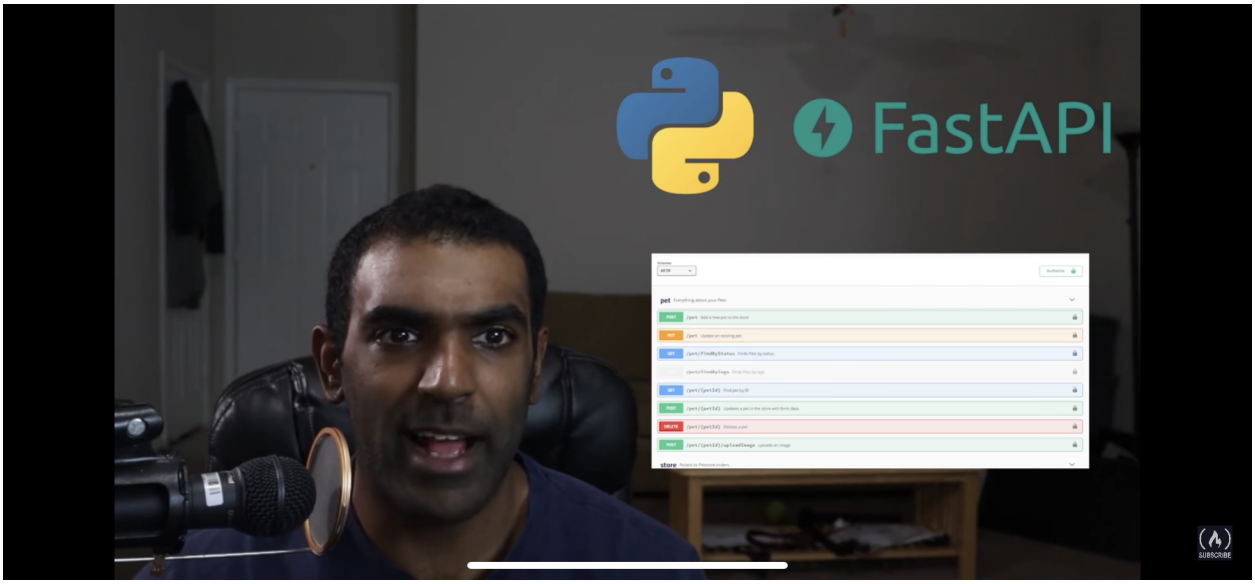
Python API Development Course (Part 1)

https://youtu.be/0sOvCWFmrtA

FastAPI

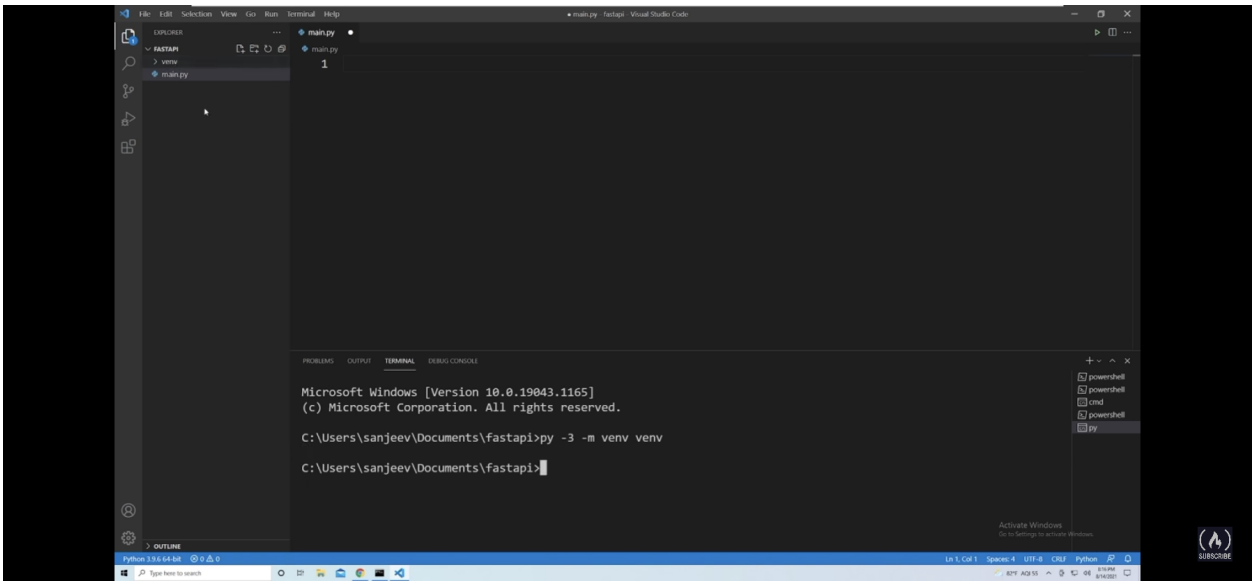


Development Environment Setup

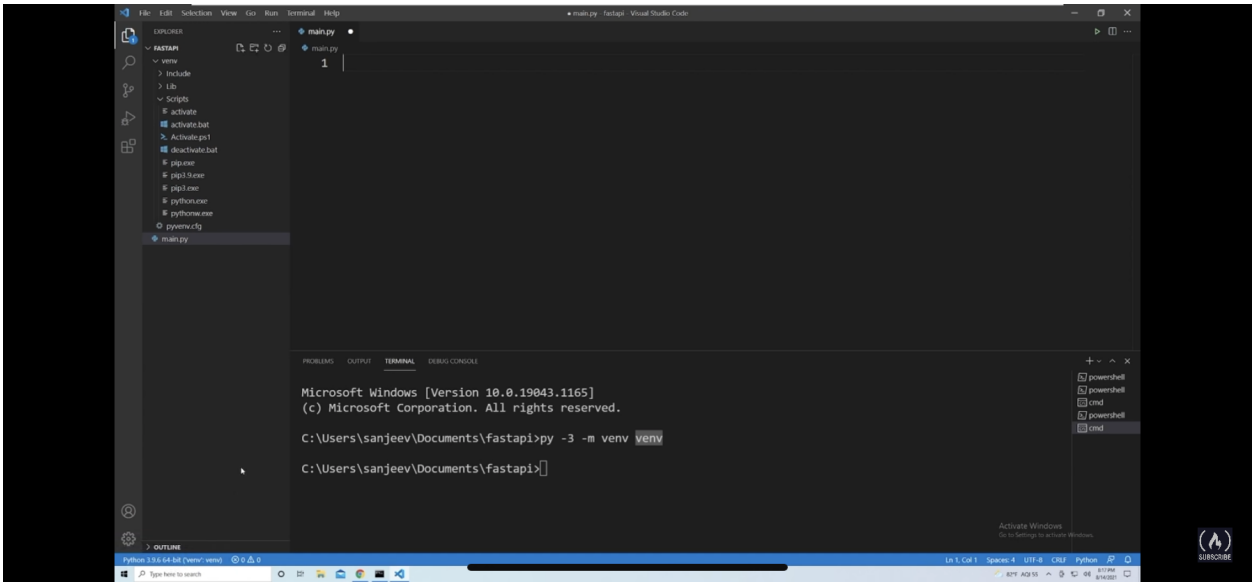
- Install Python(version 3)
- Install and Configure VSCode
- Setup Virtual Environment

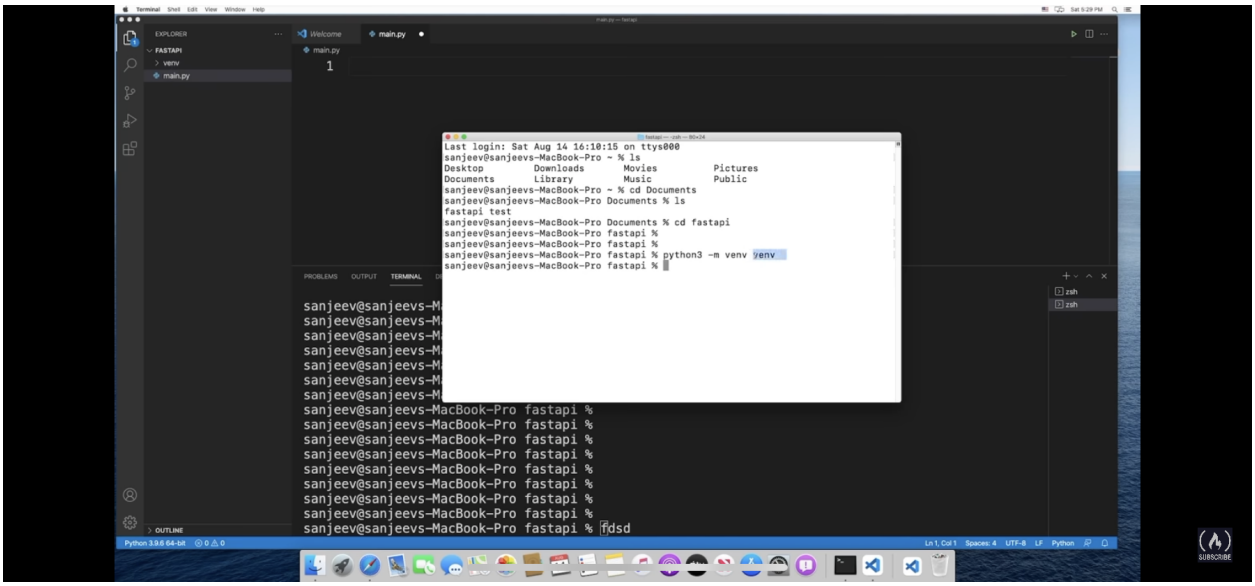
Setting Virtual environments

Python Virtual Environments

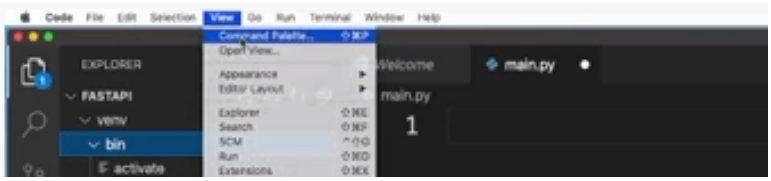

Microsoft Windows [Version 10.0.19043.1165]
(c) Microsoft Corporation. All rights reserved.
C:\Users\sanjeev\Documents\fastapi>py -3 -m venv venv
C:\Users\sanjeev\Documents\fastapi>

Enter v env


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>

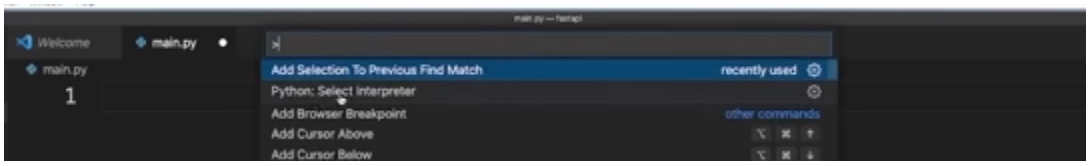


Selecting an interpreter

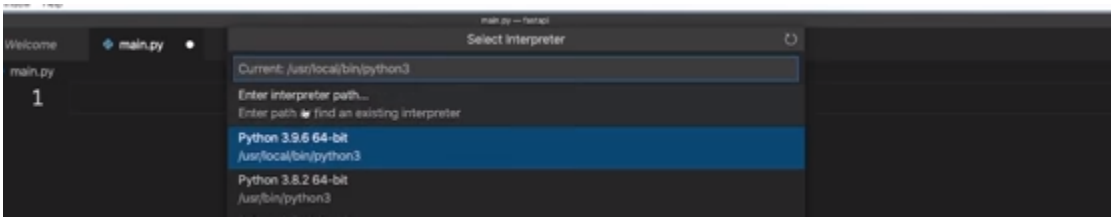


view > command palette

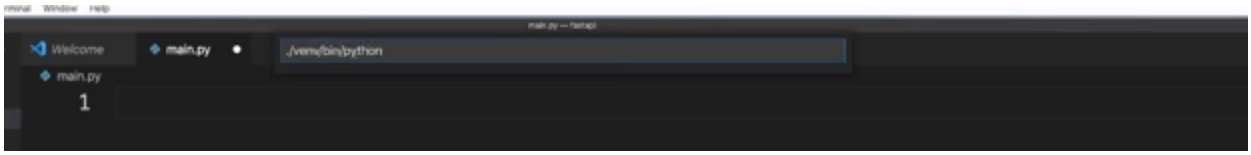
- select python interpreter

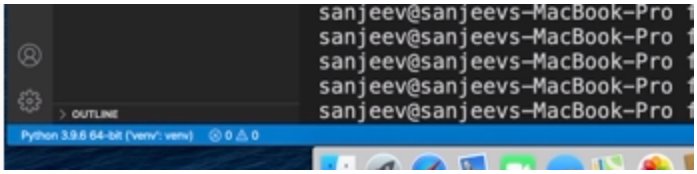


- currently using global one

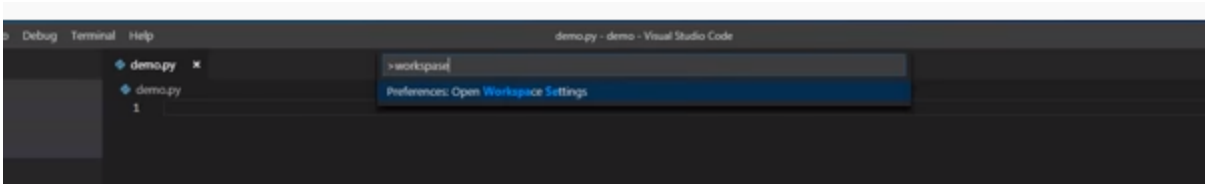


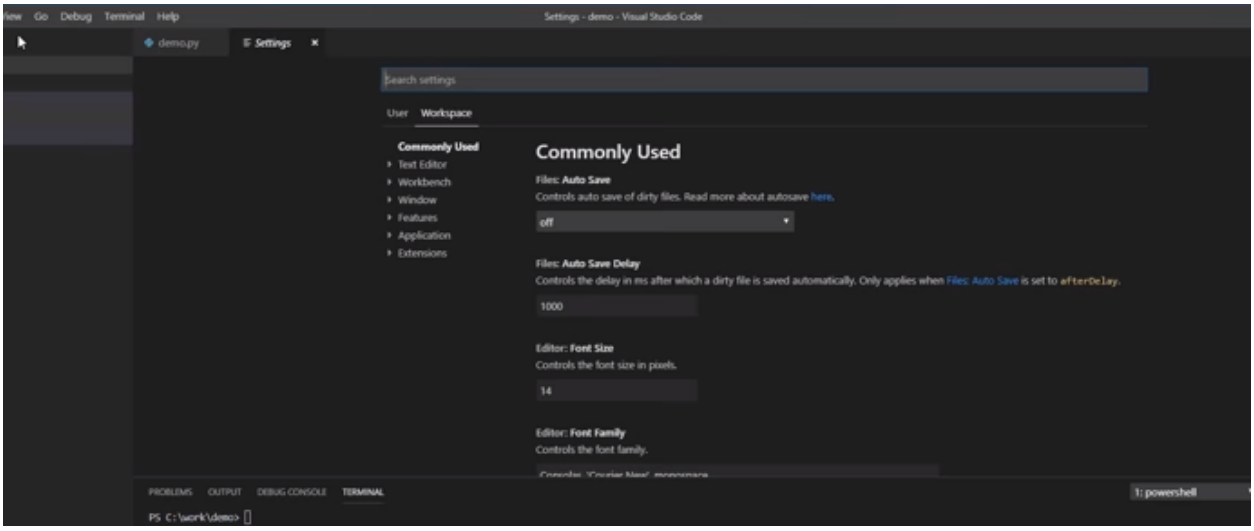
- pass in our directory to our own env.



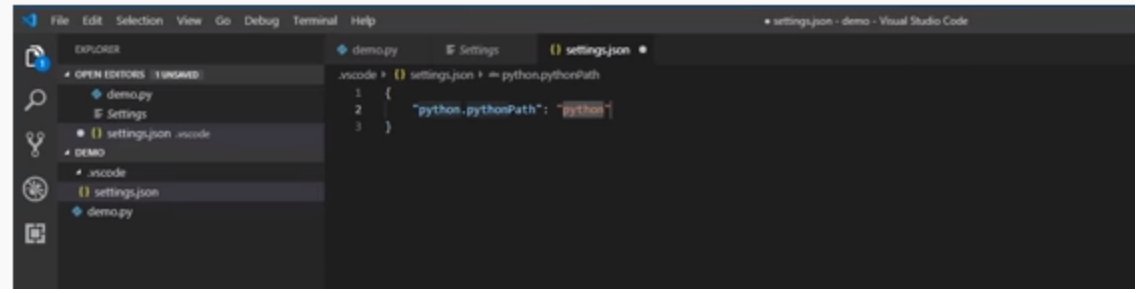


Alternate method



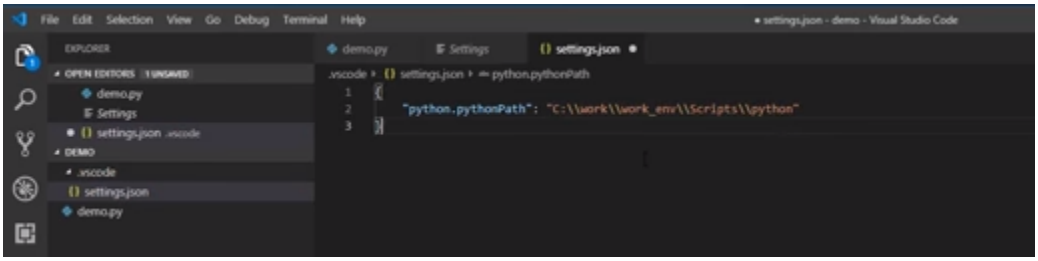


open json view on right hand toggle

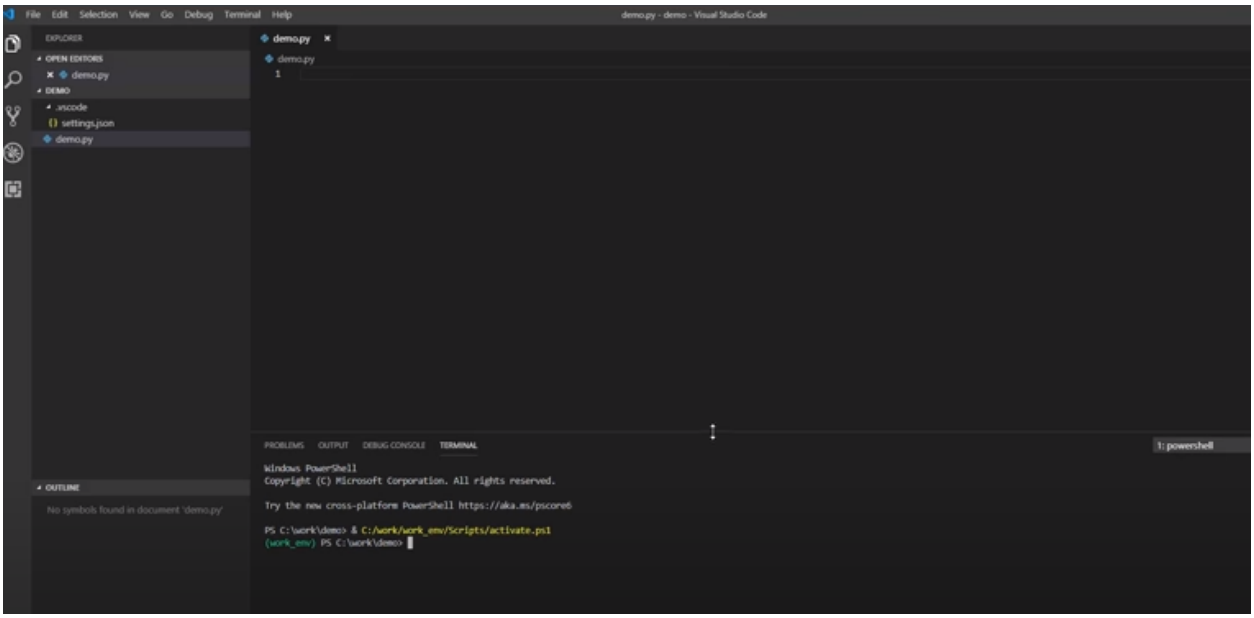


add path to python .exe

- remember to add \\



work env should now be entered

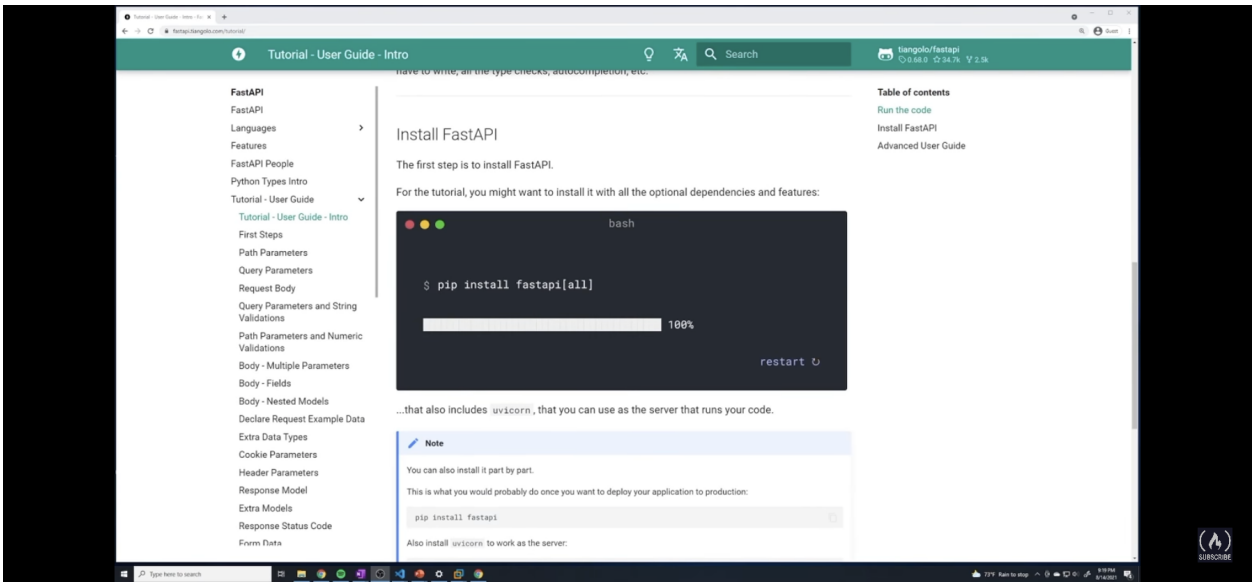


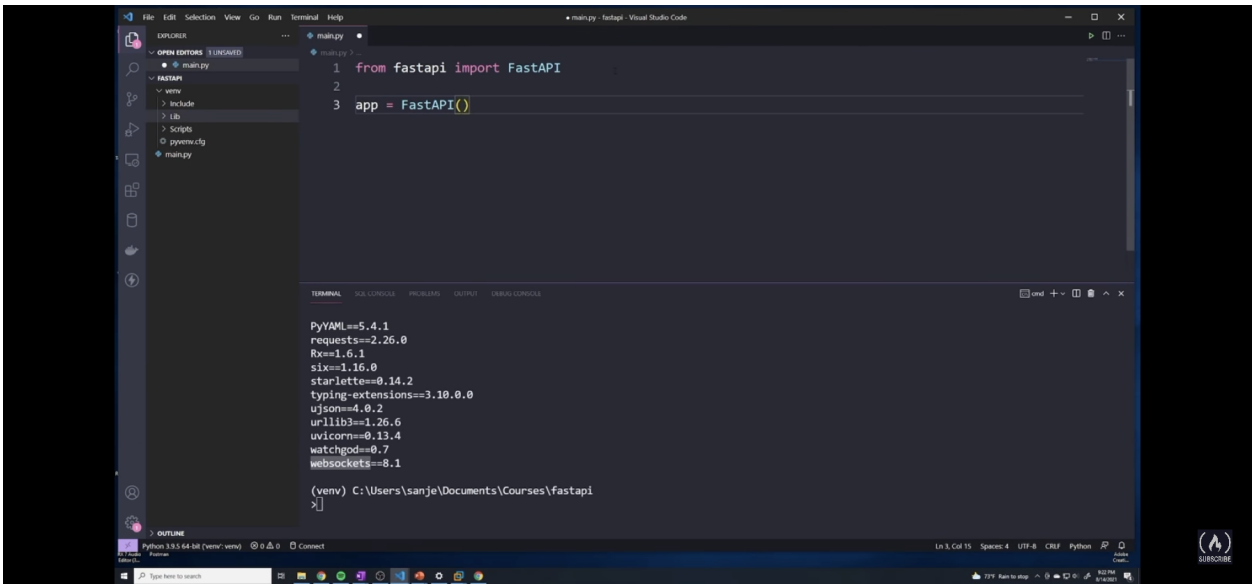
Activate env by navigating to the **Scripts** folder of the actiavte.bat file

