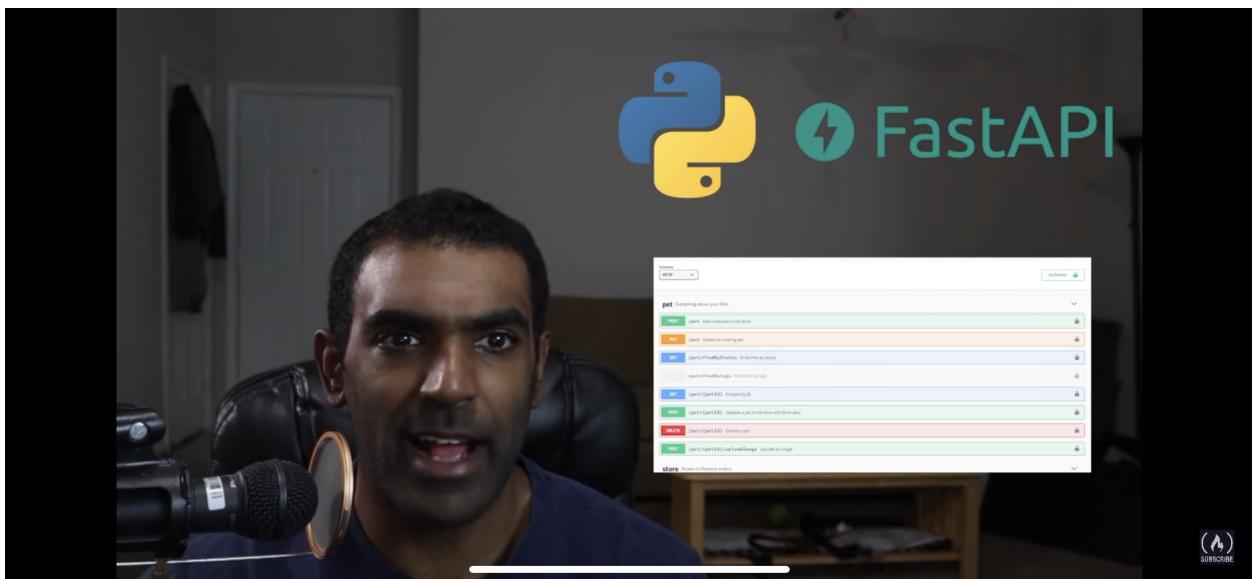




# Python API Development Course (Part 1)

<https://youtu.be/0sOvCWFmrtA>

FastAPI



## Development Environment Setup



- Install Python(version 3)
- Install and Configure VSCode
- Setup Virtual Environment



### Setting Virtual environments

## Python Virtual Environments



A screenshot of the Visual Studio Code interface. The Explorer sidebar shows a project named 'FASTAPI' with a 'venv' folder containing 'main.py'. The main editor area has 'main.py' open with the number '1' at the top. Below the editor is a terminal window titled 'TERMINAL'. The terminal output shows:

```
Microsoft Windows [Version 10.0.19043.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Users\sanjeev\Documents\fastapi>py -3 -m venv venv
C:\Users\sanjeev\Documents\fastapi>
```

The status bar at the bottom indicates 'Line 1, Col 1' and 'Spaces: 4 - UTF-8 - CR LF - Python'.

Enter v env

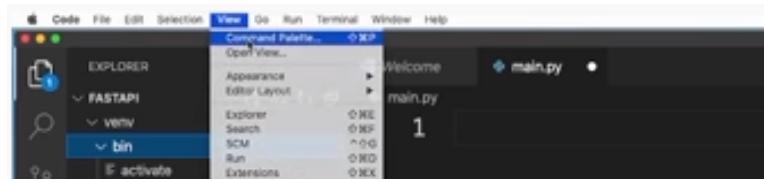
A screenshot of the Visual Studio Code interface. The Explorer sidebar shows a project named 'FASTAPI' with a 'venv' folder containing 'activate', 'activate.bat', 'deactivate.bat', and 'main.py'. The main editor area has 'main.py' open with the number '1' at the top. Below the editor is a terminal window titled 'TERMINAL'. The terminal output shows:

```
Microsoft Windows [Version 10.0.19043.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Users\sanjeev\Documents\fastapi>py -3 -m venv venv
C:\Users\sanjeev\Documents\fastapi>
```

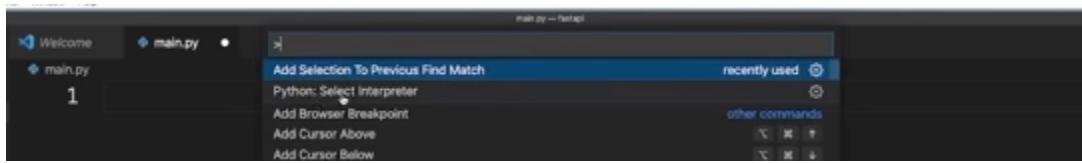
The status bar at the bottom indicates 'Line 1, Col 1' and 'Spaces: 4 - UTF-8 - CR LF - Python'.

## Selecting an interpreter

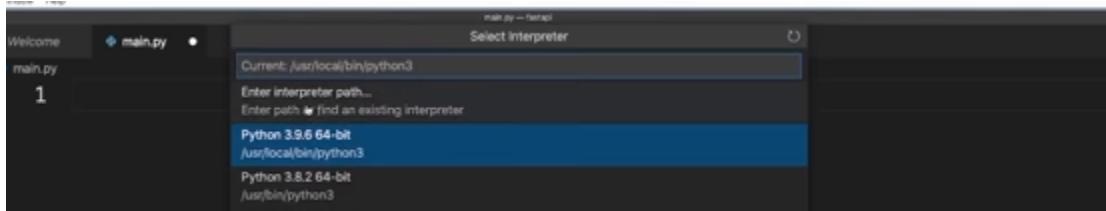


view > command palette

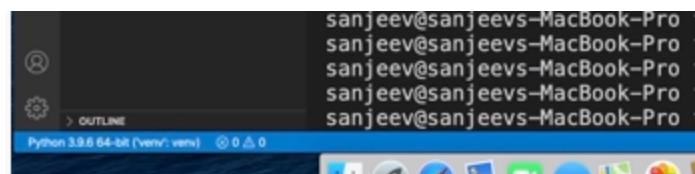
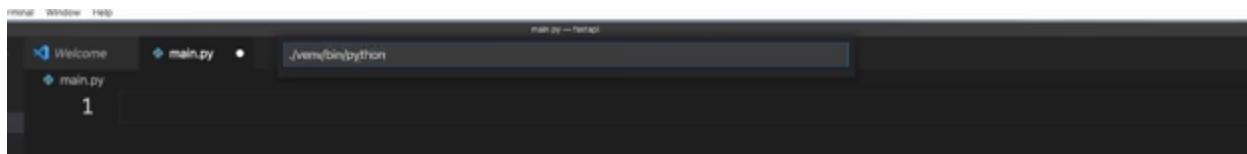
- select python interpreter



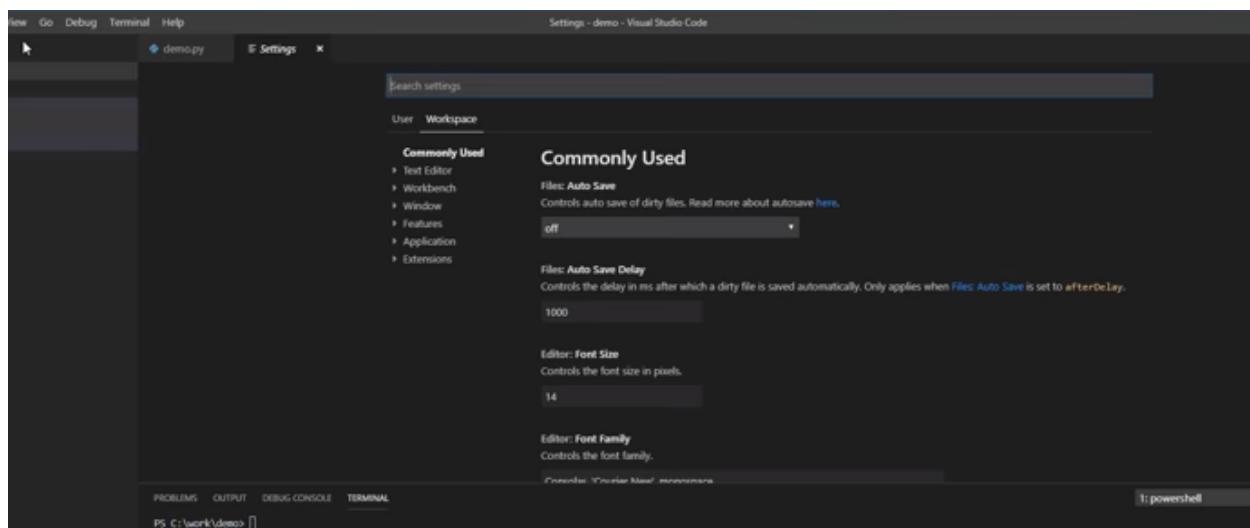
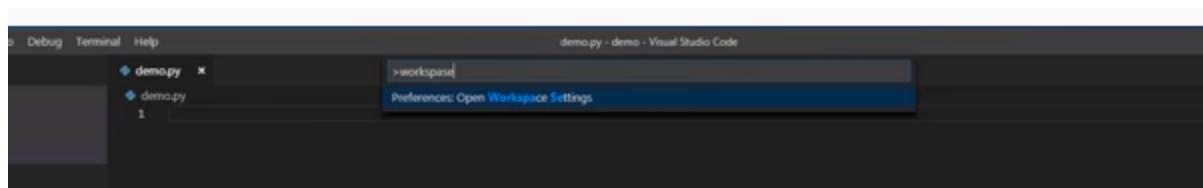
- currently using global one



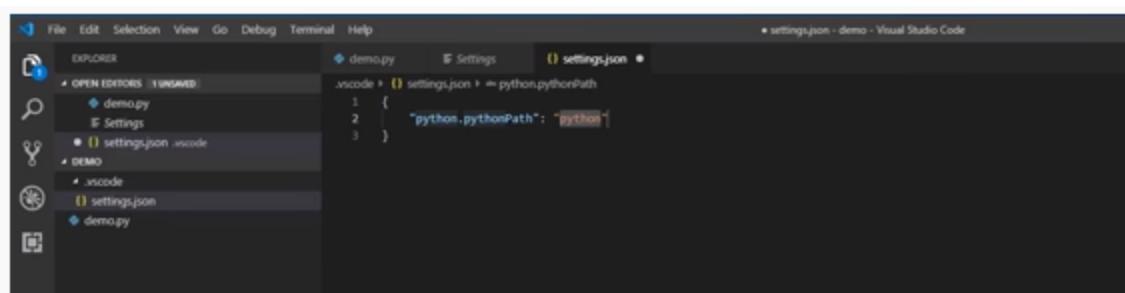
- pass in our directory to our own env.



## Alternate method

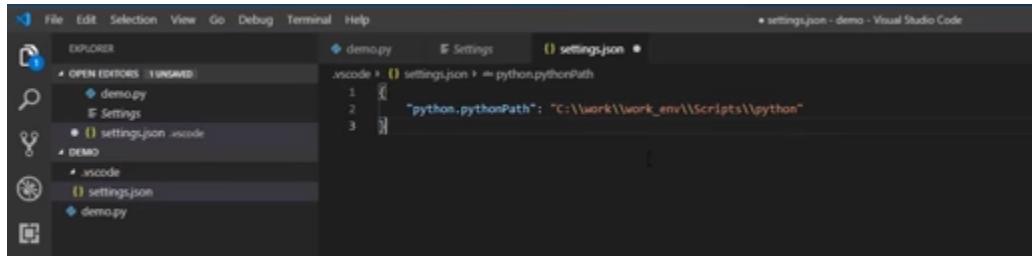


open json view on right hand toggle

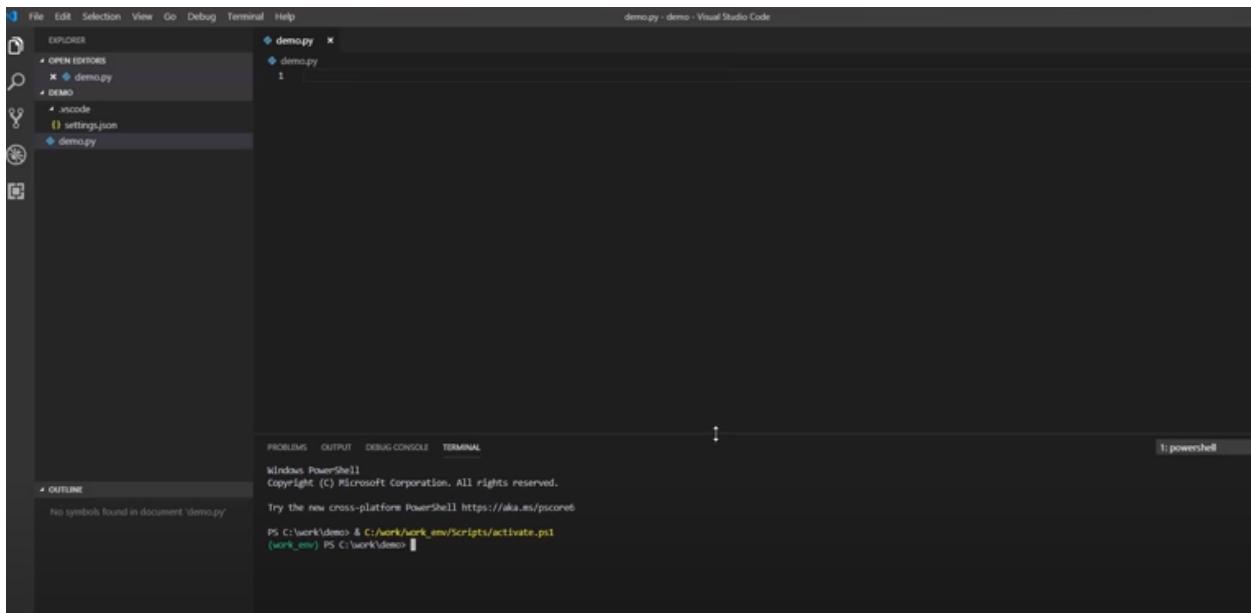


add path to python .exe

- remember to add \\



work env should now be entered



Activate env by navigating to the **Scripts** folder of the activate.bat file

- (.venv) C:/ ... will come up

## Importing Fast API

Look at documentation

The screenshot shows a web browser displaying the FastAPI User Guide - Intro page. The left sidebar contains a navigation tree with categories like FastAPI, Languages, Features, Python Types Intro, and Tutorial - User Guide. The main content area is titled "Install FastAPI" and contains instructions for installing the package via pip. It includes a terminal window showing the command "pip install fastapi[all]" being run, with a progress bar at 100% completion. A note section provides additional deployment information.

The screenshot shows a Visual Studio Code interface. The Explorer sidebar shows a project structure with files like main.py, pyenv.cfg, and a .gitignore file. The main editor area has a Python script with the following code:

```
from fastapi import FastAPI
app = FastAPI()
```

The Terminal tab shows the output of a pip install command:

```
PyYAML==5.4.1
requests==2.26.0
Rxe==1.6.1
six==1.16.0
starlette==0.14.2
typing-extensions==3.10.0.0
ujson==4.0.2
urllib3==1.26.6
uvicorn==0.13.4
watchgod==0.7
websockets==8.1
```

Create an object from the imported FastAPI class

The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there is a folder named 'fastapi' containing a file 'main.py'. The main.py file contains the following code:

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
async def root():
    return {"message": "Hello World"}
```

In the Terminal tab, the user runs the command `python -m pip install -U autopep8`. The output shows the installation of autopep8 version 1.5.7, pycodestyle version 2.7.0, and tomli version 0.10.2. A warning is displayed about the pip version being 21.1.1, while 21.2.4 is available.

```
>c:/Users/sanje/Documents/Courses/fastapi/venv/Scripts/python.exe -m pip install -U autopep8
Collecting autopep8
  Using cached autopep8-1.5.7-py2.py3-none-any.whl (45 kB)
Collecting pycodestyle
  Using cached pycodestyle-2.7.0-py2.py3-none-any.whl (41 kB)
Collecting tomli
  Using cached tomli-0.10.2-py2.py3-none-any.whl (16 kB)
Collecting pycodestyle<=2.7.0
  Using cached pycodestyle-2.7.0-py3-none-any.whl (41 kB)
Successfully collected autopep8-1.5.7, pycodestyle, autopep8, tomli-0.10.2
WARNING: You are using pip version 21.1.1; however, version 21.2.4 is available.
You should consider upgrading via the 'c:/Users/sanje/Documents/Courses/fastapi/venv/Scripts/python.exe -m pip install --upgrade pip' command.
```

Uvicorn is installed already

Call uvicorn main:app

The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there is a folder named 'fastapi' containing a file 'main.py'. The main.py file contains the same code as before:

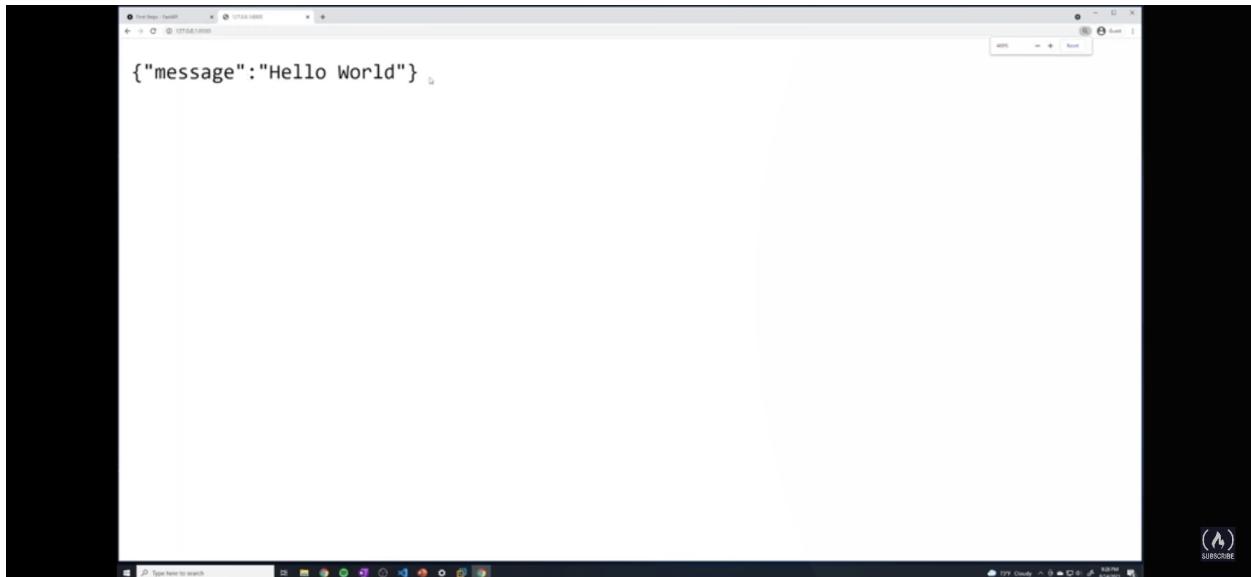
```
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
async def root():
    return {"message": "Hello World"}
```

In the Terminal tab, the user runs the command `uvicorn main:app`. The output shows the server starting on port 8000.

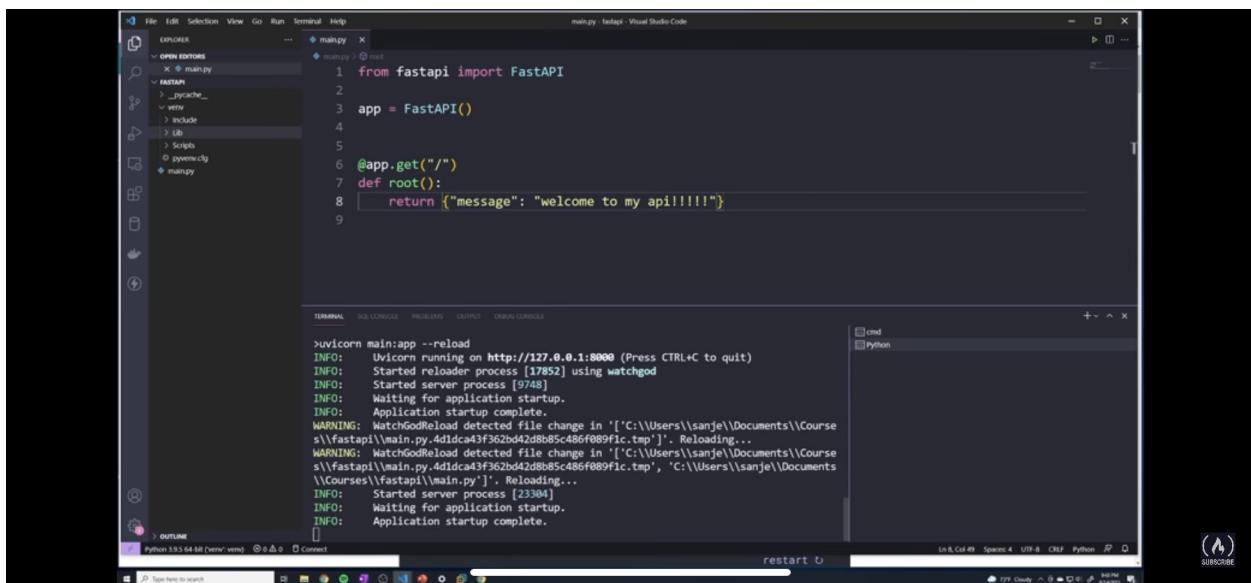
```
WARNING: You are using pip version 21.1.1; however, version 21.2.4 is available.
You should consider upgrading via the 'c:/Users/sanje/Documents/Courses/fastapi/venv/Scripts/python.exe -m pip install --upgrade pip' command.

