
```

Roadmap #9 - loading the setting

The settings in the json file can be imported using the json library.

- in main.py add the json library
- load the settings.json file

```
import json

...

def load_env():
    with open('settings.json') as f:
        env = json.load(f)
    return env

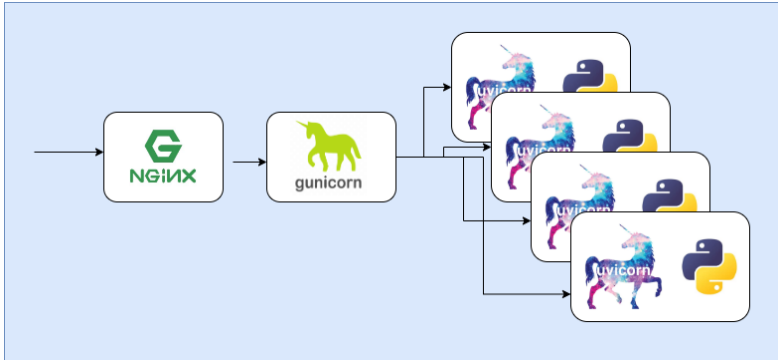
def configure():
    api.mount('/static', StaticFiles(directory='static'), name='static')
    api.include_router(greencity_api.router)
    api.include_router(index.router)
    env = load_env()
    print(env['setting'])
```


Section 6

Deployment

Deployment to a remote server

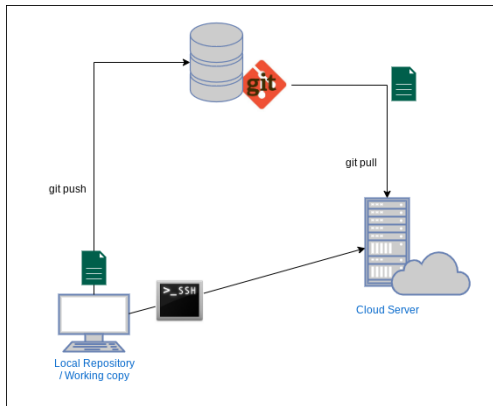
Overall deployment layout:



Roadmap #10

Cloud server

- configure server manually
- clone git repo and setup virtual environment
- setup nginx
- setup gunicorn



Roadmap #10 - configure server

Always update the server first:

- ssh into the remote server as root
- update server

```
apt update  
apt upgrade -y
```

Secure the server:

- Setup ssh-key login only by editing `/etc/ssh/sshd_config`

```
PasswordAuthentication no  
UsePAM no
```

- reboot server to apply the new kernel

Roadmap #10 - configure server

Install python and build dependencies

```
apt install -y -q python3-pip python3-dev python3-venv build-essential
```

Setup firewall and fail2ban

- allow port 22 for ssh
- allow port 80 for http request
- allow port 443 is needed for https and SSL traffic
- fail2ban will block IP with too many failed login attempts

```
ufw allow 22  
ufw allow 80  
ufw allow 443  
ufw enable  
apt install -y fail2ban
```

Roadmap #10 - configure server

Prepare the app folder

```
mkdir /apps  
chmod 777 /apps  
mkdir -p /apps/logs/greencity_api
```

Add a dummy user called apiuser to run the app later

- use -M not to create user home directory
- use -L to lock the user from login

```
useradd -M apiuser  
usermod -L apiuser
```

Set the permission of /apps folder to apiuser

```
apt install acl -y  
setfacl -m u:apiuser:rwX /apps/logs/greencity_api
```

Roadmap #10 - setup virtual env

Next, setup a virtual environment and clone the project

- setup virtual env

```
cd /apps
python3 -m venv venv
. /apps/venv/bin/activate
pip install --upgrade pip setuptools wheel
```

- clone repo and copy settings.json