- (.venv) C:/ … will come up

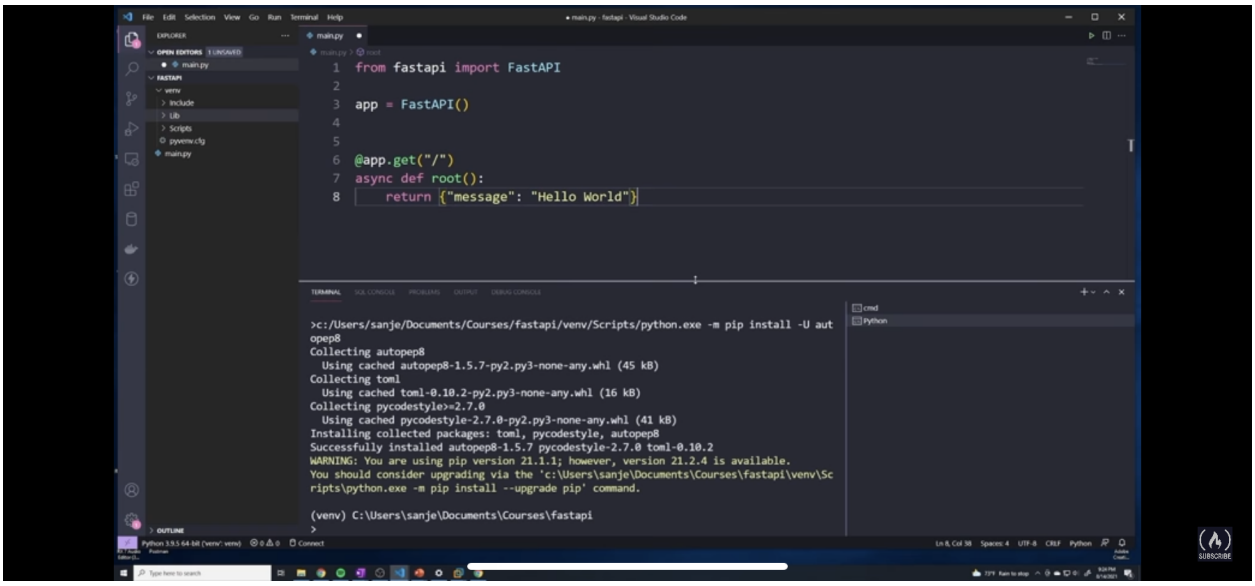
Importing Fast API

Look at documentation



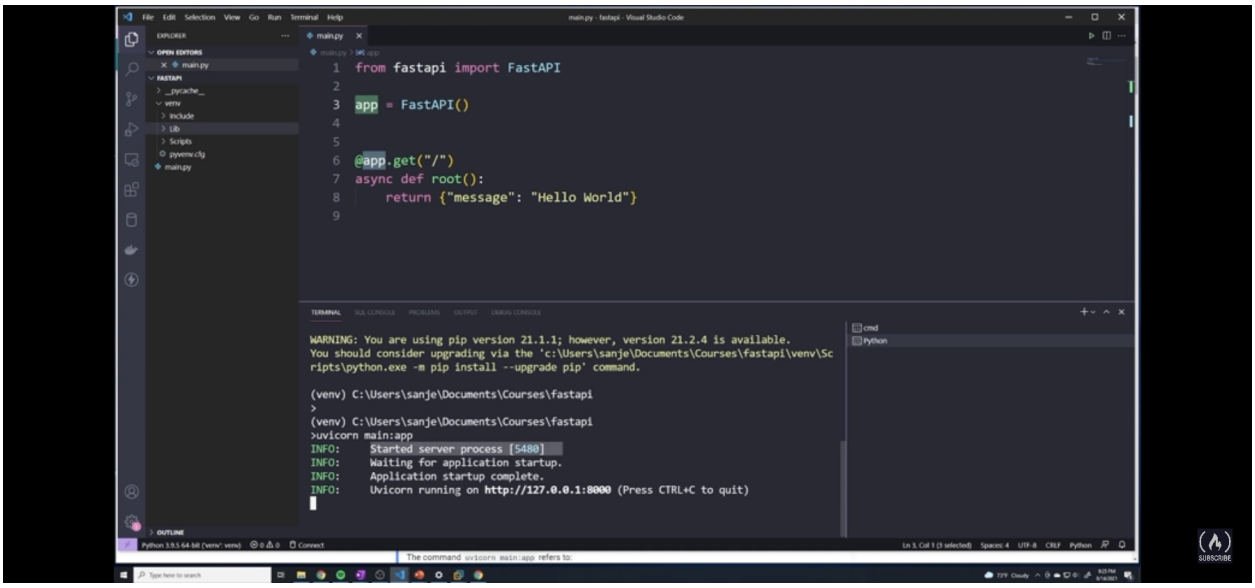


Create an object from the imported FastAPI class

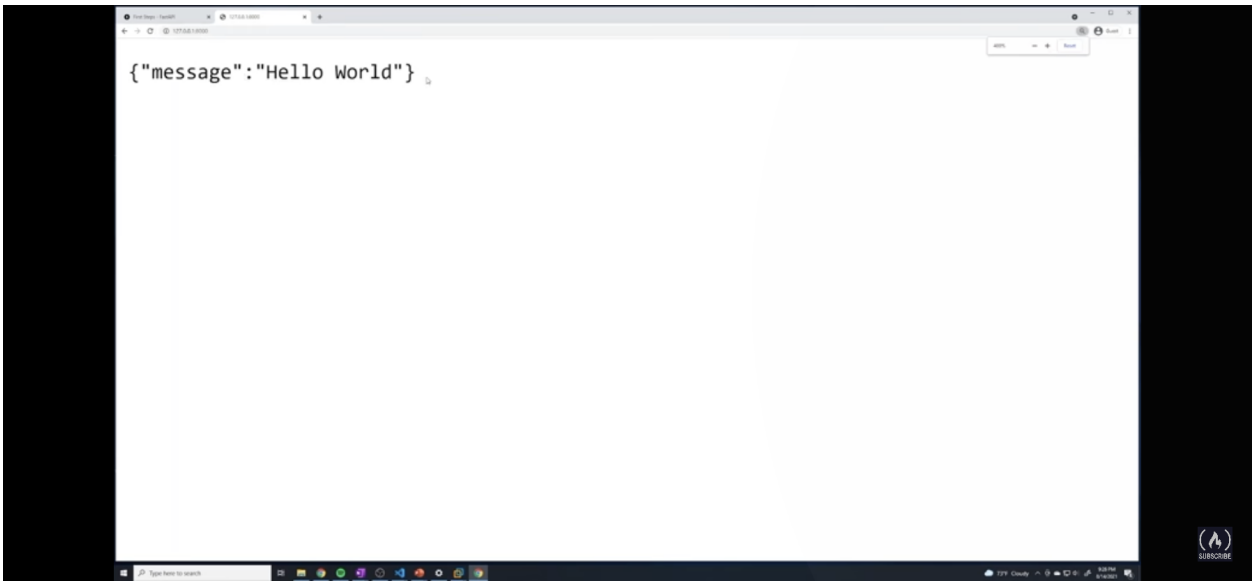


Uvicorn is installed already

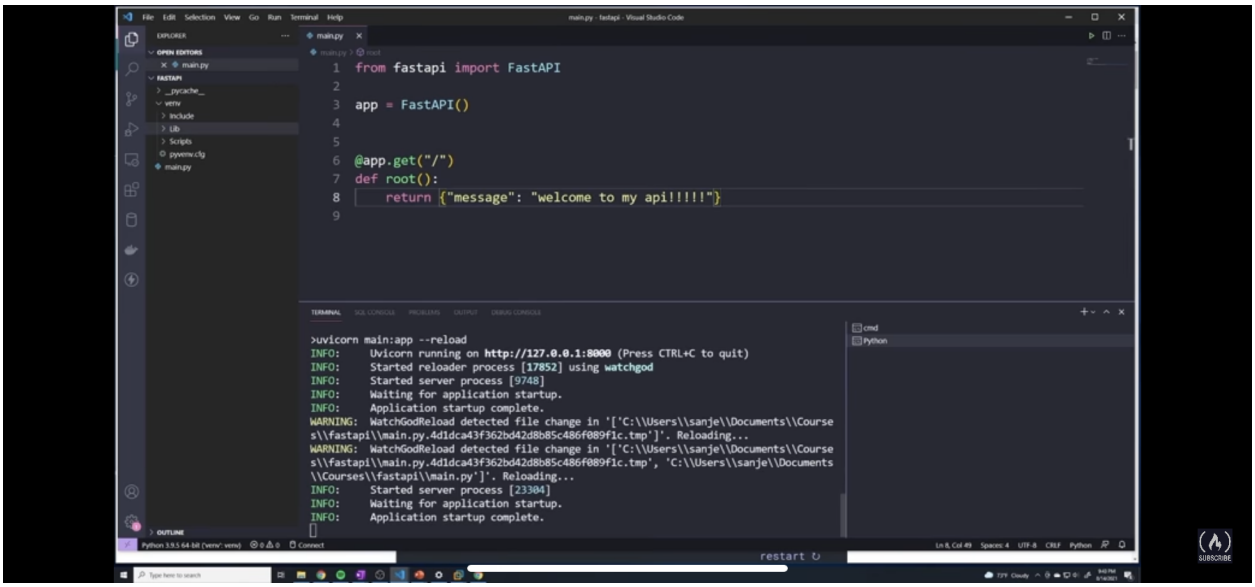
Call uvicorn main:app



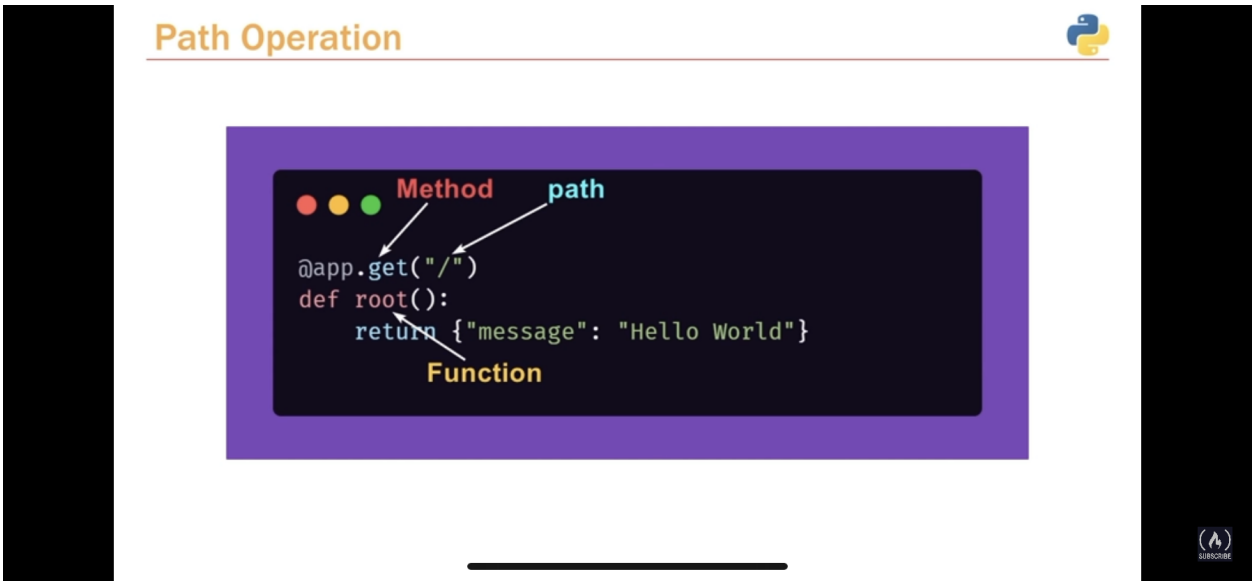
Program will start and be accessible on that port



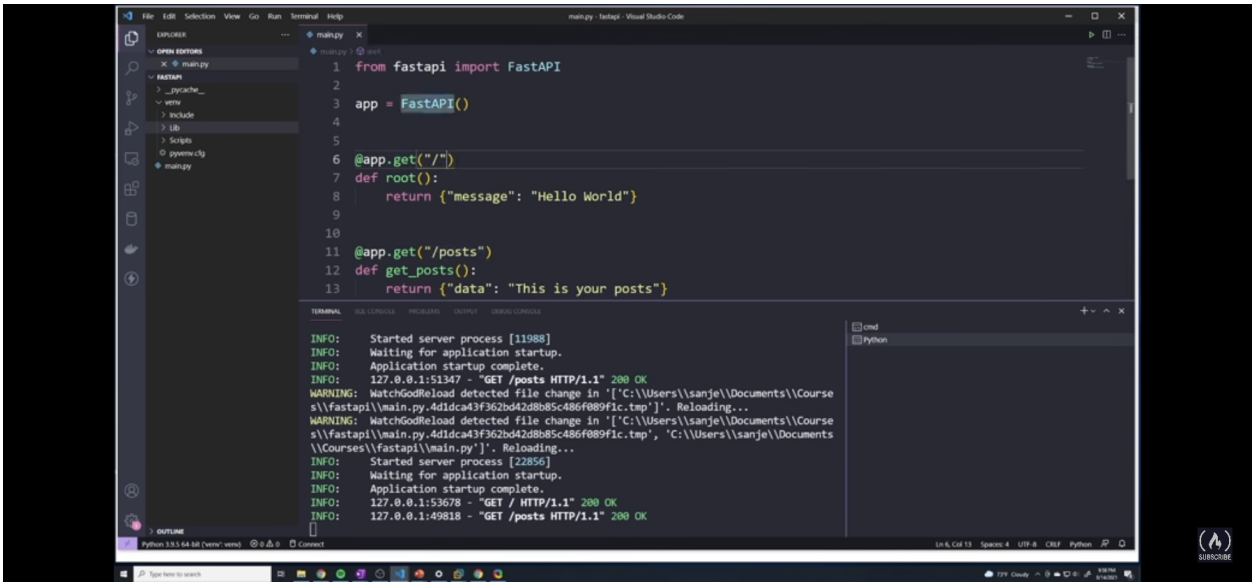
Use —reload flag to auto update server



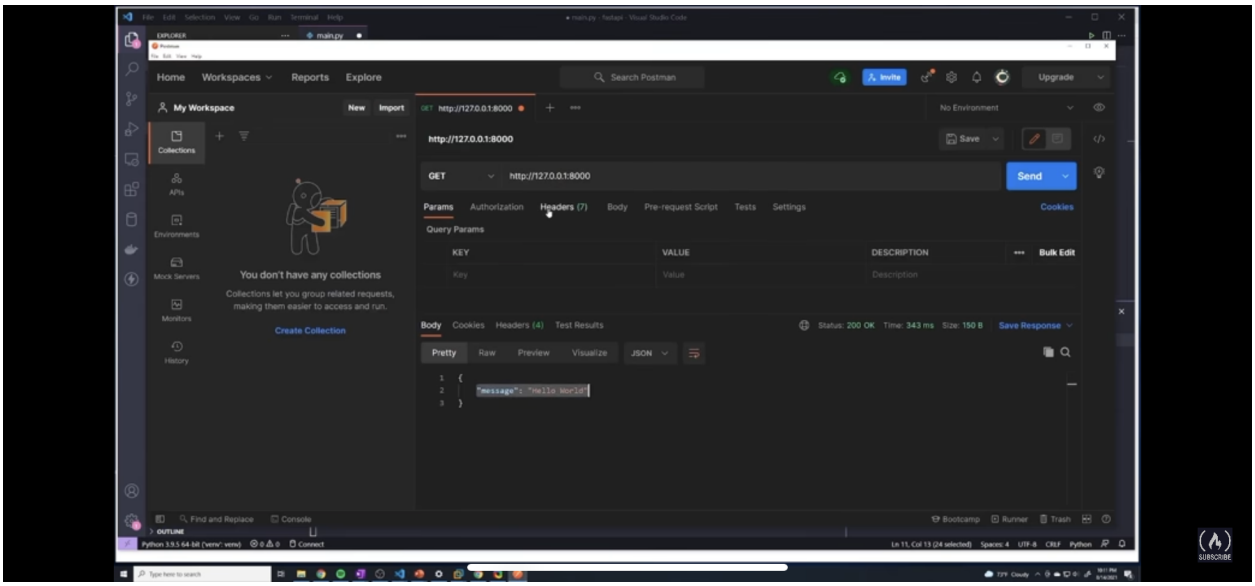
Overview of http methods

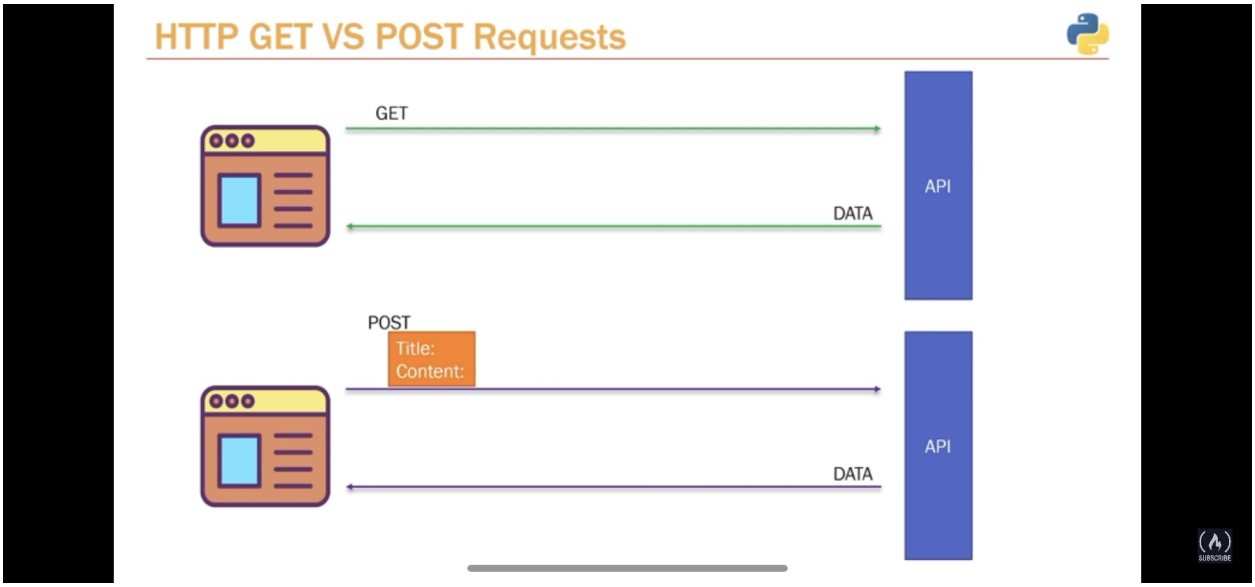


Adding different methods



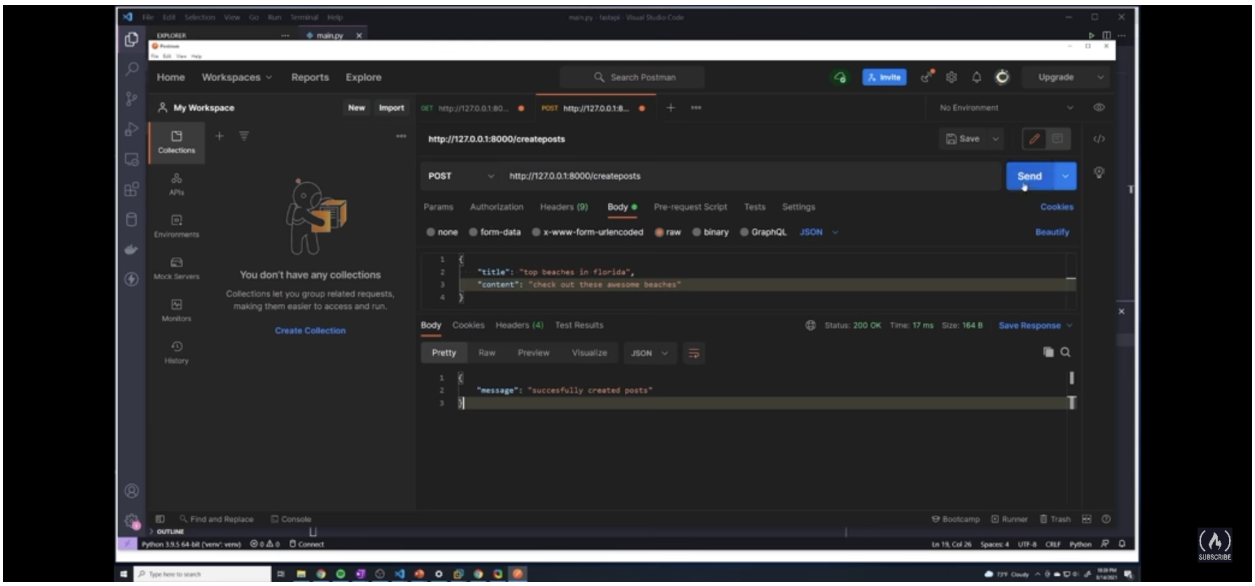
Use Postman to test API





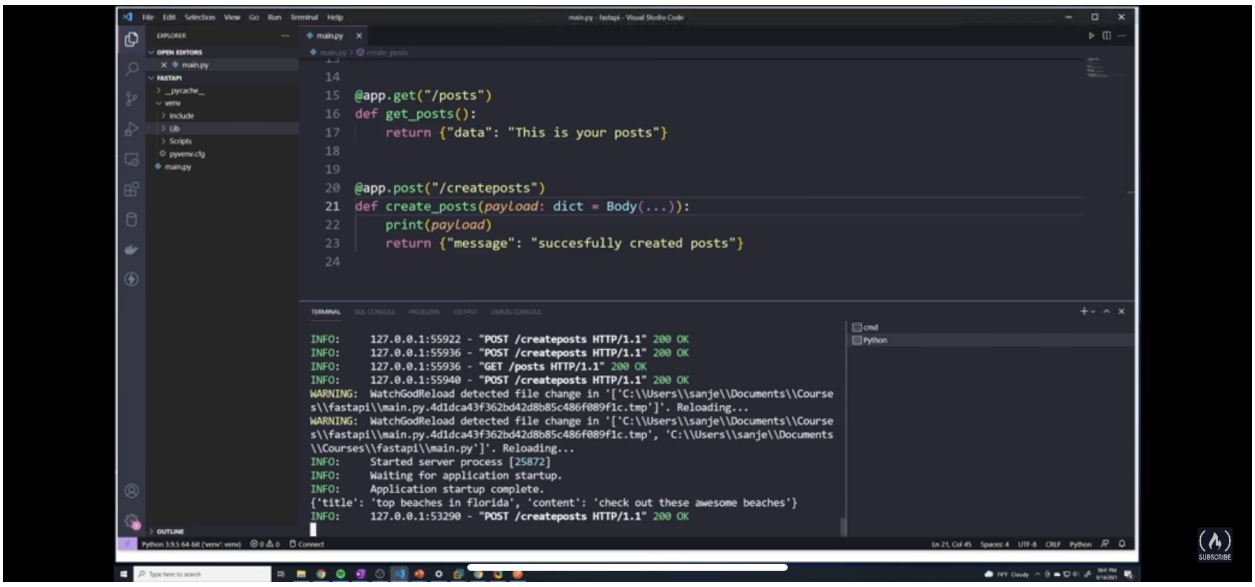
Whole point of POST is to send data to our API

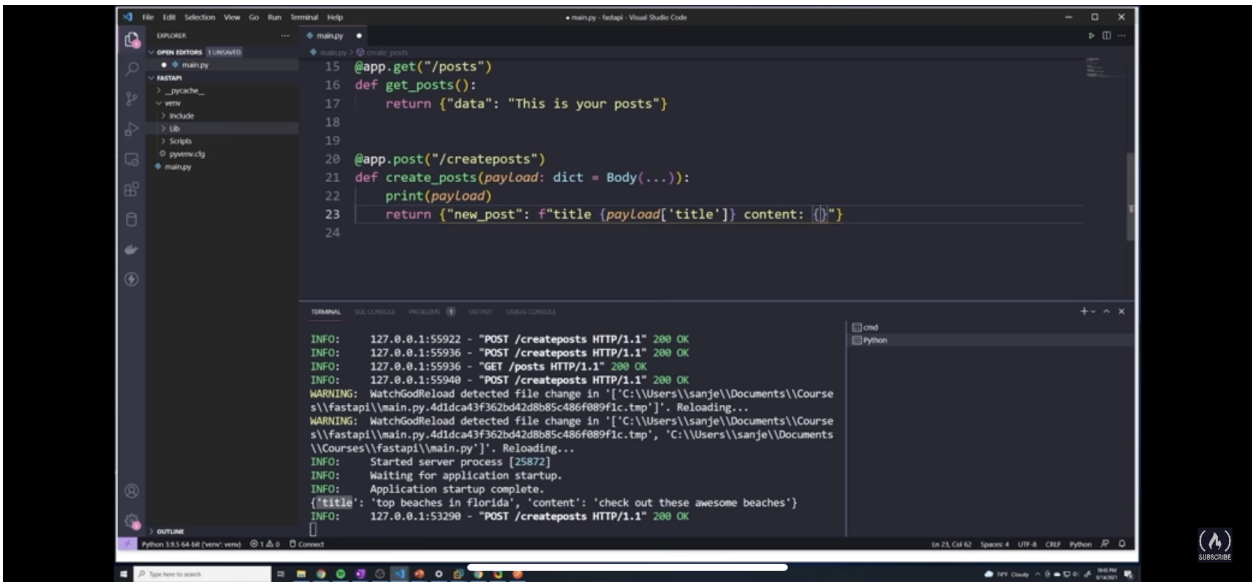
- we can do this in the body
- Send some data to the body using postman

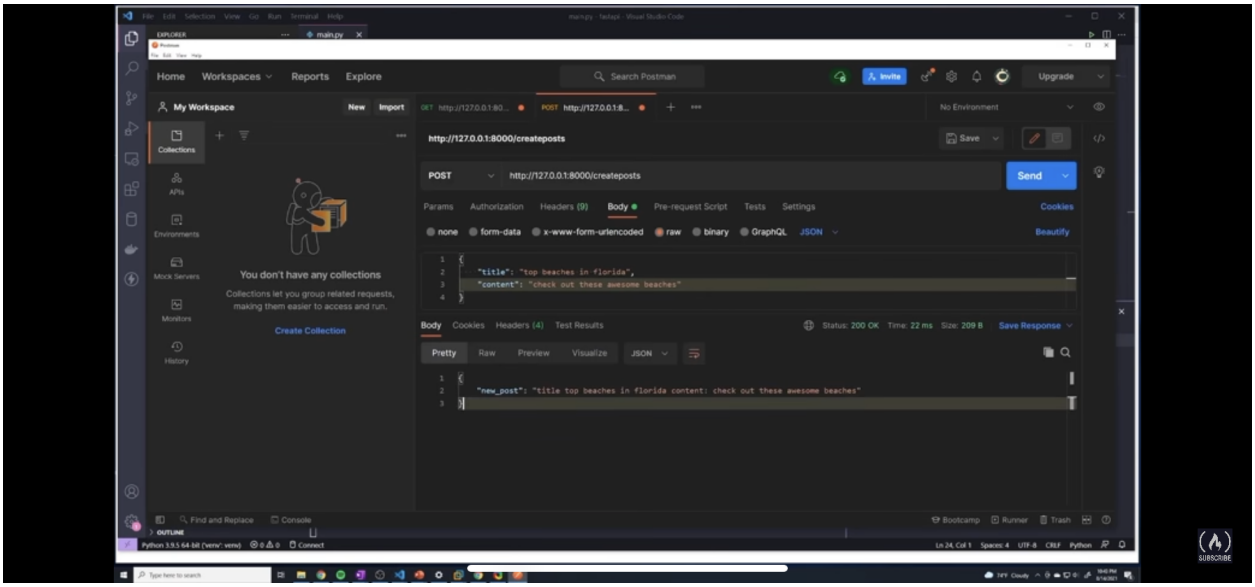


We need to get the payload somehow

- 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





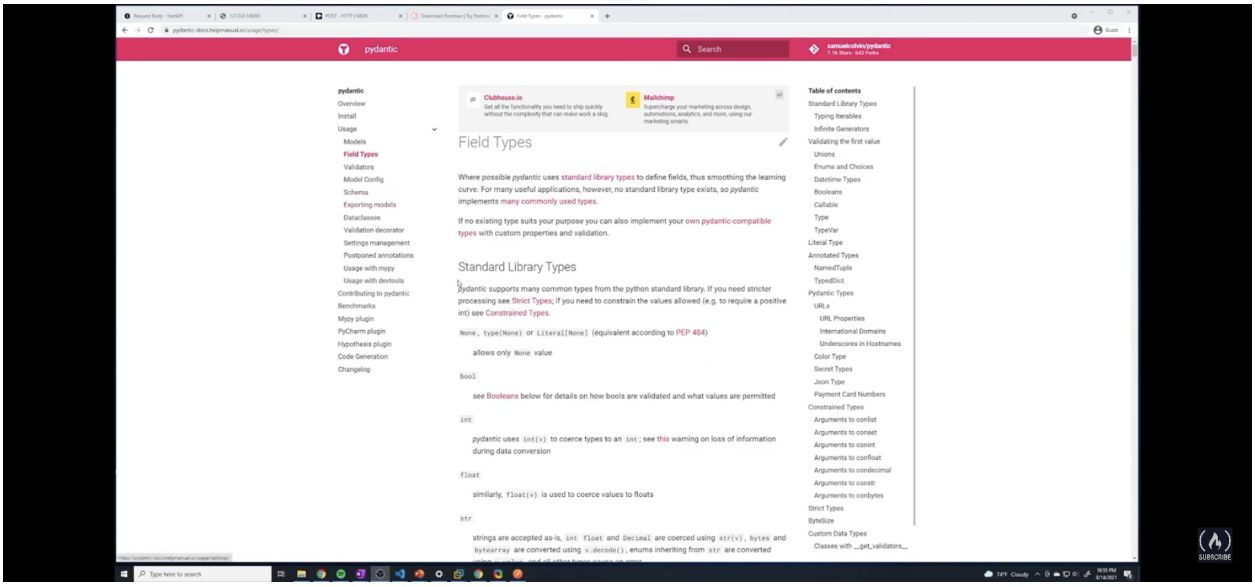


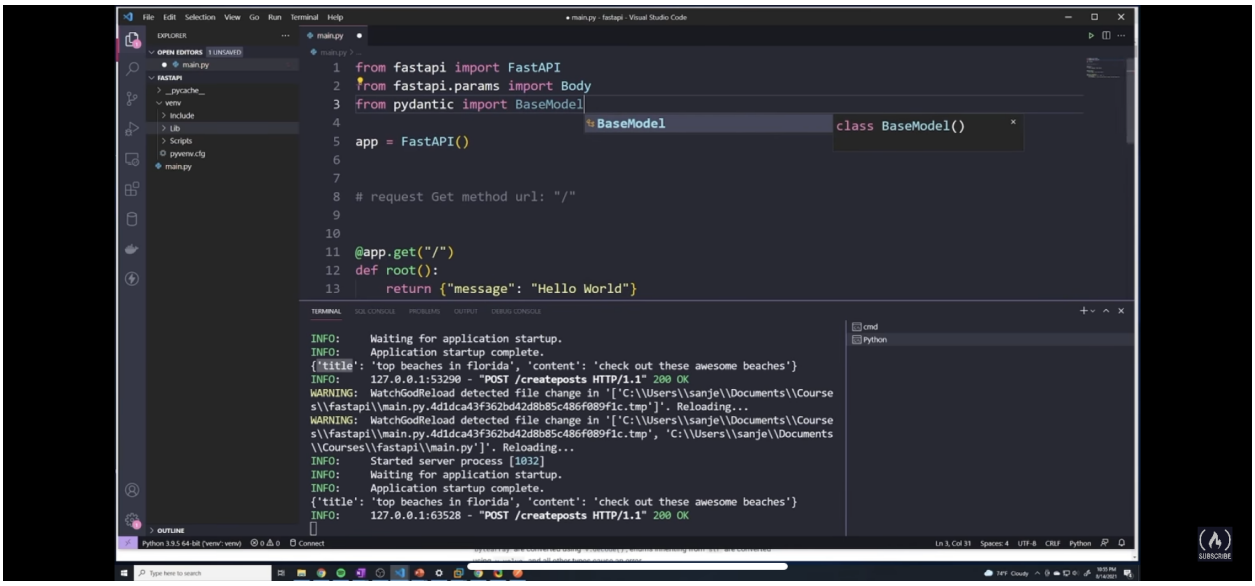
Why we need a Schema?

Why We Need Schema

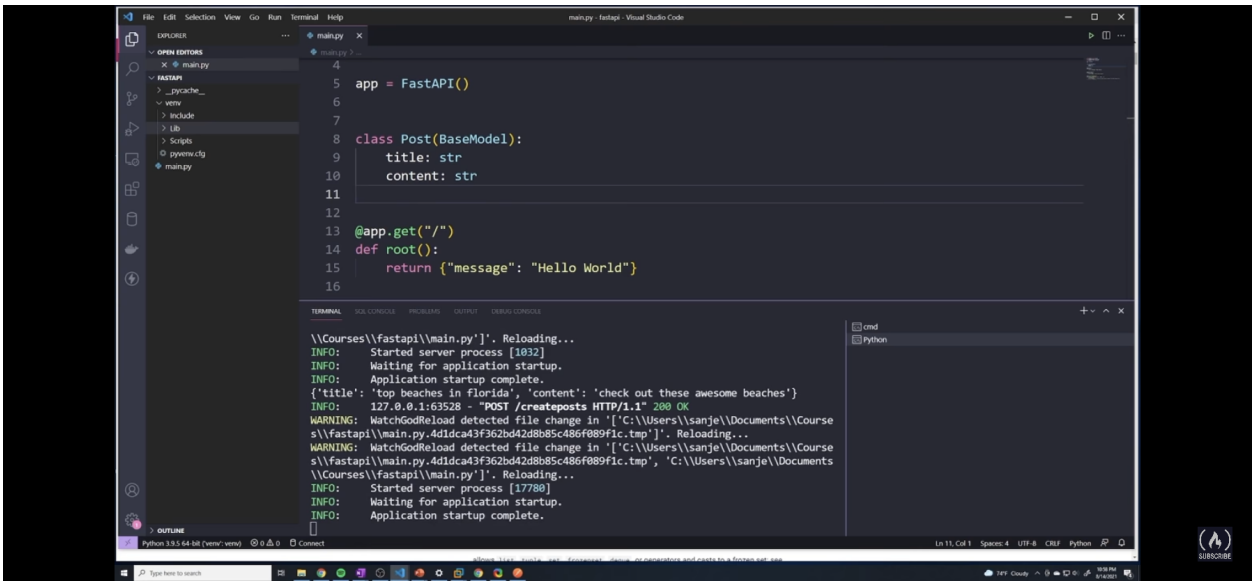
- It's a pain to get all the values from the body
- The client can send whatever data they want
- The data isn't getting validated
- We ultimately want to force the client to send data in a schema that we expect

So we don't just sent any kind of data

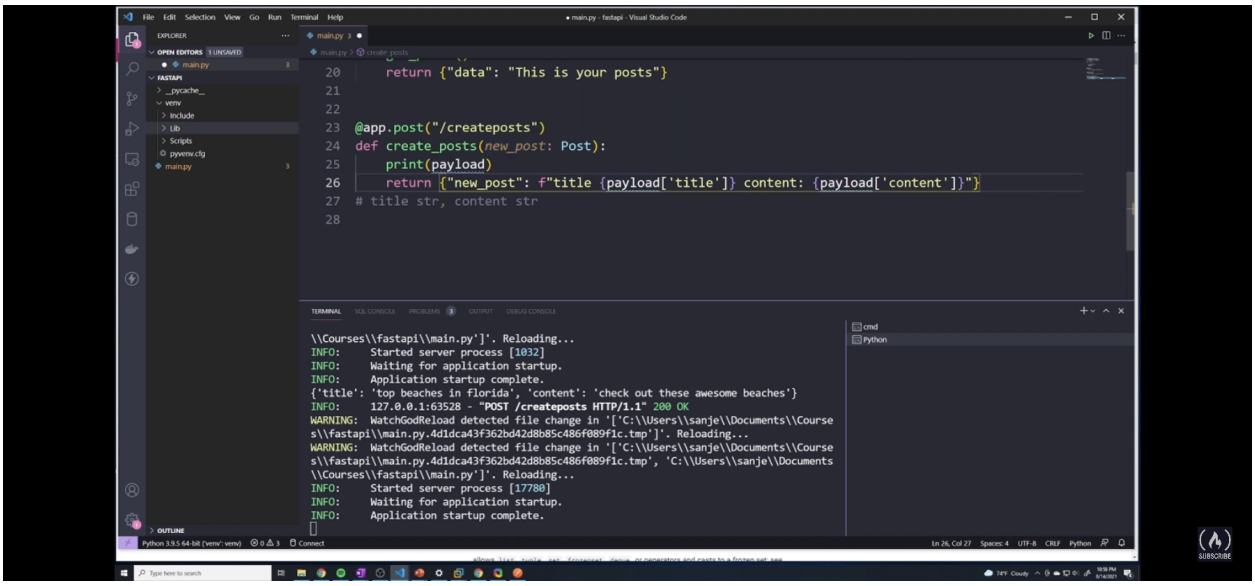




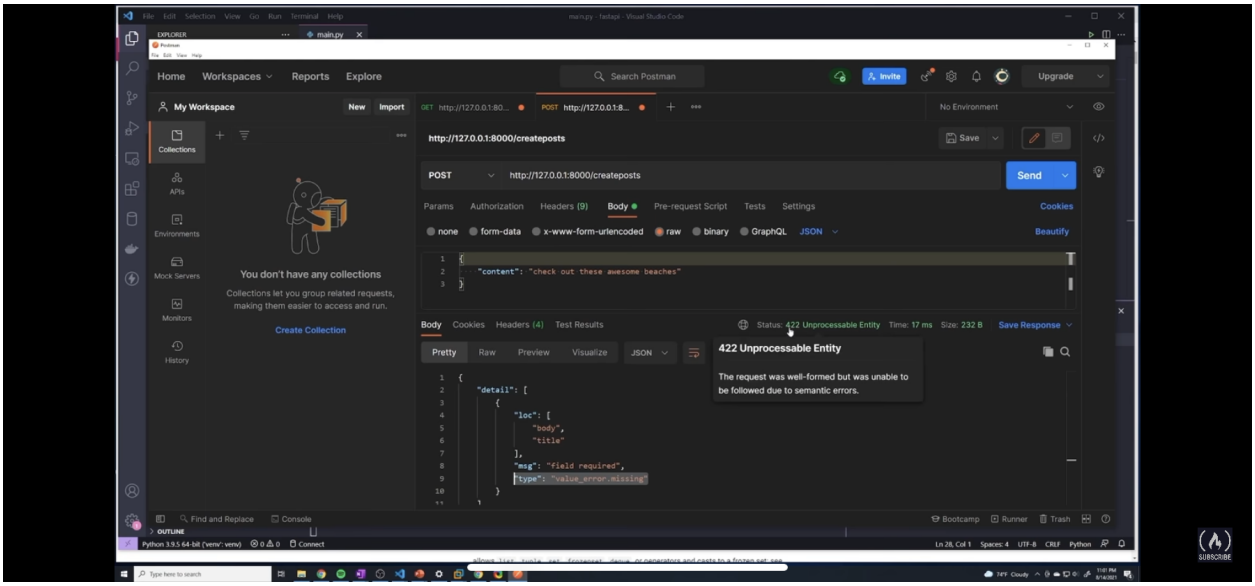
- use Pydantic Libary to make a structure
- Create a Post class