(venv) C:\Users\sanje\Documents\Courses\fastapi>uvicorn main:app
INFO:     Started server process [5480]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Program will start and be accessible on that port



Use —reload flag to auto update server



## Overview of http methods

## Path Operation

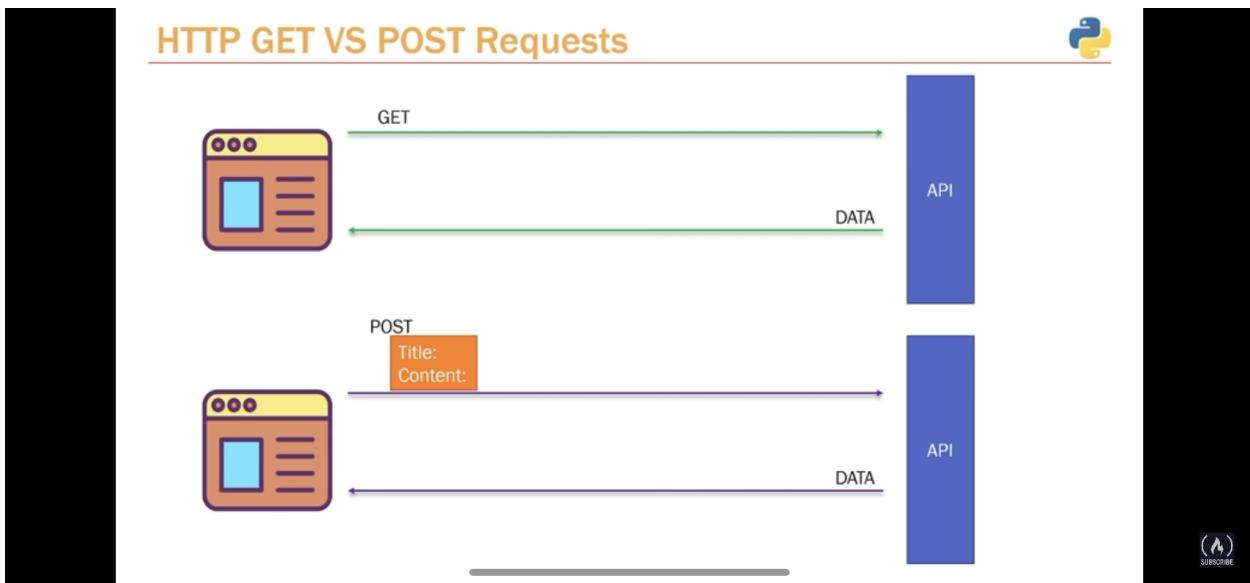
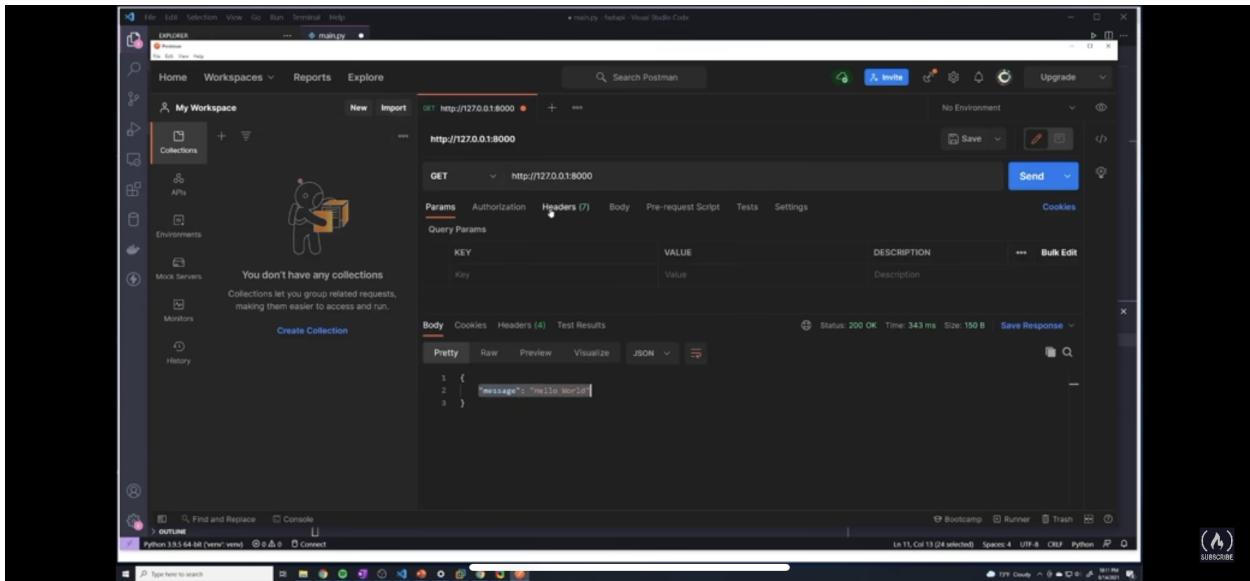


## Adding different methods

The screenshot shows the Visual Studio Code interface with the following details:

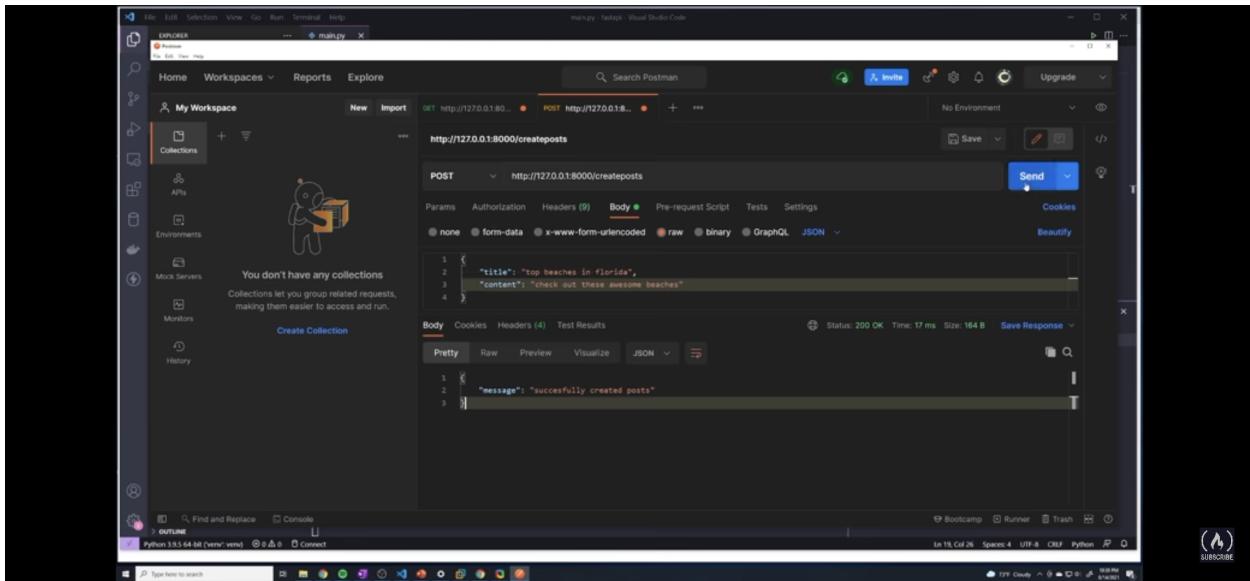
- File Structure (Explorer):** The project structure includes a main.py file under the FASTAPI folder.
- Code Editor (main.py):** The code defines a FastAPI application with a root endpoint returning "Hello World" and a posts endpoint returning a sample post.
- Terminal Output:** The server has started successfully, and a GET request to the root endpoint returns the expected response.
- Output Panel:** Shows logs from the server process, including file change detection by WatchdogReload.

## Use Postman to test API



Whole point of POST is to send data to our API

- we can do this in the body
- Send some data to the body using postman



We need to get the payload somehow

- the function can take an arg if a dict and take in the body sent to it
- And storing It in payload - can then print it

```

14
15     @app.get("/posts")
16     def get_posts():
17         return {"data": "This is your posts"}
18
19
20     @app.post("/createposts")
21     def create_posts(payload: dict = Body(...)):
22         print(payload)
23         return {"message": "succesfully created posts"}
24

```

```

INFO: 127.0.0.1:55022 - "POST /createposts HTTP/1.1" 200 OK
INFO: 127.0.0.1:55026 - "POST /createposts HTTP/1.1" 200 OK
INFO: 127.0.0.1:55026 - "GET /posts HTTP/1.1" 200 OK
INFO: 127.0.0.1:55049 - "POST /createposts HTTP/1.1" 200 OK
WARNING: WatchGodReload detected file change in '[C:\Users\sanje\Documents\Course s\fastapi\main.py.4d1dcfa3f362bd42d8885c486f089fc.tmp]'. Reloading...
WARNING: WatchGodReload detected file change in '[C:\Users\sanje\Documents\Course s\fastapi\main.py]'. Reloading...
INFO: Started server process [25872]
INFO: Waiting for application startup.
INFO: Application startup complete.
{'title': 'top beaches in florida', 'content': 'check out these awesome beaches'}
INFO: 127.0.0.1:53290 - "POST /createposts HTTP/1.1" 200 OK

```

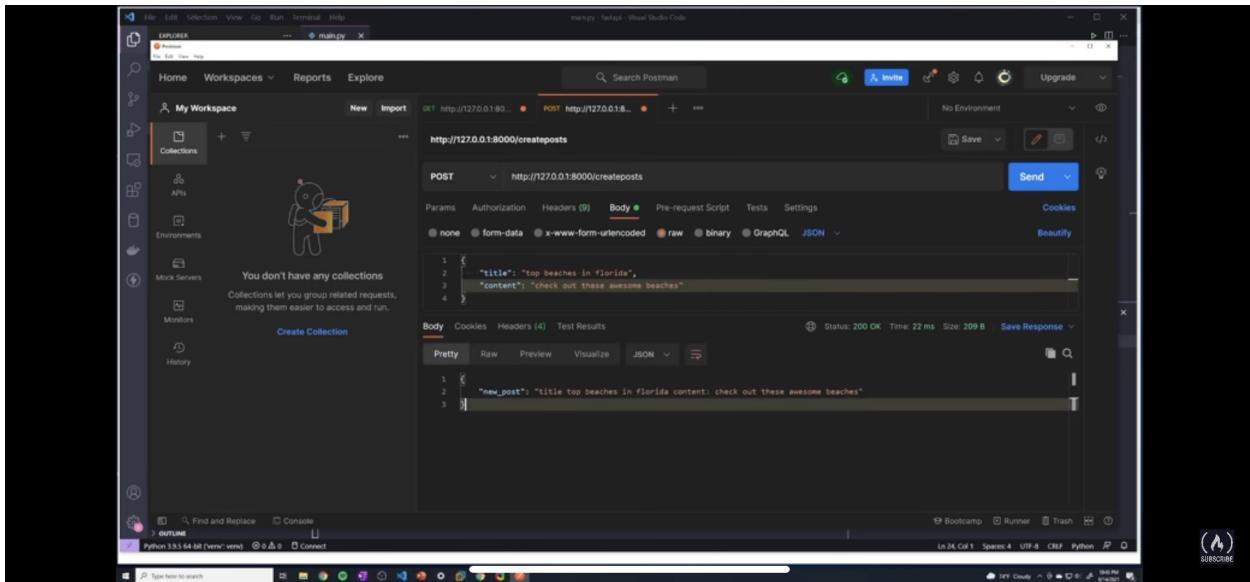
A screenshot of Visual Studio Code showing a Python file named `main.py` with code for a FastAPI application. The code defines two routes: a GET route for posts and a POST route for creating new posts. The terminal below shows the application running and responding to a POST request with a JSON payload containing a title and content.

```
File Edit Selection View Go Run Terminal Help
OPEN EDITORS 1 (UNSAVED)
main.py
> _pycache_
> __pycache__
> include
> Lib
> Scripts
> pyvenv.cfg
main.py

15 @app.get("/")
16 def get_posts():
17     return {"data": "This is your posts"}
18
19
20 @app.post("/createposts")
21 def create_posts(payload: dict = Body(...)):
22     print(payload)
23     return {"new_post": f"title {payload['title']} content: {payload['content']}"}

TERMINAL  SQL LINTER PROBLEMS OUTPUT DEBUG CONSOLE
INFO: 127.0.0.1:55922 - "POST /createposts HTTP/1.1" 200 OK
INFO: 127.0.0.1:55936 - "POST /createposts HTTP/1.1" 200 OK
INFO: 127.0.0.1:55936 - "GET /posts HTTP/1.1" 200 OK
INFO: 127.0.0.1:55946 - "POST /createposts HTTP/1.1" 200 OK
WARNING: WatchdogReloader detected file change in ['C:\Users\sanje\Documents\Course s\fastapi\main.py']. Reloading...
WARNING: WatchdogReloader detected file change in ['C:\Users\sanje\Documents\Course s\fastapi\main.py']. Reloading...
INFO: Started server process [25872]
INFO: Waiting for application startup.
INFO: Application startup complete.
{"title": "top beaches in florida", "content": "check out these awesome beaches"}
INFO: 127.0.0.1:53296 - "POST /createposts HTTP/1.1" 200 OK

cmd
Python 3.8.5 64 bit (venv\venv) ① Connect
In 24, Col 62  Spaces: 4  UTF-8  CRLF  Python ② O
Type here to search
( 🔍 ) SUBSCRIBE
```



## Why we need a Schema?

## Why We Need Schema



- It's a pain to get all the values from the body
- The client can send whatever data they want
- The data isn't getting validated
- We ultimately want to force the client to send data in a schema that we expect



So we don't just sent any kind of data

The screenshot shows a browser window displaying the Pydantic documentation on field types. The left sidebar contains a navigation menu with links like Overview, Install, Usage, Models, Field Types (which is currently selected), Validators, Model Config, Schema, Exporting models, Dataclasses, Validation decorator, Settings management, Postponed annotations, Usage with mypy, Usage with typeguard, Contributing to pydantic, Benchmarks, MyPy plugin, Pydantic plugin, Hypothesis plugin, Code Generation, and Changelog. The main content area is titled "Field Types" and discusses standard library types such as None, bool, int, float, and str. It also covers annotated types like NamedTuple, TypeDict, and Pydantic Types. A sidebar on the right lists other topics like Standard Library Types, TypeVar, Literal Type, and Strict Types. The top right corner of the slide features a "SUBSCRIBE" button with a bell icon.

```

1 from fastapi import FastAPI
2 from fastapi.params import Body
3 from pydantic import BaseModel
4
5 app = FastAPI()
6
7
8 # request Get method url: "/"
9
10
11 @app.get("/")
12 def root():
13     return {"message": "Hello World"}

```

INFO: Waiting for application startup.  
INFO: Application startup complete.  
{'title': 'top beaches in florida', 'content': 'check out these awesome beaches'}  
INFO: 127.0.0.1:53296 - "POST /createposts HTTP/1.1" 200 OK  
WARNING: WatchGodReload detected file change in '[C:\Users\Sanje\Documents\Course s\fastapi\main.py 44dca43f362bd42d885c486f089f1c.tmp]'. Reloading...  
WARNING: WatchGodReload detected file change in '[C:\Users\Sanje\Documents\Course s\fastapi\main.py 44dca43f362bd42d885c486f089f1c.tmp]', C:\Users\Sanje\Documents\Courses\fastapi\main.py'. Reloading...  
INFO: Started server process [1032]  
INFO: Waiting for application startup.  
INFO: Application startup complete.  
{'title': 'top beaches in florida', 'content': 'check out these awesome beaches'}  
INFO: 127.0.0.1:63528 - "POST /createposts HTTP/1.1" 200 OK

- use Pydantic Library to make a structure
- Create a Post class

```

1 from fastapi import FastAPI
2 from fastapi.params import Body
3 from pydantic import BaseModel
4
5 app = FastAPI()
6
7
8 class Post(BaseModel):
9     title: str
10    content: str
11
12
13 @app.get("/")
14 def root():
15     return {"message": "Hello World"}
16

```

\\Courses\\fastapi\\main.py'. Reloading...  
INFO: Started server process [1032]  
INFO: Waiting for application startup.  
INFO: Application startup complete.  
{'title': 'top beaches in florida', 'content': 'check out these awesome beaches'}  
INFO: 127.0.0.1:63528 - "POST /createposts HTTP/1.1" 200 OK  
WARNING: WatchGodReload detected file change in '[C:\Users\Sanje\Documents\Course s\fastapi\main.py 44dca43f362bd42d885c486f089f1c.tmp]'. Reloading...  
WARNING: WatchGodReload detected file change in '[C:\Users\Sanje\Documents\Course s\fastapi\main.py 44dca43f362bd42d885c486f089f1c.tmp]', C:\Users\Sanje\Documents\Courses\fastapi\main.py'. Reloading...  
INFO: Started server process [17780]  
INFO: Waiting for application startup.  
INFO: Application startup complete.

