**Fast API**

Used for web development. Fast in performance. Fast for developing as well.   
You can create RESTful APIs, Backend communication with better security. Quick and Easy to use and learn. It has OpenAPI standards.   
Handles all business logic. FA has capability for Full stack which includes front end as well.  
  
pip install fastapi  
pip install uvicorn

pip install fastapi  
pip install uvicorn – this is the server. Django has inbuilt server. But   
uvicorn filenamewithoutextension:nameoffastapiinstancewhichisapp --reload   
The above will start the server. You will see the url in logs, when opened in browser, you can see output.  
You can add /docs to url to get API swagger. We can test API from here. We don’t need postman etc.

FastAPI has swagger UI making it easy to see what all APIs are provided by Project. You can open by url/docs  
  
After coding, to run the application on server, run below command  
uvicorn filename:appwhichisinstanceofFastAPI –reload  
--reload makes sure everytime you update the code, it reloads the app with new changes  
  
Path parameters – which comes as part of url - /getuserbyid/1  
  
Order of API creation matters with path parameters. For ex: you could have /books/book and another as /books/{dynamic\_param}  
  
Query parameter is url?parameter=value  
If you want to pass a path or query param value which contains space, use %20 in place of space  
You can have both pathparam and queryparam  
url/{pathparam}?{queryparam}=value  
  
@app.get(“/getstudent/{classnumber}”)  
async def get\_student(classnumber: str,studentnumber: str)

Pydantic   
used to create schema for data modelling, validation. You can validate both request or   
response data. Enums are used to define allowed values only.  
In POST/PUT, if you do not want to use pydantic for validating body, you can import Body from fastapi and use Body() instead of UserDetail below  
  
1. You can import Field from fastapi which can be used to set conditions like allowed, not allowed, range.  
When you declare the schema class, you can say id:str=Field(gt=0, lt=10)  
Similarly, you can use, min\_length, max\_length etc.  
  
id: Optional[int] = None  
You can also write this using Field.  
id: Optional[int] = Field(description=”ID is not needed”,default=None)  
  
2. Within the schema class, you can define below structure to show example which can be seen on swagger UI.   
  
    model\_config = {  
 "json\_schema\_extra": {  
 "example": {  
 'id': "A new book",  
 "name": "give user detail"  
3. You can import Path from fastapi to validate path parameter  
 async def read\_book(book\_id: int = Path(get=0))  
 You can import Query from fastapi to validate path parameter  
 async def read\_book(book\_id: int = Query(get=0))

Using routes

|  |  |
| --- | --- |
| from fastapi import FastAPI, Body, HTTPException  from pydantic import BaseModel  from enum import Enum  class GenreChoices(str, Enum):      fiction = "Fiction"      fantasy = "Fantasy"      mystery = "Mystery"  class UserDetail(BaseModel):      id: int      name: str      age: int      fav\_genre: GenreChoices  user\_db = {      1: {"id":1,"name": "Alice","age":25,"fav\_genre":"Fiction"},      2: {"id":2,"name": "Bob","age":27,"fav\_genre":"Fantasy"}  }  app = FastAPI()  @app.get('/')  async def index():      return {"name": "First Data"}  @app.get('/about')  async def about():      return 'exceptional company'  //below is path param  @app.get("/get\_user/{user\_id}", response\_model=UserDetail)  async def get\_user(user\_id: int):      user = user\_db.get(user\_id)      if not user:          raise HTTPException(status\_code=404,detail="User not found")      return user  //below is query param  @app.get("/get\_usery\_query", response\_model=UserDetail)  async def get\_user\_query(user\_id: int):      user = user\_db.get(user\_id)      if not user:          raise HTTPException(status\_code=404,detail="User not found")      return user  @app.post("/add\_user", response\_model=UserDetail)  def add\_user(user\_data: UserDetail): //you can also use Body()      user\_db[user\_data.id] = user\_data.dict()      print('user is added')      return user\_db[user\_data.id] | from fastapi import FastAPI, HTTPException, APIRouter  from pydantic import BaseModel  from enum import Enum  class GenreChoices(str, Enum):      fiction = "Fiction"      fantasy = "Fantasy"      mystery = "Mystery"  class UserDetail(BaseModel):      id: int      name: str      age: int      fav\_genre: GenreChoices  user\_db = {      1: {"id":1,"name": "Alice","age":25,"fav\_genre":"Fiction"},      2: {"id":2,"name": "Bob","age":27,"fav\_genre":"Fantasy"}  }  router = APIRouter()  app = FastAPI()  @router.get('/')  async def index():      return {"name": "First Data"}  @router.get('/about')  async def about():      return 'exceptional company'  @router.get("/get\_user/{user\_id}", response\_model=UserDetail)  async def get\_user(user\_id: int):      user = user\_db.get(user\_id)      if not user:          raise HTTPException(status\_code=404,detail="User not found")      return user  @router.get("/get\_usery\_query", response\_model=UserDetail)  async def get\_user\_query(user\_id: int):      user = user\_db.get(user\_id)      if not user:          raise HTTPException(status\_code=404,detail="User not found")      return user  @router.post("/add\_user", response\_model=UserDetail)  def add\_user(user\_data: UserDetail):      user\_db[user\_data.id] = user\_data.dict()      print('user is added')      return user\_db[user\_data.id]  app.include\_router(router,prefix="/myroute",tags=["User Details"]) |