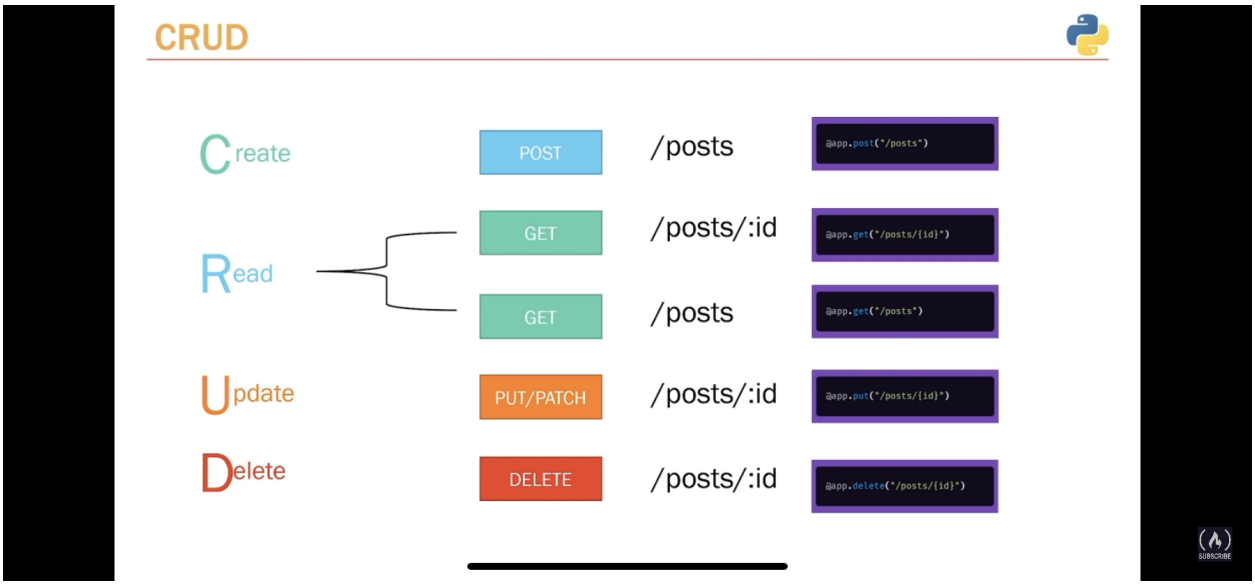


Pass in the class as an arg



validates all by itself - if wrong structure

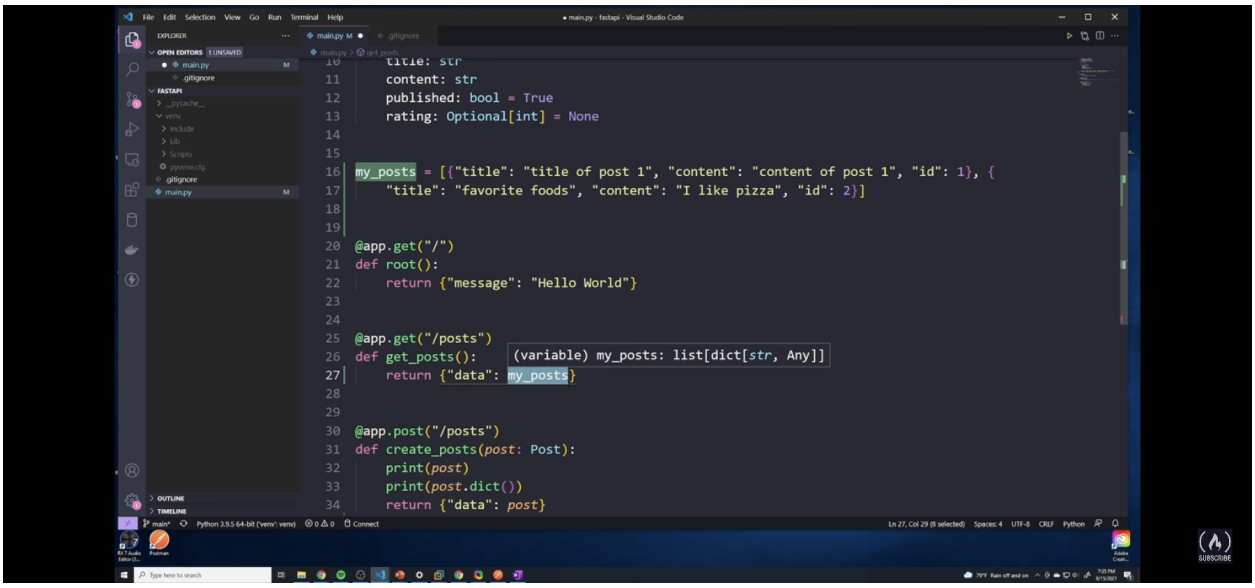


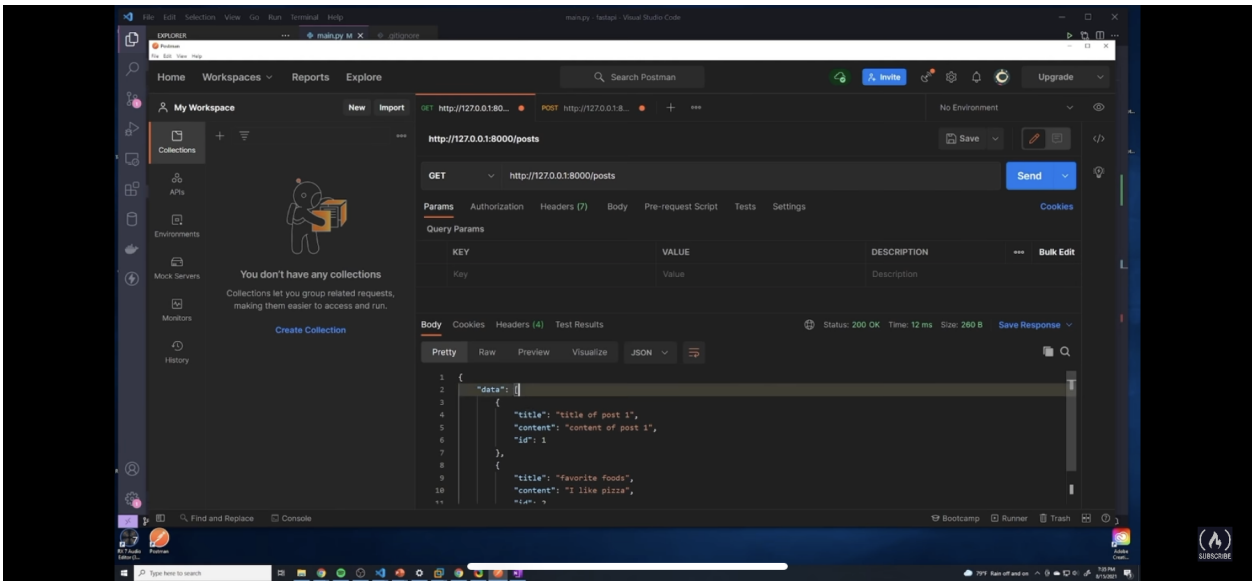


Every app should be able to Create, Read, Update, Delete

Passing Arrays

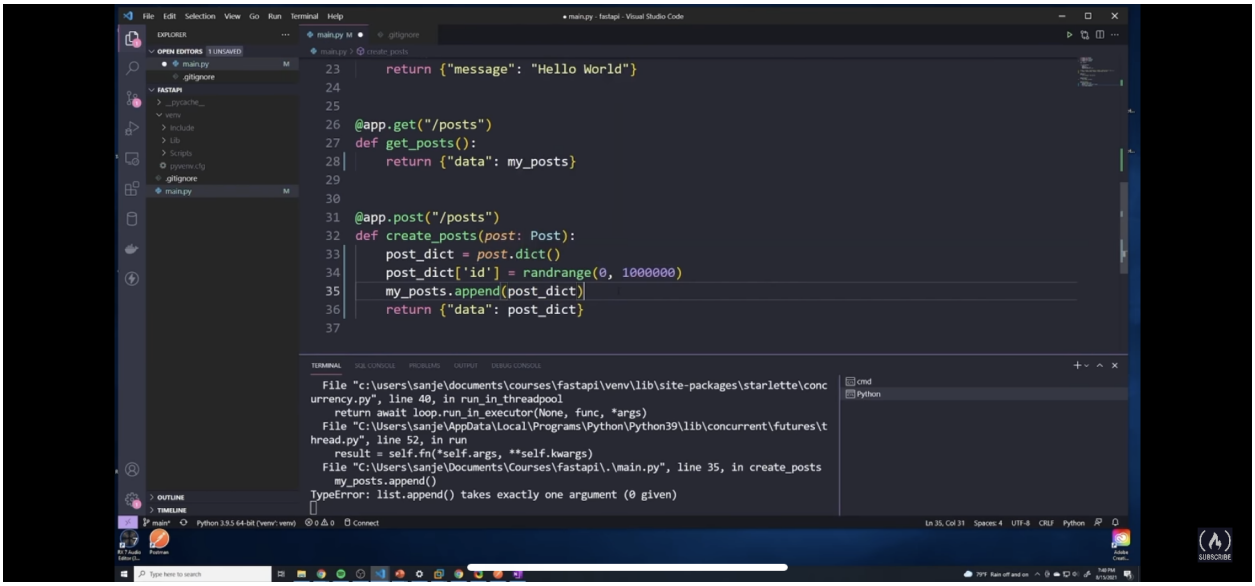
Can pass in arrays which will get auto-serialized



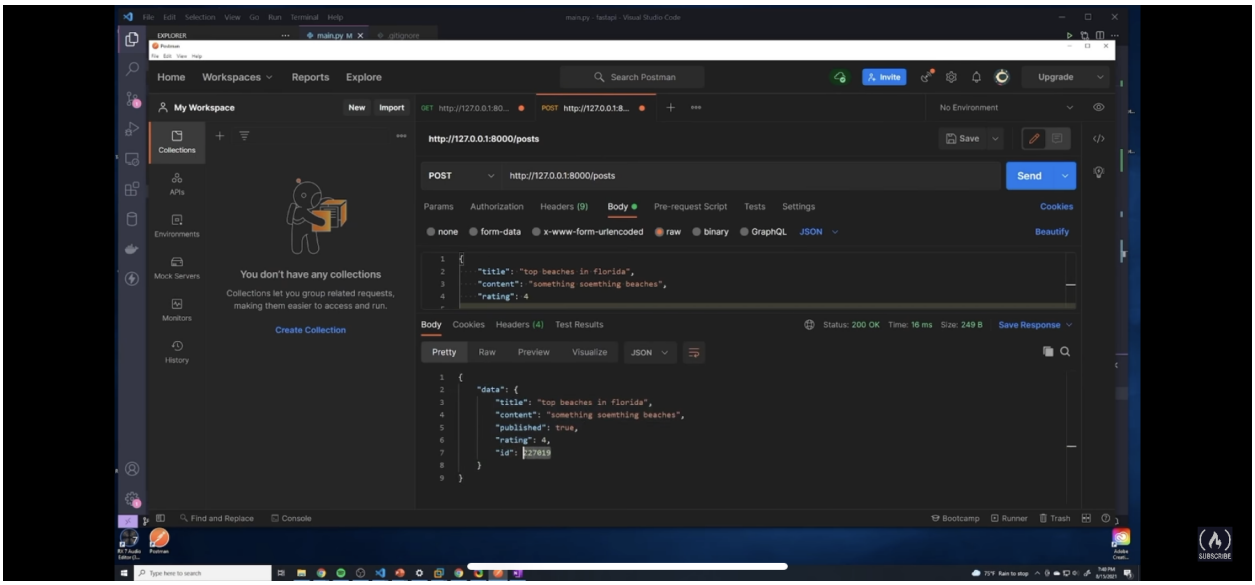


Saving the data from a post request to an Array

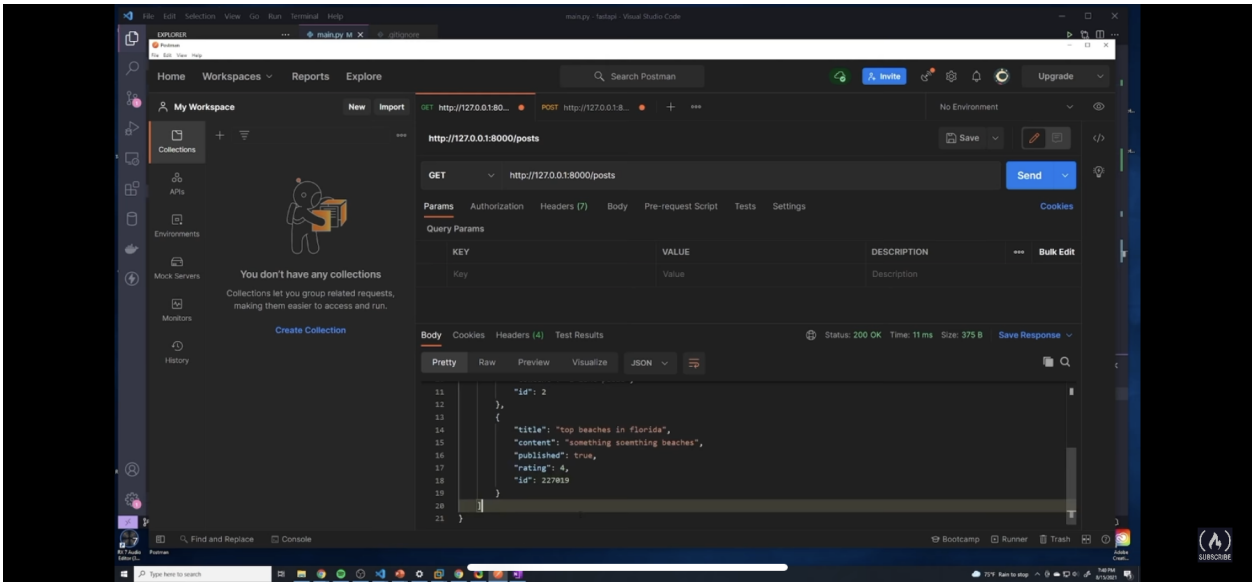
There is a chance that the id's will be hit - so make them bigger



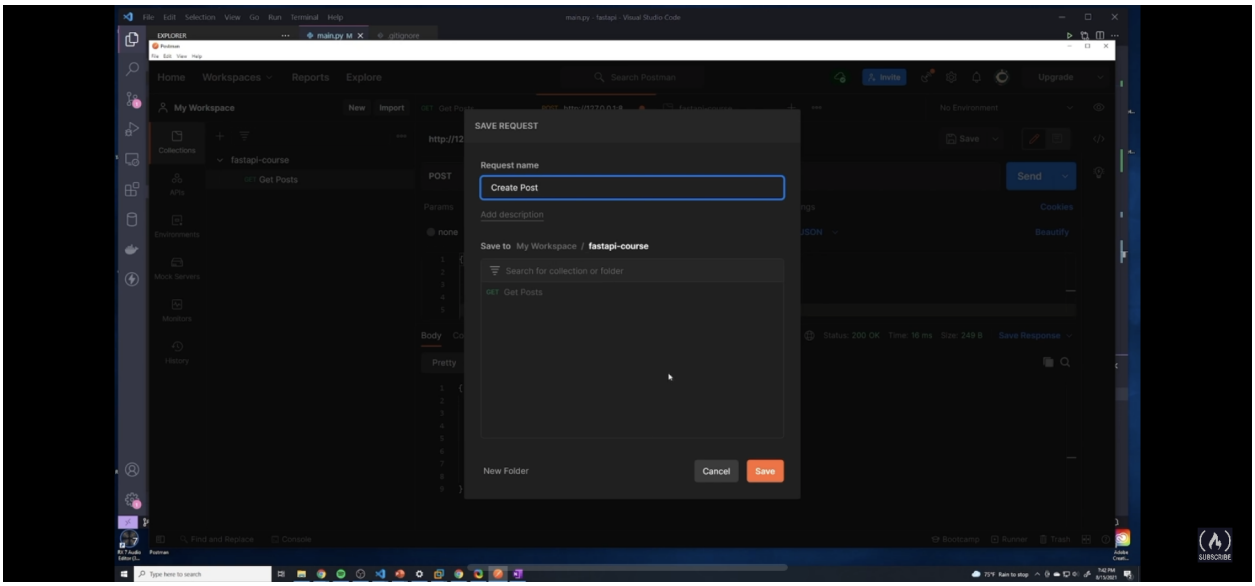
Run a Post request and the id increases



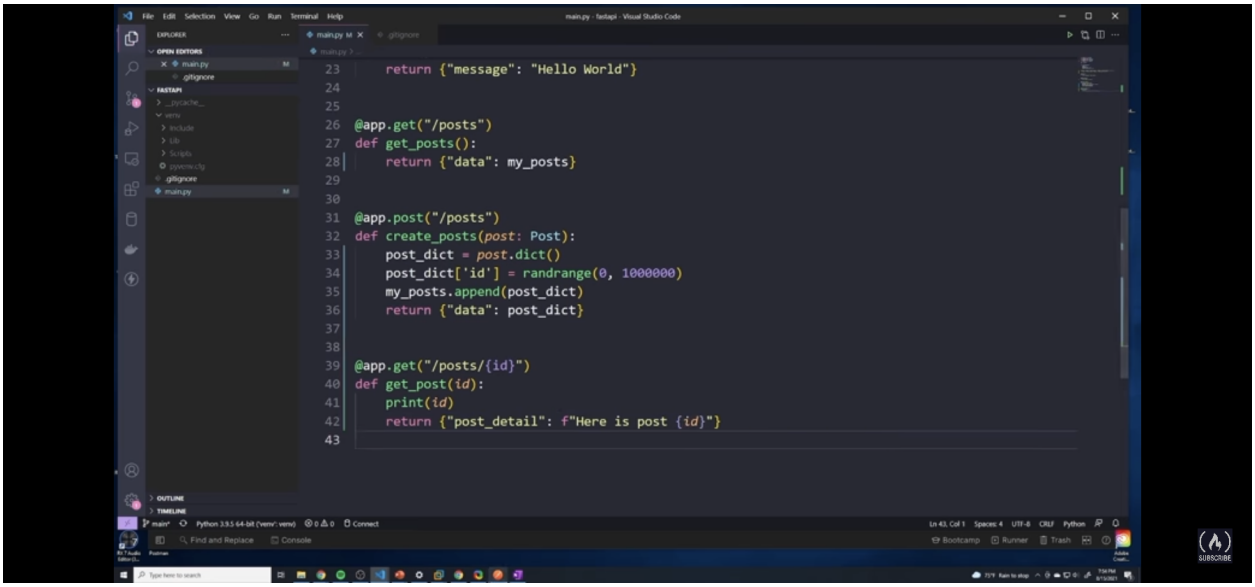
Run a get request to retrieve the data



Can save requests too

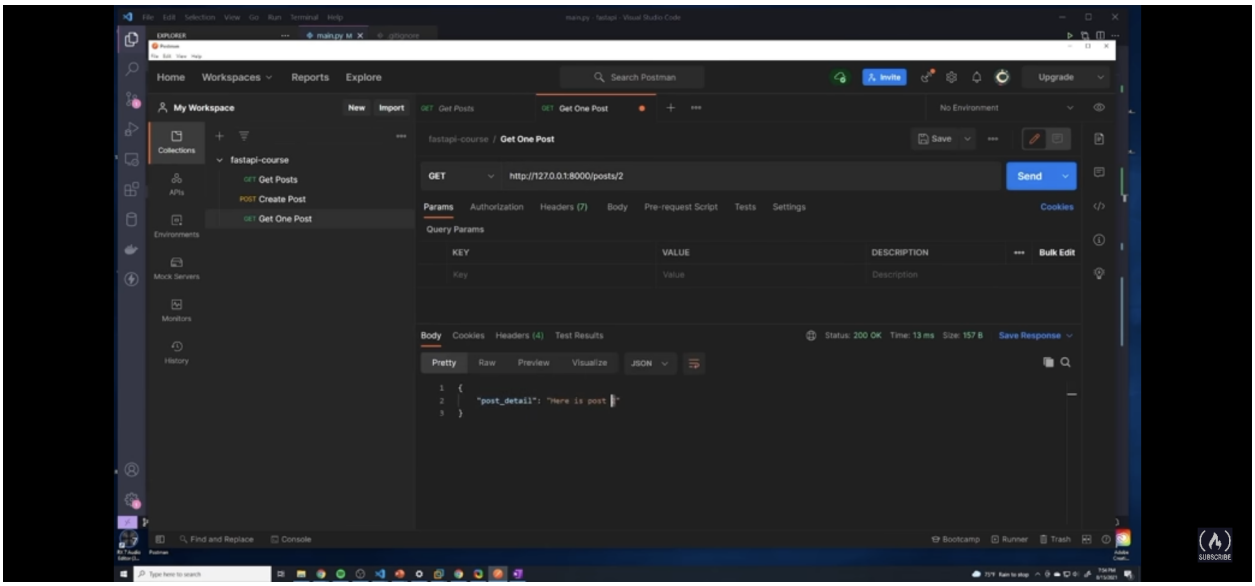


Get a specific id

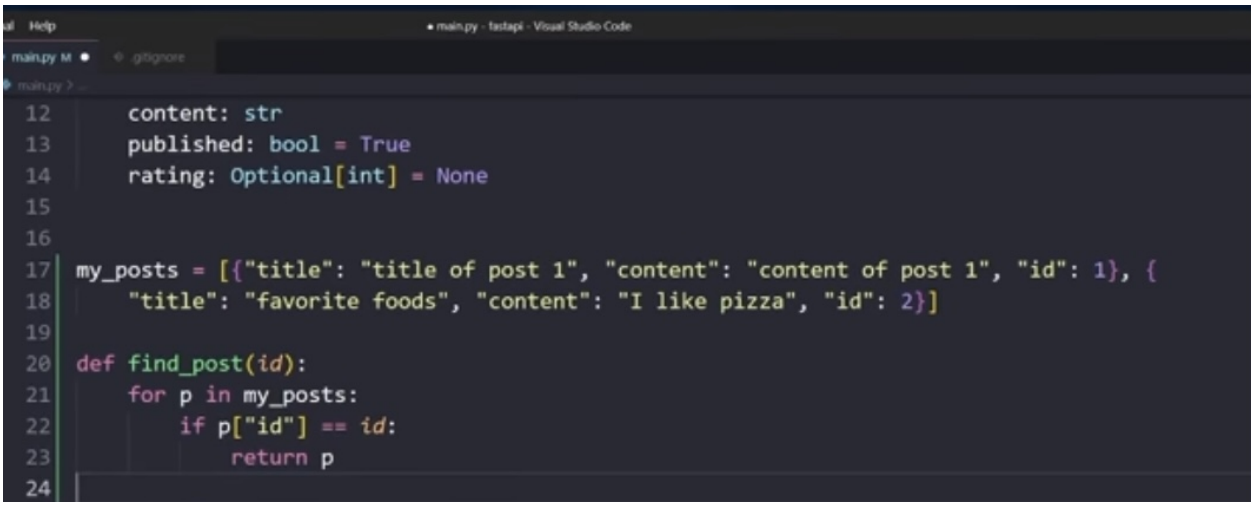


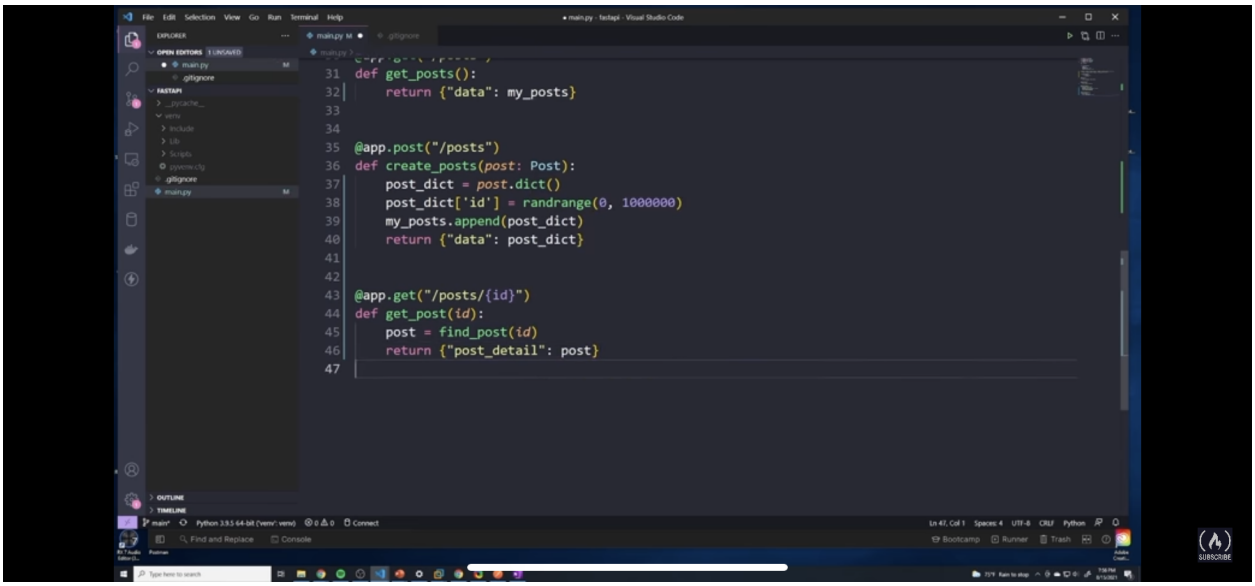
```python
    return {"message": "Hello World"}


@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'] = randrange(0, 1000000)
    my_posts.append(post_dict)
    return {"data": post_dict}


@app.get("/posts/{id}")
def get_post(id):
    print(id)
    return {"post_detail": f"Here is post {id}"}
```