Pass in the class as an arg

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface. The left sidebar (Explorer) displays a file tree with the following structure:

- OPEN EDITORS (1 UNSAVED)
- main.py
- > .pycache\_
- > venv
- > Include
- > Lib
- > Scripts
- pyenv.cfg
- main.py

The main editor area contains the following Python code:

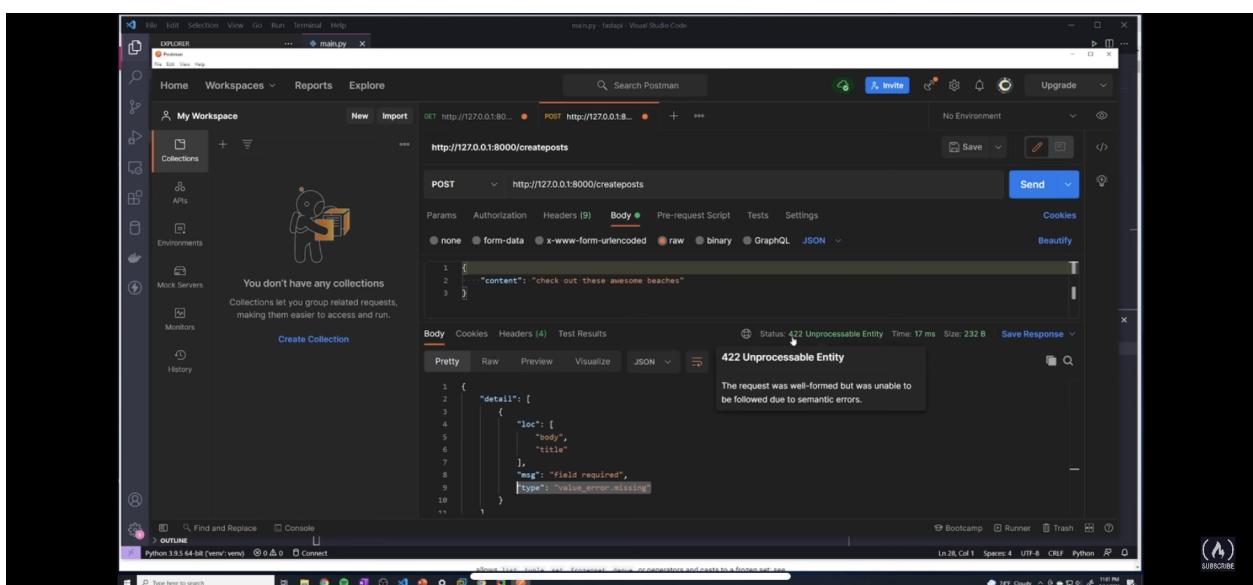
```
return {"data": "This is your posts"}  
20  
21  
22  
23 @app.post("/createposts")  
24 def create_posts(new_post: Post):  
25     print(payload)  
26     return {"new_post": f"title {payload['title']} content: {payload['content']}"}  
27 # title str, content str  
28
```

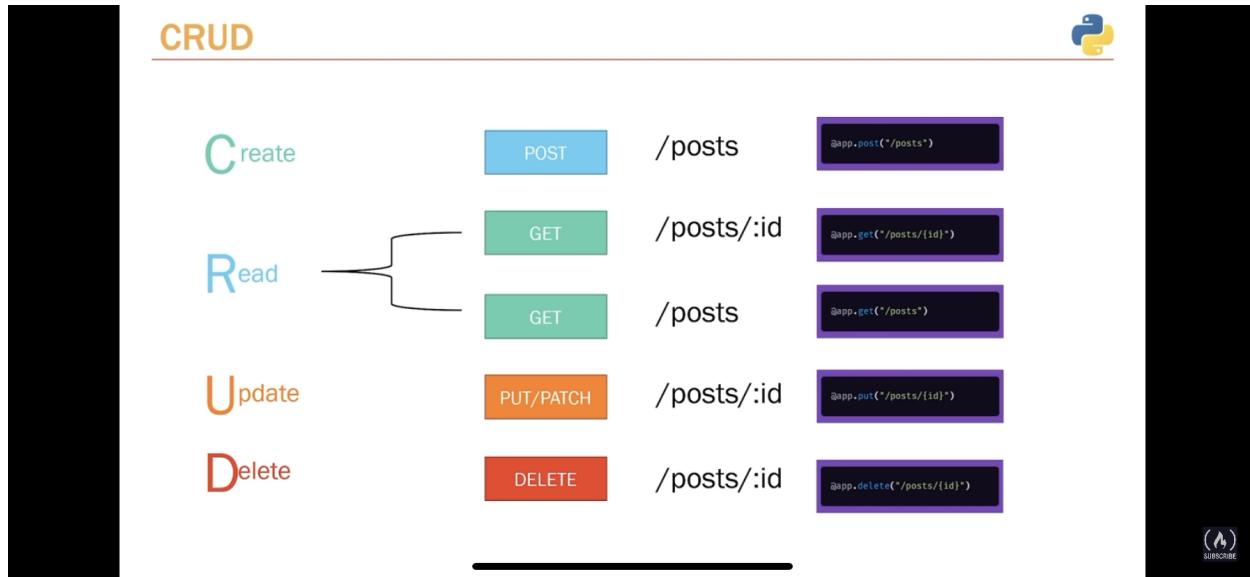
The terminal at the bottom shows the output of running the application:

```
\\"C:\\Courses\\Fastapi\\main.py\". Reloading...  
INFO: Started server process [1832]  
INFO: Waiting for application startup.  
INFO: Application startup complete.  
{'title': 'top beaches in florida', 'content': 'check out these awesome beaches'}  
INFO: 127.0.0.1:63528 - "POST /createposts HTTP/1.1" 200 OK  
WARNING: WatchGodReload detected file change in '[C:\\Users\\sanje\\Documents\\Courses\\Fastapi\\main.py]'. Reloading...  
WARNING: WatchGodReload detected file change in '[C:\\Users\\sanje\\Documents\\Courses\\Fastapi\\main.py]'. Reloading...  
INFO: Started server process [17780]  
INFO: Waiting for application startup.  
INFO: Application startup complete.
```

The status bar at the bottom indicates the environment is Python 3.9.5 64-bit (venv: venv), and the current file is main.py - fastapi - Visual Studio Code.

validates all by itself - if wrong structure





Every app should be able to Create, Read, Update, Delete

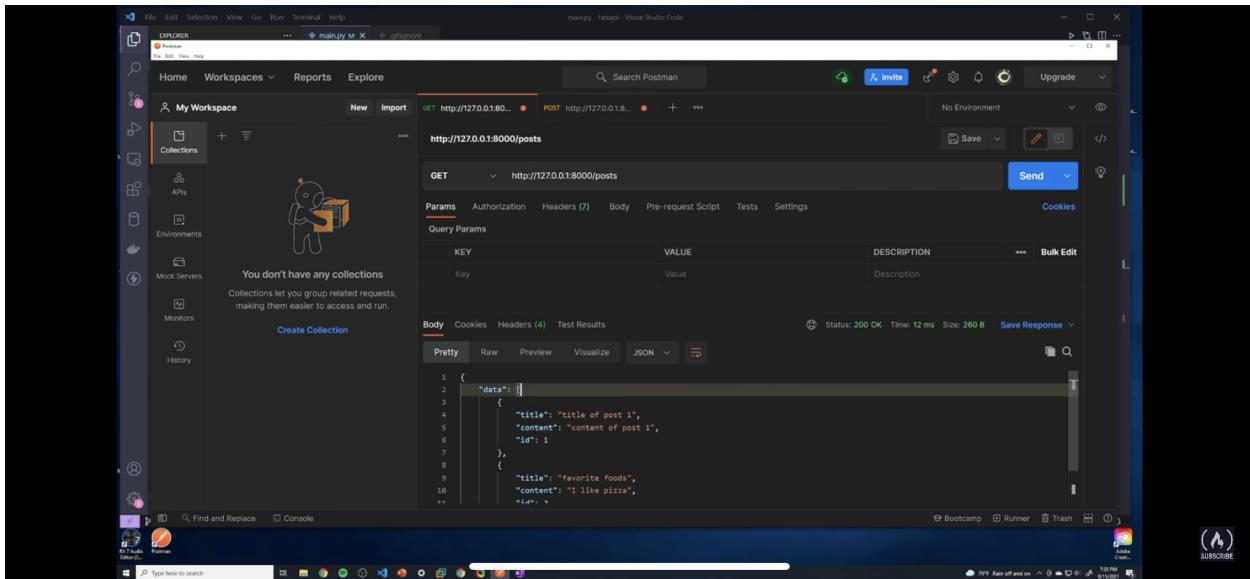
## Passing Arrays

Can pass in arrays which will get auto-serialized

```

File Edit Selection View Go Run Terminal Help
OPEN EDITORS TUNGED
main.py M .gitignore
mainpy > get_posts
    title: str
    content: str
    published: bool = True
    rating: Optional[int] = None
my_posts = [{"title": "title of post 1", "content": "content of post 1", "id": 1}, {"title": "favorite foods", "content": "I like pizza", "id": 2}]
@app.get("/")
def root():
    return {"message": "Hello World"}
@app.get("/posts")
def get_posts():
    (variable) my_posts: list[dict[str, Any]]
    return {"data": my_posts}
@app.post("/posts")
def create_posts(post: Post):
    print(post)
    print(post.dict())
    return {"data": post}

```

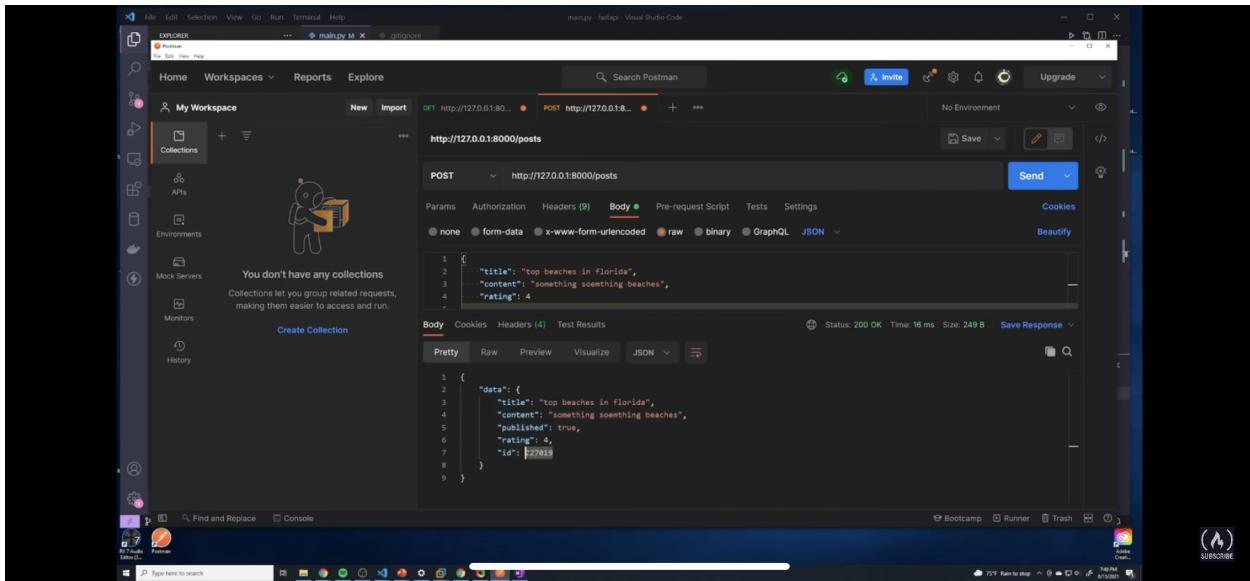


## Saving the data from a post request to an Array

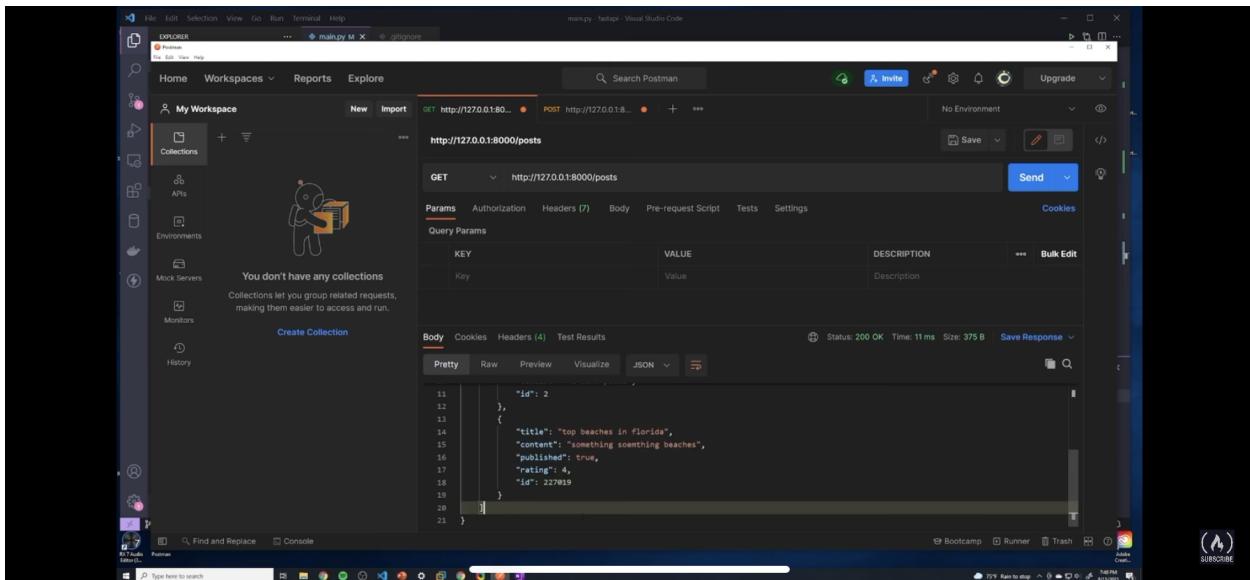
There is a chance that the id's will be hit - so make them bigger

```
return {"message": "Hello World"}  
@app.get("/posts")  
def get_posts():  
    return {"data": my_posts}  
  
@app.post("/posts")  
def create_posts(post: Post):  
    post_dict = post.dict()  
    post_dict['id'] = randint(0, 1000000)  
    my_posts.append(post_dict)  
    return {"data": post_dict}
```

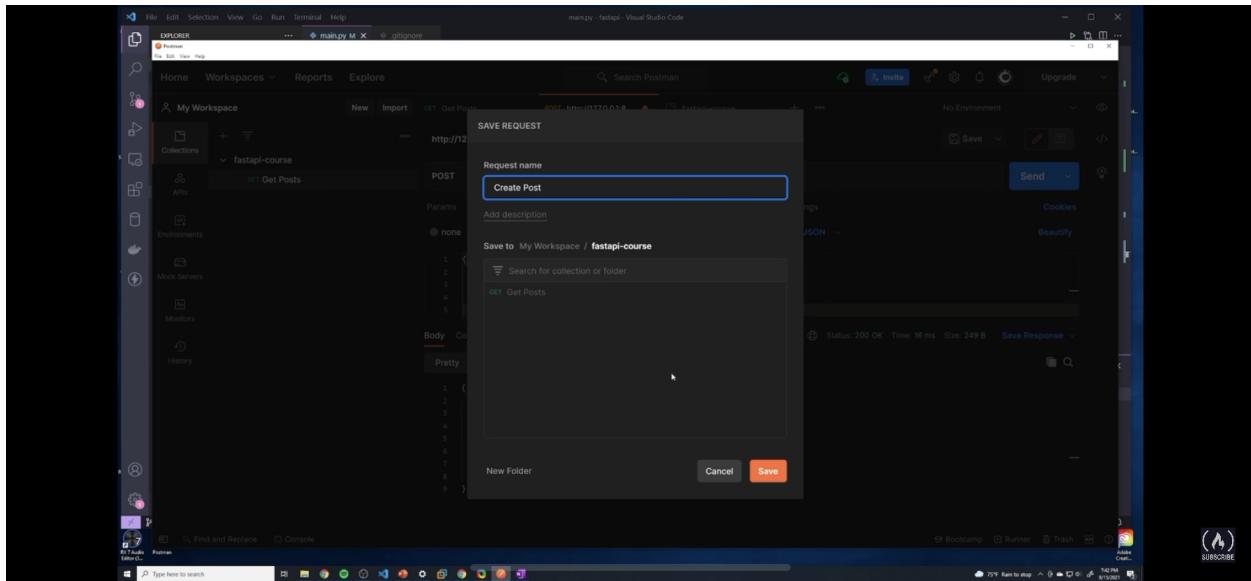
Run a Post request and the id increases



Run a get request to retrieve the data



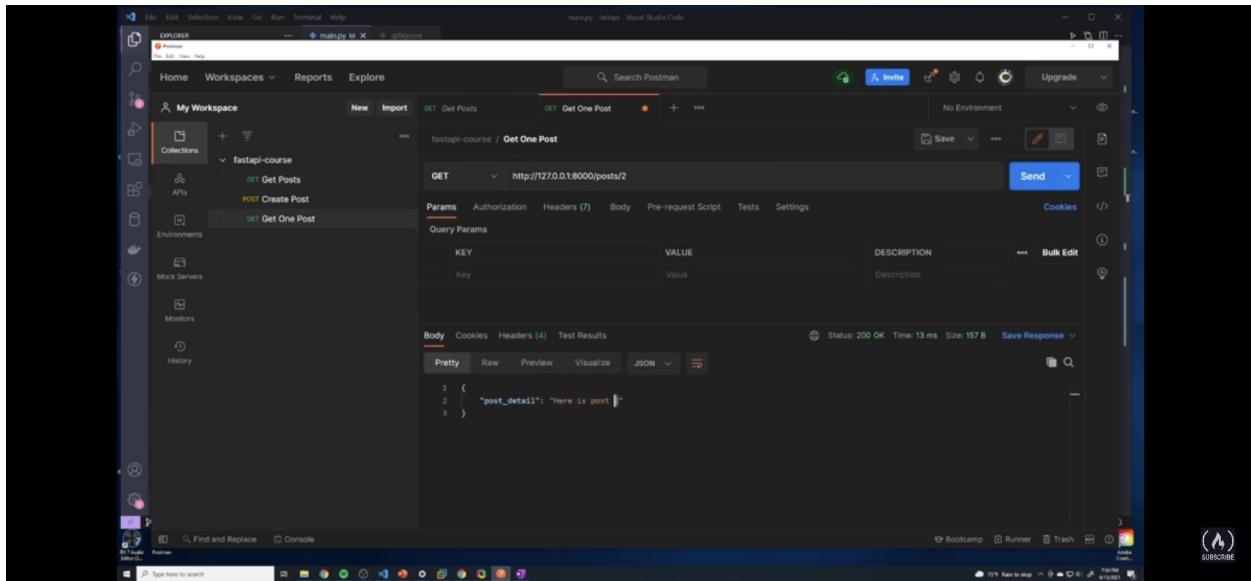
Can save requests too



## Get a specific id

```
23     return {"message": "Hello World"}
24
25
26 @app.get("/posts")
27 def get_posts():
28     return {"data": my_posts}
29
30
31 @app.post("/posts")
32 def create_posts(post: Post):
33     post_dict = post.dict()
34     post_dict['id'] = randrange(0, 1000000)
35     my_posts.append(post_dict)
36     return {"data": post_dict}
37
38
39 @app.get("/posts/{id}")
40 def get_post(id):
41     print(id)
42     return {"post_detail": f"Here is post {id}"}
```

## Test in Postman



Refactor and use a function to find id we want

```
content: str
published: bool = True
rating: Optional[int] = None

my_posts = [{"title": "title of post 1", "content": "content of post 1", "id": 1}, {
    "title": "favorite foods", "content": "I like pizza", "id": 2}]

def find_post(id):
    for p in my_posts:
        if p["id"] == id:
            return p
```

A screenshot of Visual Studio Code showing a Python file named `main.py`. The code defines a FastAPI application with endpoints for creating and retrieving posts. The `get_posts` endpoint returns a list of posts, while the `get_post` endpoint returns a single post by ID. The code uses `randrange` to generate random IDs.

```
def get_posts():
    return {"data": my_posts}

@app.post("/posts")
def create_posts(post: Post):
    post_dict = post.dict()
    post_dict['id'] = randrange(0, 1000000)
    my_posts.append(post_dict)
    return {"data": post_dict}

@app.get("/posts/{id}")
def get_post(id):
    post = find_post(id)
    return {"post_detail": post}
```

Need to make sure we only send ints

A screenshot of Visual Studio Code showing the same `main.py` file. The code has been modified to include type hints for the `id` parameter in the `get_post` endpoint. The terminal shows an error message indicating that the value passed to `int()` is not a valid integer literal.

```
def get_post(id: int):

    post = find_post(int(id))
    print(post)
    return {"post_detail": post}
```

```
File "c:\users\sanje\documents\courses\fastapi\venv\lib\site-packages\starlette\concurrency.py", line 40, in run_in_threadpool
    return await loop.run_in_executor(None, func, *args)
  File "C:\Users\sanje\AppData\Local\Programs\Python\Python39\lib\concurrent\futures\thread.py", line 52, in run
    result = self.fn(*self.args, **self.kwargs)
  File "C:\Users\sanje\Documents\Courses\FastAPI\main.py", line 48, in get_post
    post = find_post(int(id))
ValueError: invalid literal for int() with base 10: 'asdfasdf'
```

Return most recent post

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows a project structure with files like `main.py`, `fastapi`, `venv`, and `githooks`.
- Code Editor:** Displays Python code for a FastAPI application. The code includes routes for creating posts, getting a post by ID, and getting the latest post.

```
35
36
37 @app.post("/posts")
38 def create_posts(post: Post):
39     post_dict = post.dict()
40     post_dict['id'] = randrange(0, 1000000)
41     my_posts.append(post_dict)
42     return {"data": post_dict}
43
44
45 @app.get("/posts/{id}")
46 def get_post(id: int):
47
48     post = find_post(id)
49     print(post)
50     return {"post_detail": post}
51
52
53 @app.get("/posts/latest")
54 def get_latest_post():
55     post = my_posts[len(my_posts)-1]
56     return {"detail": post}
```

- Bottom Status Bar:** Shows the current file is `main.py`, the Python version is 3.9.5 (64-bit), and the terminal shows "Connect".
- Bottom Taskbar:** Includes icons for Bootcamp, Runner, Trash, and others.