```
git clone https://github.com/yongkheng/greencity app_repo
cp /apps/app_repo/settings_dummy.json /apps/app_repo/settings.json
```

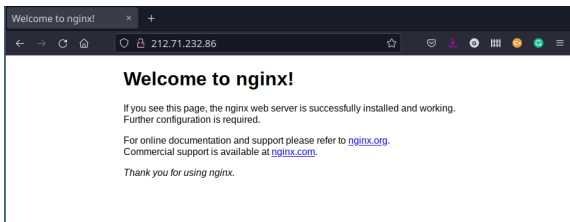
- install dependencies

```
pip install /apps/app_repo/requirements.txt
```

Roadmap #10 - setup nginx

First, install nginx

```
apt install -y nginx
```



- this is the default page at `/etc/nginx/sites-enabled/default`
- remove the default page, add in `greencity.nginx`, and restart

```
rm /etc/nginx/sites-enabled/default
cp /apps/app_repo/server/nginx/greencity.nginx /etc/nginx/sites-enabled/
update-rc.d nginx enable
service nginx restart
```


Roadmap #10 - setup gunicorn

Next, install gunicorn and async uvloop

```
pip install --upgrade gunicorn uvloop
```

Finally, add the service to systemd to ensure the api would start when system reboot.

```
cp /apps/app_repo/server/systemd/greencity.service /etc/systemd/system/  
systemctl start greencity  
systemctl enable greencity
```

Roadmap #10 - wrap up

To complete the entire process:

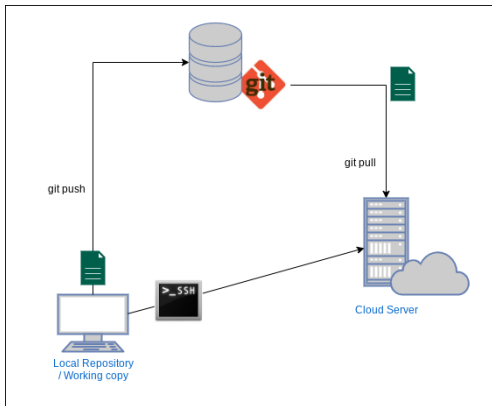
- remember to set your api-key in the `settings.json`
- you might want to
 - ▷ register a domain name for your api
 - ▷ contact your IT department to enable https
 - ▷ if you are doing it yourself, you can use certbot

```
add-apt-repository ppa:certbot/certbot
apt install python3-certbot-nginx
certbot --nginx -d your_api_domain_name.com
```

Roadmap #11

Cloud server

- update server manually
- automate setup with shell script



Roadmap #11 - configure server

Always update the server first:

- ssh into the remote server as root
- update server

```
apt update  
apt upgrade -y
```

Secure the server:

- Setup ssh-key login only by editing `/etc/ssh/sshd_config`

```
PasswordAuthentication no  
UsePAM no
```

- reboot server to apply the new kernel

Roadmap #11 - automate setup

Copy automation script to the remote server with scp:

```
scp server_setup.sh root@<ip_address>:/root
```

Ssh into the remote server and run the shell script:

```
sh server_setup.sh
```

Remember to set the api-key in settings.json.

Roadmap #12 - Dockerize FastAPI



Deployment of a microservice can also be done on a container. The advantages of containerize deployment are:

- Easier management - pinned version to avoid broken service due to upgrade
- Improved security - isolation of services from host system
- Scalability - fast to spin up the uniform services
- Flexibility - work on virtualized infrastructure or bare metal servers
- Lightweight - compared to virtual machines
- Integration - allow CI/CD pipelines

Roadmap #12 - Dockerize FastAPI

Local or remote computer

- install docker
- create Dockerfile
- build docker image
- run and stop docker container

Roadmap #12 - Dockerize FastAPI

The process of creating docker image can be done on a local computer. To avoid the complication of setting up docker on a local machine, hard disk storage issue, and take advantage of the network speed, we will build the docker image from the server.

- login to the server as root
- shut down the app and `nginx`. (we will run from the app from Docker)

```
systemctl stop greencity  
systemctl stop nginx
```


Roadmap #12 - Dockerize FastAPI

Install docker on the remote server:

```
apt install docker.io
```

At the server, the source code of the api app is at /apps/app_repo.

- change directory to /apps
- create a Dockerfile (next slide)

Tips of building docker

- every command in the Dockerfile will be built as a layer
- put the commands that less modify to the front
- putting commands that have frequent modifications at the back will save the build time of the docker image

Roadmap #12 - Dockerize FastAPI

```
FROM ubuntu:latest

RUN apt update && apt upgrade -y
RUN apt install -y -q build-essential python3-pip python3-dev
RUN pip3 install -U pip setuptools wheel
RUN pip3 install gunicorn uvloop httptools

RUN addgroup --system app && adduser --system --group app
RUN mkdir /code
WORKDIR /code

COPY ./app_repo/requirements.txt /code/requirements.txt
RUN pip3 install --no-cache-dir --upgrade -r /code/requirements.txt

COPY ./app_repo /code/app

WORKDIR /code/app
CMD gunicorn -w 4 -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:80 main:api
```

Roadmap #12 - Dockerize FastAPI

Then, build the docker image and run.

```
docker build -t myimage .  
docker run -d --name mycontainer -p 80:80 myimage
```

To stop the docker container:

```
docker stop mycontainer
```

To restart the stopped container:

```
docker start mycontainer
```

To delete the container

```
docker stop mycontainer  
docker rm mycontainer
```

Roadmap #13 - Deploy to Heroku



Heroku is a cloud platform for delivering web apps and hosting since 2007.

- Own by Salesforce
- Convenient web app deployment testing using free Dyno
- Starting Nov 2022, free Dynos will no longer available

To deploy on Heroku, you will need a Heroku account.

- create a Heroku account at <https://signup.heroku.com>
- then login to the dashboard

Roadmap #13 - Deploy to Heroku

Local or remote computer

- create Heroku account
- install `heroku-cli`
- create Heroku app
- create and modify `Dockerfile`
- build docker image
- push docker image to Heroku docker image registry
- run and release the Heroku container

Roadmap #13 - Deploy to Heroku

We can create Heroku Dyno (mini virtual machine) from Heroku website. Alternatively, a more convenient way is using its command line interface:

- see <https://devcenter.heroku.com/articles/heroku-cli>

```
curl https://cli-assets.heroku.com/install.sh | sh
```

- Once the `heroku-cli` is installed, you will need to authenticate your account by `heroku login`
- On a local machine, usually a browser will be launched and presenting the heroku login page
- On a server, you can login using the headless login

```
heroku login -i
```

```
heroku: Enter your login credentials
```

```
Email: <your email address>
```

```
Password: <heroku API token>
```

```
Logged in as <your email address>
```

Roadmap #13 - Deploy to Heroku

- After successful login, you can create a new Heroku app

```
heroku create
Creating app... done,   secure-hamlet-95506
https://secure-hamlet-95506.herokuapp.com/ | https://git.heroku.com/secure-hamlet-95506.git
```

- Note that the name of the Dyno is generated randomly. In this example, the Dyno name is `secure-hamlet-95506`, yours will be different.
- Remember for the following example, you should replace `secure-hamlet-95506` with your own Dyno name.

Roadmap #13 - Deploy to Heroku

Deployment of the web app to Heroku is via docker. We will need to modify the Dockerfile for the use of Heroku app.

```
...  
  
COPY ./app_repo /code/app  
  
WORKDIR /code/app  
  
EXPOSE $PORT  
  
CMD gunicorn -w 4 -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:$PORT main:api
```

- note that the variable \$PORT is the magic port number that assigned by Heroku
- the value of \$PORT changes on every run of the docker container.

Roadmap #13 - Deploy to Heroku

- Now, login to Heroku docker container registry:

```
heroku container:login
```

- then, build the docker container

```
docker build -f Dockerfile -t registry.heroku.com/secure-hamlet-95506/web .
```

- Here, `registry.heroku.com/secure-hamlet-95506/web` is the docker image name.
- the docker image is now built in the remote server and later will be pushed to the Heroku register.
- If the docker image is built properly, the image can be pull down to run in the Heroku app.

Roadmap #13 - Deploy to Heroku

- test run the Heroku docker locally