Test in Postman

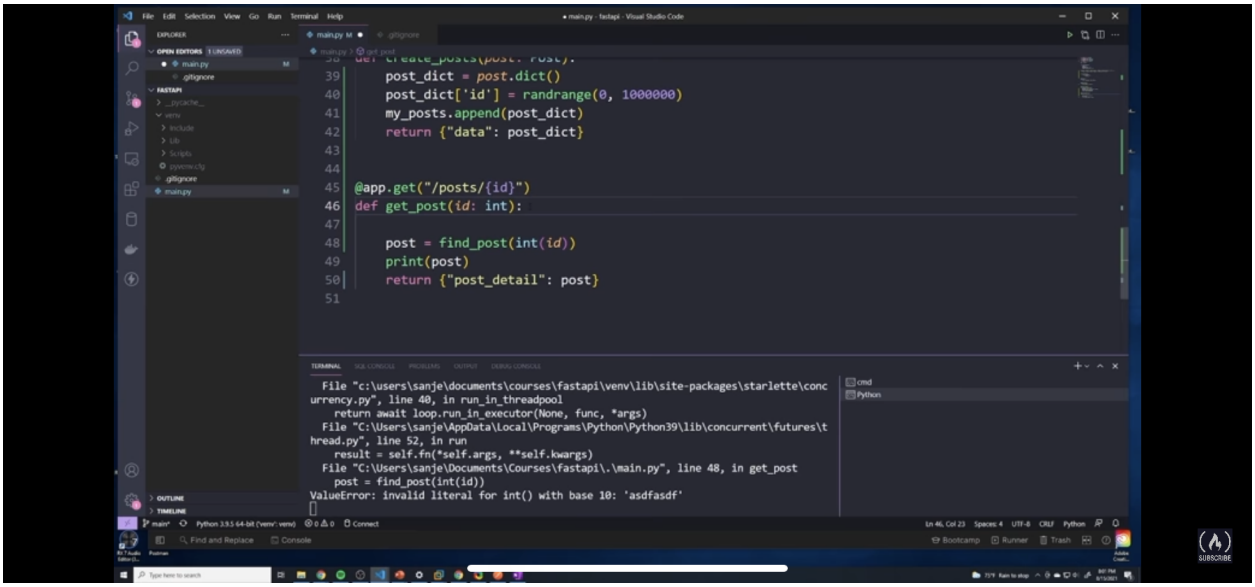


Refactor and use a function to find id we want

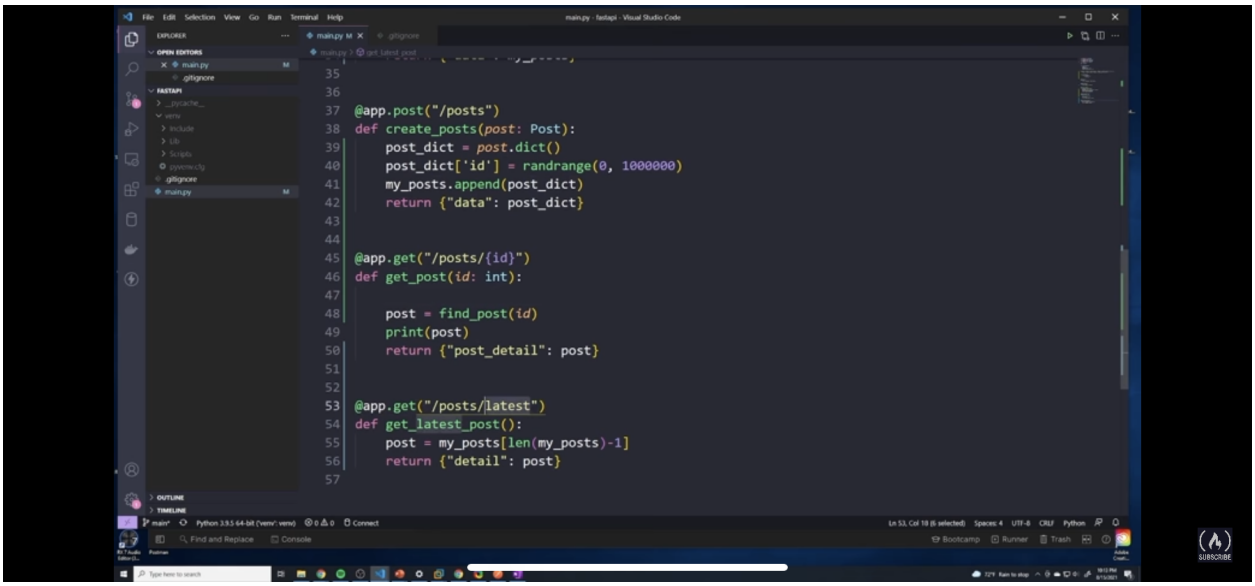




Need to make sure we only send ints

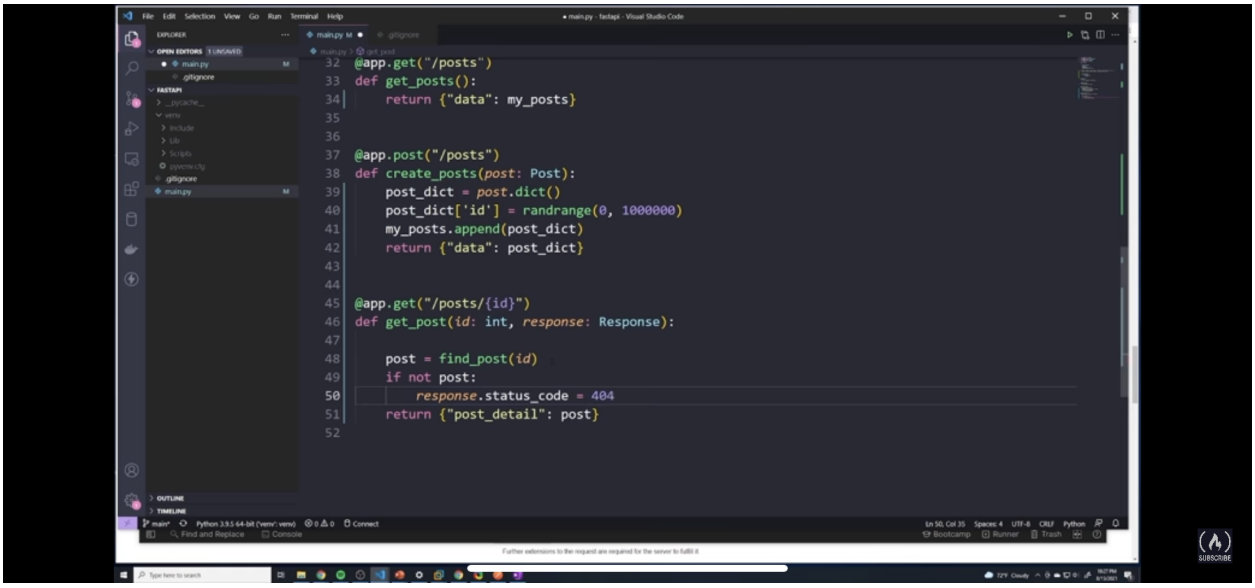


Return most recent post

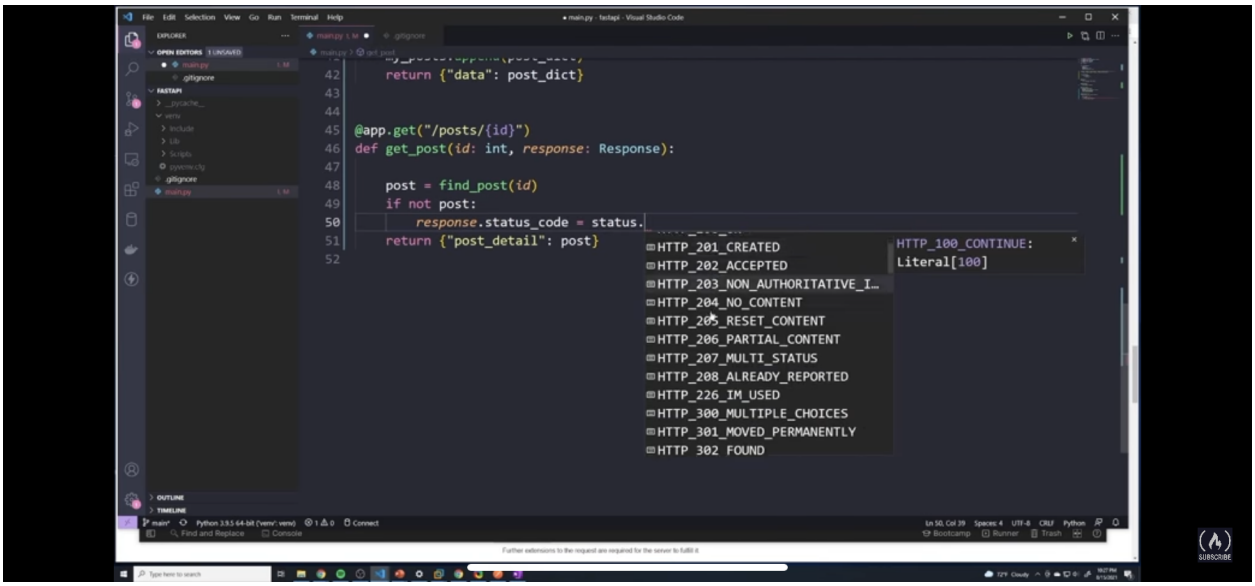


Working with Response Codes

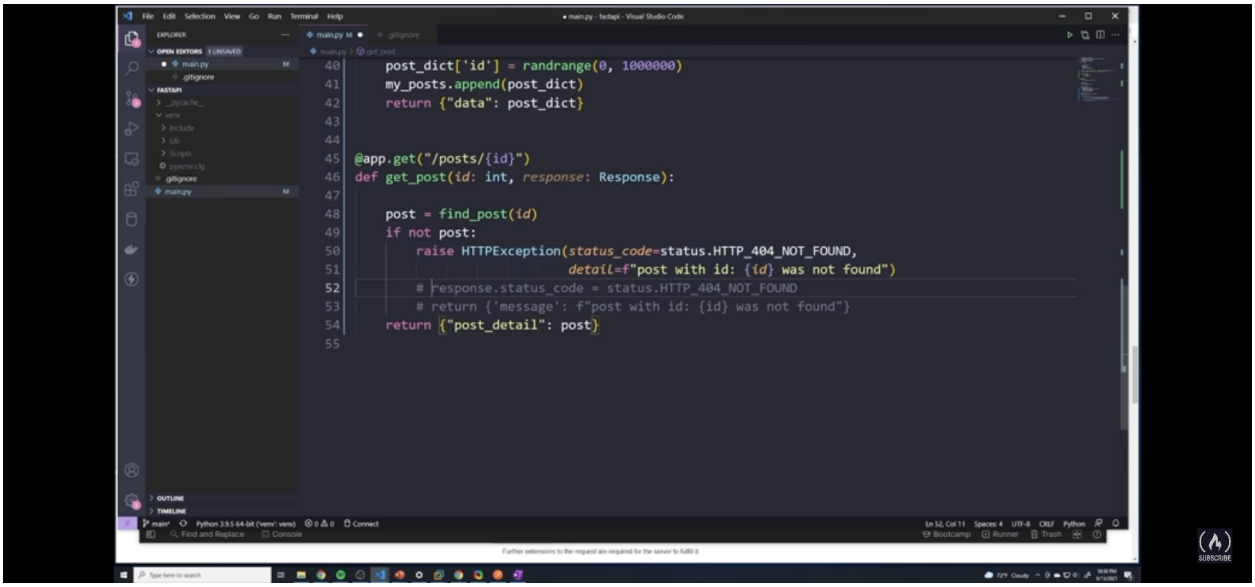
If a post is not found



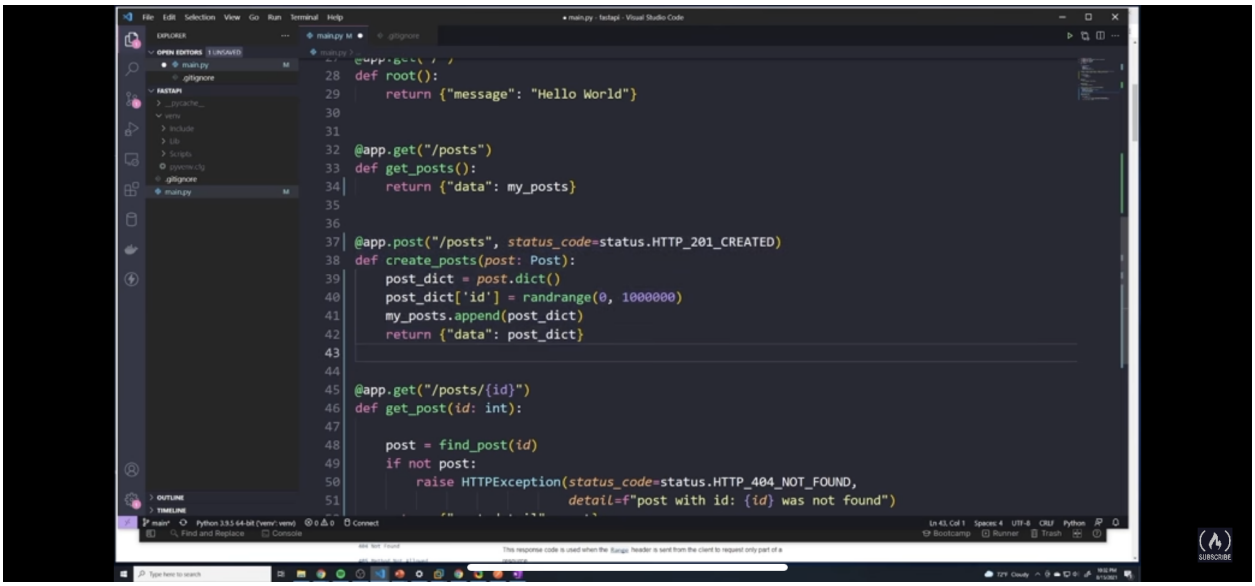
Return a status code 404



Or write status or add httpException



Send a 201 created when a Post request is sent



Deleting Posts

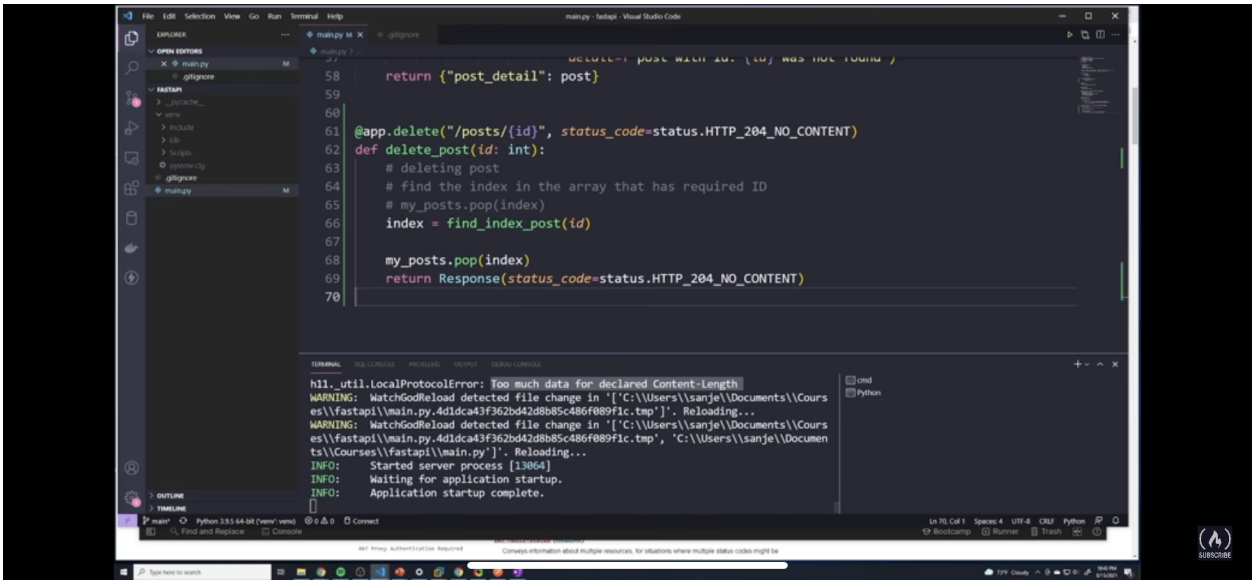
Loop through posts and return the index

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

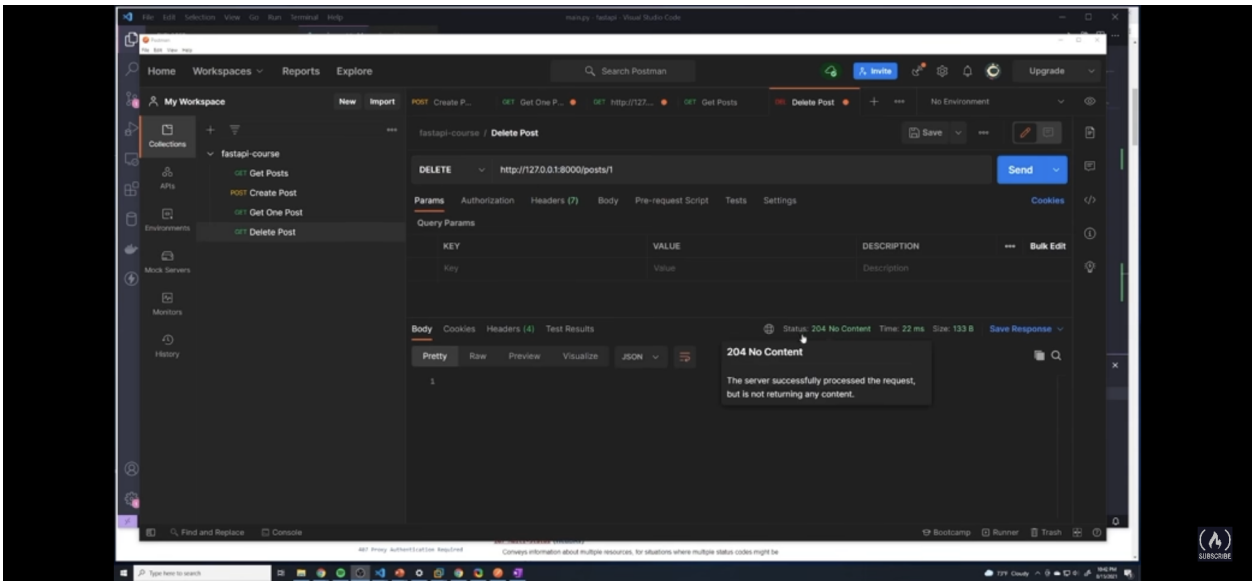
Pop from the dict the item with the id that you want to delete

```python

@app.delete("/posts/{id}")
def delete_post():
    # deleting post
    # find the index in the array that has required ID
    # my_posts.pop(index)
    index = find_index_post(id)

    my_posts.pop(index)
    return {'message': 'post was succesfully deleted'}

```

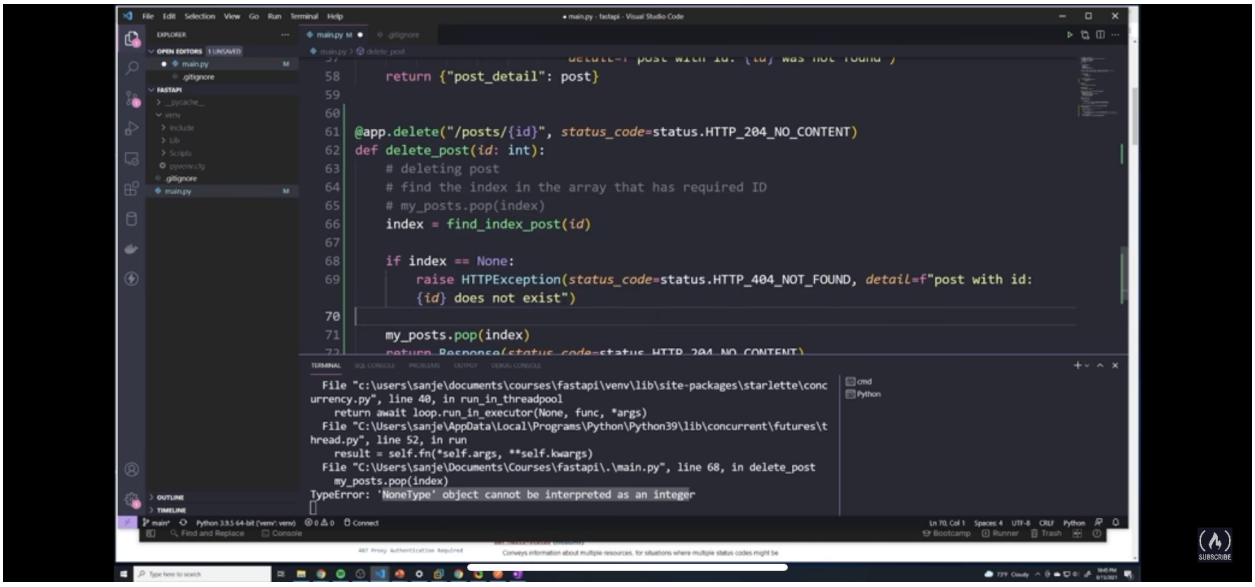
Add status code 204 : No content



Test in Postman

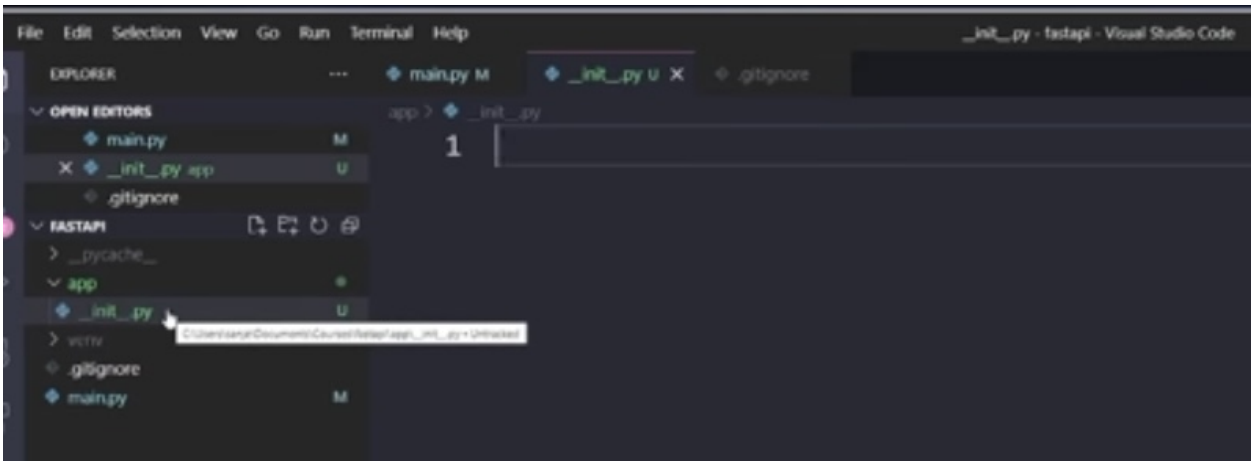


Add exception Handling for id not existing

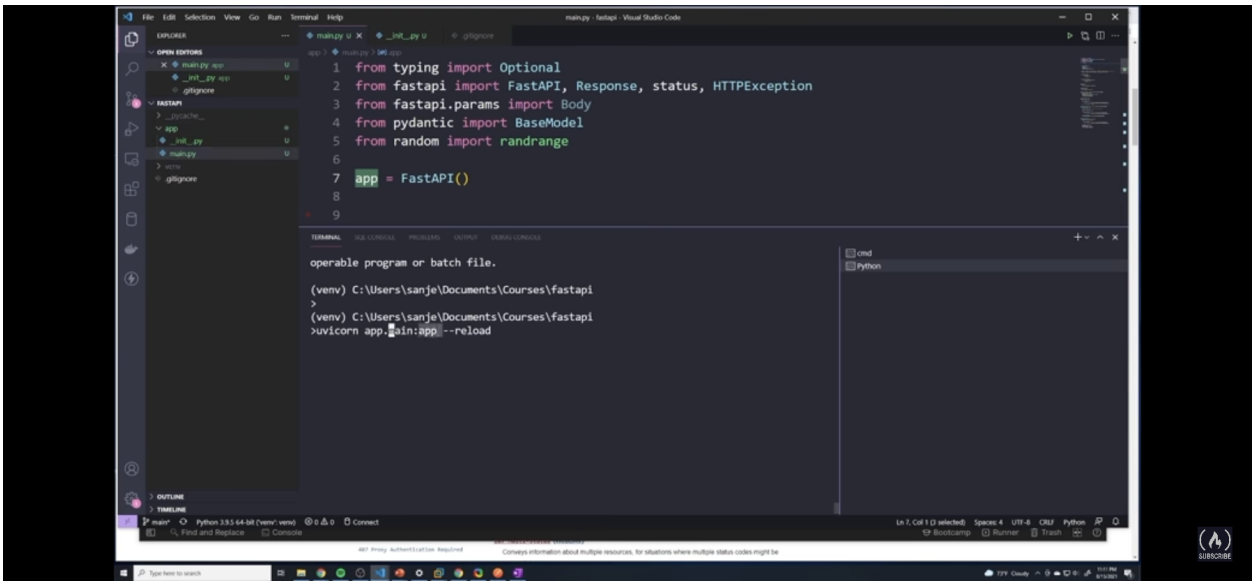


Python Packages

Create a folder called app



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



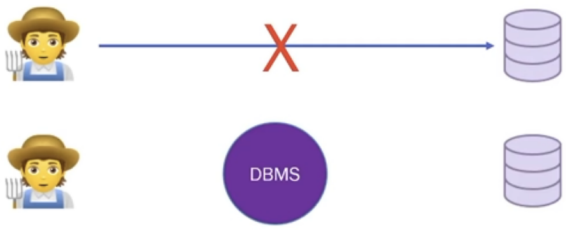
Working with Databases

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

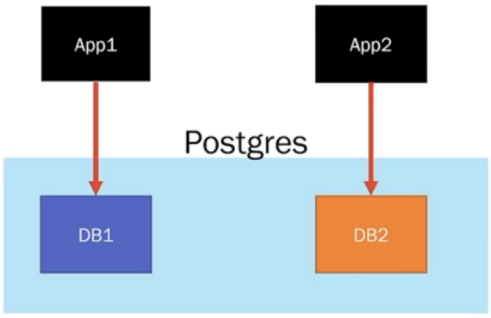
Relational	NoSQL
MYSQL	MongoDB
POSTGRESQL	DynamoDB
ORACLE	ORACLE
SQL SERVER	SQL SERVER

Relational Database & SQL

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

Postgres

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

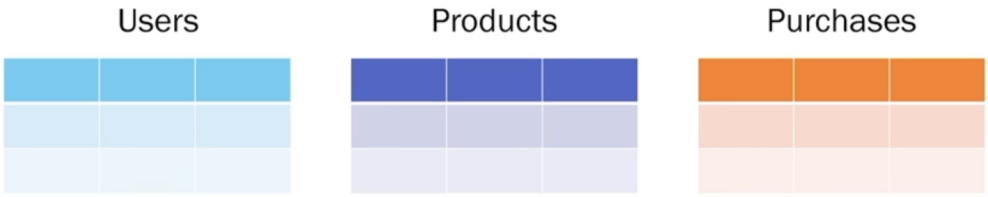


Postgres

- 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

Tables

- A table represents a subject or event in an application



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

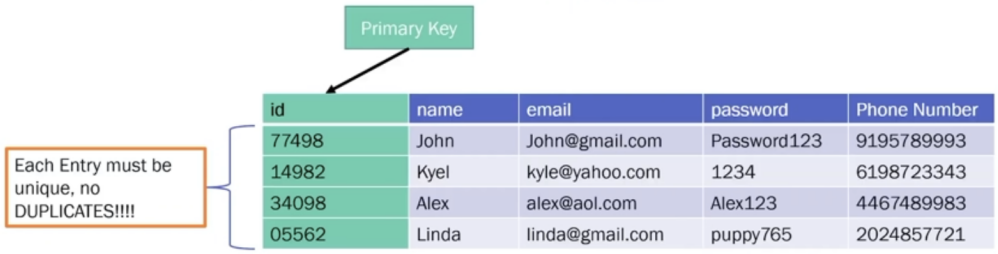
Columns

Rows

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

Primary Key

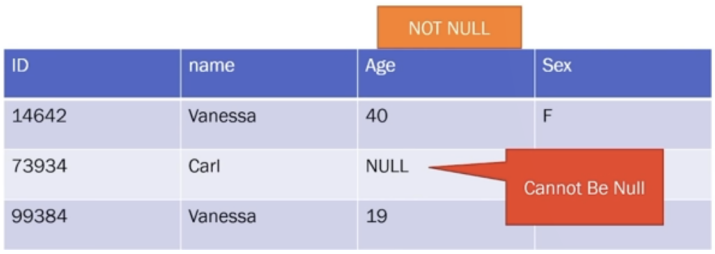
- Is a column or group of columns that uniquely identifies each row in a table
- Table can have one and only one primary key



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

Null Constraints

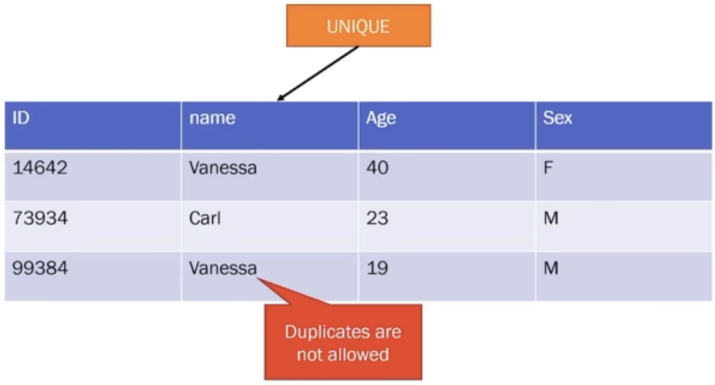
- By default, when adding a new entry to a database, any column can be left blank. When a column is left blank, it has a null value
- If you need column to be properly filled in to create a new record, a NOT NULL constraint can be added to the column to ensure that the column is never left blank



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

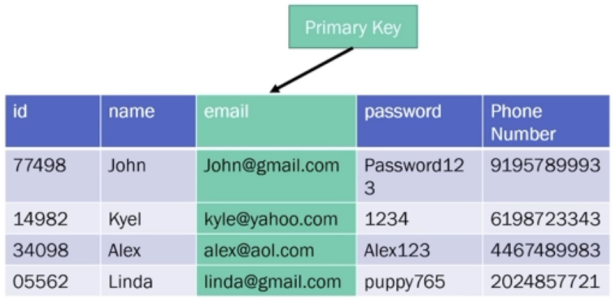
Unique Constraints

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

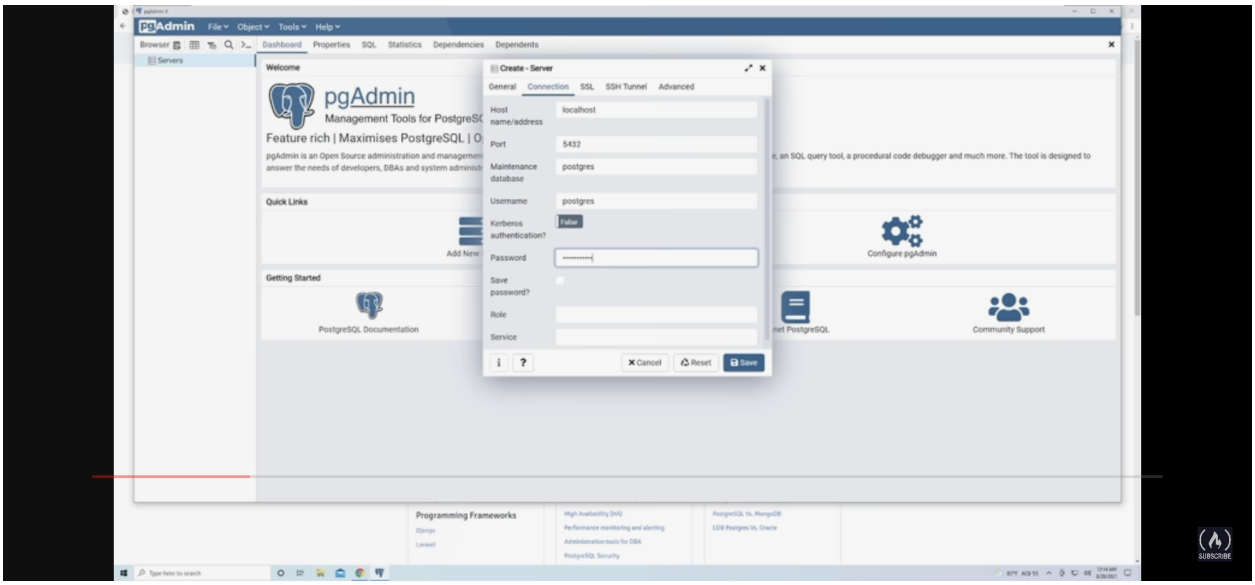


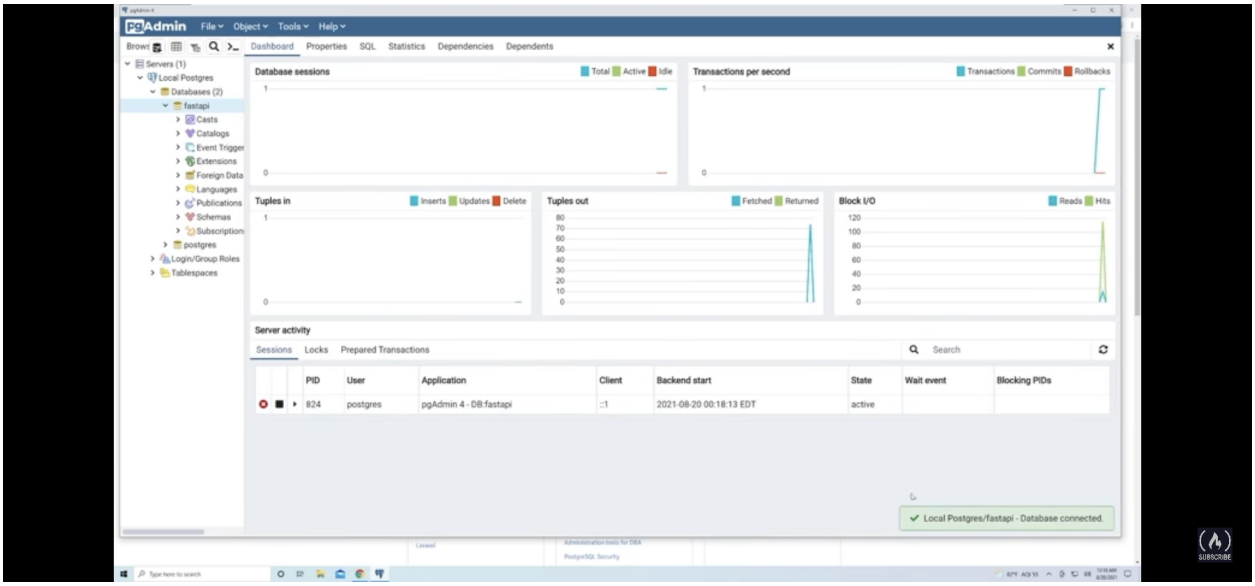
Primary Key

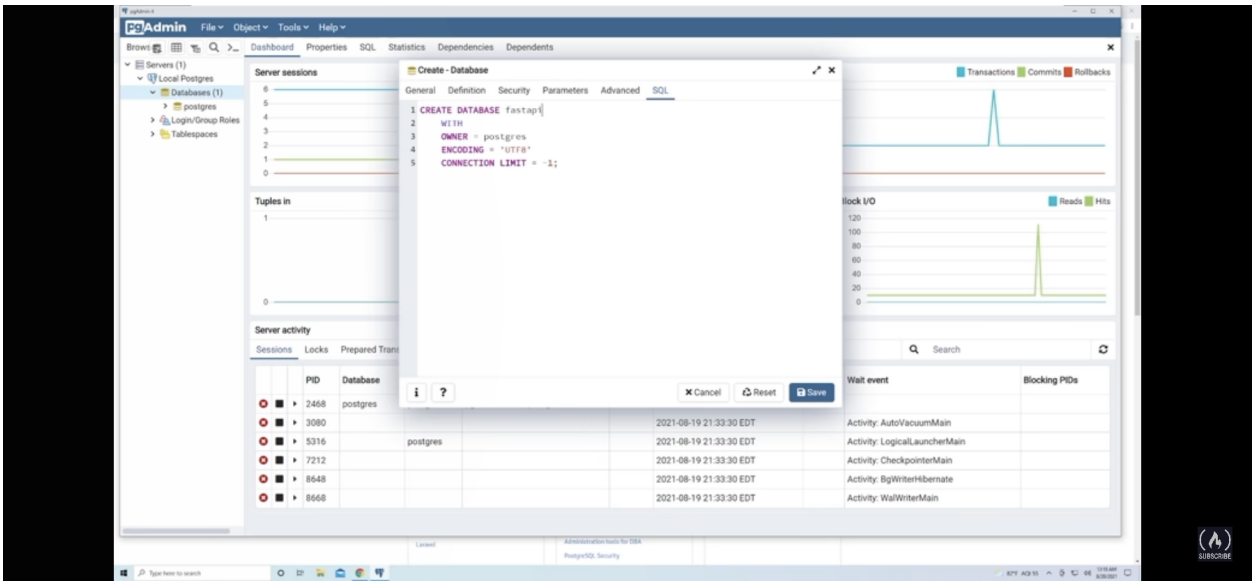
- The Primary Key does not have to be the ID column always. It's up to you to decide which column uniquely defines each record
- In this example, since an email can only be registered once, the email column can also be used as the primary key