# Working with Response Codes

If a post is not found

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure with files: `main.py`, `__init__.py`, `models.py`, `schemas.py`, `fastapi`, `venv`, `include`, `lib`, `scripts`, and `gitignore`.
- Code Editor:** The `main.py` file is open, displaying the following Python code for a FastAPI application:

```
main.py M ● ④ .gitignore

OPEN EDITORS: UNSTAGED
main.py
32     def get_posts():
33         return {"data": my_posts}
34
35
36
37     @app.post("/posts")
38     def create_posts(post: Post):
39         post_dict = post.dict()
40         post_dict['id'] = randrange(0, 1000000)
41         my_posts.append(post_dict)
42         return {"data": post_dict}
43
44
45     @app.get("/posts/{id}")
46     def get_post(id: int, response: Response):
47
48         post = find_post(id)
49         if not post:
50             response.status_code = 404
51         return {"post_detail": post}
```

- Bottom Status Bar:** Shows the current environment as "Python 3.9.5 64-bit (venv, venv)" and other status indicators like "In 50, Col 35".
- Taskbar:** Shows the taskbar with various pinned icons.

Return a status code 404

A screenshot of Visual Studio Code showing a Python file named `main.py`. The code defines a function `get_post` which returns a dictionary for a post by ID. Inside the function, there is a line of code where the variable `response.status_code` is assigned a value. A code completion dropdown is open, listing various HTTP status codes such as `HTTP_201_CREATED`, `HTTP_202_ACCEPTED`, etc. The status code `HTTP_100_CONTINUE` is highlighted.

```
def get_post(id: int, response: Response):
    post = find_post(id)
    if not post:
        response.status_code = status.HTTP_100_CONTINUE
        return {"post_detail": post}
```

Or write status or add httpException

A screenshot of Visual Studio Code showing the same `main.py` file. The code now includes a `raise HTTPException` statement instead of directly setting the `status_code`. The message part of the exception is set to a string: "post with id: {id} was not found".

```
def get_post(id: int, response: Response):
    post = find_post(id)
    if not post:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"post with id: {id} was not found")
    return {"post_detail": post}
```

Send a 201 created when a Post request is sent

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure with files: `main.py`, `gignore`, and a folder named `mainpy`.
- Code Editor:** Displays Python code for a FastAPI application. The code includes routes for `/posts` and `/posts/{id}`, and a helper function `find_post`. It uses `status.HTTP_201_CREATED` and `status.HTTP_404_NOT_FOUND`.
- Bottom Status Bar:** Shows the status bar with tabs for `main`, `Python 3.8 64-bit (venv: venv)`, and `Connect`. It also displays file statistics like `43 Col 1`, `Spaces 4`, `UTF-8`, and `Python`.

## **Deleting Posts**

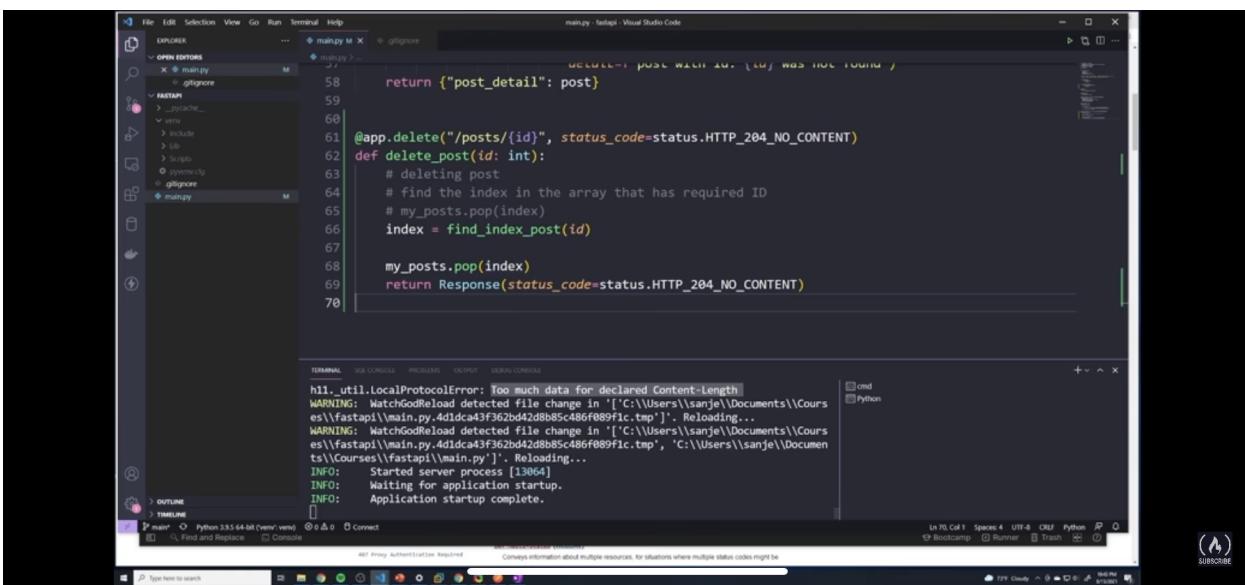
Loop through posts and return the index

```
25
26 def find_index_post(id):
27     for i, p in enumerate(my_posts):
28         if p['id'] == id:
29             return i
30
31
```

Pop from the dict the item with the id that you want to delete

```
59
60
61 @app.delete("/posts/{id}")
62 def delete_post():
63     # deleting post
64     # find the index in the array that has required ID
65     # my_posts.pop(index)
66     index = find_index_post(id)
67
68     my_posts.pop(index)
69     return {'message': 'post was successfully deleted'}
70
```

Add status code 204 : No content

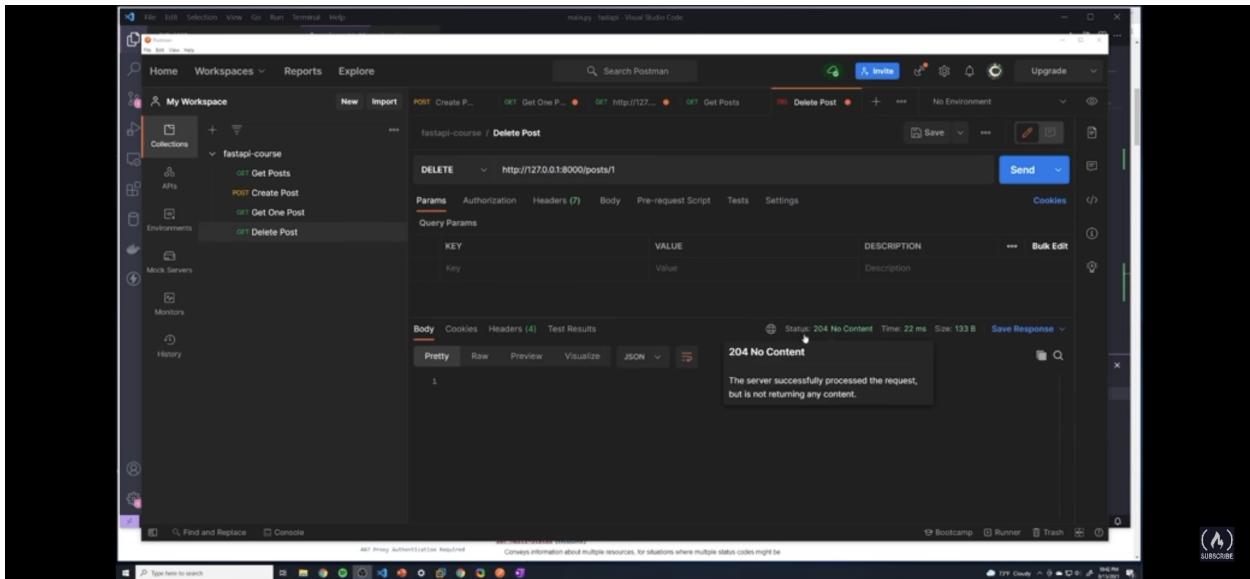


The screenshot shows the Visual Studio Code interface with the main.py file open. The code has been modified to include a status code for the delete operation:

```
58     return {"post_detail": post}
59
60
61 @app.delete("/posts/{id}", status_code=status.HTTP_204_NO_CONTENT)
62 def delete_post(id: int):
63     # deleting post
64     # find the index in the array that has required ID
65     # my_posts.pop(index)
66     index = find_index_post(id)
67
68     my_posts.pop(index)
69     return Response(status_code=status.HTTP_204_NO_CONTENT)
70
```

The terminal at the bottom shows application logs and a command prompt.

Test in Postman



Add exception Handling for id not existing

```

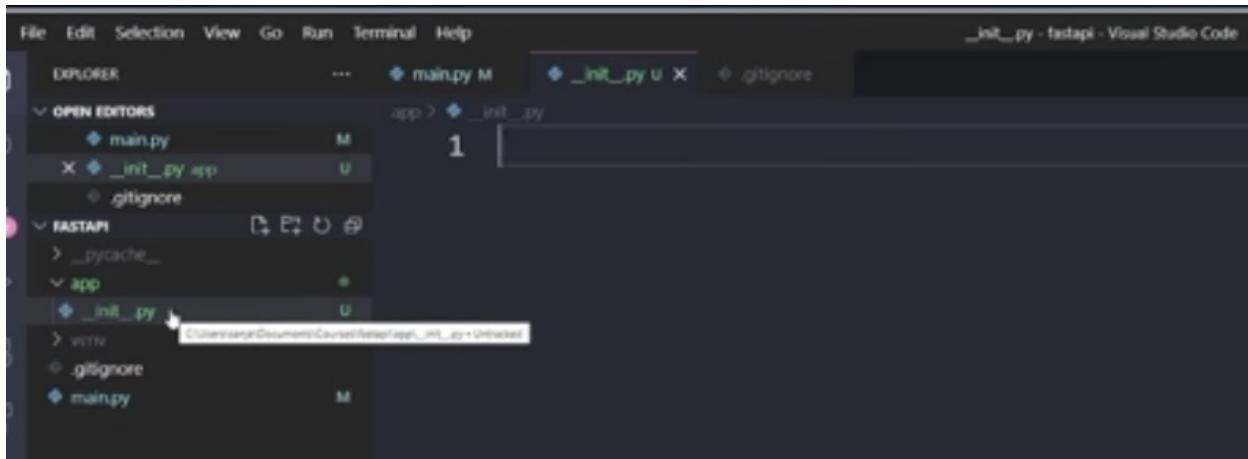
File Edit Selection View Go Run Terminal Help
main.py M • gllignore
mainpy > delete_post
      detail=f"post with ID: {id} was not found."
      return {"post_detail": post}
@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_post(id: int):
    # deleting post
    # find the index in the array that has required ID
    # my_posts.pop(index)
    index = find_index_post(id)

    if index == None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=f"post with id: {id} does not exist")
    my_posts.pop(index)
    return Response(status_code=status.HTTP_204_NO_CONTENT)

```

## Python Packages

Create a folder called app



- this needs to become a package so needs an init.py file

```
from typing import Optional
from fastapi import FastAPI, Response, status, HTTPException
from pydantic import BaseModel
from random import randrange

app = FastAPI()
```

The screenshot shows the code for the `app/\_init\_.py` file in Visual Studio Code. The code defines a FastAPI application object named `app`:

```
from typing import Optional
from fastapi import FastAPI, Response, status, HTTPException
from pydantic import BaseModel
from random import randrange

app = FastAPI()
```

## Working with Databases

## What is a Database



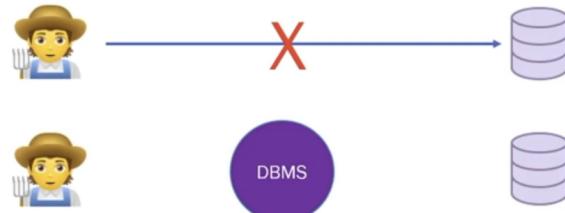
- Database is a collection of organized data that can be easily accessed and managed



## DBMS



- We don't work or interact with databases directly.
- Instead we make use of a software referred to as a Database Management System(DBMS)



## Popular DBMS



### Relational

- MYSQL
- POSTGRESQL
- ORACLE
- SQL SERVER

### NoSQL

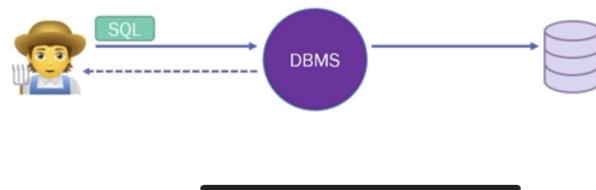
- MongoDB
- DynamoDB
- ORACLE
- SQL SERVER

( 🔍 )  
SUBSCRIBE

## Relational Database & SQL



- Structured Query Language(SQL) – Language used to communicate with DBMS

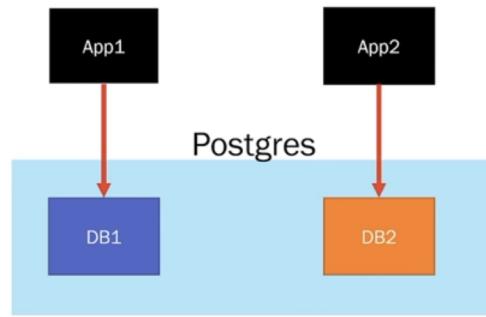


( 🔍 )  
SUBSCRIBE

## Postgres



- Each instance of postgres can be carved into multiple separate databases



(⤴)  
SUBSCRIBE

## Postgres



- By default every Postgres installation comes with one database already created called “postgres”
- This is important because Postgres requires you to specify the name of a database to make a connection. So there needs to always be one database

(⤴)  
SUBSCRIBE

## Tables



- A table represents a subject or event in an application

Users		

Products		

Purchases		



## Columns Vs Rows



- A table is made up of columns and rows
- Each Column represents a different attribute
- Each row represents a different entry in the table

Columns

Rows

ID	name	Age	Sex
14642	Vanessa	40	F
73934	Carl	23	M
99384	George	19	M





## Primary Key

- Is a column or group of columns that uniquely identifies each row in a table
- Table can have one and only one primary key

A diagram illustrating a primary key. At the top, a green box labeled "Primary Key" has an arrow pointing down to the first column of a table. The first column is labeled "id" and is also highlighted in green. To the left of the table, a callout box contains the text: "Each Entry must be unique, no DUPLICATES!!!!". The table has five columns: id, name, email, password, and Phone Number. It contains four rows of data.

id	name	email	password	Phone Number
77498	John	John@gmail.com	Password123	9195789993
14982	Kyel	kyle@yahoo.com	1234	6198723343
34098	Alex	alex@aol.com	Alex123	4467489983
05562	Linda	linda@gmail.com	puppy765	2024857721

( 🔍 )  
SUBSCRIBE



## Null Constraints

- By default, when adding a new entry to a database, any column can be left blank. When a column is left blank, it has a null value
- If you need column to be properly filled in to create a new record, a NOT NULL constraint can be added to the column to ensure that the column is never left blank

A diagram illustrating a NOT NULL constraint. An orange box labeled "NOT NULL" is positioned above the "Age" column header. A red arrow points from this box to the "Age" column in the second row, where the value is listed as "NULL". A red callout box to the right of the table contains the text: "Cannot Be Null". The table has four columns: ID, name, Age, and Sex. It contains three rows of data.

ID	name	Age	Sex
14642	Vanessa	40	F
73934	Carl	NULL	
99384	Vanessa	19	

( 🔍 )  
SUBSCRIBE



## Unique Constraints



- A UNIQUE constraint can be applied to any column to make sure every record has a unique value for that column

The diagram illustrates a database table with four columns: ID, name, Age, and Sex. An orange box labeled "UNIQUE" is positioned above the table, with a black arrow pointing down to the "name" column. A red callout box below the table contains the text "Duplicates are not allowed". A red arrow points from this callout to the second row of the table, which shows "Vanessa" listed twice under the "name" column.