```
docker run -d --name heroku-app -e PORT=8765 -p 8080:8765 \
registry.heroku.com/secure-hamlet-95506/web:latest
```

- check the docker run status

```
docker container ls -a

curl localhost:8080
```

- wind down the docker

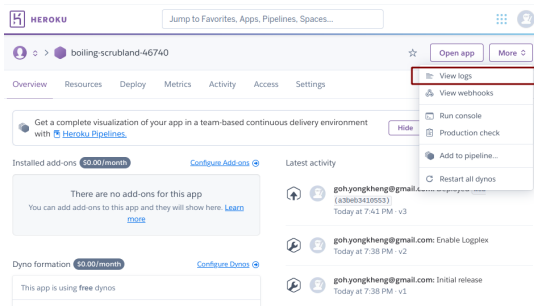
```
docker container stop heroku-app
docker container rm heroku-app
```

Roadmap #13 - Deploy to Heroku

If the docker container run without problem, then we can push the docker image to the Heroku registry, then release the image

```
docker push registry.heroku.com/secure-hamlet-95506/web:latest
heroku container:release web --app secure-hamlet-95506
```

- for debug, check the dashboard of the heroku app log.



Roadmap #13 - Deploy to Heroku

To spin up the docker on another Dyno can use the same process, but but much faster

```
heroku create
docker build -f Dockerfile -t registry.heroku.com/<APP_NAME>/web .
docker push registry.heroku.com/<APP_NAME>/web:latest
heroku container:release web --app <APP_NAME>
```

To remove the container

```
heroku container:rm web --app <APP_NAME>
```

Section 7

FastAPI with Database

Roadmap #14 - FastAPI with sqlite3

Next is to create a web app with database: a URL shortener.

Local or remote computer

- create project folder and virtual environment
- install dependencies
- create external file for environmental variables
- create sqlite3 database and models
- create FastAPI app
- adding front end



Roadmap #14 - FastAPI with sqlite3

First setup the project folder and virtual environment:

```
mkdir roadmap_14
cd roadmap_14
mkdir shortener_app
python -m venv venv
source venv/bin/activate
```

Then create a requirements.txt in the shortener_app folder:

```
fastapi
uvicorn
sqlalchemy
python-dotenv
validators
jinja2
python-multipart
```

and install:

```
pip install -r shortener_app/requirements.txt
```

Roadmap #14 - FastAPI with sqlite3



In this project we are using `sqlite` as the database:

- advantage of using `sqlite`:
 - ▷ is a single file database
 - ▷ cross platform
 - ▷ suitable for small to medium size applications
 - ▷ stable and mature opensource project

Roadmap #14 - FastAPI with sqlite3

During the development of an app, it make sense to split the development database and production database. A flexible way to achieve the separation is via environment variables.

- create a `.env` file to store environment variables

```
ENV_NAME="Development"  
BASE_URL="http://127.0.0.1:8000"  
DB_URL="sqlite:///./shortener.db"
```

- we can modify `.env` or create a separate file for production environment
 - ▷ the domain name and the location to the database will be different from development environment
 - ▷ this `.env` should be ignored from uploading to project git repo

Roadmap #14 - FastAPI with sqlite3

Then, we want to load the environment variables to the FastAPI project.

- create a file `shortener_app/config.py`

```
# shortener_app/config.py
from functools import lru_cache
from pydantic import BaseSettings

class Settings(BaseSettings):
    env_name: str
    base_url: str
    db_url: str

    class Config:
        env_file = ".env"

@lru_cache
def get_settings() -> Settings:
    settings = Settings()
    print(f"Loading settings for: {settings.env_name}")
    return settings
```

- `get_settings()` will be called many times in the app
- caching `get_settings()` for improved performance.

Roadmap #14 - FastAPI with sqlite3

Next, is to setup the data exchange models for the web app and the database.

- create `shortener_app/database.py`

```
# shortener_app/database.py

from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

from .config import get_settings

engine = create_engine(get_settings().db_url,
                      connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