POSTGRESQL


pgAdmin
Create - Server
Welcome
Quick Links
Getting Started
Programming Frameworks

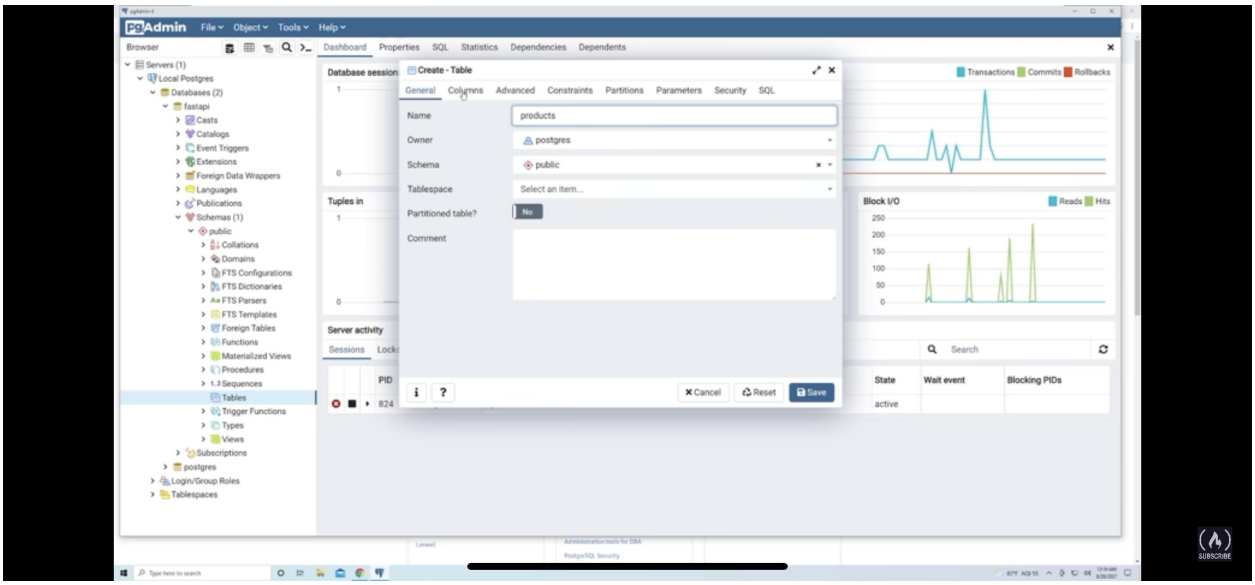

pgAdmin
Database sessions
Transactions per second
Tuples in
Tuples out
Block I/O
Server activity
Local Postgres/fastapi - Database connected.

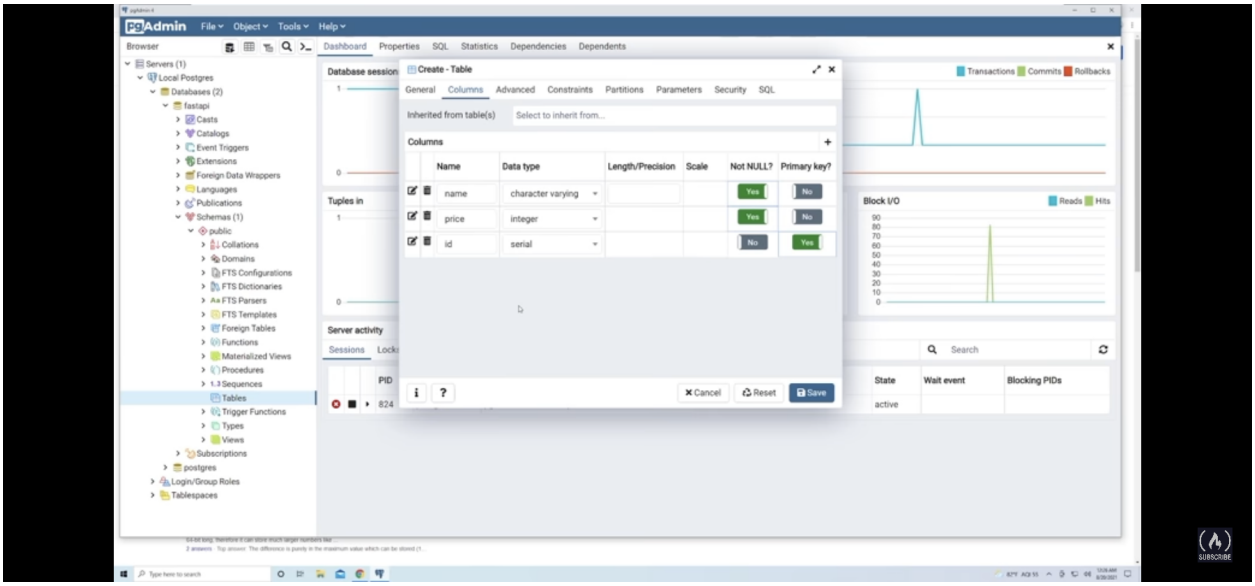


Postgres DataTypes

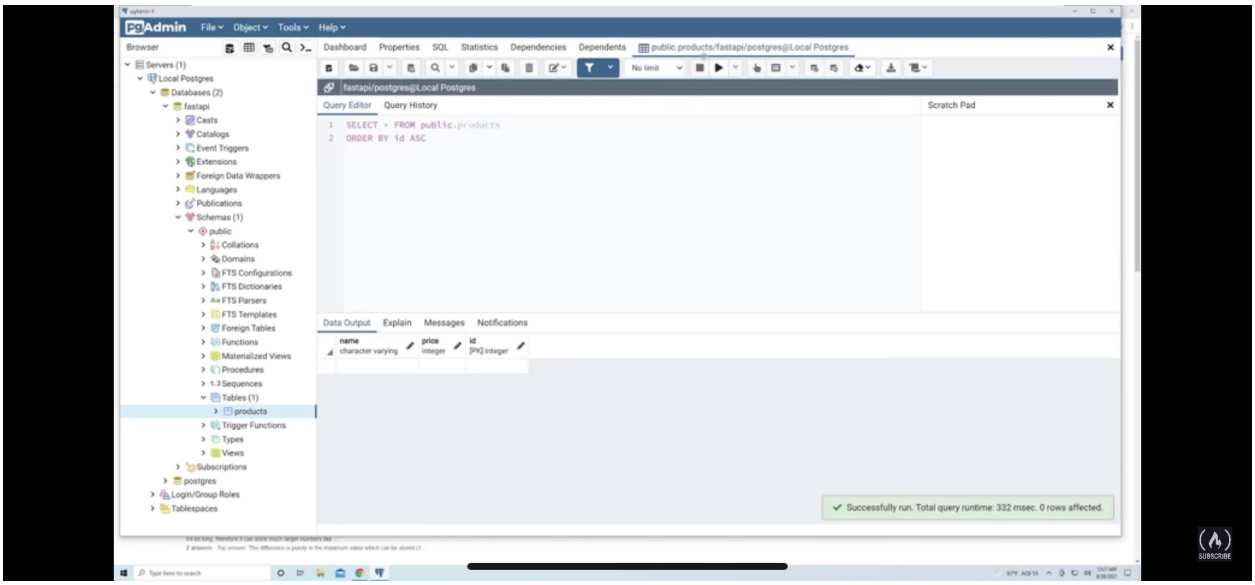
- Databases have datatypes just like any programming language

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

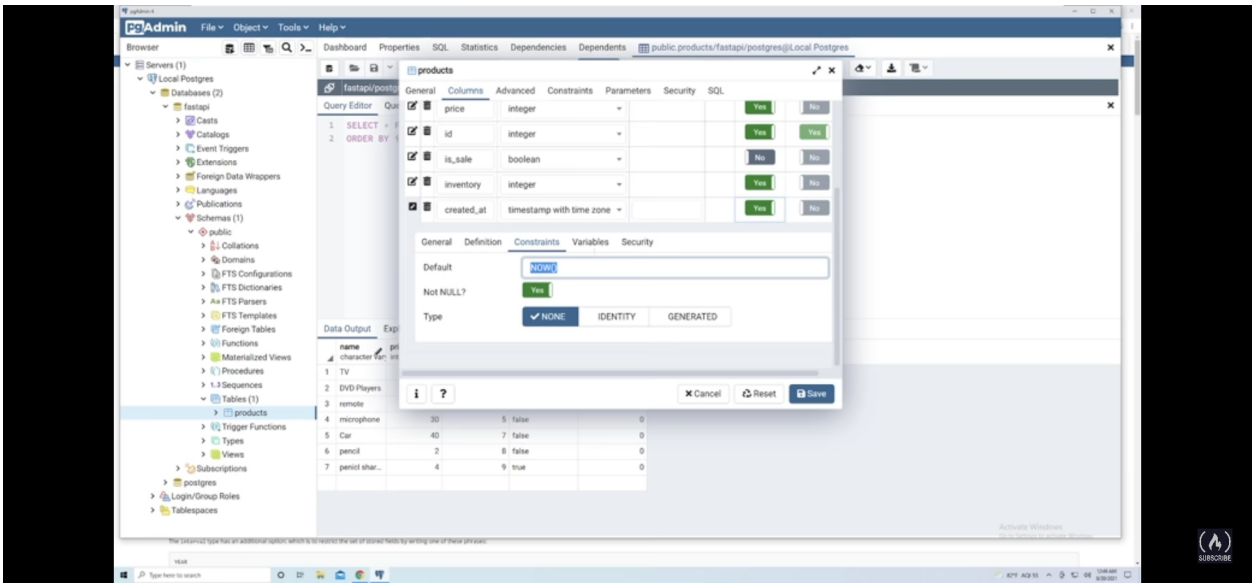


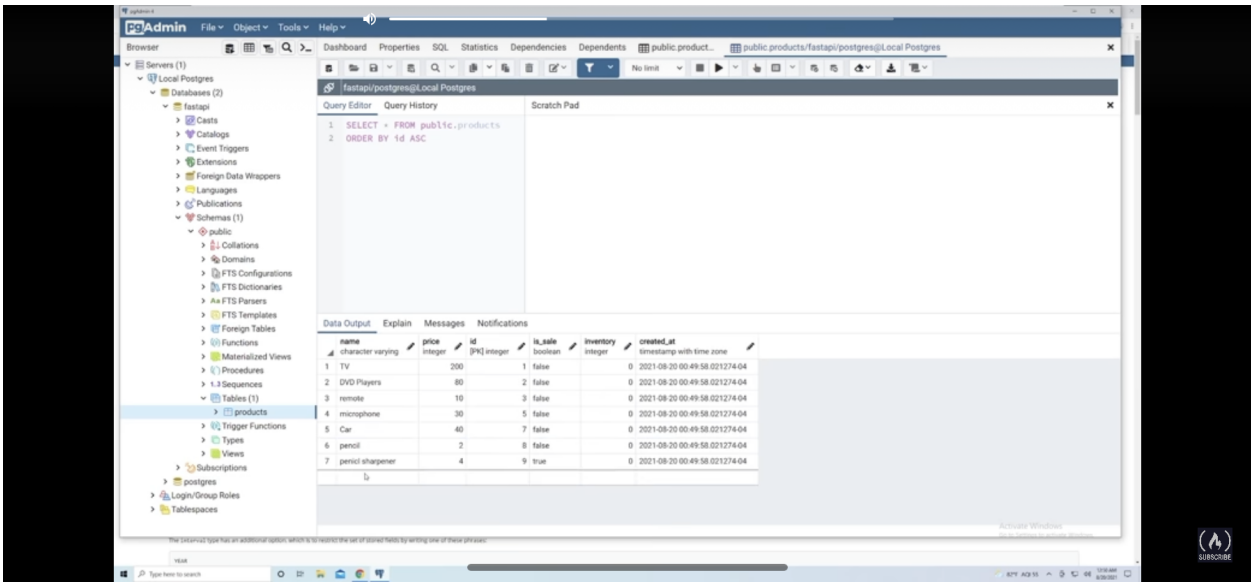


View top x rows



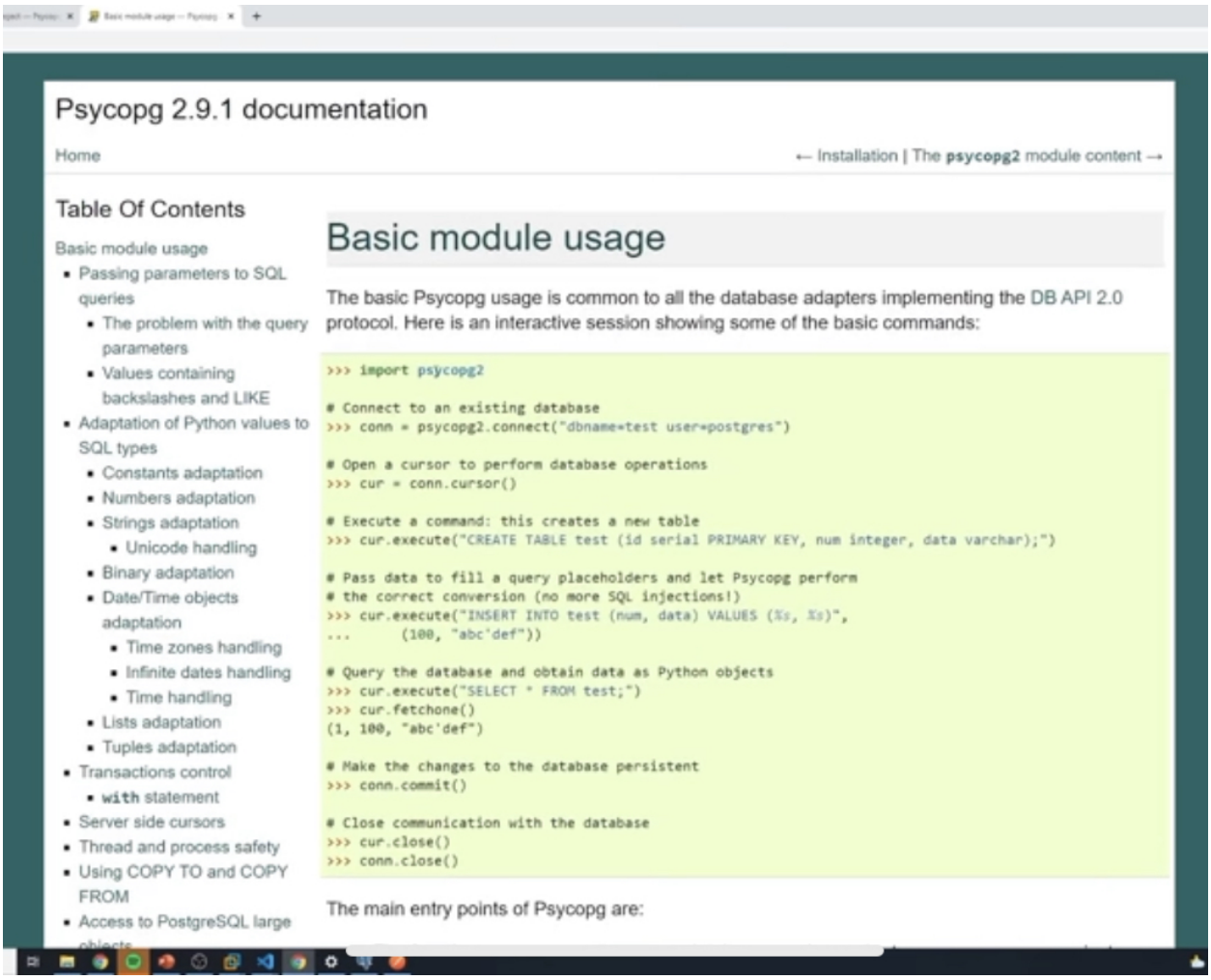
Setting data





Connecting to SQL with our API

pip install and Import Psycopg

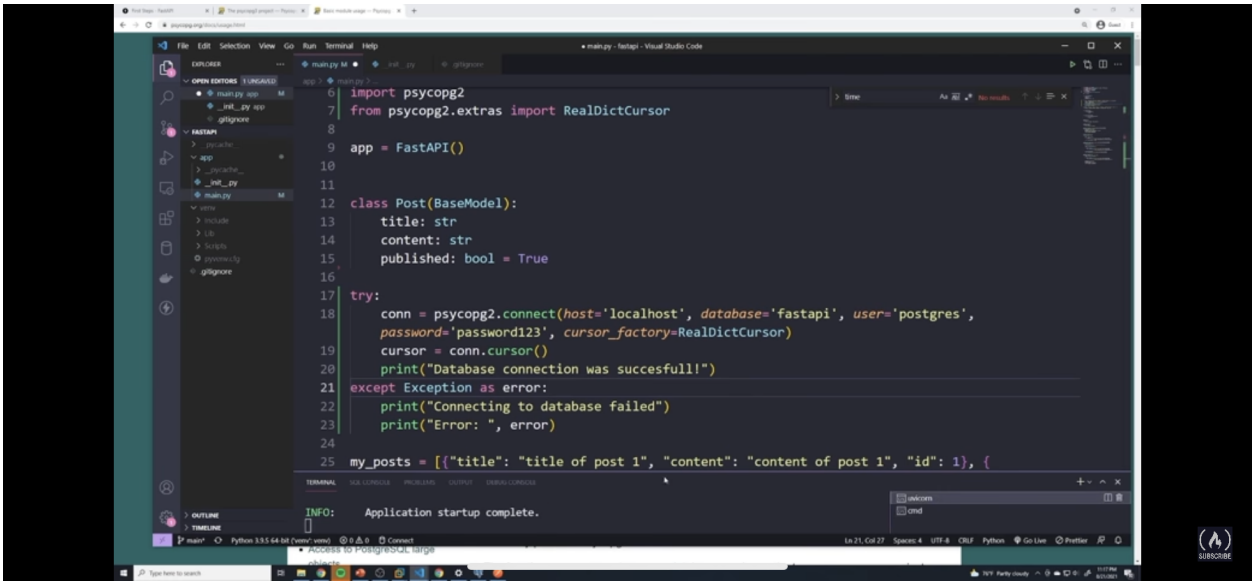


Connect to Postgres instance

```python
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)
```

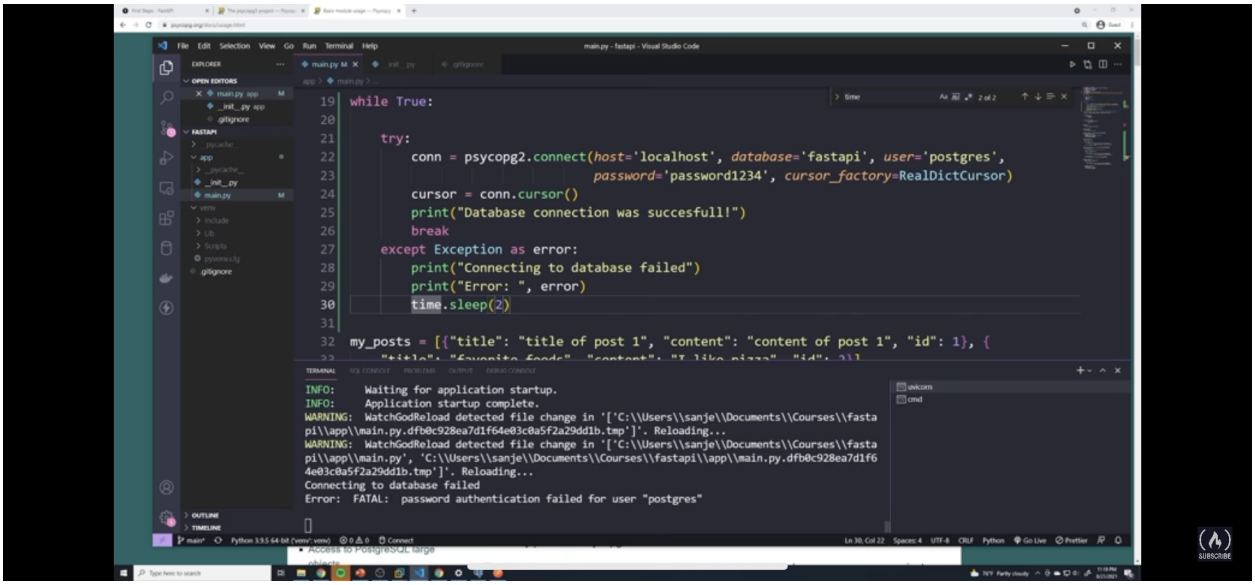
Real dict cursor gives you the column name aswell as value - gives east to use dict

Add cursor and exception handling



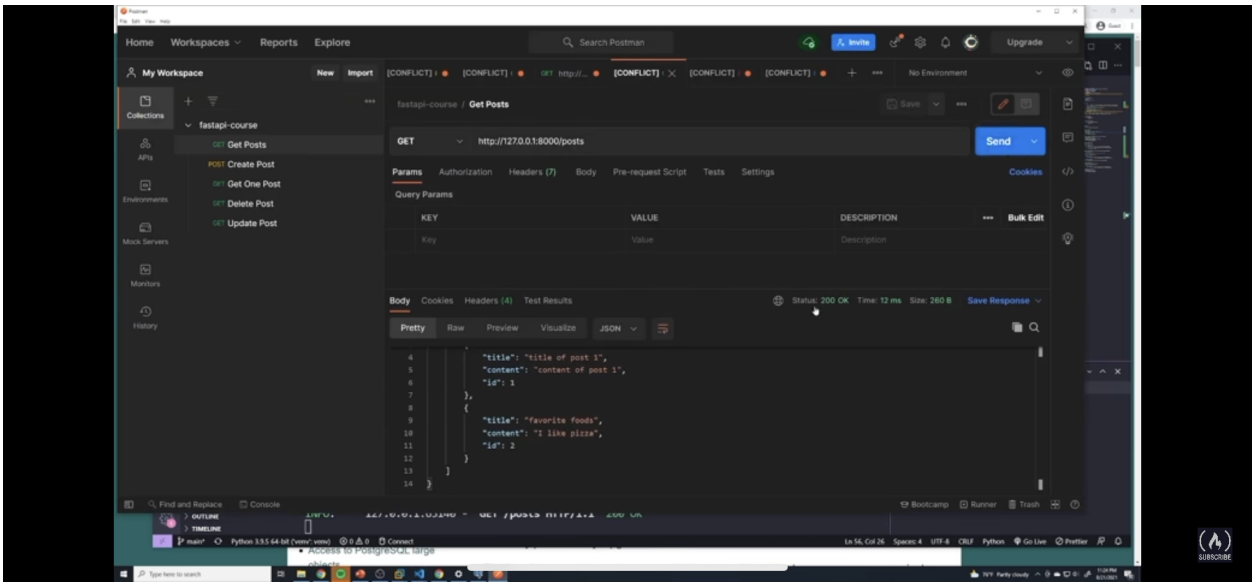
Add ability to keep trying to connect but add a wait period using a while loop

- import time libary

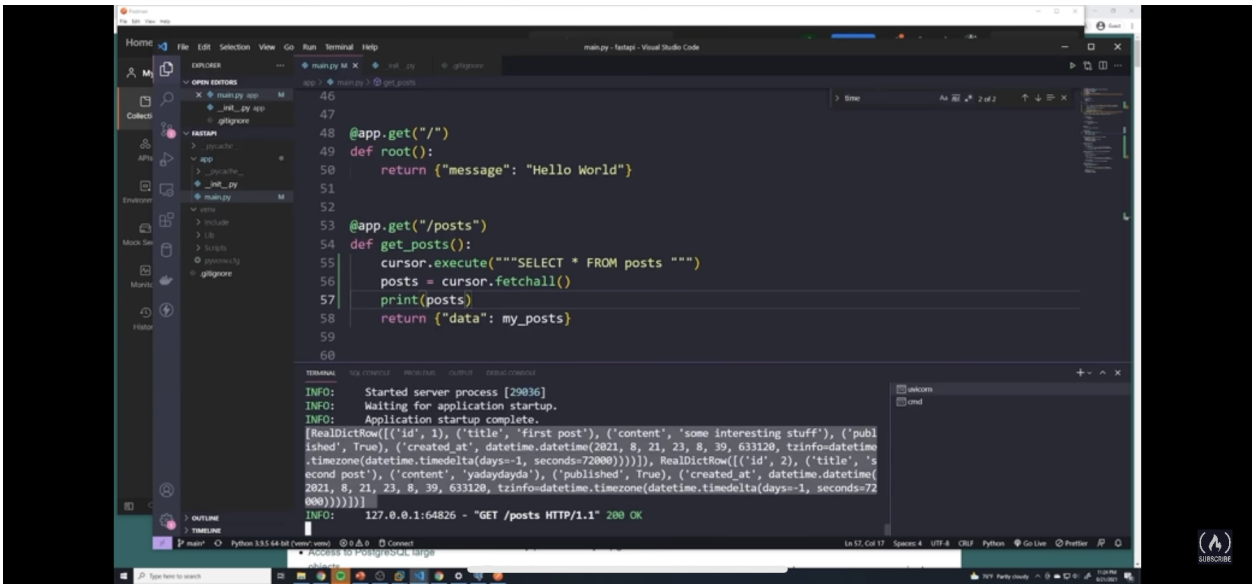


Connecting our database to http methods

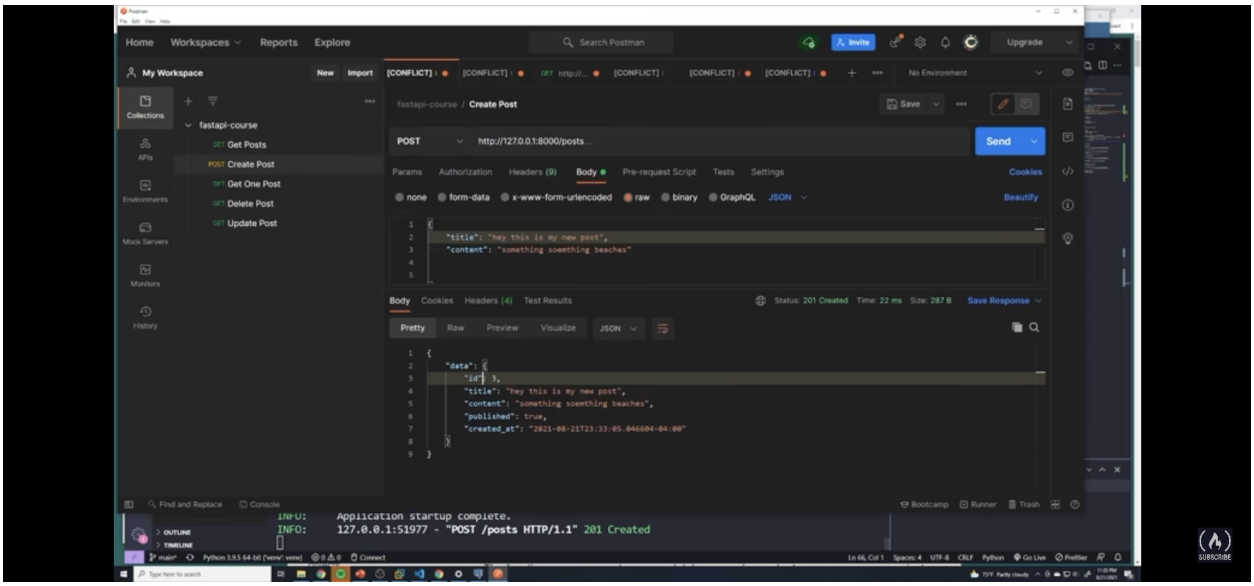
Write the sql query using the cursor method