ID	name	Age	Sex
14642	Vanessa	40	F
73934	Carl	23	M
99384	Vanessa	19	M



## Primary Key

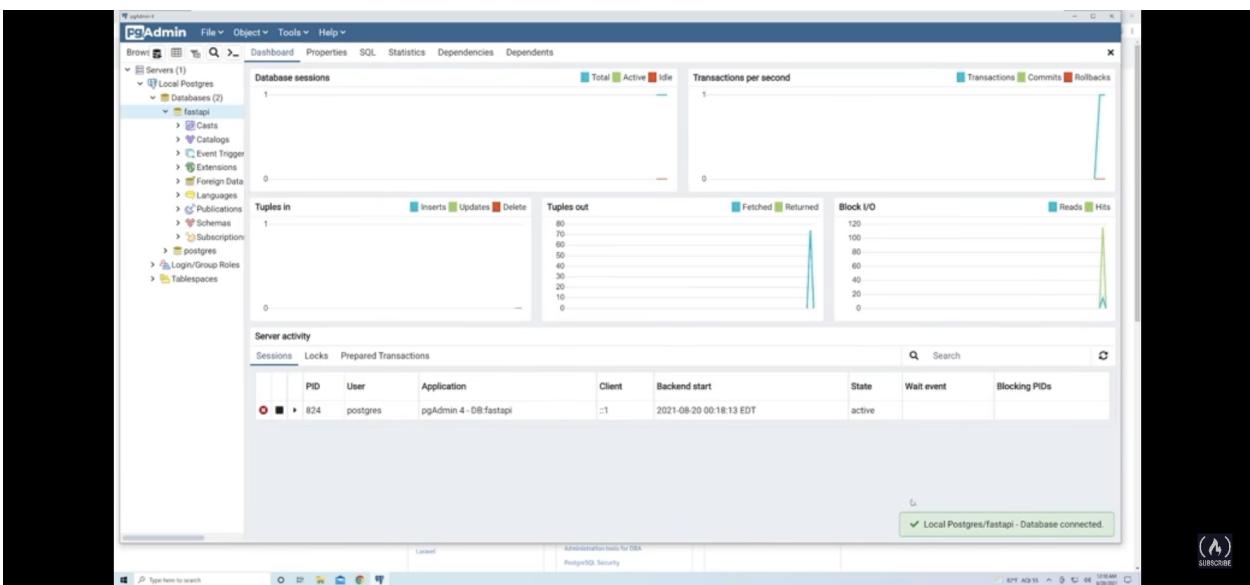
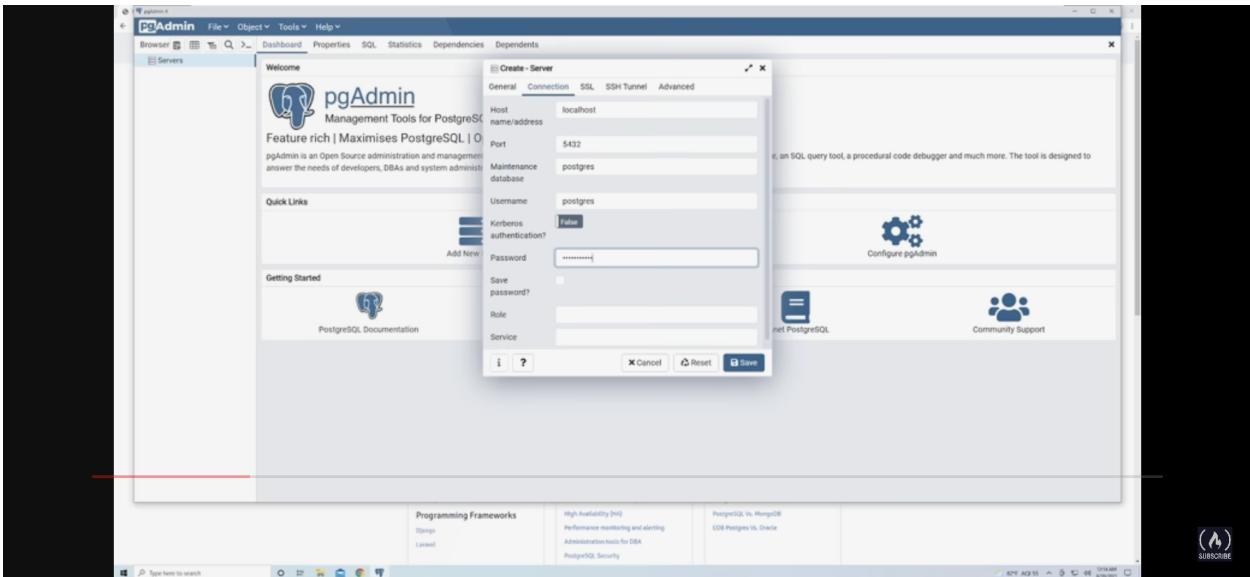


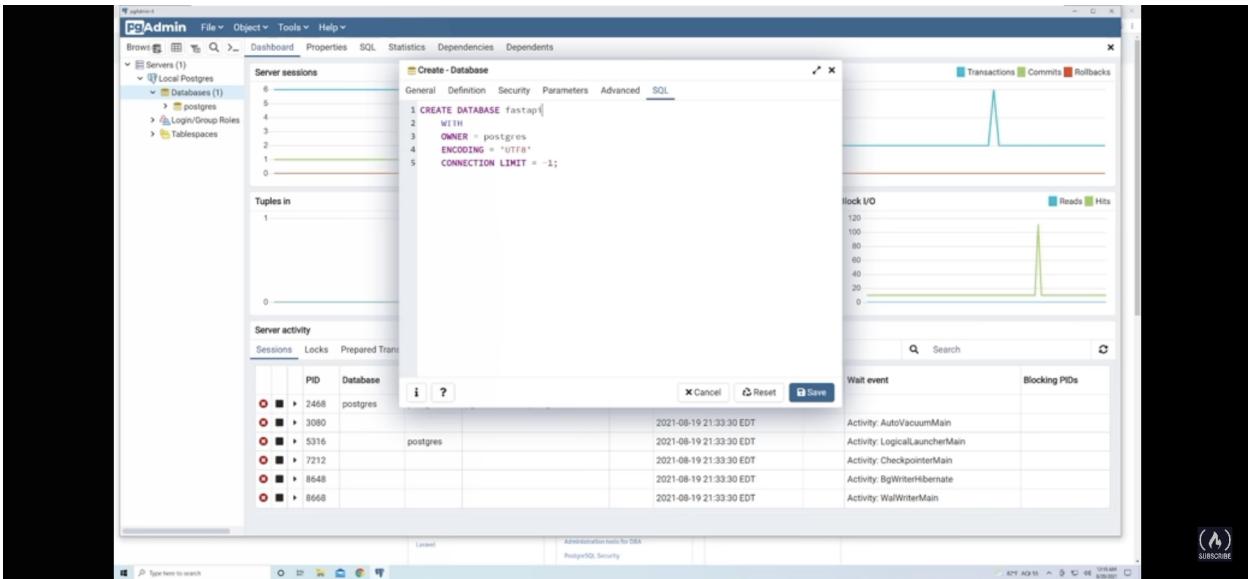
- The Primary Key does not have to be the ID column always. It's up to you to decide which column uniquely defines each record
- In this example, since an email can only be registered once, the email column can also be used as the primary key

The diagram illustrates a database table with five columns: id, name, email, password, and Phone Number. A green box labeled "Primary Key" is positioned above the table, with a black arrow pointing down to the "email" column. The "email" column is highlighted with a green background in all rows.

id	name	email	password	Phone Number
77498	John	John@gmail.com	Password123	9195789993
14982	Kyel	kyle@yahoo.com	1234	6198723343
34098	Alex	alex@aol.com	Alex123	4467489983
05562	Linda	linda@gmail.com	puppy765	2024857721

# POSTGRESQL





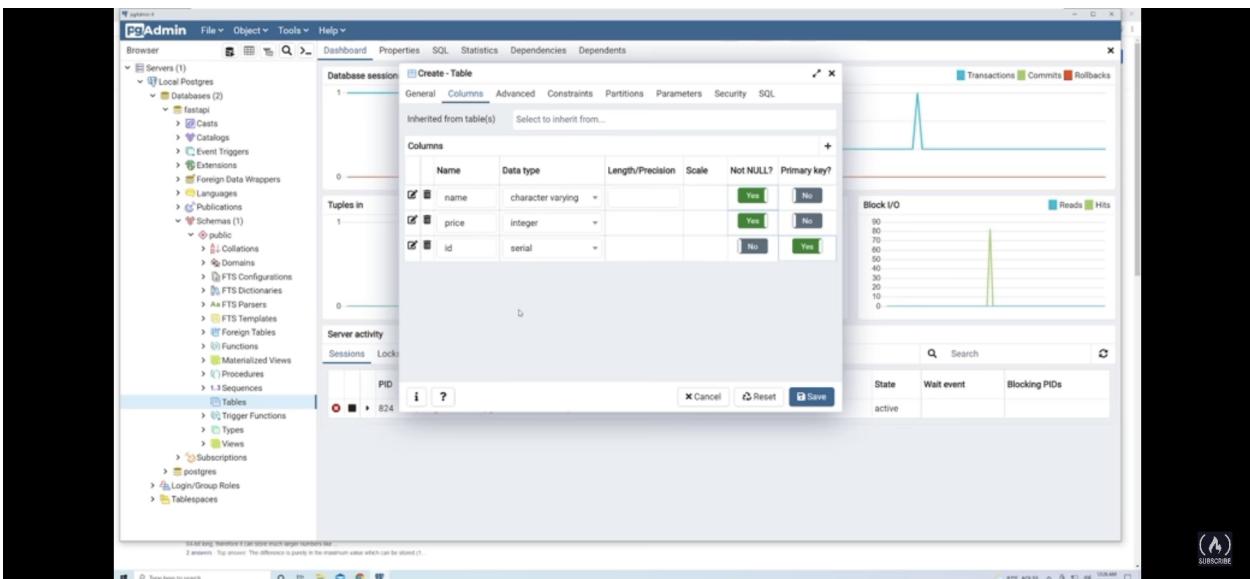
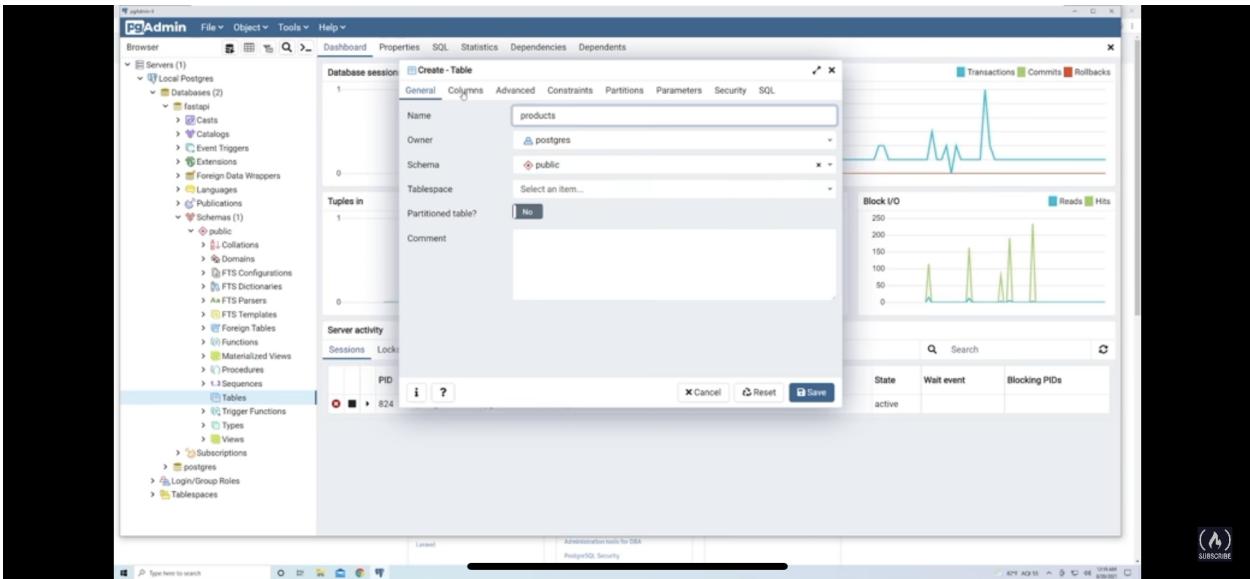
## Postgres DataTypes



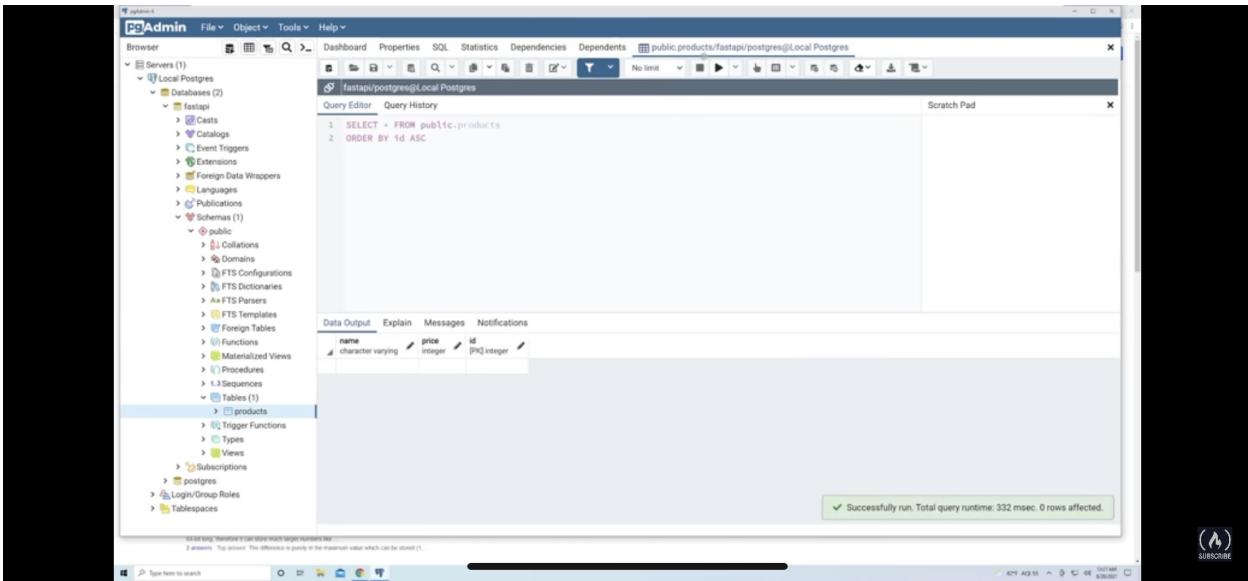
- Databases have datatypes just like any programming language

Data Type	Postgres	Python
Numeric	Int, decimal, precision	Int, float
Text	Varchar, text	string
bool	boolean	boolean
sequence	array	list

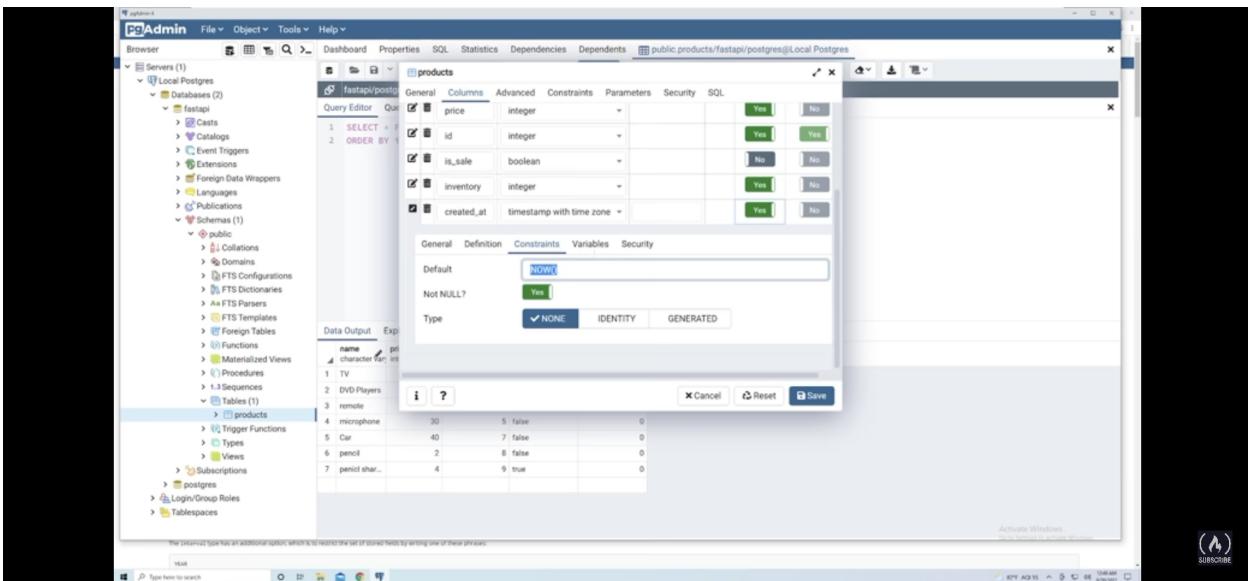


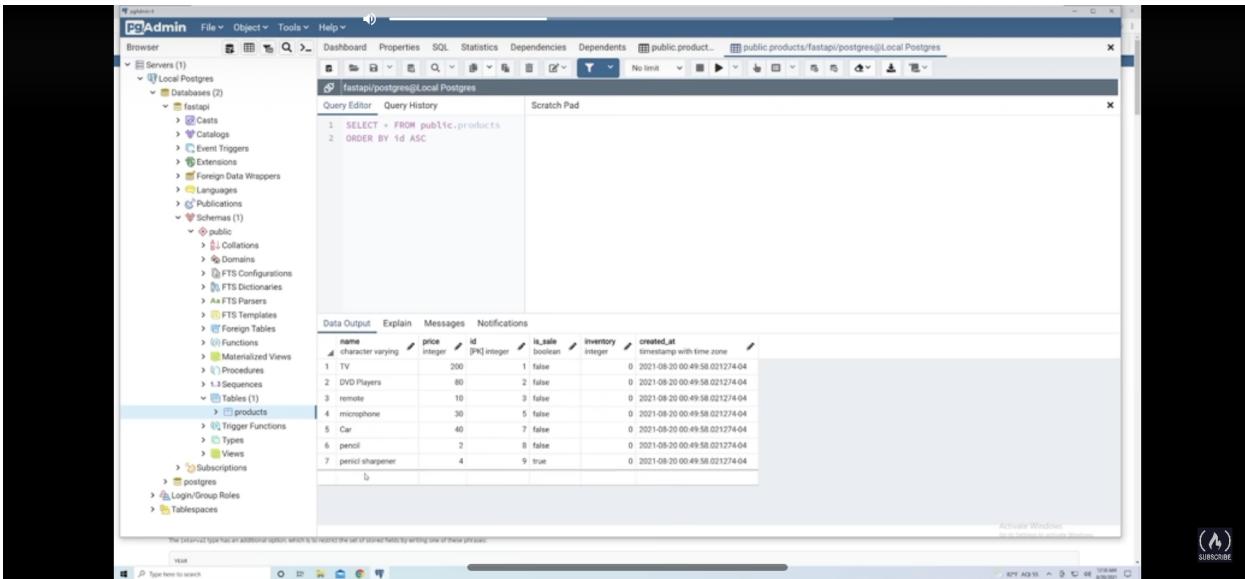


View top x rows



## Setting data





## Connecting to SQL with our API

pip install and Import Psycopg

The screenshot shows a web browser displaying the Psycopg 2.9.1 documentation. The title is "Psycopg 2.9.1 documentation". The main content area is titled "Basic module usage". It includes a brief description: "The basic Psycopg usage is common to all the database adapters implementing the DB API 2.0 protocol. Here is an interactive session showing some of the basic commands:" followed by a code block:

```

>>> import psycopg2
>>> # Connect to an existing database
>>> conn = psycopg2.connect("dbname=test user=postgres")
>>> # Open a cursor to perform database operations
>>> cur = conn.cursor()
>>> # Execute a command: this creates a new table
>>> cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);")
>>> # Pass data to fill a query placeholders and let Psycopg perform
>>> # the correct conversion (no more SQL injections!)
>>> cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",
...             (100, "abc'def"))
>>> # Query the database and obtain data as Python objects
>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchone()
(1, 100, "abc'def")
>>> # Make the changes to the database persistent
>>> conn.commit()
>>> # Close communication with the database
>>> cur.close()
>>> conn.close()

```

Below the code block, there is a note: "The main entry points of Psycopg are:".