Roadmap #14 - FastAPI with sqlite3

- create `shortener_app/models.py`

```
# shortener_app/models.py
from sqlalchemy import Column, Integer, String
from .database import Base

class URL(Base):
    __tablename__ = "urls"
    id = Column(Integer, primary_key=True)
    key = Column(String, unique=True, index=True)
    target_url = Column(String, index=True)
```

- eventually, the sql table will be generated in the file specified by `get_settings().db_url`
- `id`, `key`, `'target_url'` are the fields in the `urls` table

Roadmap #14 - FastAPI with sqlite3

Finally, we are to connect the database to the main program:

- we will have a rather long `main.py`:

```
# shortener_app/main.py

import secrets
import validators
import string
from fastapi import Depends, FastAPI, HTTPException, Request
from fastapi.responses import RedirectResponse
from starlette.templating import Jinja2Templates
from sqlalchemy.orm import Session
from . import models
from .database import SessionLocal, engine

app = FastAPI()
models.Base.metadata.create_all(bind=engine)
templates = Jinja2Templates("./shortener_app")
```

- we add an empty file `__init__.py` to declare that `shortener_app` folder is a package

⚡ FastAPI

Roadmap #14 - FastAPI with sqlite3

- some helper functions in main.py

```
def raise_bad_request(message):  
    raise HTTPException(status_code=400, detail=message)  
  
def raise_not_found(request):  
    message = f"URL '{request.url}' doesn't exist"  
    raise HTTPException(status_code=404, detail=message)  
  
def get_db():  
    db = SessionLocal()  
    try:  
        yield db  
    finally:  
        db.close()
```

- the web root

```
@app.get("/")  
def read_root(request: Request):  
    return templates.TemplateResponse("index.html", {"request": request})
```

Roadmap #14 - FastAPI with sqlite3

- POST endpoint to submit URL to be shortened

```
@app.post("/url")
async def create_url(request: Request, db: Session = Depends(get_db)):
    form = await request.form()
    form = dict(form)
    url = form["url"]
    if not validators.url(url):
        raise_bad_request(message="Your provided URL is not valid")

    chars = string.ascii_uppercase + string.digits
    key = "".join(secrets.choice(chars) for _ in range(5))
    db_url = models.URL(target_url=url, key=key)
    db.add(db_url)
    db.commit()
    db.refresh(db_url)

    url_short = f"http://127.0.0.1:8000/{key}"
    return templates.TemplateResponse("link.html",
    {
        "request": request,
        "url": url,
        "url_short": url_short,
    },
    )
```

Roadmap #14 - FastAPI with sqlite3

- lastly, forwarding the short URL to the target URL

```
@app.get("/{url_key}")
def forward_to_target_url(
    url_key: str, request: Request, db: Session = Depends(get_db)
):
    db_url = db.query(models.URL).filter(models.URL.key == url_key).first()
    if db_url:
        return RedirectResponse(db_url.target_url)
    else:
        raise_not_found(request)
```


Roadmap #14 - FastAPI with sqlite3

We still need to provide the two html files for the front end:

- index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple URL Shortener</title>
  </head>
  <body>
    <form action="/url" method="post">
      <input type="url" name="url" placeholder="https://www.example.com" />
      <input type="submit" value="Submit" />
    </form>

  </body>
</html>
```



Roadmap #14 - FastAPI with sqlite3

- link.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple URL Shortener</title>
  </head>
  <body>

    URL: {{url}}
    <br>
    shortened URL: <a href="{{url_short}}">{{url_short}}</a>
  </body>
</html>
```



Section 8

Summary

Summary of the different Roadmaps

No.	Summary	No.	Summary
1.	minimal FastAPI	8.	pydantic data models
2.	setup git and ssh	9.	refactor and APIRouter
3.	end points and data	10.	deploy via nginx and gunicorn
4.	rendering html pages	11.	automate server deployment
5.	adding static files	12.	deploy via docker
6.	end points and external api	13.	deploy to Heroku
7.	async end points	14.	FastAPI with database

Thank you