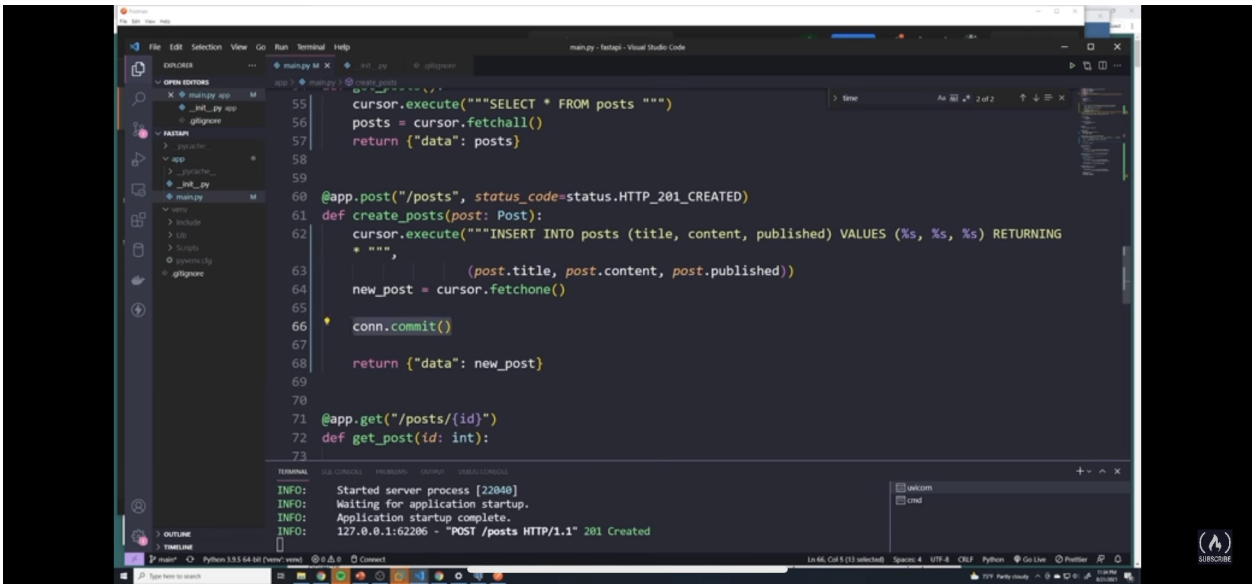


Now retrieved data from db



Connecting a post request to our db



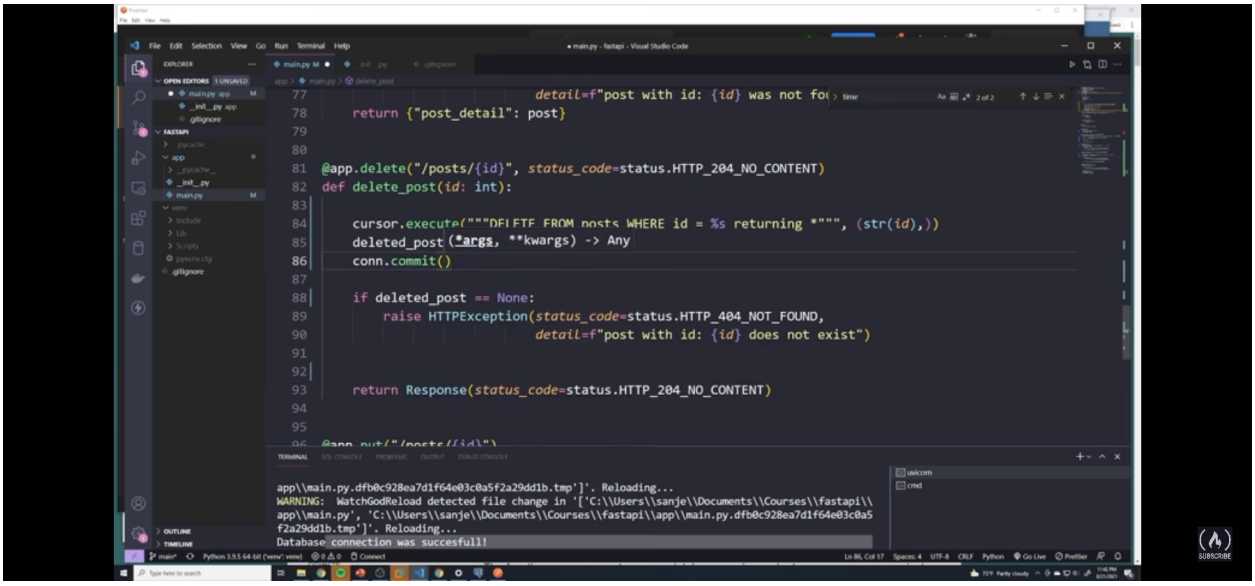


Need conn.commit() to actually move staged changes to db

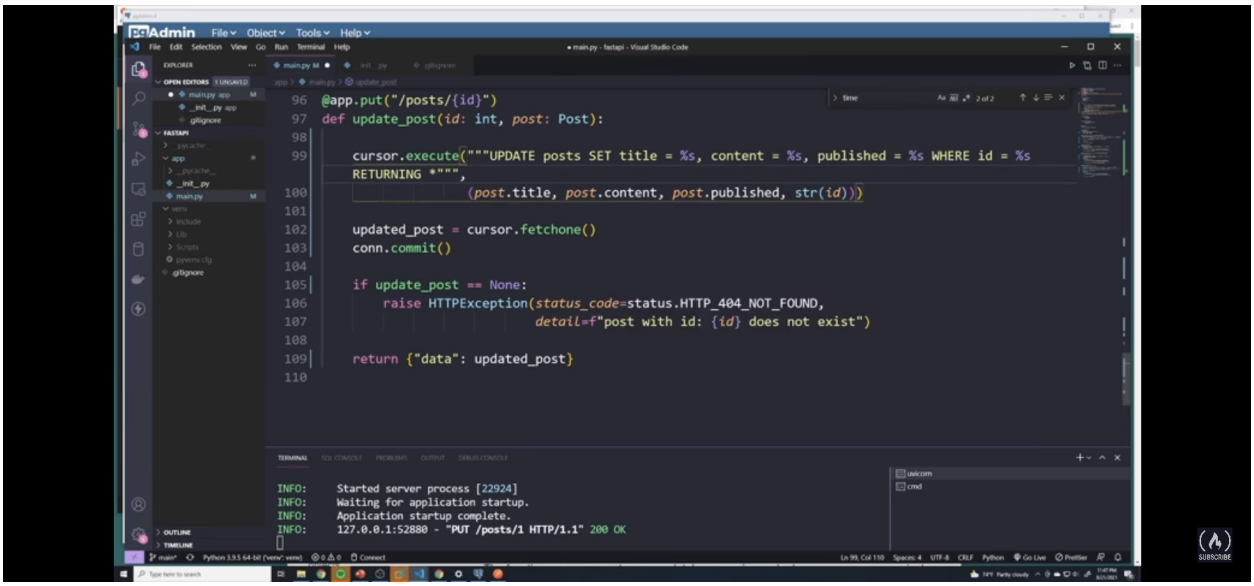
Connecting db to Get Post specific Id

```python
@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}
```

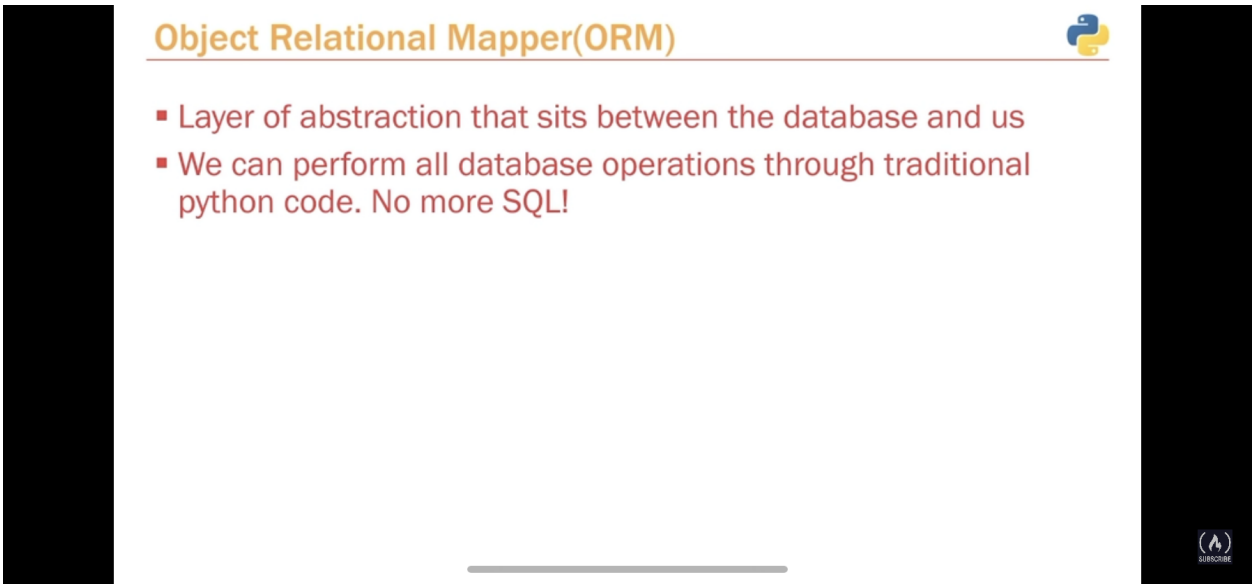
Connecting Db to Delete Post



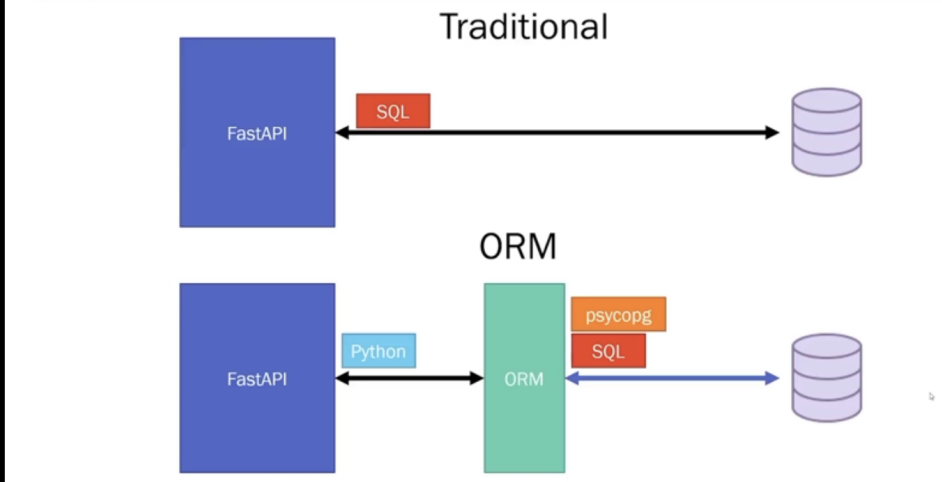
Connecting Db to Update Post specific Id



Using an ORM (SQLALCHEMY)



Object Relational Mapper(ORM)



What can ORMs Do

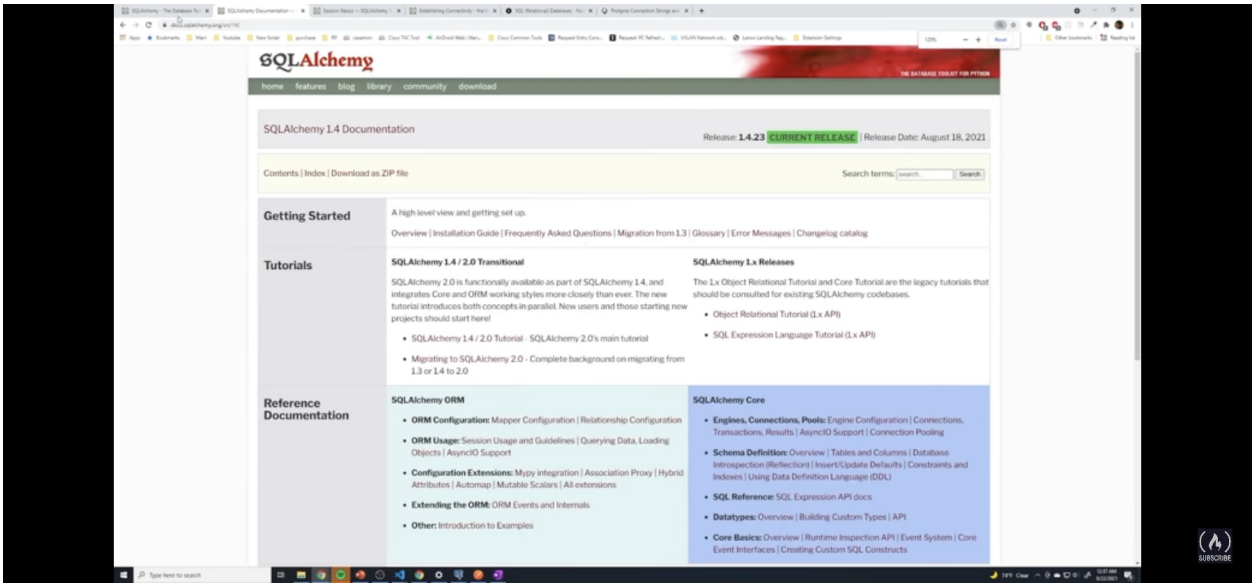
- Instead of manually defining tables in postgres, we can define our tables as python models
- Queries can be made exclusively through python code. No SQL is necessary