## Connect to Postgres instance

```

6 import psycopg2
7 from psycopg2.extras import RealDictCursor
8
9 app = FastAPI()
10
11
12 class Post(BaseModel):
13     title: str
14     content: str
15     published: bool = True
16
17 try:
18     conn = psycopg2.connect(host='localhost', database='fastapi', user='postgres',
19                             password='password123', cursor_factory=RealDictCursor)

```

Real dict cursor gives you the column name aswell as value - gives east to use dict

## Add cursor and exception handling

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure with files like `main.py`, `app.py`, and `__init__.py`.
- Code Editor:** Displays the `main.py` file containing Python code for a FastAPI application connecting to PostgreSQL.
- Terminal:** Shows the message "Application startup complete."
- Status Bar:** Shows the Python version (Python 3.5.5 64-bit) and the current file (`main.py`).

```
import psycopg2
from psycopg2.extras import RealDictCursor

app = FastAPI()

class Post(BaseModel):
    title: str
    content: str
    published: bool = True

try:
    conn = psycopg2.connect(host='localhost', database='fastapi', user='postgres',
                           password='password123', cursor_factory=RealDictCursor)
    cursor = conn.cursor()
    print("Database connection was successful!")
except Exception as error:
    print("Connecting to database failed")
    print("Error: ", error)

my_posts = [{"title": "title of post 1", "content": "content of post 1", "id": 1}, {
```

Add ability to keep trying to connect but add a wait period using a while loop

- import time library

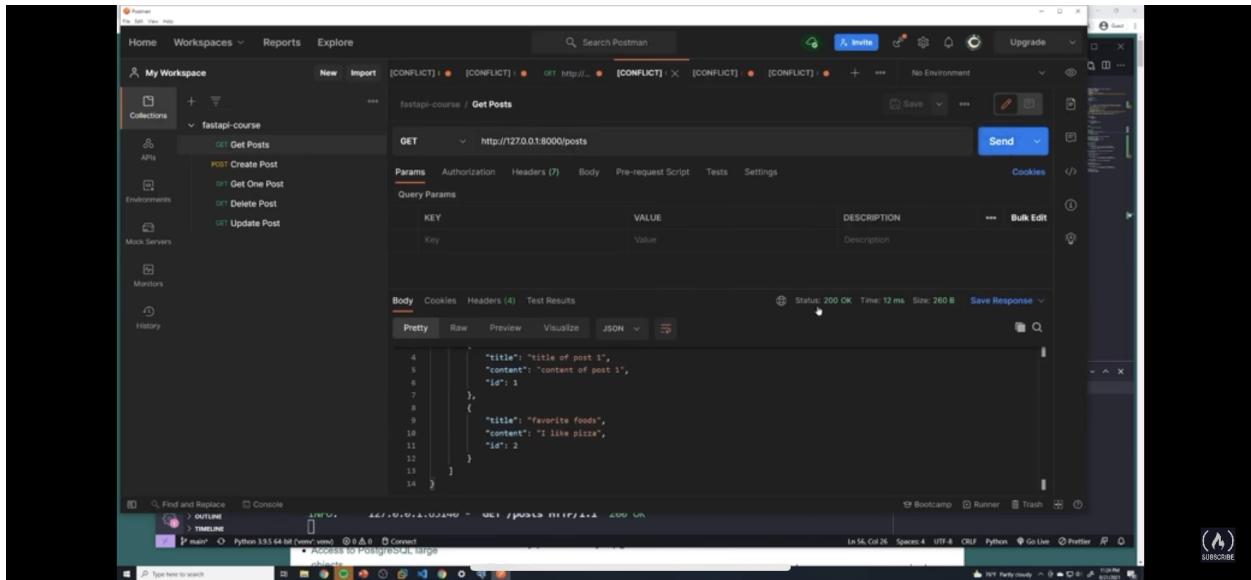
```
while True:
    try:
        conn = psycopg2.connect(host='localhost', database='fastapi', user='postgres',
                               password='password1234', cursor_factory=RealDictCursor)
        cursor = conn.cursor()
        print("Database connection was successful!")
        break
    except Exception as error:
        print("Connecting to database failed")
        print("Error: ", error)
        time.sleep(2)

my_posts = [{"title": "title of post 1", "content": "content of post 1", "id": 1}, {"title": "title of post 2", "content": "content of post 2", "id": 2}]

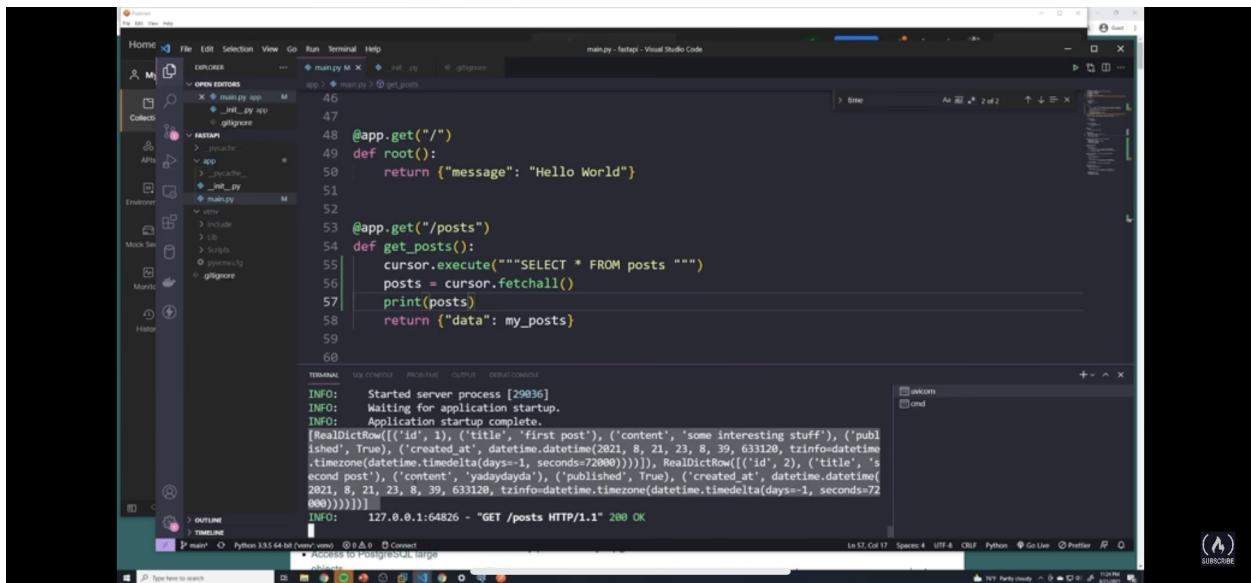
INFO: Waiting for application startup.
INFO: Application startup complete.
WARNING: WatchGodReload detected file change in '[C:\Users\sanje\Documents\Courses\fastapi\app\main.py,dfb0c928ea7d1f64e3c0a5f2a29ddib.tmp]'. Reloading...
WARNING: WatchGodReload detected file change in '[C:\Users\sanje\Documents\Courses\fastapi\app\main.py,dfb0c928ea7d1f64e3c0a5f2a29ddib.tmp]'. Reloading...
Connecting to database failed
Error: FATAL: password authentication failed for user "postgres"
```

## Connecting our database to http methods

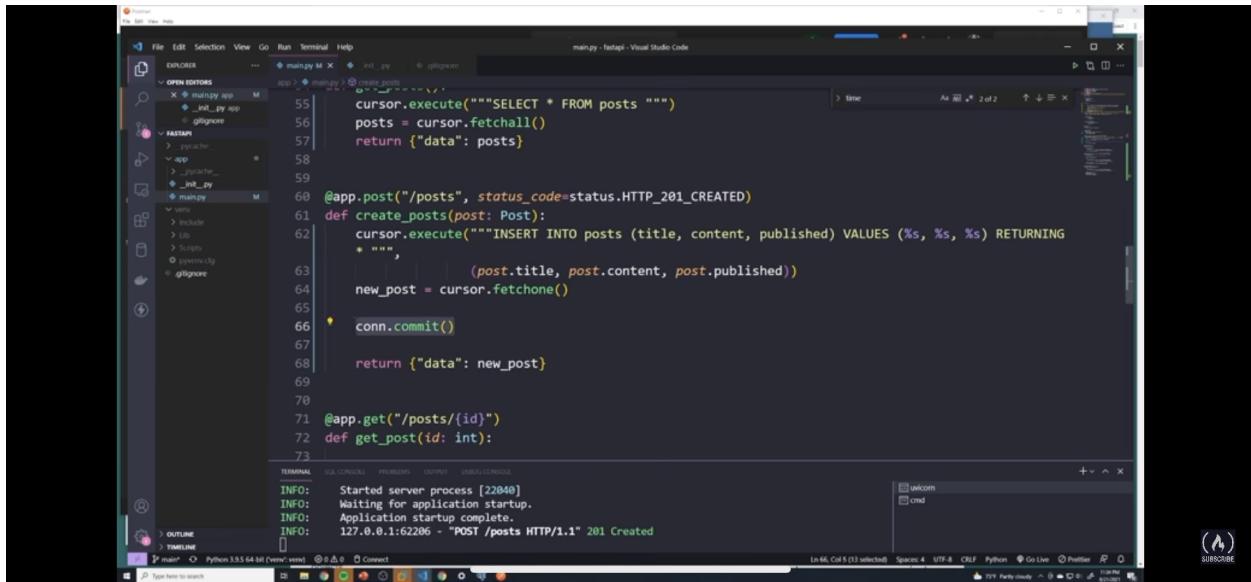
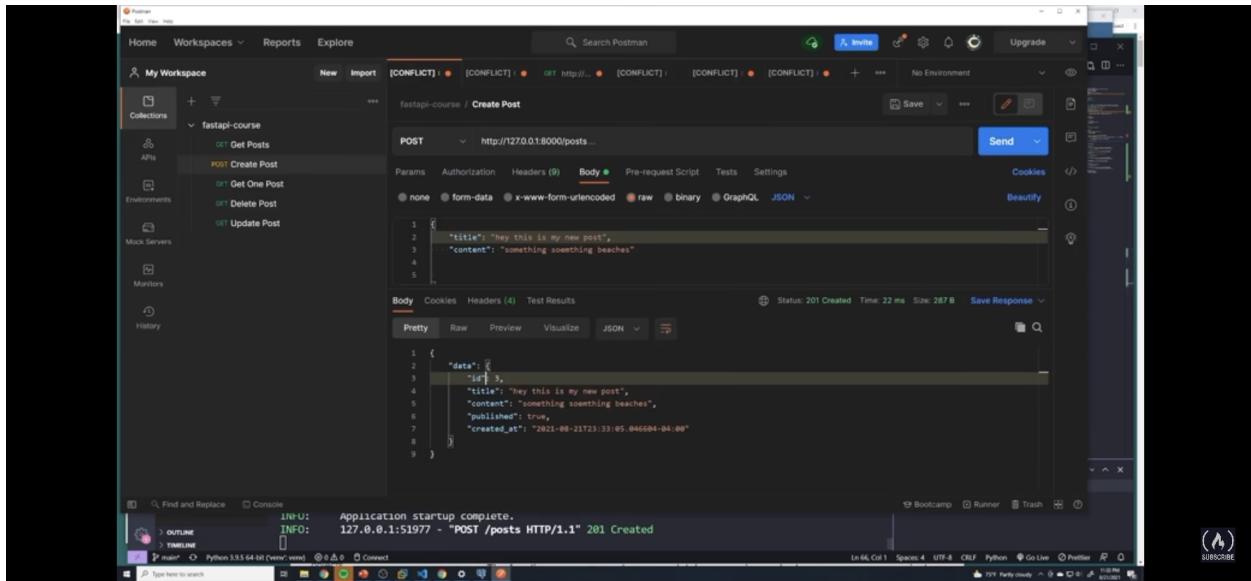
Write the sql query using the cursor method



Now retrieved data from db



Connecting a post request to our db



Need conn.commit() to actually move staged changes to db

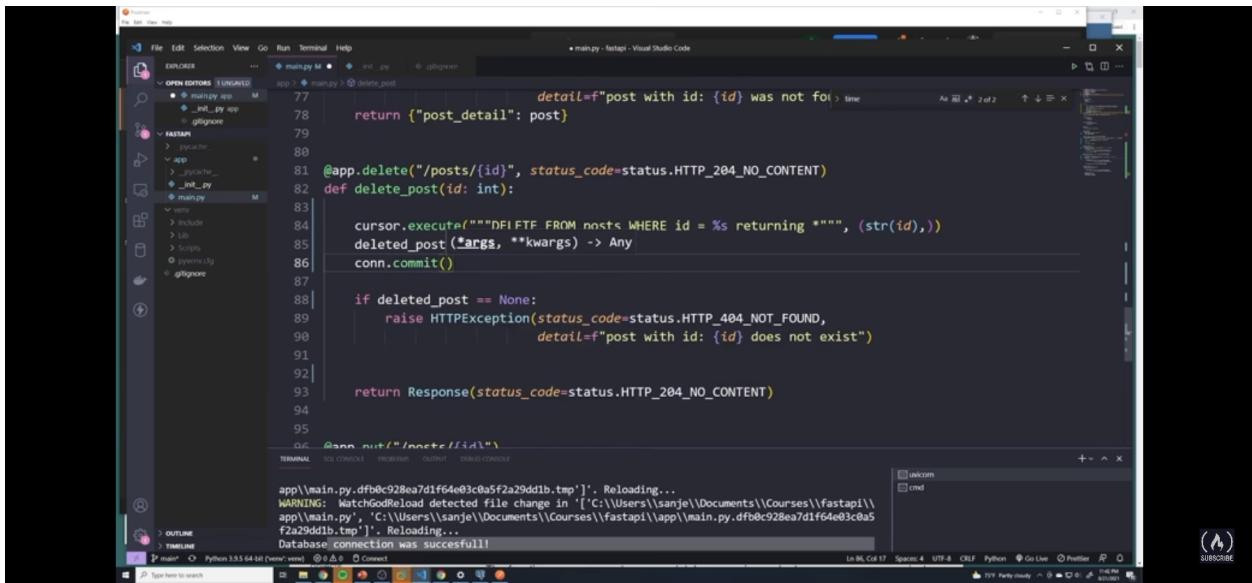
## Connecting db to Get Post specific Id

```

@app.get("/posts/{id}")
def get_post(id: str):
    cursor.execute("""SELECT * from posts WHERE id = %s """, (str(id)))
    post = cursor.fetchone()
    if not post:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"post with id: {id} was not found")
    return {"post_detail": post}

```

## Connecting Db to Delete Post



The screenshot shows the Visual Studio Code interface with the main.py file open. The code implements a delete endpoint for posts:

```

@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_post(id: int):
    cursor.execute("""DELETE FROM posts WHERE id = %s returning *""", (str(id),))
    deleted_post = args, **kwargs) -> Any
    conn.commit()

    if deleted_post == None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"post with id: {id} does not exist")

    return Response(status_code=status.HTTP_204_NO_CONTENT)

```

The terminal below shows a successful reload of the application after changes were made.

## Connecting Db to Update Post specific Id

```
main.py M main.py > update_post
96 @app.put("/posts/{id}")
97 def update_post(id: int, post: Post):
98     cursor.execute("""UPDATE posts SET title = %s, content = %s, published = %s WHERE id = %s
99     RETURNING *""", (post.title, post.content, post.published, str(id)))
100
101     updated_post = cursor.fetchone()
102     conn.commit()
103
104     if updated_post == None:
105         raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
106                             detail=f"post with id: {id} does not exist")
107
108
109     return {"data": updated_post}
110
```

TERMINAL

```
INFO:     Started server process [33924]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     127.0.0.1:52880 - "PUT /posts/1 HTTP/1.1" 200 OK
```

## Using an ORM (SQLALCHEMY)

### Object Relational Mapper(ORM)



- Layer of abstraction that sits between the database and us
- We can perform all database operations through traditional python code. No more SQL!

(A)  
SUBSCRIBE

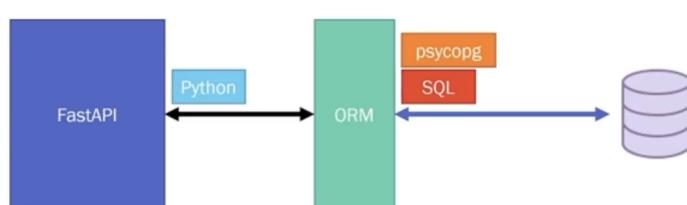
## Object Relational Mapper(ORM)



Traditional



ORM



( SUBSCRIBE

## What can ORMs Do

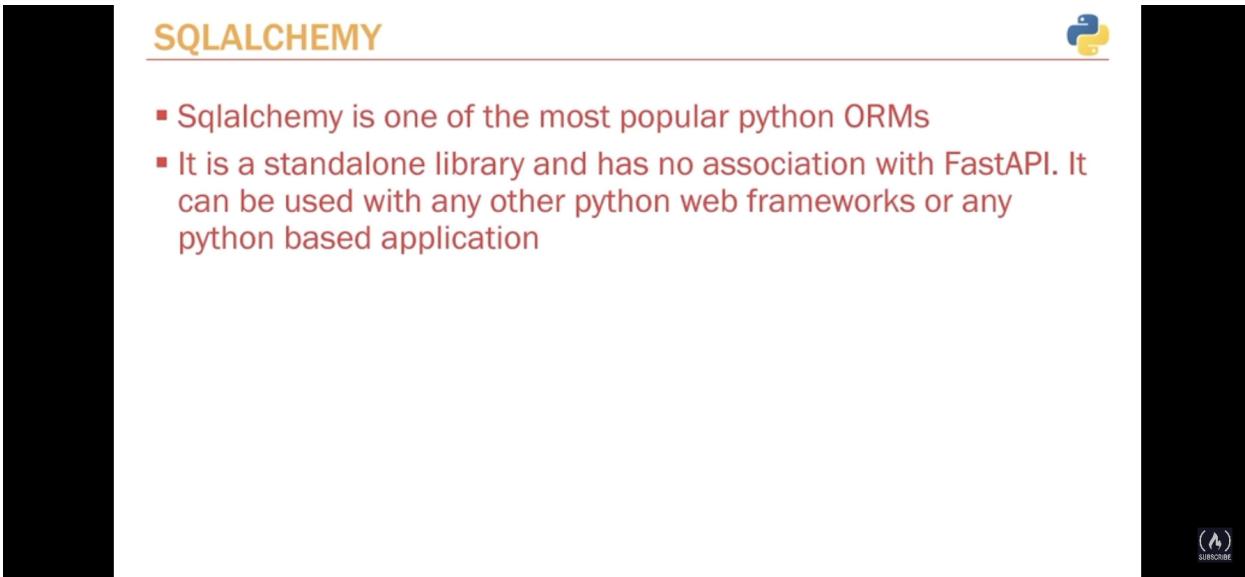


- Instead of manually defining tables in postgres, we can define our tables as python models
- Queries can be made exclusively through python code. No SQL is necessary

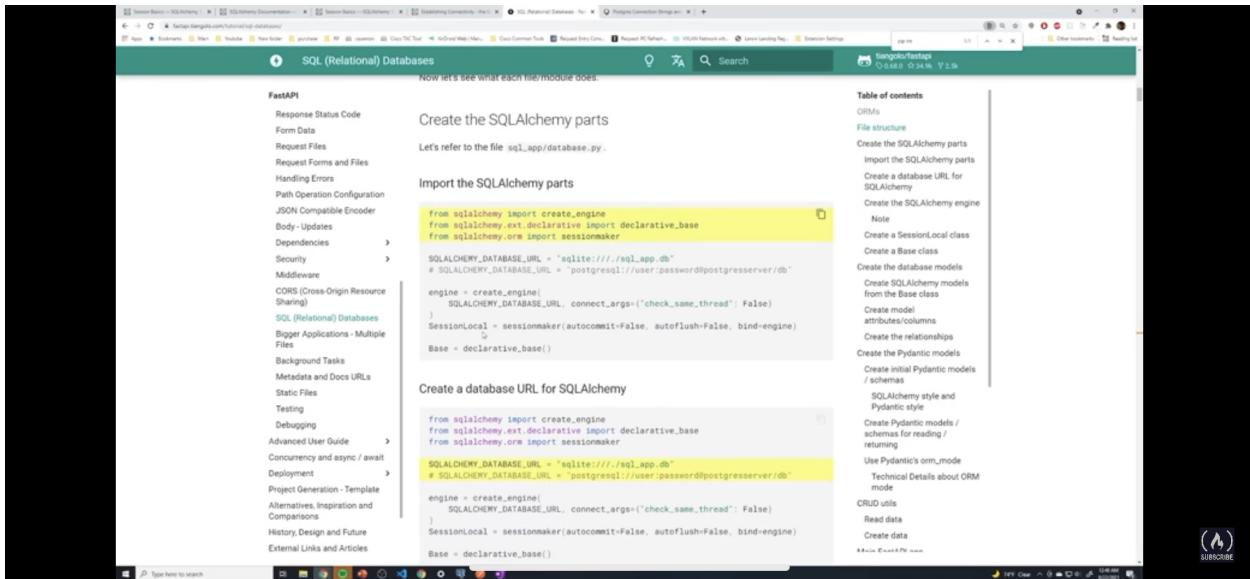
```
● ● ●  
class Post(Base):  
    __tablename__ = "posts"  
  
    id = Column(Integer, primary_key=True, index=True)  
    title = Column(String, index=True, nullable=False)  
    content = Column(String, nullable=False)  
    published = Column(Boolean)
```

```
● ● ●  
db.query(models.Post).filter(models.Post.id == id).first()
```

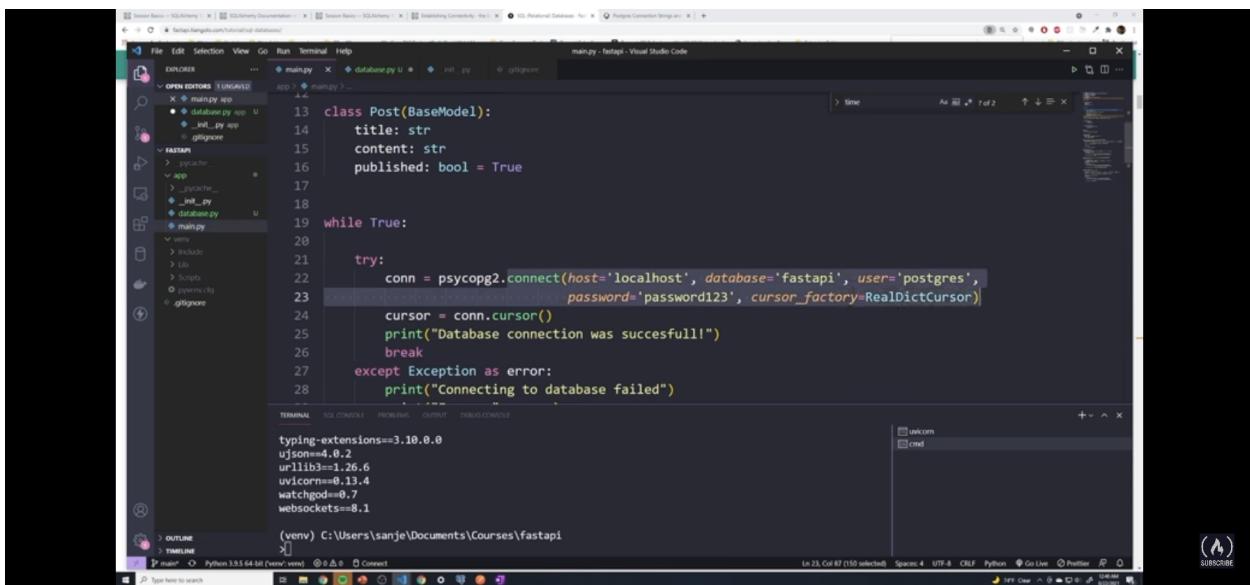
( SUBSCRIBE



A screenshot of the SQLAlchemy 1.4 Documentation page. The title bar says "SQLAlchemy 1.4 Documentation" and "Release: 1.4.23 CURRENT RELEASE | Release Date: August 18, 2021". The page contains sections for "Getting Started", "Tutorials", and "Reference Documentation". The "Reference Documentation" section is expanded, showing sub-sections like "SQLAlchemy Core", "SQLAlchemy ORM", and "SQLAlchemy 1.x Releases". A "Search terms" input field is at the top right. On the far right, there is a "SUBSCRIBE" button with a bell icon.