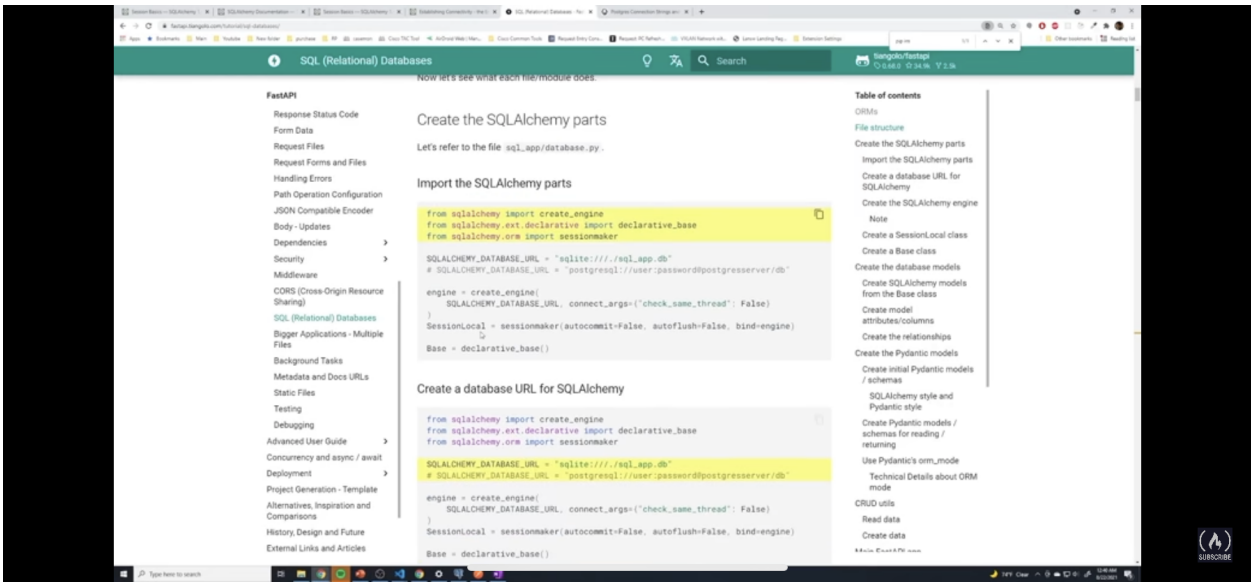
```python
class Post(Base):
    __tablename__ = "posts"

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

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

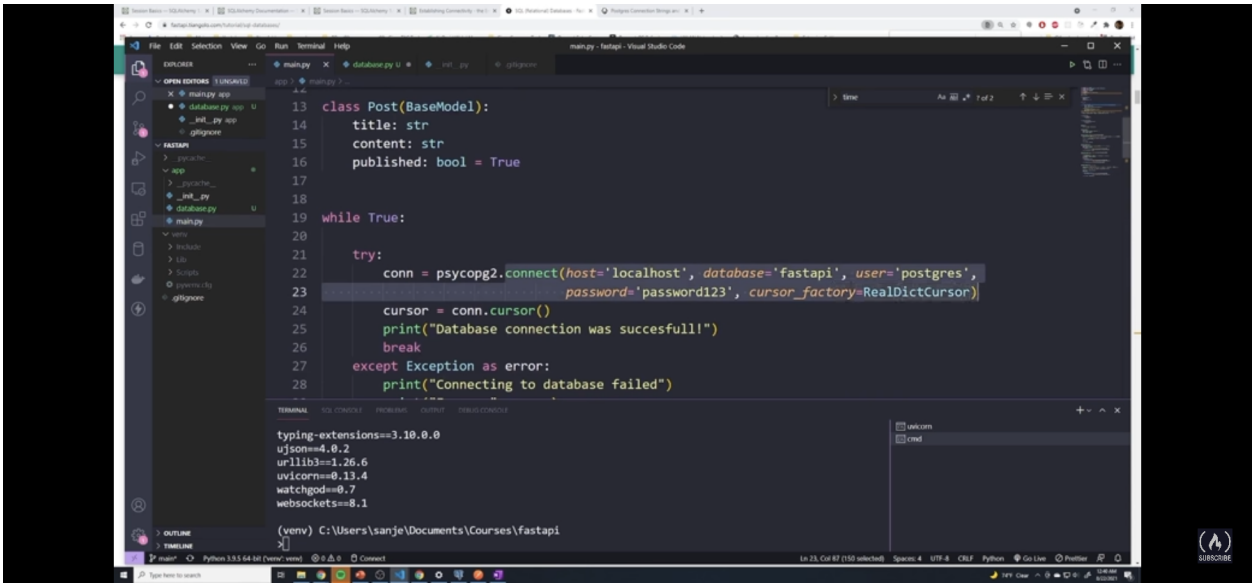
SQLALCHEMY

- Sqlalchemy is one of the most popular python ORMs
- It is a standalone library and has no association with FastAPI. It can be used with any other python web frameworks or any python based application

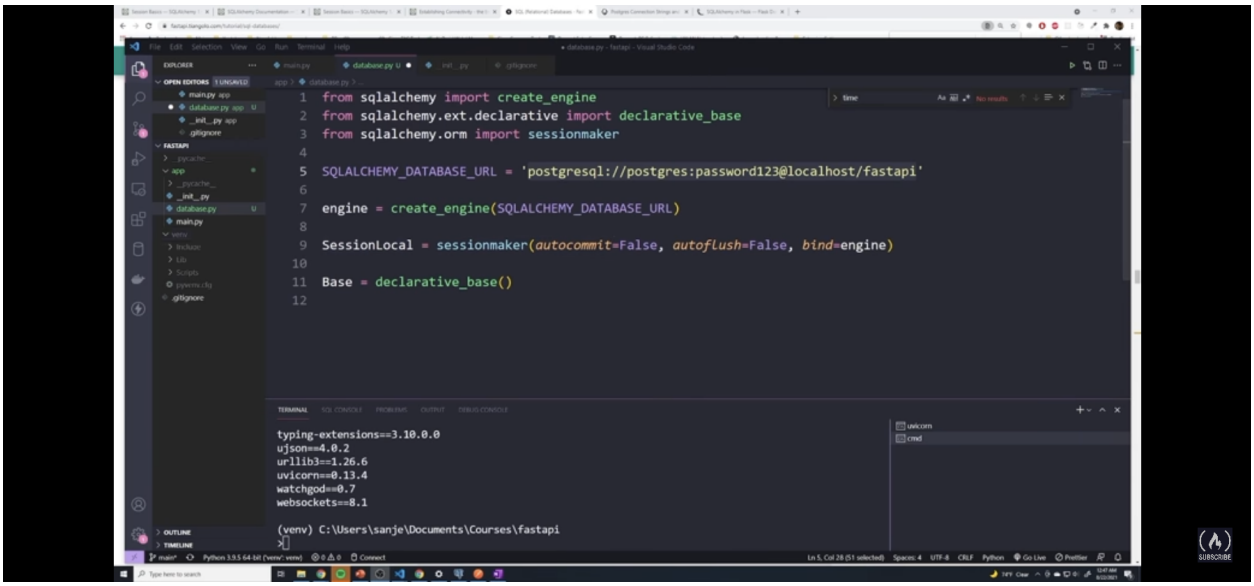




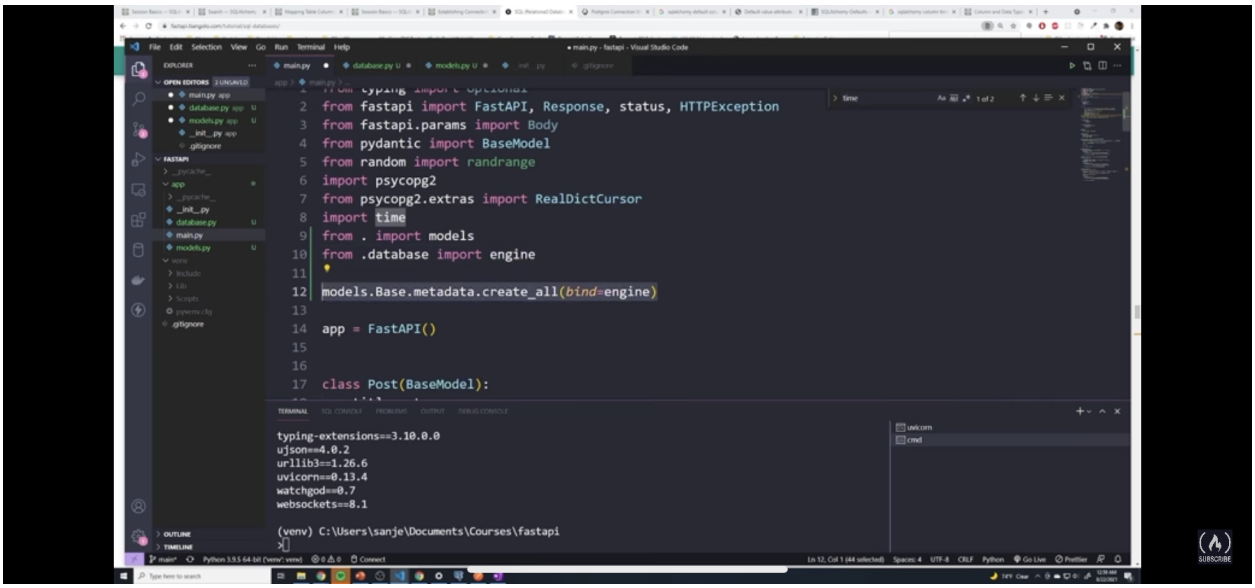
Before we connected on our main file

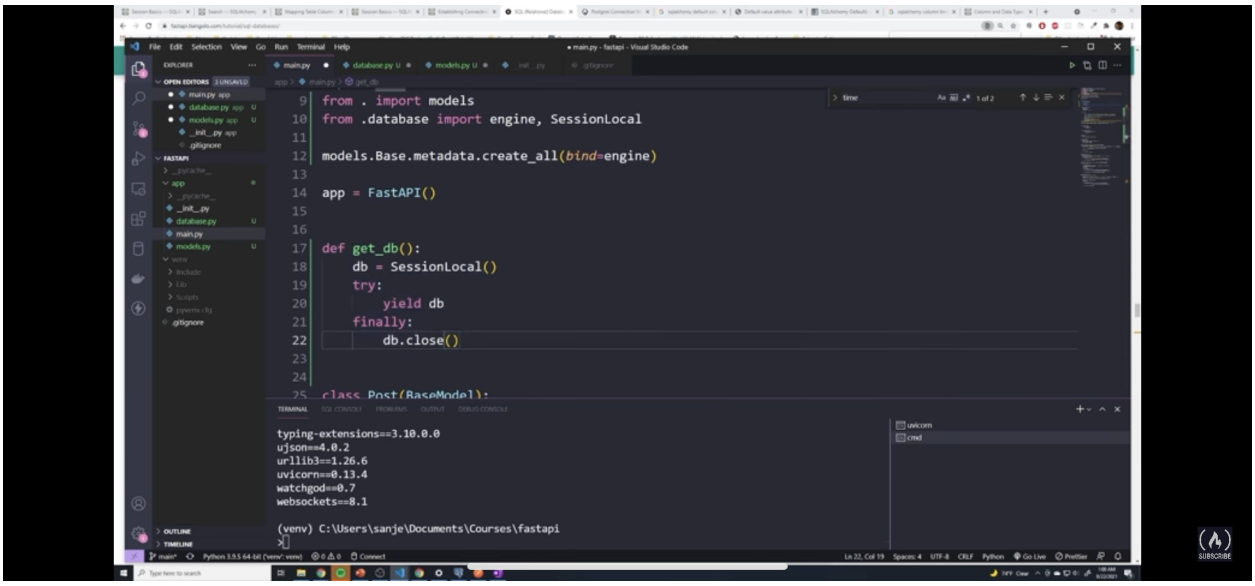


Create a database.py file to connect

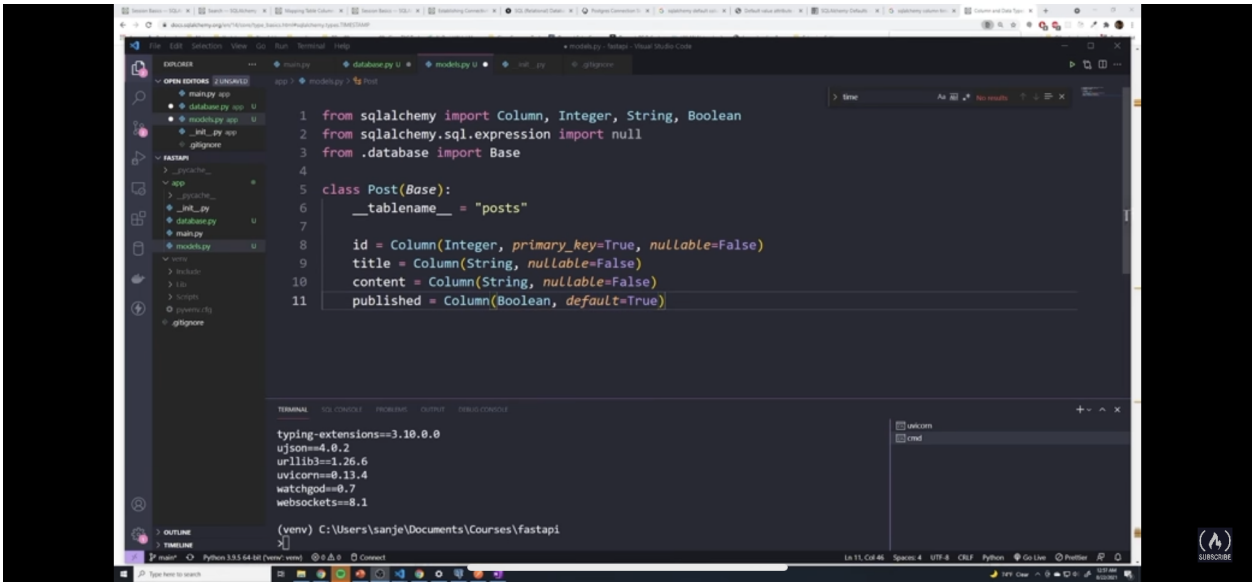


Creating db Models



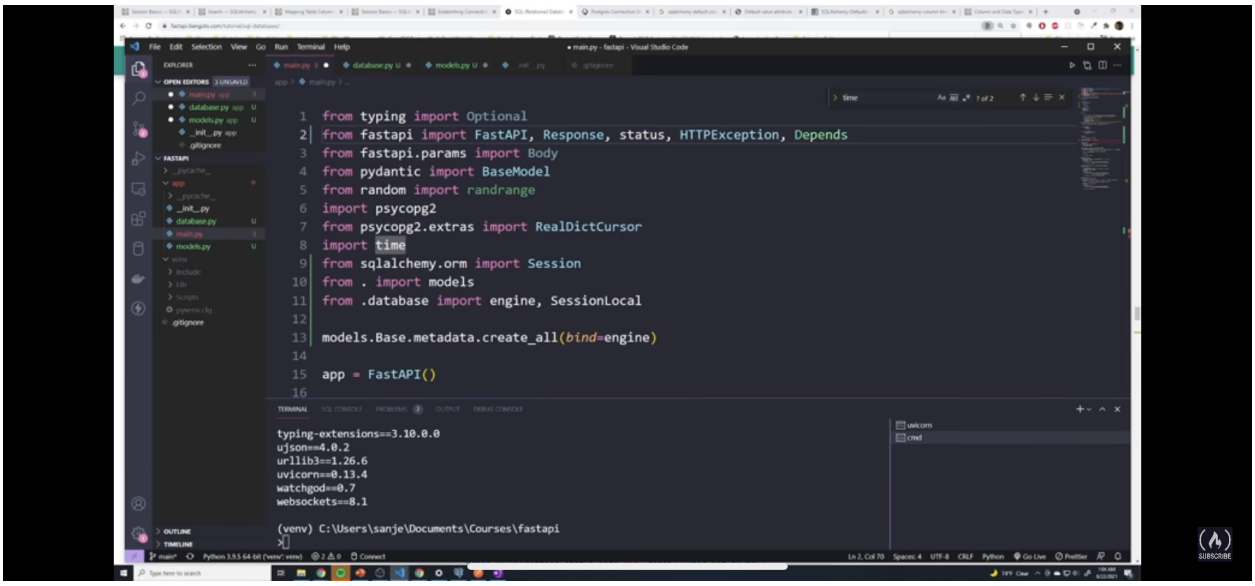


Create a model for the db

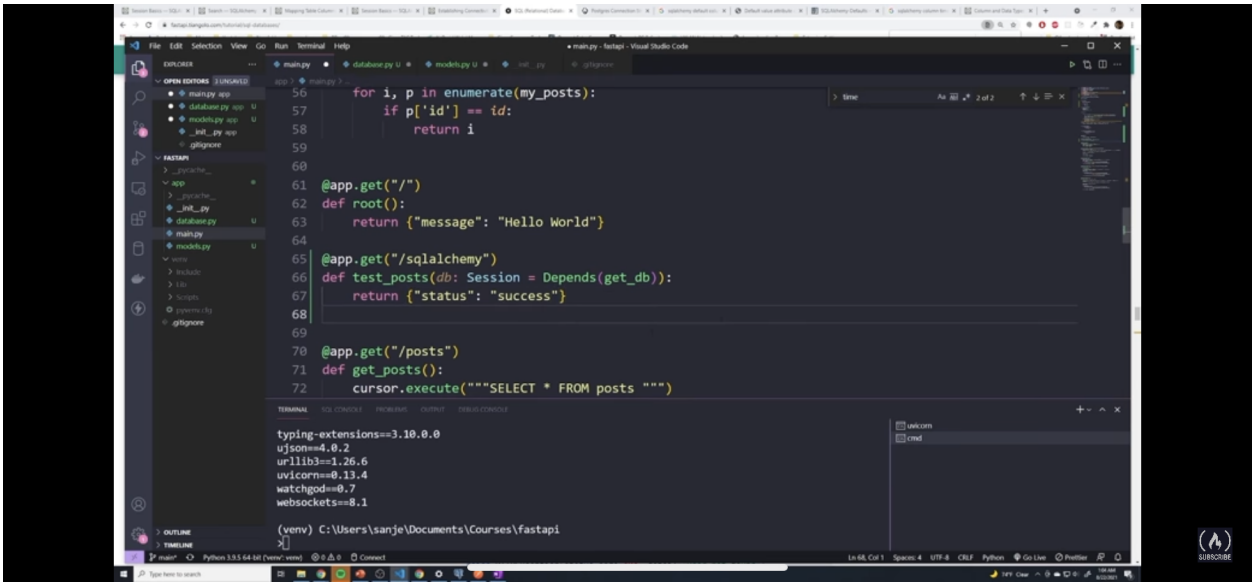


Sessions

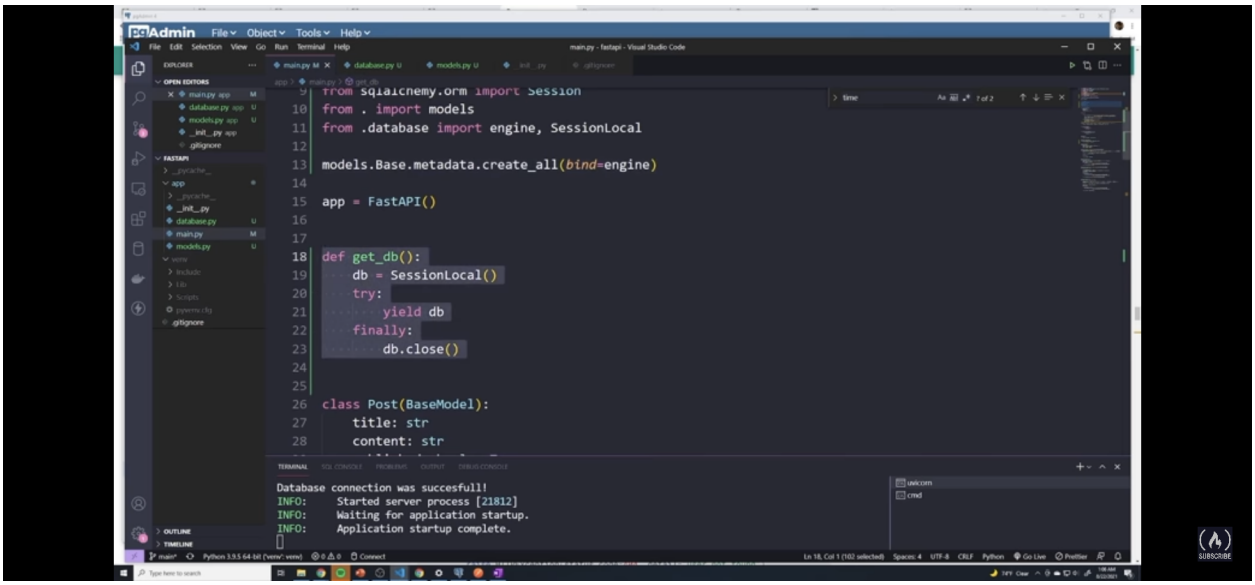
Import sessions from sql alchemy and depends method from FastAPI



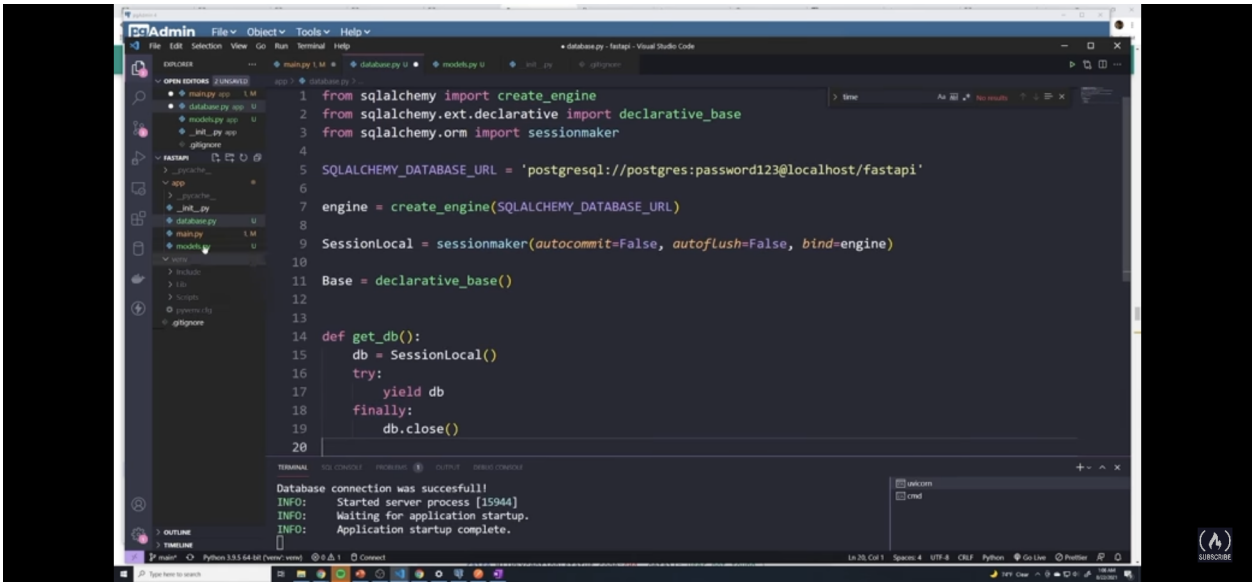
update get('/sqlAlchemy')



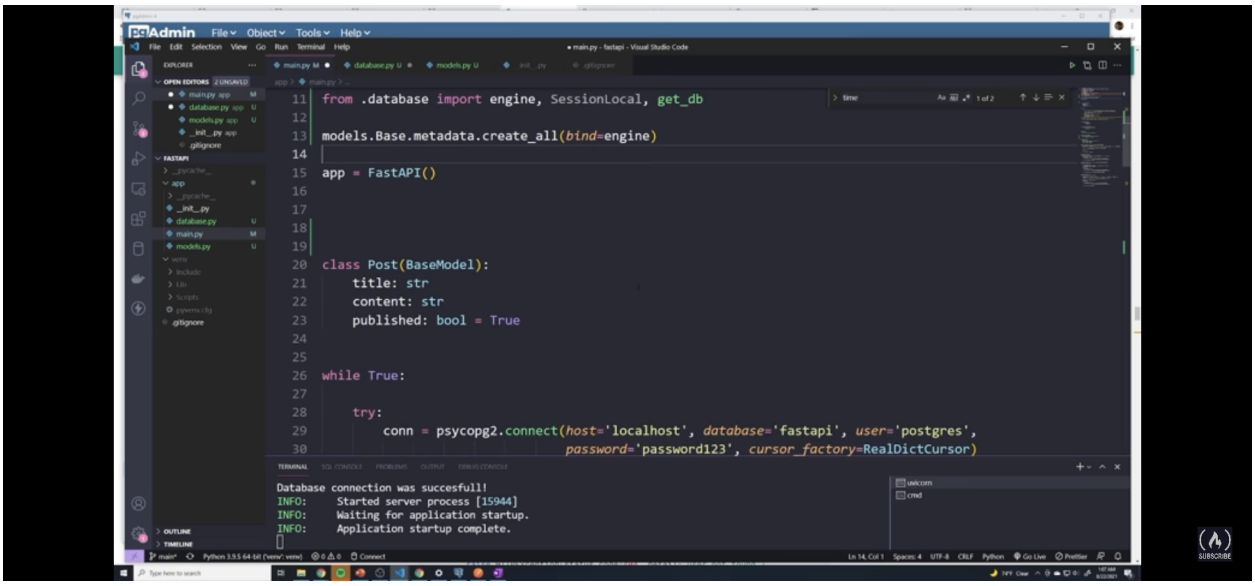
Clean up API



- move db related code into db file

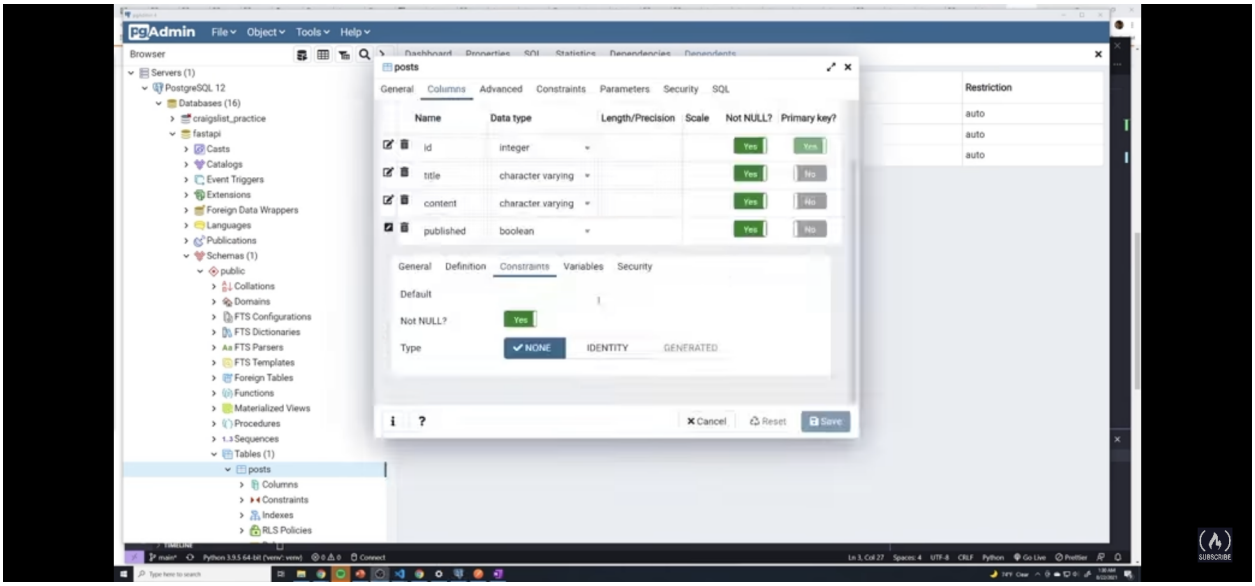


Import the function

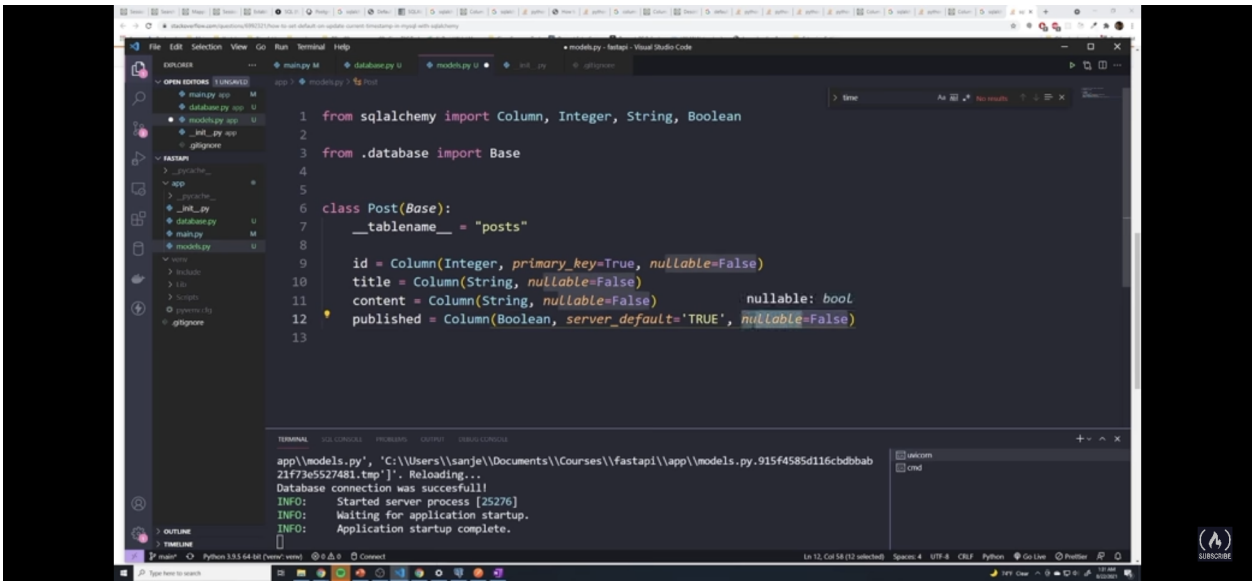


Adding Created at field

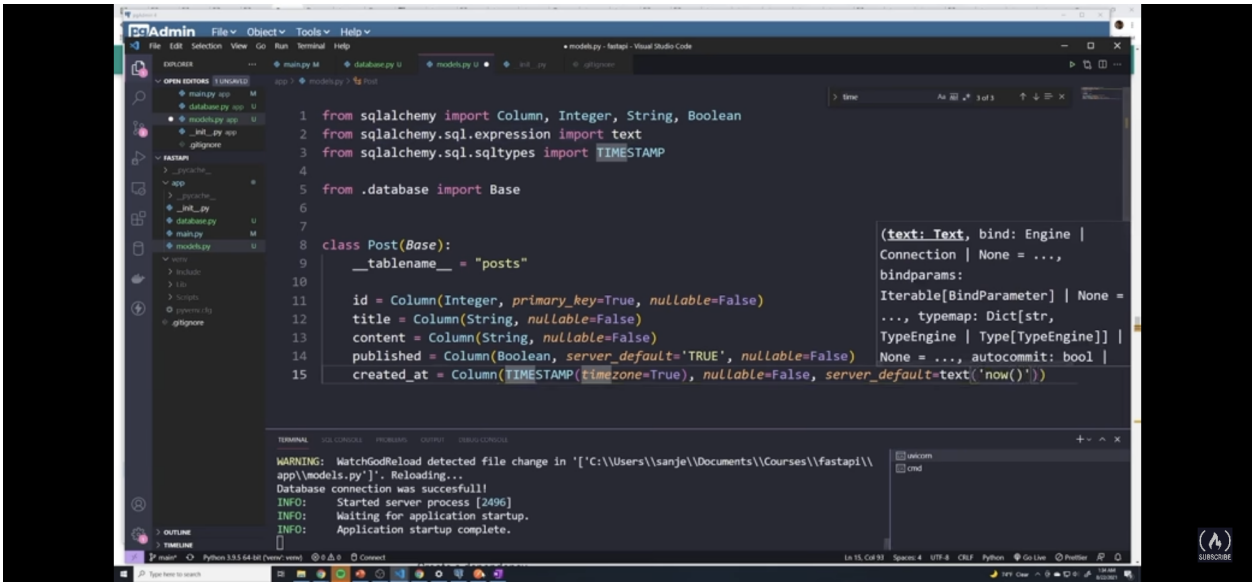
No default set



Need to update our model

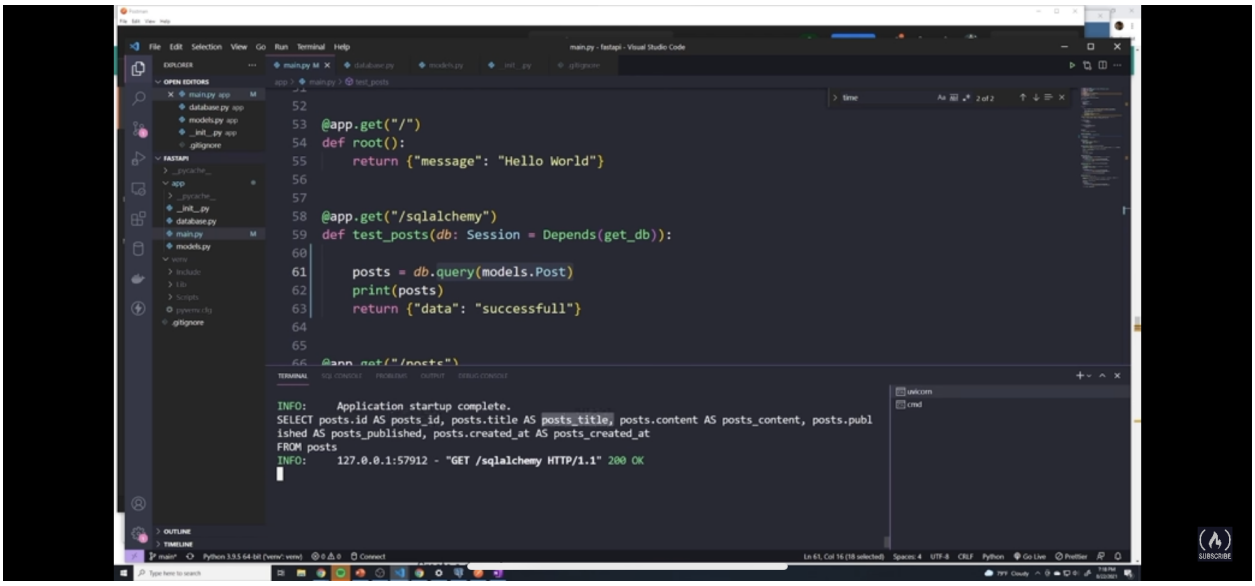


Add in Created_at functionality with defaults



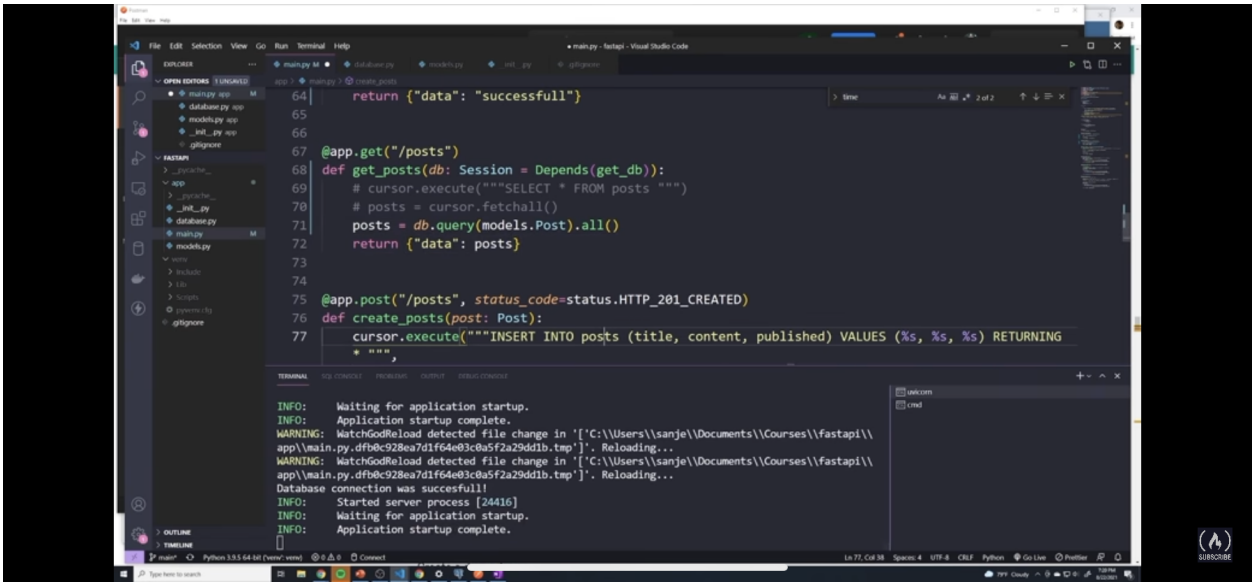
SQL alchemy has now created the correct tables

- If you print the statement created
- Looks like sql

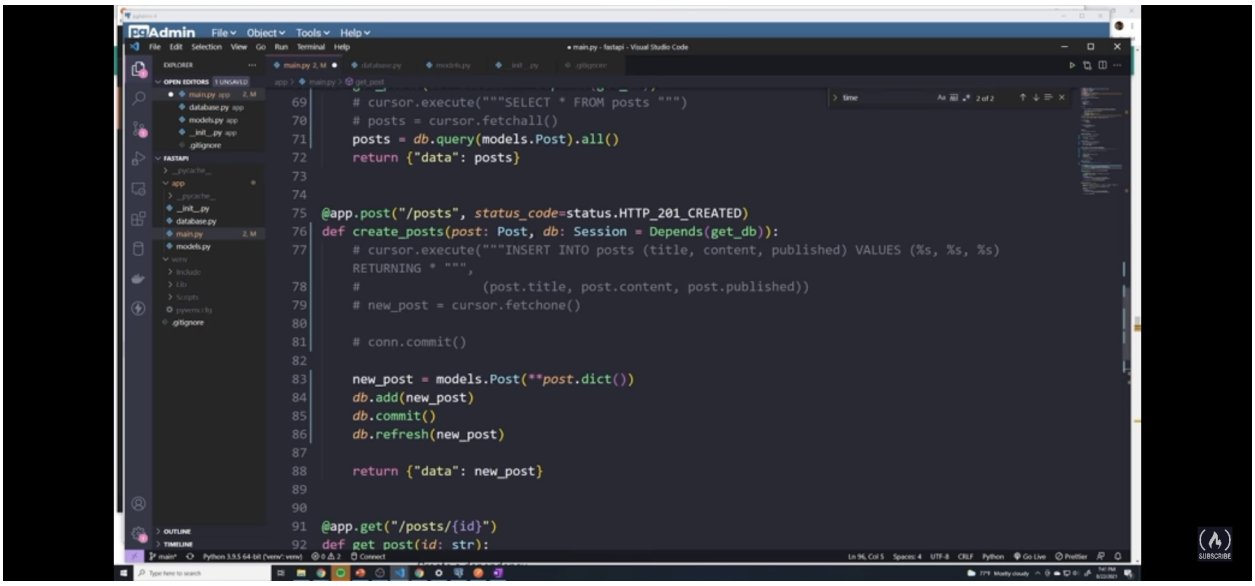


Update methods to use SQLALCHEMY

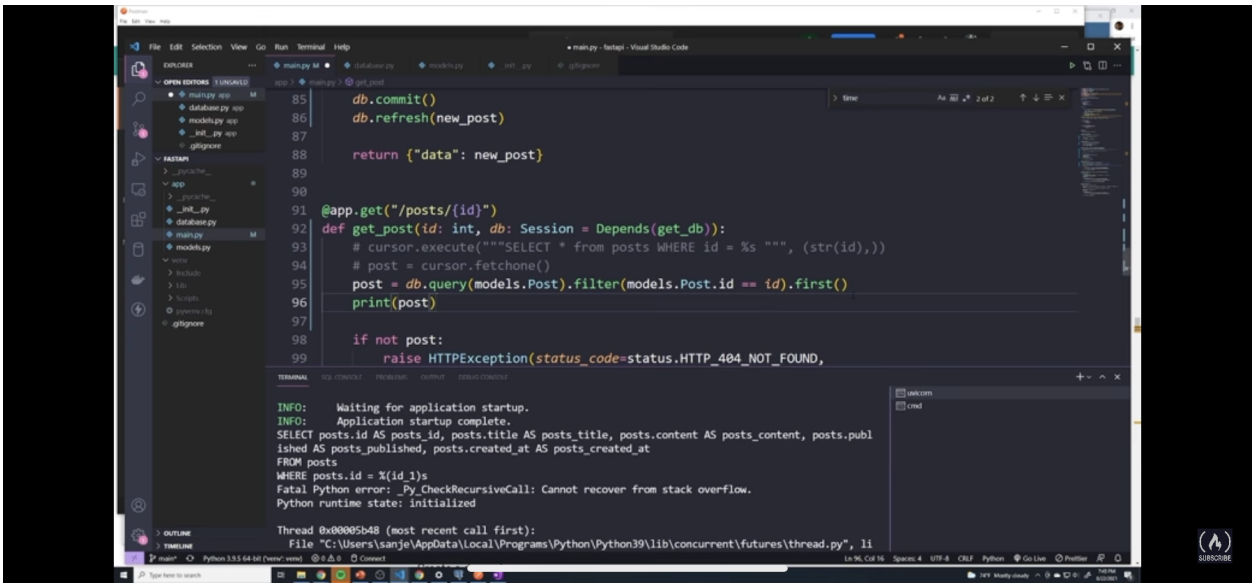
Get



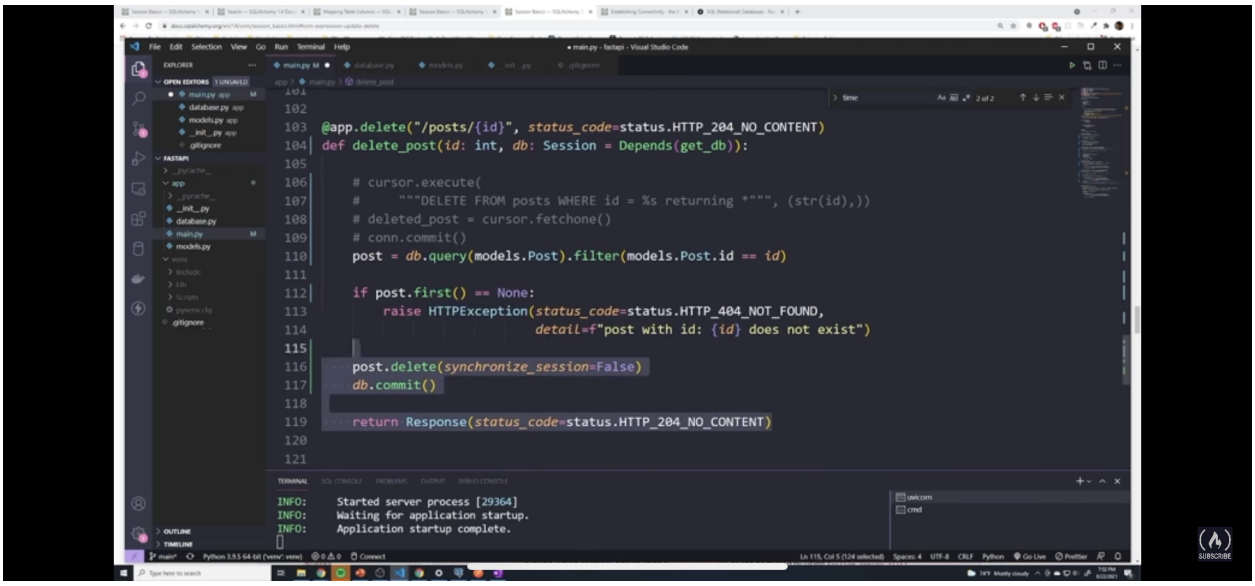
Post



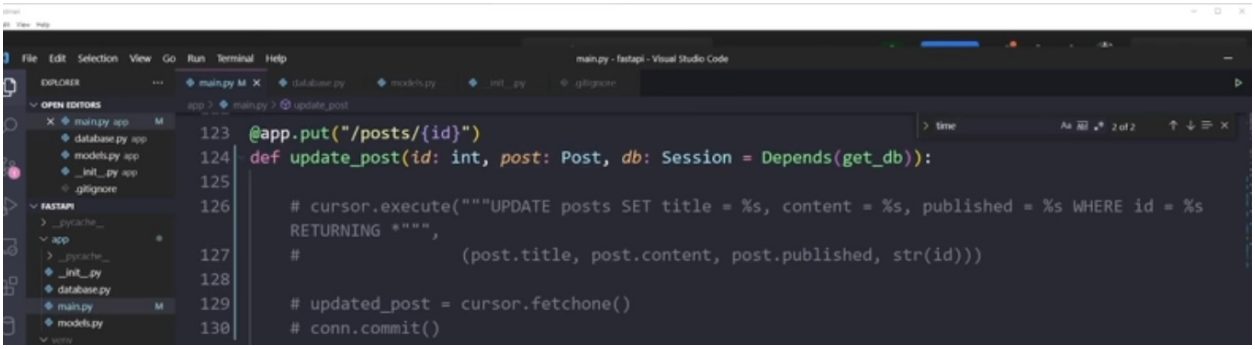
Get(id)

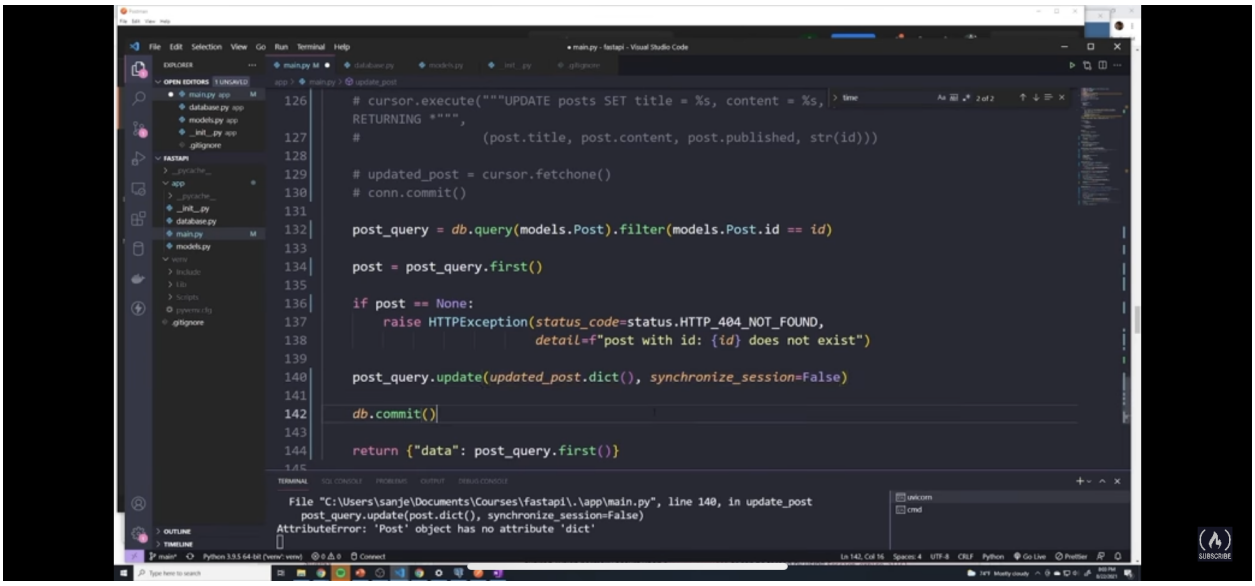


Delete(id)


@app.delete("/posts/{id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_post(id: int, db: Session = Depends(get_db)):
# cursor.execute(
# """DELETE FROM posts WHERE id = %s returning *""", (str(id),))
# deleted_post = cursor.fetchone()
# conn.commit()
post = db.query(models.Post).filter(models.Post.id == id)
if post.first() == None:
raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail=f"post with id: {id} does not exist")
post.delete(synchronize_session=False)
db.commit()
return Response(status_code=status.HTTP_204_NO_CONTENT)
INFO: Started server process [29364]
INFO: Waiting for application startup.
INFO: Application startup complete.

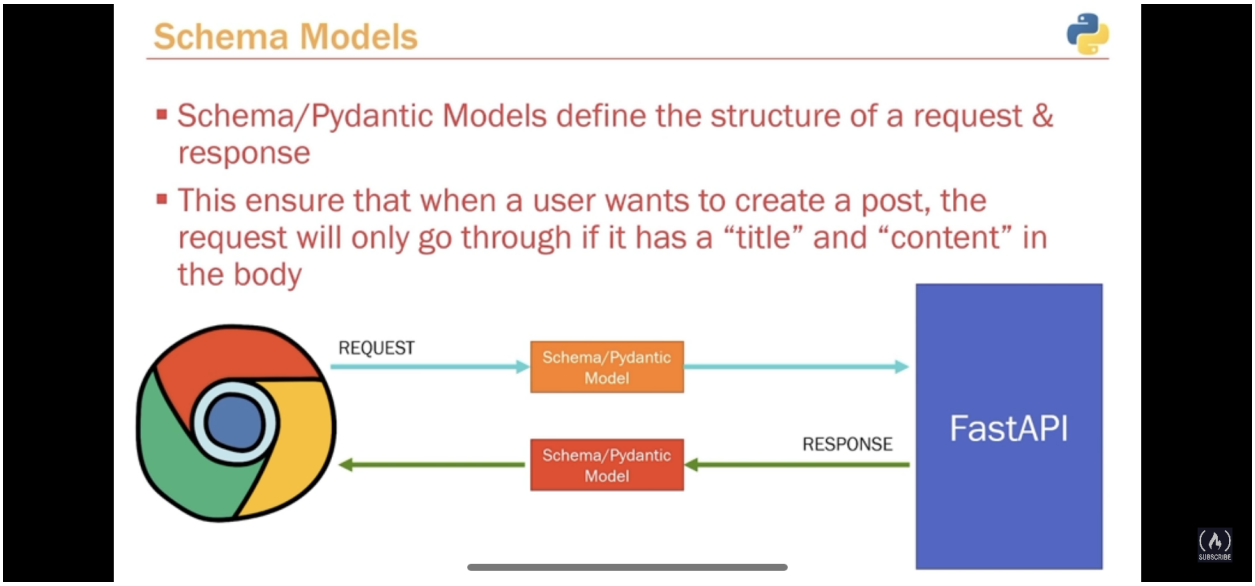
Update

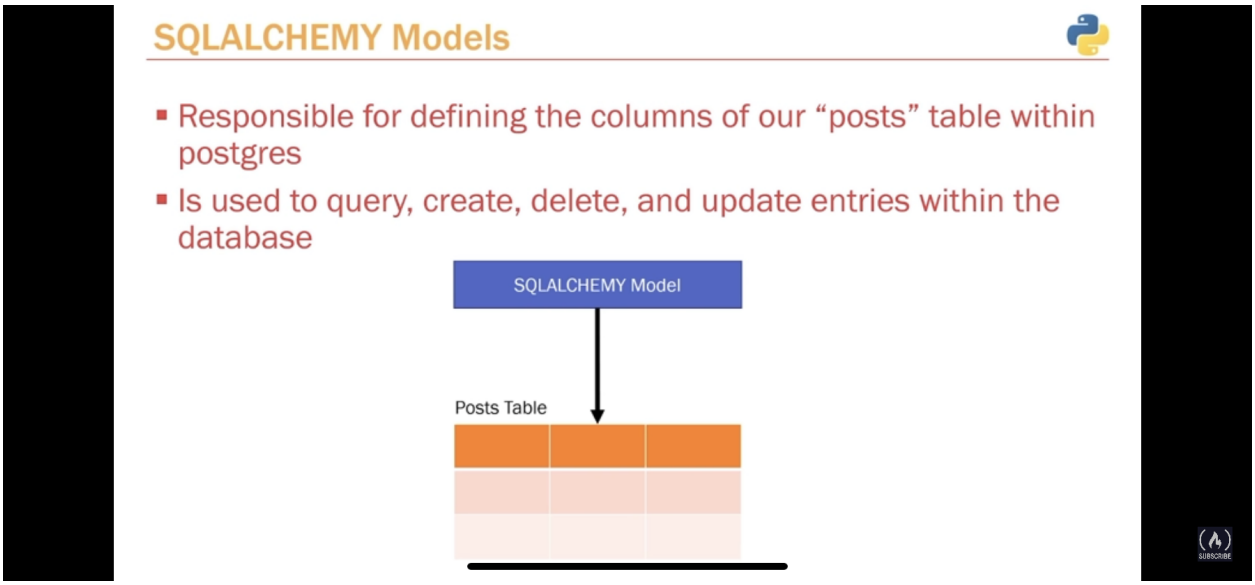

@app.put("/posts/{id}")
def update_post(id: int, post: Post, db: Session = Depends(get_db)):
# cursor.execute("""UPDATE posts SET title = %s, content = %s, published = %s WHERE id = %s
RETURNING *""",
(post.title, post.content, post.published, str(id)))
updated_post = cursor.fetchone()
conn.commit()



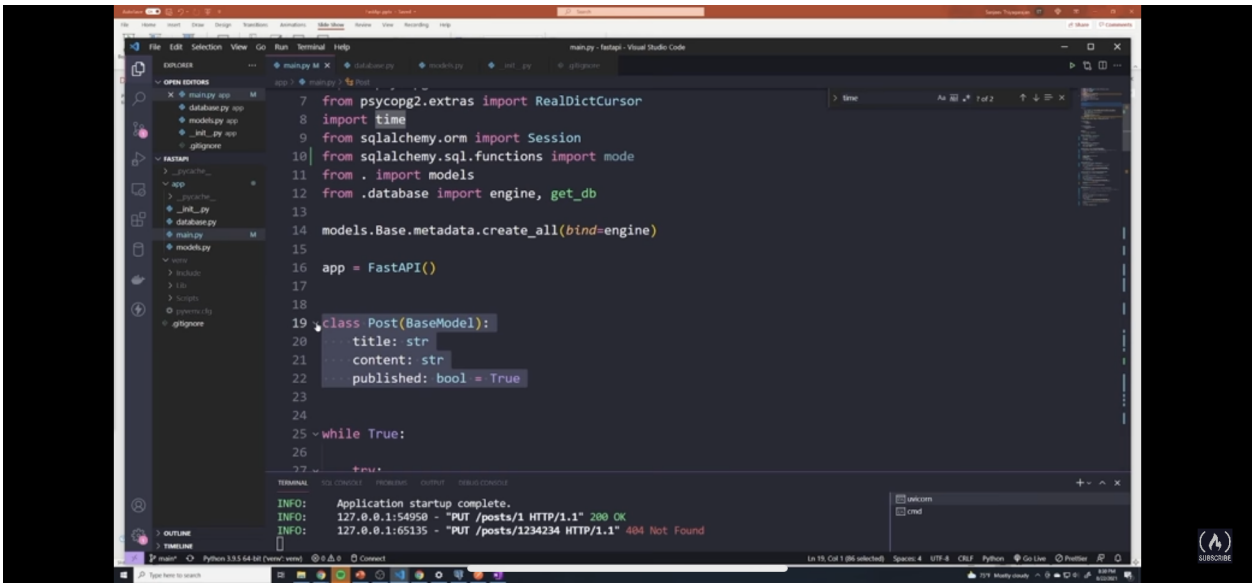
Schema Models

We don’t want to give the client the freedom to send what ever data they want

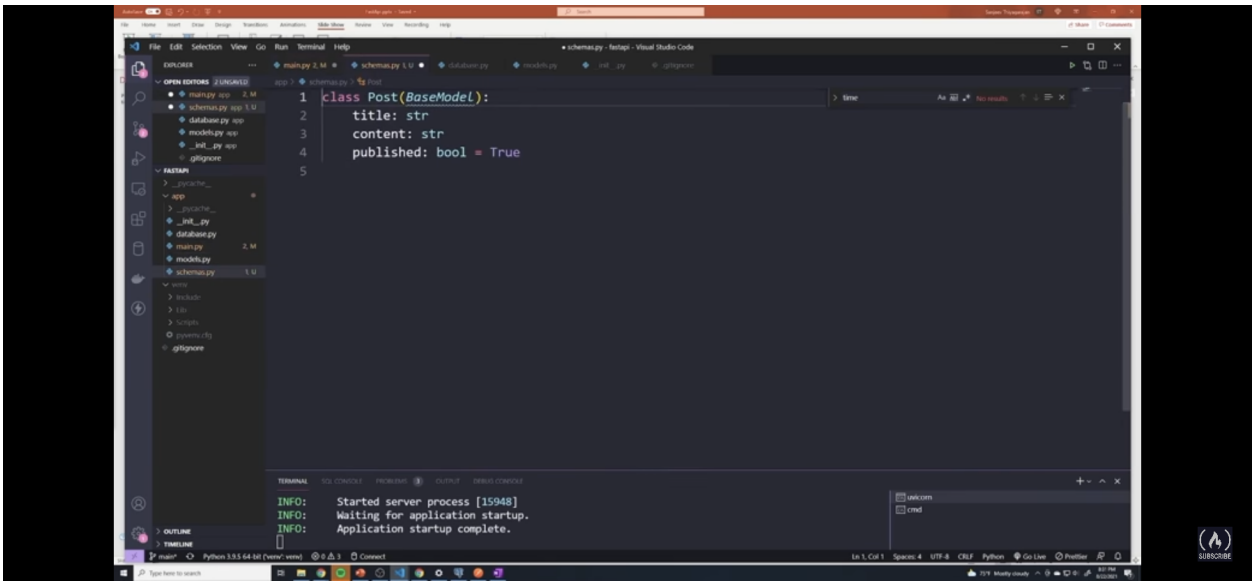




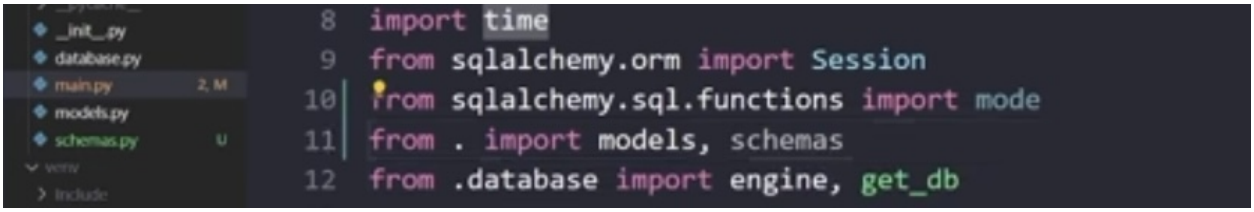
Schema folder



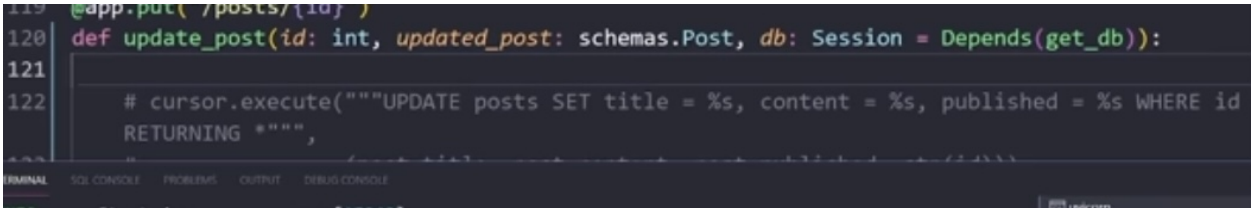
De-clutter the main file



Import schemas

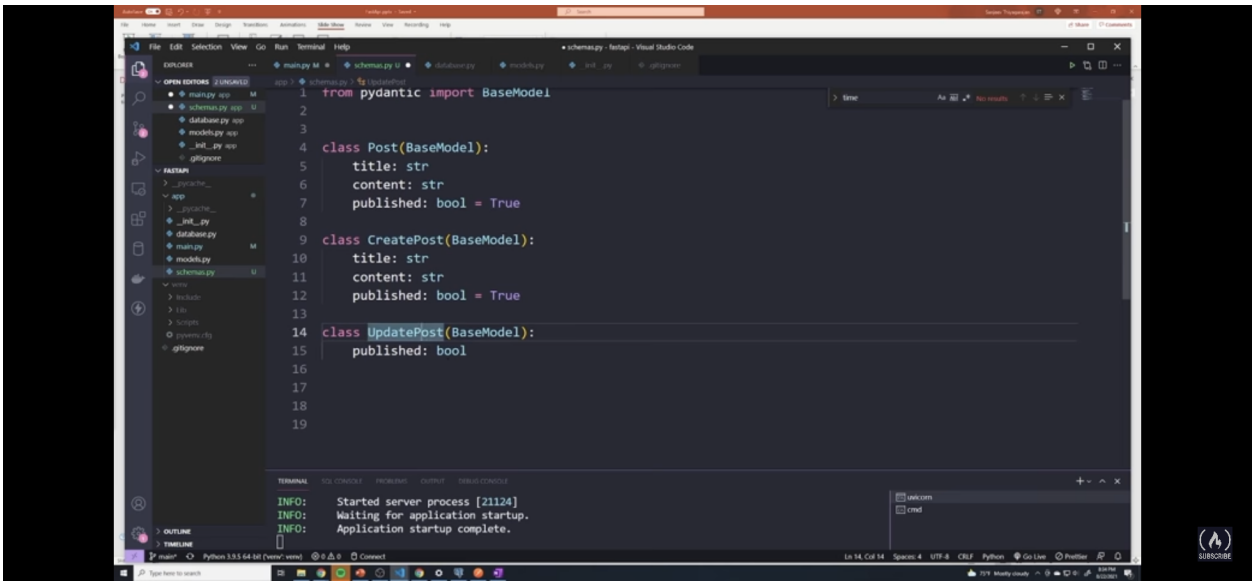


Now it's schemas.Post for the class

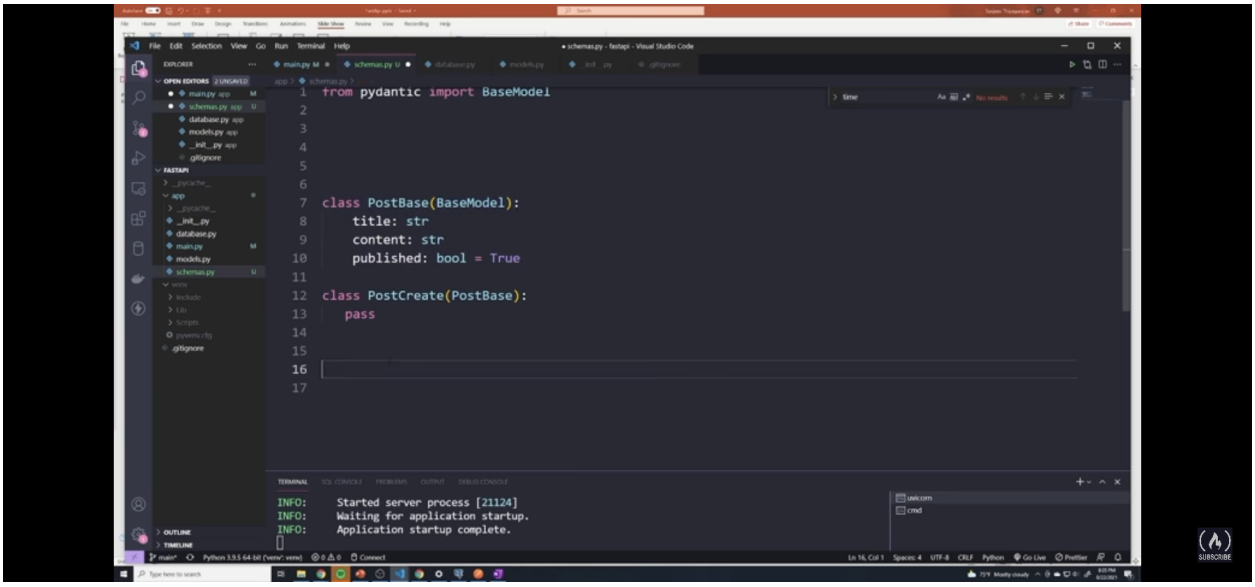


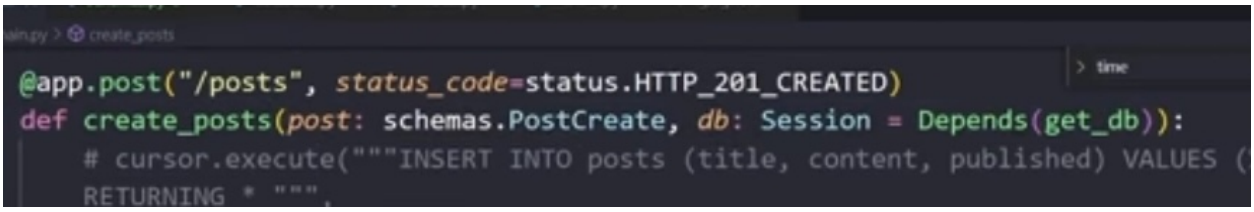
Creating new Schemas for Requests

- you could create loads of individual schemas



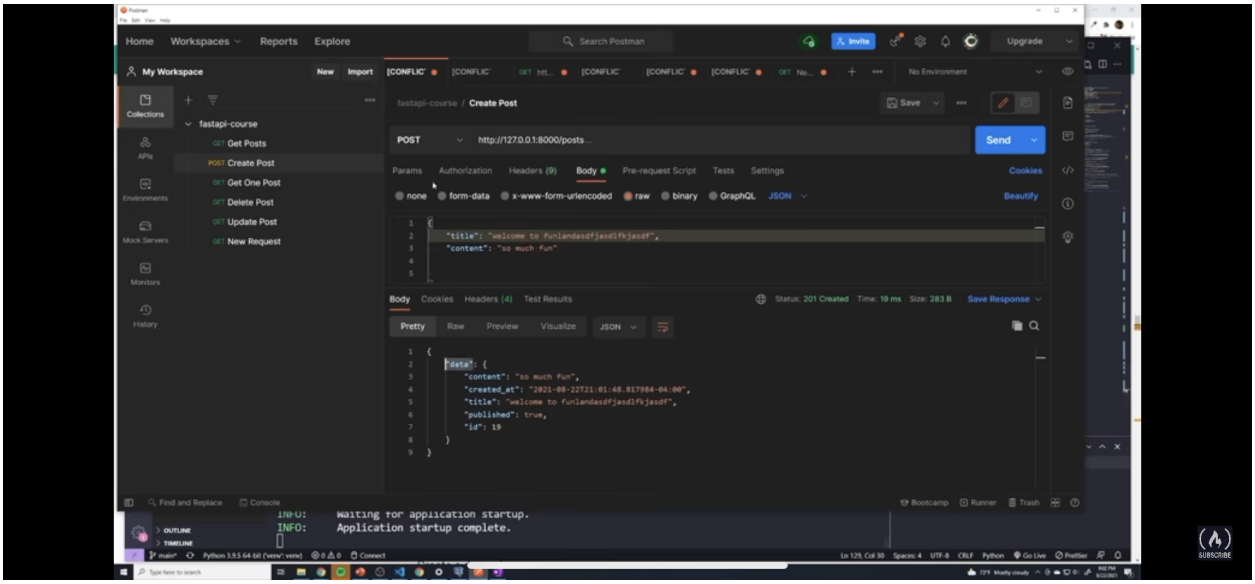
Or use inheritance from a base schema



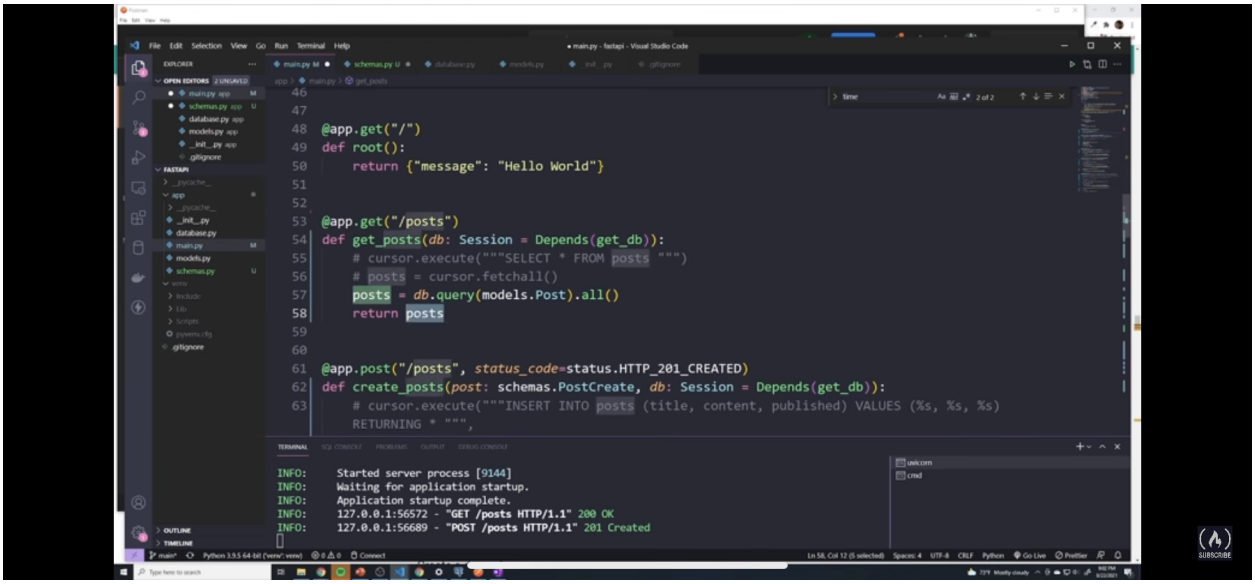


Creating schema’s for responses

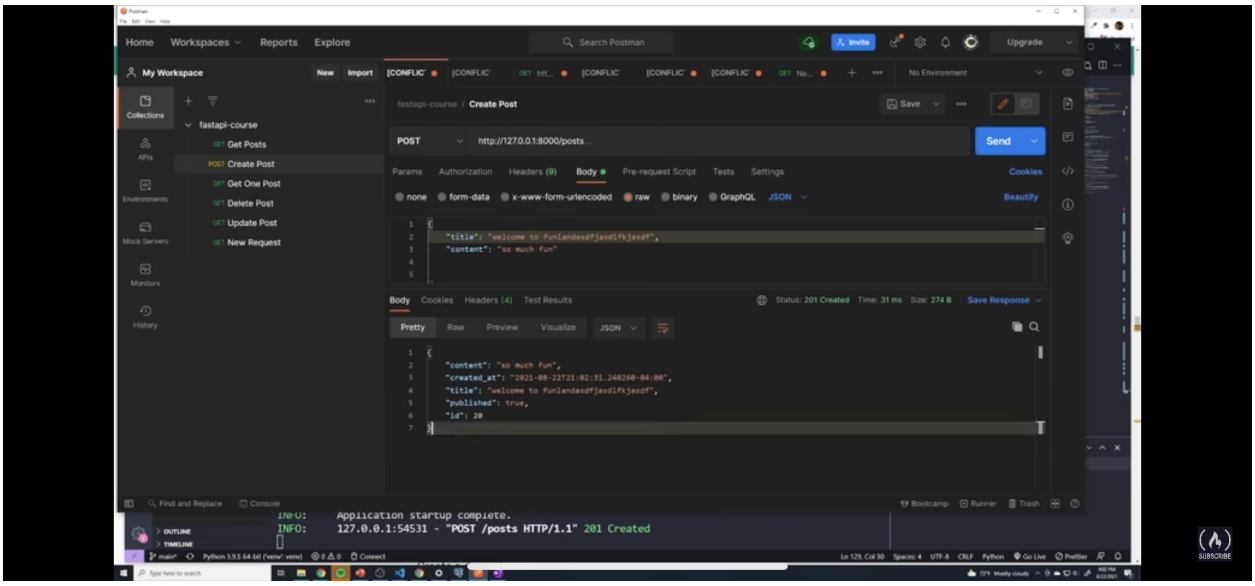
Remove returning a dict for all (fast api auto-serialises)



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

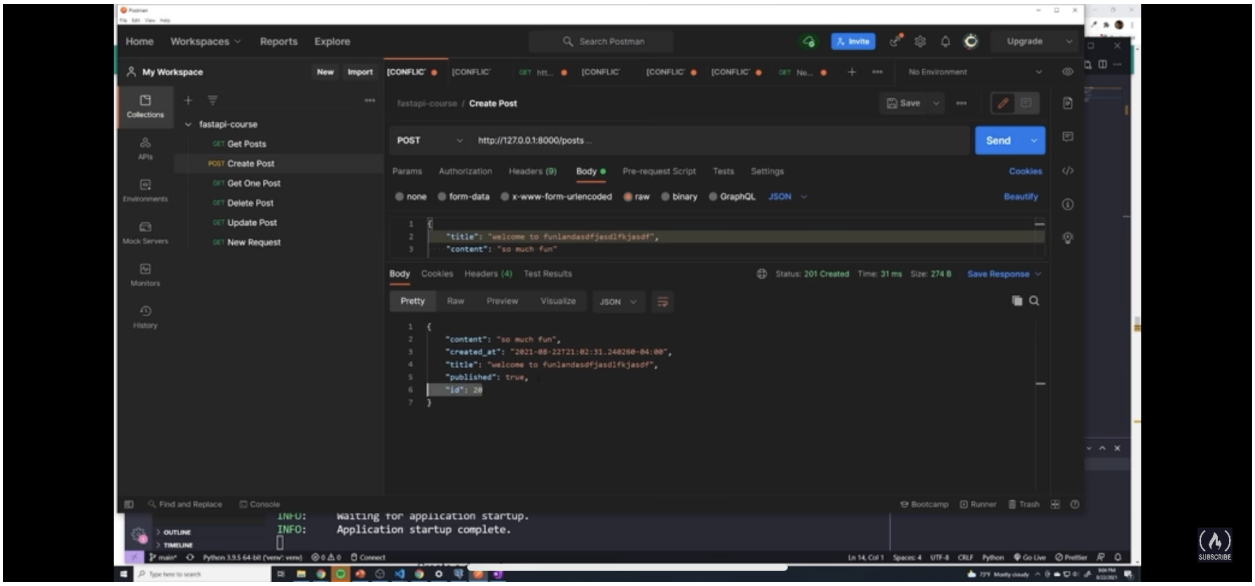


Now is more clean

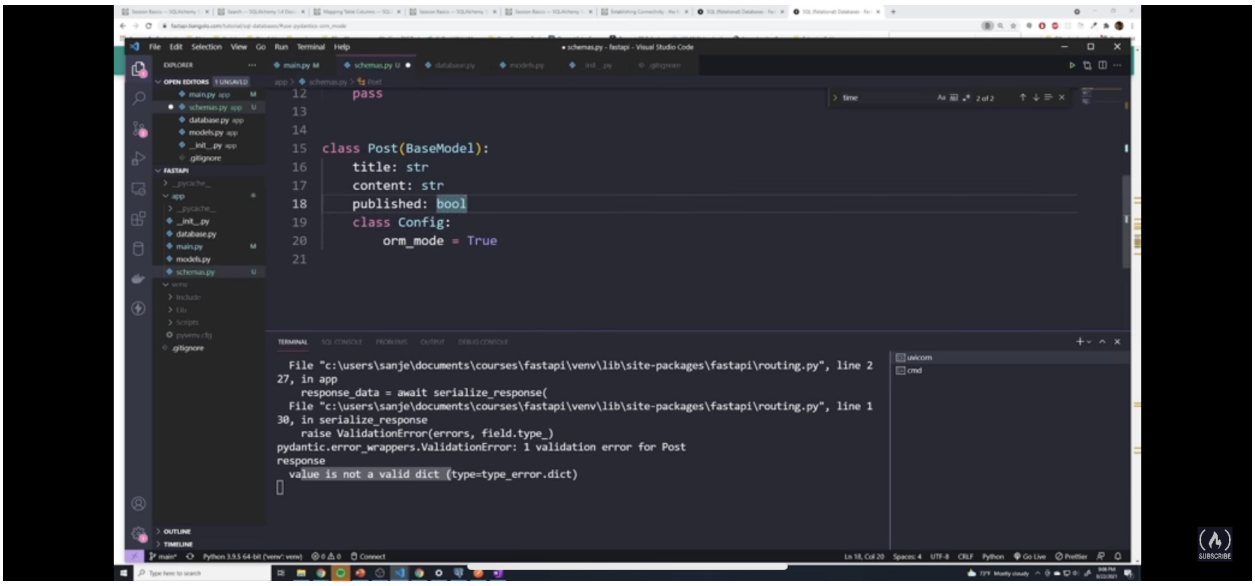


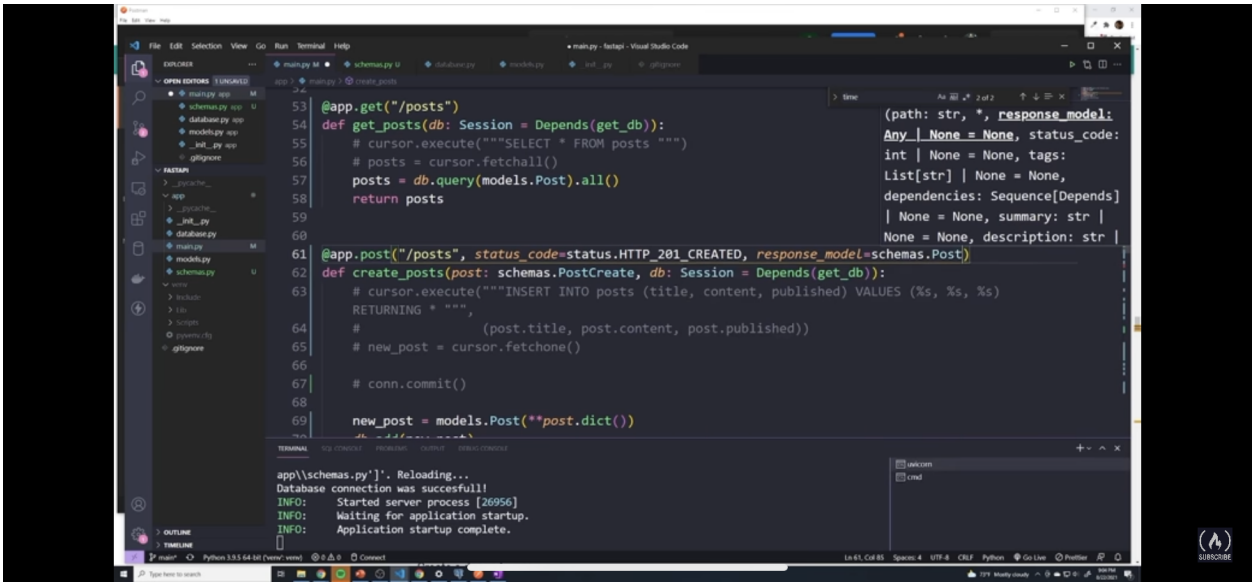
Defining our response model

Previously returned all data from reponse

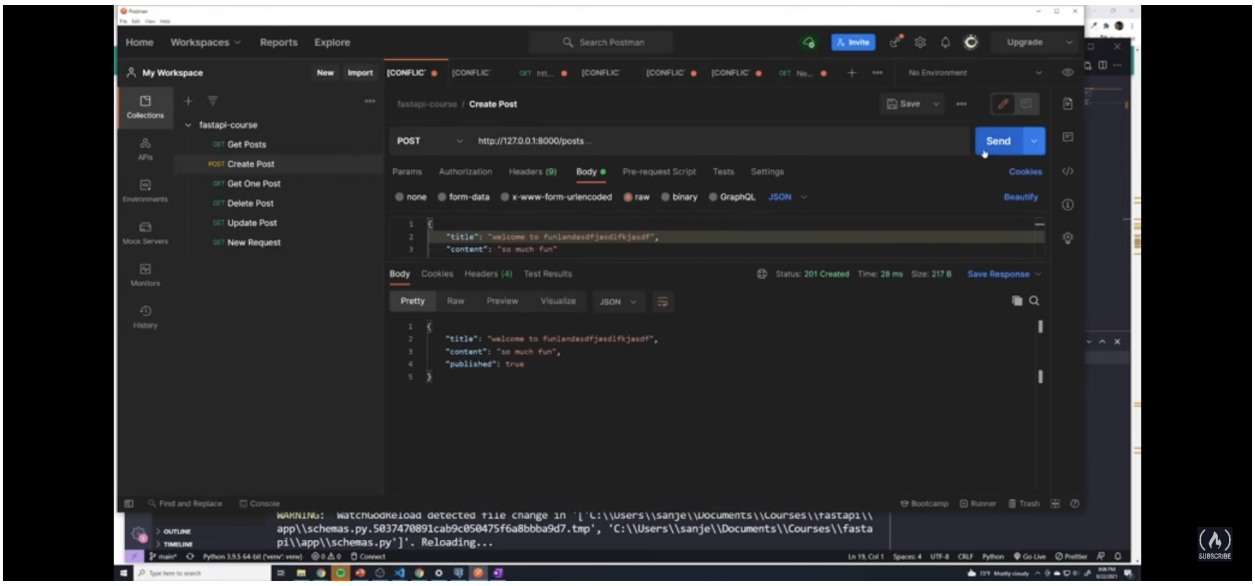


In our model - we only want to return certain info to the client





Now they only see limited info



Or add other fields to return