Before we connected on our main file



Create a `database.py` file to connect

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under the `OPEN EDITORS` tab:
  - `main.py`
  - `database.py`
  - `__init__.py`
  - `app` folder containing `__init__.py` and `models.py`
  - `fastapi` folder containing `__init__.py`, `app`, `pycache`, and `__pycache__`
  - `man.py`
  - `ignore`
  - `gitignore`
- Code Editor:** Displays `database.py` with the following code:

```
1 from sqlalchemy import create_engine
2 from sqlalchemy.ext.declarative import declarative_base
3 from sqlalchemy.orm import sessionmaker
4
5 SQLALCHEMY_DATABASE_URL = 'postgresql://postgres:password123@localhost/fastapi'
6
7 engine = create_engine(SQLALCHEMY_DATABASE_URL)
8
9 SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
10
11 Base = declarative_base()
12
```
- Terminal:** Shows the installed dependencies:

```
typing-extensions==3.10.0.0
ujson==4.0.2
urllib3==1.26.6
uvicorn==0.13.4
watchgod==0.7
websocket==8.1
```
- Output:** Shows the logs for the `unicorn` process.
- Status Bar:** Shows the file path as `(venv) C:\Users\sanje\Documents\Courses\fastapi`, the Python version as `Python 3.5.5 64-bit [venv]`, and the connection status as `Connected`.

# Creating db Models

```
from . import models
from .database import engine, SessionLocal
models.Base.metadata.create_all(bind=engine)

app = FastAPI()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

class Post(BaseModel):
    typing_extensions==3.10.0.0
    ujson==4.0.2
    urllib3==1.26.6
    uvicorn==0.13.4
    watchgod==0.7
    websockets==8.1
```

Create a model for the db

```
from sqlalchemy import Column, Integer, String, Boolean
from sqlalchemy.sql.expression import null
from .database import Base

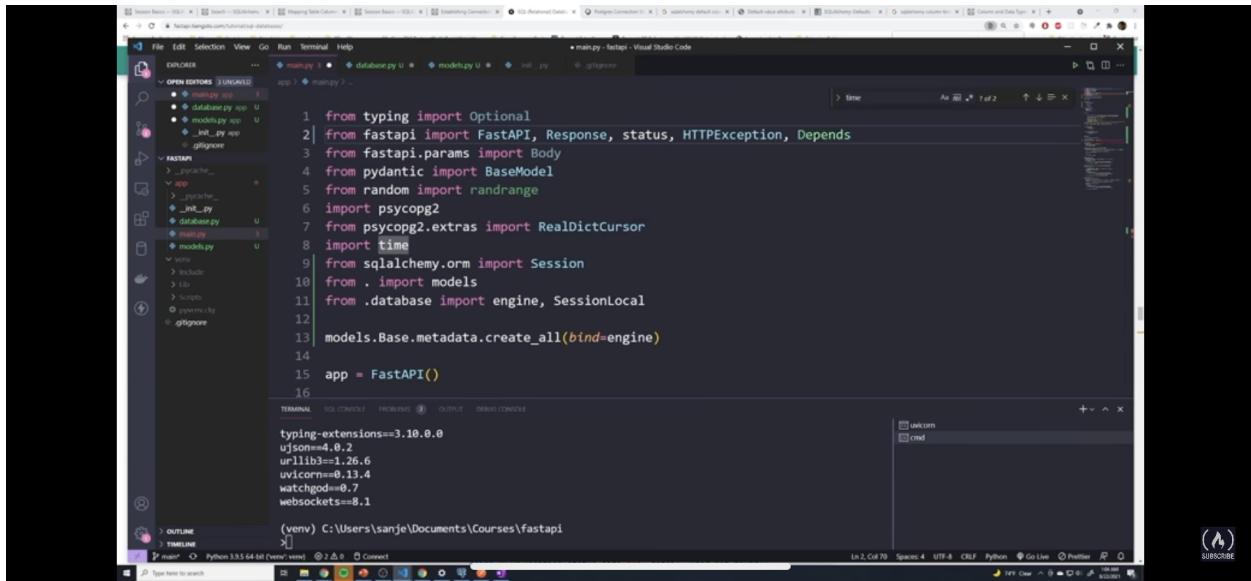
class Post(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True, nullable=False)
    title = Column(String, nullable=False)
    content = Column(String, nullable=False)
    published = Column(Boolean, default=True)

typing_extensions==3.10.0.0
ujson==4.0.2
urllib3==1.26.6
uvicorn==0.13.4
watchgod==0.7
websockets==8.1
```

## Sessions

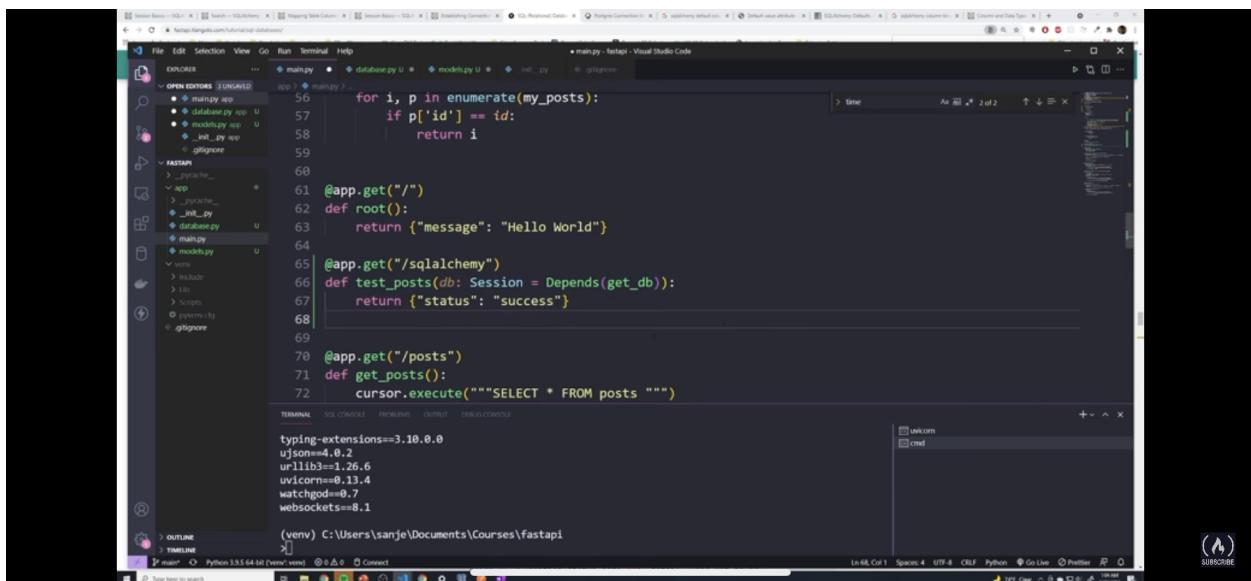
Import sessions from sql alchemy and depends method from FastAPI



```
from typing import Optional
from fastapi import FastAPI, Response, status, HTTPException, Depends
from pydantic import BaseModel
from random import randrange
import psycopg2
from psycopg2.extras import RealDictCursor
import time
from sqlalchemy.orm import Session
from . import models
from .database import engine, SessionLocal
from models import Base
Base.metadata.create_all(bind=engine)

app = FastAPI()
```

update get('/sqlAlchemy')



```
for i, p in enumerate(my_posts):
    if p['id'] == id:
        return i

@app.get("/")
def root():
    return {"message": "Hello World"}

@app.get("/sqlalchemy")
def test_posts(db: Session = Depends(get_db)):
    return {"status": "success"}

@app.get("/posts")
def get_posts():
    cursor.execute("""SELECT * FROM posts """)
```

## Clean up API

The screenshot shows the Visual Studio Code interface with the file structure for a FastAPI application. The `main.py` file contains the following code:

```
from sqlalchemy.orm import Session
from . import models
from .database import engine, SessionLocal

models.Base.metadata.create_all(bind=engine)

app = FastAPI()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

class Post(BaseModel):
    title: str
    content: str
```

The terminal output shows the application starting successfully:

```
Database connection was successful!
INFO: Started server process [21812]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

- move db related code into db file

The screenshot shows the Visual Studio Code interface with the file structure for a FastAPI application. The `database.py` file contains the following code:

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

SQLALCHEMY_DATABASE_URL = 'postgresql://postgres:password123@localhost/fastapi'

engine = create_engine(SQLALCHEMY_DATABASE_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

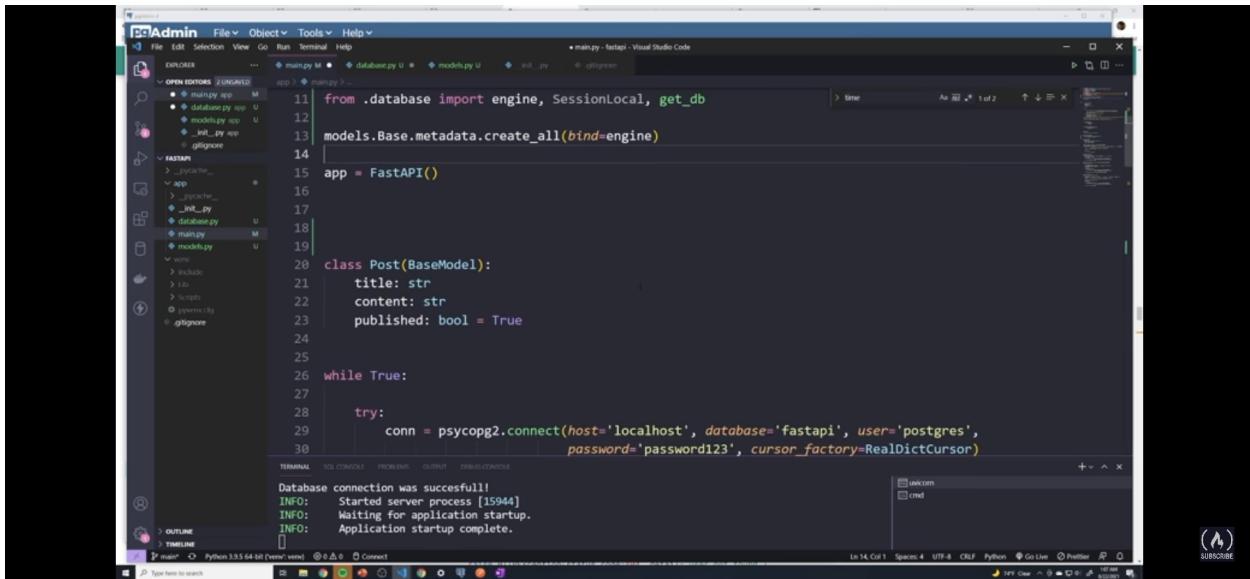
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

The terminal output shows the application starting successfully:

```
Database connection was successful!
INFO: Started server process [15944]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Import the function



```
from .database import engine, SessionLocal, get_db
models.Base.metadata.create_all(bind=engine)

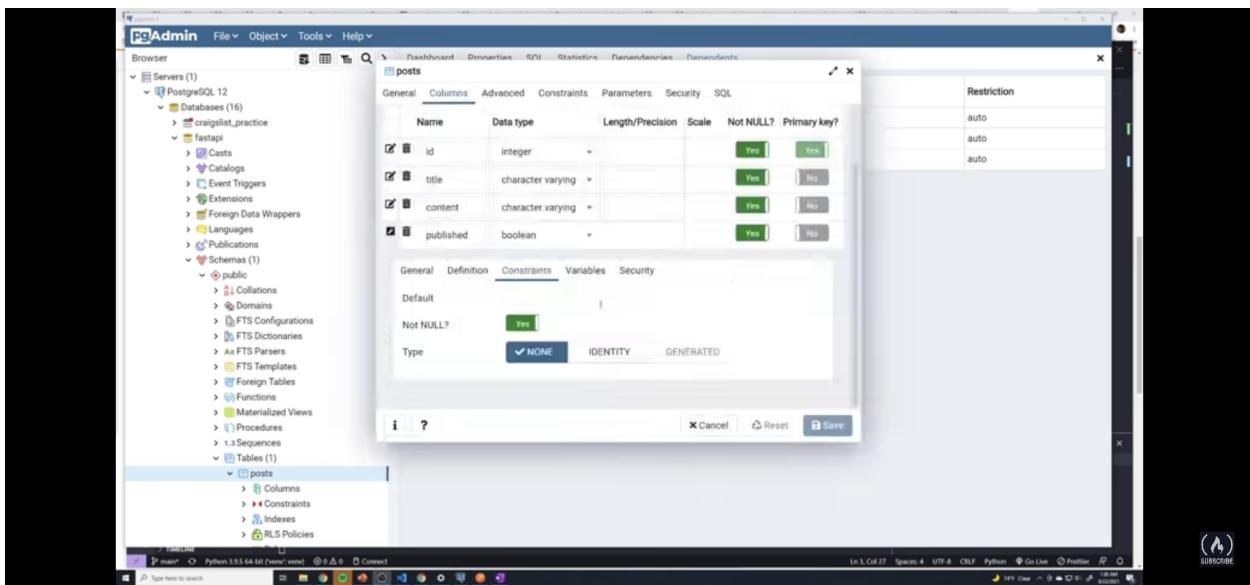
app = FastAPI()

class Post(BaseModel):
    title: str
    content: str
    published: bool = True

while True:
    try:
        conn = psycopg2.connect(host='localhost', database='fastapi', user='postgres', password='password123', cursor_factory=RealDictCursor)
        Database connection was successful!
        INFO: Started server process [15944]
        INFO: Waiting for application startup.
        INFO: Application startup complete.
    except Exception as e:
        print(f'Error connecting to PostgreSQL: {e}')
        time.sleep(1)
```

## Adding Created at field

No default set



Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Restriction
id	integer			Yes	No	auto
title	character varying			Yes	No	auto
content	character varying			Yes	No	auto
published	boolean			Yes	No	auto

Need to update our model

```

from sqlalchemy import Column, Integer, String, Boolean
from .database import Base

class Post(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True, nullable=False)
    title = Column(String, nullable=False)
    content = Column(String, nullable=False)
    published = Column(Boolean, server_default='TRUE', nullable=False)

    created_at = Column(TIMESTAMP(timezone=True), nullable=False, server_default=text('now()'))

```

Add in Created\_at functionality with defaults

```

from sqlalchemy import Column, Integer, String, Boolean
from sqlalchemy.sql.expression import text
from sqlalchemy.sql.sqltypes import TIMESTAMP

from .database import Base

class Post(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True, nullable=False)
    title = Column(String, nullable=False)
    content = Column(String, nullable=False)
    published = Column(Boolean, server_default='TRUE', nullable=False)
    created_at = Column(TIMESTAMP(timezone=True), nullable=False, server_default=text('now()'))

```

SQL alchemy has now created the correct tables

- If you print the statement created
- Looks like sql

```
main.py M - fastapi - Visual Studio Code
File Edit Selection View Go Run Terminal Help
main.py X database.py models.py init_py
app > main.py > test_posts
52 @app.get("/")
53 def root():
54     return {"message": "Hello World"}
55
56
57
58 @app.get("/sqlalchemy")
59 def test_posts(db: Session = Depends(get_db)):
60
61     posts = db.query(models.Post)
62     print(posts)
63     return {"data": "successfull"}
64
65
66
67 @app.get("/posts")
68
69
70
71
72
73
74
75
76
77
```

```
INFO: Application startup complete.
SELECT posts.id AS posts_id, posts.title AS posts_title, posts.content AS posts_content, posts.published AS posts_published, posts.created_at AS posts_created_at
FROM posts
INFO: 127.0.0.1:57912 - "GET /sqlalchemy HTTP/1.1" 200 OK
```

## Update methods to use SQLALCHEMY

Get

```
main.py M - fastapi - Visual Studio Code
File Edit Selection View Go Run Terminal Help
main.py X database.py models.py init_py
app > main.py > create_posts
64     return {"data": "successfull"}
65
66
67 @app.get("/posts")
68 def get_posts(db: Session = Depends(get_db)):
69     # cursor.execute("SELECT * FROM posts")
70     # posts = cursor.fetchall()
71     posts = db.query(models.Post).all()
72     return {"data": posts}
73
74
75 @app.post("/posts", status_code=status.HTTP_201_CREATED)
76 def create_posts(post: Post):
77     cursor.execute("""INSERT INTO posts (title, content, published) VALUES (%s, %s, %s) RETURNING *""",
```

```
INFO: Waiting for application startup.
INFO: Application startup complete.
WARNING: Watchdog detected file change in '[C:\Users\sanje\Documents\Courses\fastapi\app\main.py, df9c0c28be0f66e40d05f2a299db1b.tmp]'. Reloading..
WARNING: WatchdogReload detected file change in '[C:\Users\sanje\Documents\Courses\fastapi\app\main.py, df9c0c28be0f66e40d05f2a299db1b.tmp]'. Reloading..
Database connection was successful.
INFO: Started server process [24416]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Post

```
main.py 2 M  database.py  models.py  init_.py  __pycache__  
69     # cursor.execute("""SELECT * FROM posts """)  
70     # posts = cursor.fetchall()  
71     posts = db.query(models.Post).all()  
72     return {"data": posts}  
73  
74  
75     @app.post("/posts", status_code=status.HTTP_201_CREATED)  
76     def create_posts(post: Post, db: Session = Depends(get_db)):  
77         # cursor.execute("""INSERT INTO posts (title, content, published) VALUES (%s, %s, %s)  
78         # RETURNING * """,  
79         # (post.title, post.content, post.published))  
80         # new_post = cursor.fetchone()  
81         # conn.commit()  
82  
83         new_post = models.Post(**post.dict())  
84         db.add(new_post)  
85         db.commit()  
86         db.refresh(new_post)  
87  
88         return {"data": new_post}  
89  
90  
91     @app.get("/posts/{id}")  
92     def get_post(id: str):
```

## Get(id)

```
main.py 2 M  database.py  models.py  init_.py  __pycache__  
85     db.commit()  
86     db.refresh(new_post)  
87  
88     return {"data": new_post}  
89  
90  
91     @app.get("/posts/{id}")  
92     def get_post(id: int, db: Session = Depends(get_db)):  
93         # cursor.execute("""SELECT * from posts WHERE id = %s """, (str(id),))  
94         # post = cursor.fetchone()  
95         post = db.query(models.Post).filter(models.Post.id == id).first()  
96         print(post)  
97  
98         if not post:  
99             raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
```

INFO: Waiting for application startup.  
INFO: Application startup complete.  
SELECT posts.id AS posts\_id, posts.title AS posts\_title, posts.content AS posts\_content, posts.published AS posts\_published, posts.created\_at AS posts\_created\_at  
FROM posts  
WHERE posts.id = %(id\_1)s  
Fatal Python error: Py\_CheckRecursiveCall: Cannot recover from stack overflow.  
Python runtime state: initialized

Thread 0x00005b48 (most recent call first):  
File "C:\Users\sanje\AppData\Local\Programs\Python\Python39\lib\concurrent\futures\thread.py", li

## Delete(id)

The screenshot shows the Visual Studio Code interface with the main.py file open in the editor. The code implements a DELETE endpoint for posts:

```
102 @app.delete("/posts/{id}", status_code=status.HTTP_204_NO_CONTENT)
103 def delete_post(id: int, db: Session = Depends(get_db)):
104     cursor.execute(
105         f"DELETE FROM posts WHERE id = %s returning *", (str(id),))
106     deleted_post = cursor.fetchone()
107     conn.commit()
108     post = db.query(models.Post).filter(models.Post.id == id)
109
110     if post.first() == None:
111         raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
112                             detail=f"post with id: {id} does not exist")
113
114     post.delete(synchronize_session=False)
115     db.commit()
116
117     return Response(status_code=status.HTTP_204_NO_CONTENT)
```

The terminal shows the application starting up:

```
INFO: Started server process [29364]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

## Update

The screenshot shows the Visual Studio Code interface with the main.py file open in the editor. The code implements a PUT endpoint for posts:

```
123 @app.put("/posts/{id}")
124 def update_post(id: int, post: Post, db: Session = Depends(get_db)):
125
126     cursor.execute(
127         f"UPDATE posts SET title = %s, content = %s, published = %s WHERE id = %s
128         RETURNING *",
129         (post.title, post.content, post.published, str(id)))
130
131     updated_post = cursor.fetchone()
132     conn.commit()
```

```

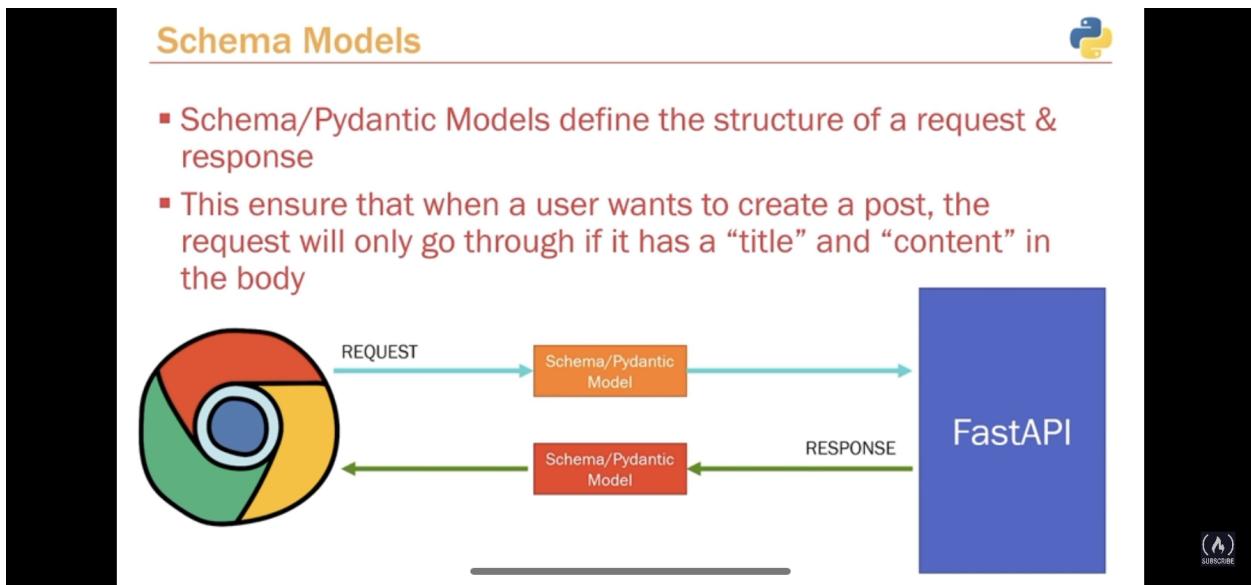
126     # cursor.execute("""UPDATE posts SET title = %s, content = %s, time
127     # RETURNING """,
128     #         (post.title, post.content, post.published, str(id)))
129
130     # updated_post = cursor.fetchone()
131     # conn.commit()
132
133     post_query = db.query(models.Post).filter(models.Post.id == id)
134
135     post = post_query.first()
136
137     if post == None:
138         raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
139                             detail=f"post with id: {id} does not exist")
140
141     post_query.update(updated_post.dict(), synchronize_session=False)
142
143     db.commit()
144
145
146     return {"data": post_query.first()}

```

File "C:/Users/sanje/Downloads/Courses/fastapi/app/main.py", line 140, in update\_post  
post\_query.update(post.dict(), synchronize\_session=False)  
AttributeError: 'Post' object has no attribute 'dict'

## Schema Models

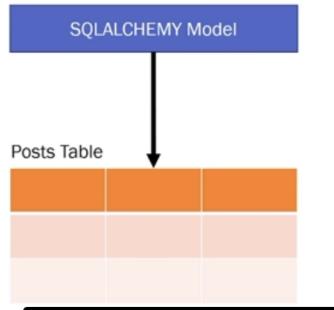
We don't want to give the client the freedom to send whatever data they want



# SQLALCHEMY Models



- Responsible for defining the columns of our “posts” table within postgres
  - Is used to query, create, delete, and update entries within the database



A small black square containing a white 'S' shape, with the word 'SUBSCRIBE' written in white capital letters below it.

## Schema folder

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure with files like `main.py`, `database.py`, `models.py`, `util.py`, and `gunicorn`.
- Code Editor:** Displays the `main.py` file content for a FastAPI application. The code includes imports for `psycopg2.extras`, `RealDictCursor`, `time`, `Session`, `mode`, and `models`. It sets up the `FastAPI` application and defines a `Post` model with fields `title`, `content`, and `published` (set to `True` by default). A `while True` loop is present at the bottom.
- Terminal:** Shows application startup logs:

```
INFO: Application startup complete.  
INFO: 127.0.0.1:54958 - "PUT /posts/1 HTTP/1.1" 200 OK  
INFO: 127.0.0.1:65135 - "PUT /posts/1234234 HTTP/1.1" 404 Not Found
```
- Status Bar:** Shows the Python version (`Python 3.5.5 64-bit (v3.5.5:17f5b0e43d, Mar 10 2018, 10:44:34) [MSC v.1900 64 bit (AMD64)]`), the current file (`main.py`), and other system information.

De-clutter the main file

```
1 class Post(BaseModel):
2     title: str
3     content: str
4     published: bool = True
```

INFO: Started server process [15948]  
INFO: Waiting for application startup.  
INFO: Application startup complete.

## Import schemas

```
8 import time
9 from sqlalchemy.orm import Session
10 from sqlalchemy.sql.functions import mode
11 from . import models, schemas
12 from .database import engine, get_db
```

Now it's schemas.Post for the class

```
119 @app.put('/posts/{id}')
120 def update_post(id: int, updated_post: schemas.Post, db: Session = Depends(get_db)):
121
122     # cursor.execute("""UPDATE posts SET title = %s, content = %s, published = %s WHERE id
123     # RETURNING *"""),
124
125     # return {"id": id, "title": updated_post.title, "content": updated_post.content,
126     # "published": updated_post.published}
```

## Creating new Schemas for Requests

- you could create loads of individual schemas

A screenshot of Visual Studio Code showing the code editor and terminal. The code editor displays three files: `main.py`, `schemas.py`, and `database.py`. The `schemas.py` file contains Pydantic schema definitions:`from pydantic import BaseModel

class Post(BaseModel):
 title: str
 content: str
 published: bool = True

class CreatePost(BaseModel):
 title: str
 content: str
 published: bool = True

class UpdatePost(BaseModel):
 published: bool`The terminal shows application startup logs:`INFO: Started server process [21124]
INFO: Waiting for application startup.
INFO: Application startup complete.`

Or use inheritance from a base schema

A screenshot of Visual Studio Code showing the code editor and terminal. The code editor displays three files: `main.py`, `schemas.py`, and `database.py`. The `schemas.py` file contains Pydantic schema definitions:`from pydantic import BaseModel

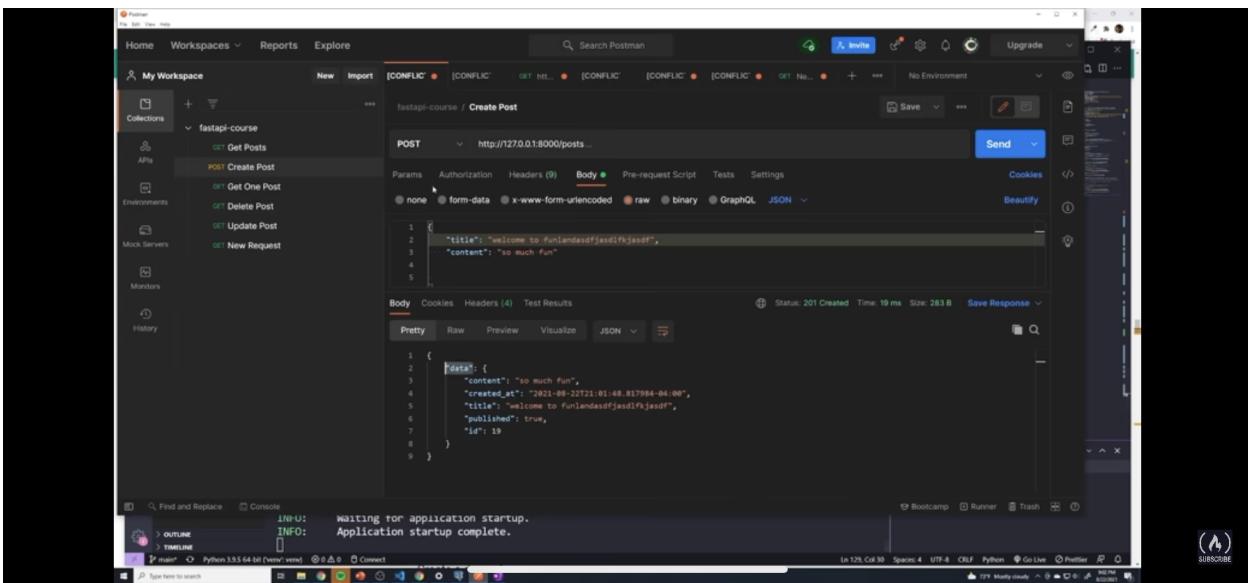
class PostBase(BaseModel):
 title: str
 content: str
 published: bool = True

class PostCreate(PostBase):
 pass`The terminal shows application startup logs:`INFO: Started server process [21124]
INFO: Waiting for application startup.
INFO: Application startup complete.`

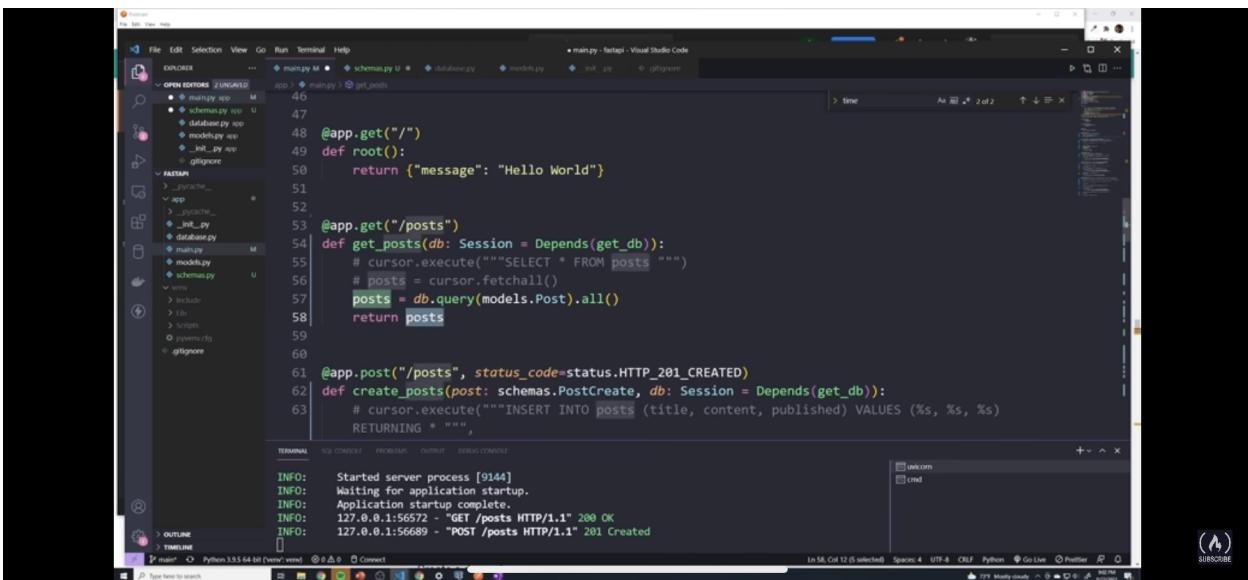
```
uvicorn > create_posts
@app.post("/posts", status_code=status.HTTP_201_CREATED)
def create_posts(post: schemas.PostCreate, db: Session = Depends(get_db)):
    # cursor.execute("""INSERT INTO posts (title, content, published) VALUES (
    RETURNING * """)
```

# Creating schema's for responses

Remove returning a dict for all (fast api auto-serialises)



- this removes data : {..}
  - Only return posts



Now is more clean

The screenshot shows the Postman application interface. In the left sidebar, there's a collection named 'fastapi-course' containing several items: 'Get Posts', 'POST Create Post', 'Get One Post', 'Delete Post', 'Update Post', and 'New Request'. The 'POST Create Post' item is selected. The main workspace shows a POST request to 'http://127.0.0.1:8000/posts...'. The 'Body' tab is selected, showing a JSON payload:

```
1 {
2     "title": "Welcome to FullstackFastAPI",
3     "content": "so much fun"
4 }
```

The 'Test Results' tab shows the response status: 201 Created. The 'Pretty' tab displays the response body:

```
1 {
2     "content": "so much fun",
3     "created_at": "2021-08-22T21:02:31.248Z0-04:00",
4     "title": "Welcome to FullstackFastAPI",
5     "published": true,
6     "id": 20
7 }
```

At the bottom, the terminal window shows application logs:

```
INFO: Application startup complete.
INFO: 127.0.0.1:54531 - "POST /posts HTTP/1.1" 201 Created
```

## Defining our response model

Previously returned all data from response

The screenshot shows the Postman application interface, similar to the previous one but with a few changes. The 'Body' tab now includes an additional field 'id': 20 at the end of the JSON object. The 'Pretty' tab shows the response body with this additional field included.

In our model - we only want to return certain info to the client

A screenshot of Visual Studio Code showing a code editor with Python code and a terminal window below it. The code editor has several tabs open, including `main.py`, `schemas.py`, `database.py`, `models.py`, `fastapi`, and `ignore`. The `main.py` tab shows the following code:

```
12     pass
13
14
15 class Post(BaseModel):
16     title: str
17     content: str
18     published: bool
19
20     class Config:
21         orm_mode = True
```

The terminal window shows an error message:

```
File "c:/users/sanje/documents/courses/fastapi/venv/lib/site-packages/fastapi/routing.py", line 2
27, in app
    response_data = await serialize_response(
  File "c:/users/sanje/documents/courses/fastapi/venv/lib/site-packages/fastapi/routing.py", line 1
30, in serialize_response
    raise ValidationError(error, field.type_)
pydantic.error_wrappers.ValidationError: 1 validation error for Post
response
    value is not a valid dict (type=type_error.dict)
[]
```

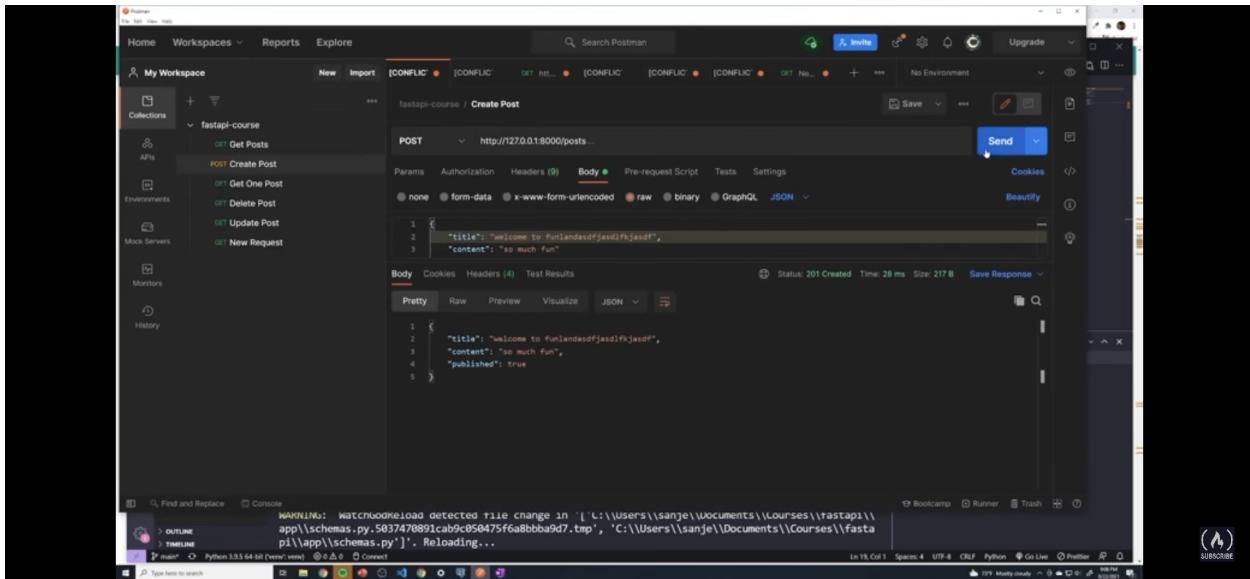
A screenshot of Visual Studio Code showing a code editor with Python code and a terminal window below it. The code editor has several tabs open, including `main.py`, `schemas.py`, `database.py`, `models.py`, `fastapi`, and `ignore`. The `main.py` tab shows the following code:

```
53 @app.get("/posts")
54 def get_posts(db: Session = Depends(get_db)):
55     # cursor.execute("""SELECT * FROM posts""")
56     # posts = cursor.fetchall()
57     posts = db.query(models.Post).all()
58     return posts
59
60 @app.post("/posts", status_code=status.HTTP_201_CREATED, response_model=schemas.Post)
61 def create_posts(post: schemas.PostCreate, db: Session = Depends(get_db)):
62     # cursor.execute("""INSERT INTO posts (title, content, published) VALUES (%s, %s, %s)
63     # RETURNING * """,
64     #                 (post.title, post.content, post.published))
65     # new_post = cursor.fetchone()
66
67     # conn.commit()
68
69     new_post = models.Post(**post.dict())
70
71     db.add(new_post)
72     db.commit()
73
74     return new_post
```

The terminal window shows the application starting up successfully:

```
app\schemas.py']. Reloading...
Database connection was successful!
INFO: Started server process [26956]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Now they only see limited info



Or add other fields to return