For testing APIs you can use Swagger or RestClient extension on VS code and write as below and then click on send:

POST urladdress  
Content-Type application/json  
{actualjson

HTTP status codes   
1xx – processing, 2xx – success, 3xx – redirection, 4xx – client error, 5xx – server error  
GET – 200, POST – 201, PUT – 204,   
400-bad request, 401- unauthorized, 404-notfound,   
500 – internalservererror  
you can raise by below statement  
 raise HTTPException(status\_code=404,detail="User not found")  
  
Starlette gets installed when you install fastapi.   
import status from starlette and You can add explicit status code by adding it after the url.  
 @app.get(“/books”,status\_code = status.HTTP\_200\_OK)

Using Asynchronous flow   
Server – Before, process, after, return  
1. App has async method, blocking process like time.sleep()  
 client is using asyncio and uses await on API ORR uses requests library  
 Result: before,completesprocess,after. In this case, if some other client makes a call, will have to wait till request 1 is completed. Meanwhile might get timeout error on client.  
2. App has async method, non-blocking process without await like asyncio.sleep()  
 client is using asyncio and uses await on API ORR uses requests library  
 Result: Returns response immediately to requests and executes sleep behind the scenes  
3. App has async method, non-blocking process with await like asyncio.sleep()  
 client is using asyncio and uses await on API ORR uses requests library  
 Result: before,completesprocess,after. Can take multiple requests and process parallely.  
 This is the advantage of having async method in API with async processes. It can handle multiple requests.  
  
Though you see similar behavior when you use client of async or requests, the difference lies in the way client itself functions. If async client calls async API, it does not block other operations on client side. It can perform other tasks. But if requests being synchronous and blocking calls async API, it blocks itself till API responds and hence does not perform other tasks.   
  
Hence its important to keep both server and client non blocking if you want to use async behavior. Hence use asyncclient while dealing with async APIs.

|  |  |  |
| --- | --- | --- |
| FastAPI  from fastapi import FastAPI  import time  app = FastAPI()  import asyncio  @app.get("/")  async def read\_root():      print('before sleeping')      await asyncio.sleep(10)      print('after sleeping')      return {"message": "Hello,  World"} | Using Async client  import httpx  import asyncio  async def call\_api():      response = 'initial\_response'      print('before response')      async with httpx.AsyncClient() as client:          response = await client.get('http://127.0.0.1:6970/',timeout=15.0)      print('after response ',response)      print('api executed')  asyncio.run(call\_api()) | Using requests  import requests  response = 'initial\_response'  print('before response')  response = requests.get('http://127.0.0.1:6970/',timeout=15)  print('after response ',response)  print('api executed') |

You can convert any process to asynchronous using asyncio. You need to create a task using asyncio.create\_task and pass the method to be executed. In this method, you need to get the event loop and run the actual operation in the event loop.

|  |
| --- |
| Import asyncio  async def method():  loop = asyncio.get\_event\_loop()  await loop.run\_in\_executor(None, s3.upload\_file)  async def main():  asyncio.create\_task(method())  asyncio.run(main()) |

CICD Pipeline for a Python/FastAPI Project  
In this project, we are going to use Gitlab CI, Makefile, Dockerfile to (all 3 in root dir) achieve CICD.  
  
Below are the different steps involved  
1. Define a Makefile  
 short command followed by the full command  
 If you want to execute in local, its make shortcommand  
 Commands include lint, test, run, build, push, clean etc  
2. Define Gitlab CI file (.gitlab-ci.yml)  
 This will have different stages. In each stage, it will perform some steps like executing corresponding make command.  
3. Dockerfile   
 You will have to provide steps to install necessary software in Docker  
  
  
Different tools involved  
1. Lint: flake8 and black -> will display linting issues. You can configure to proceed or not in G-ci  
2. Lint-fix: black -> will display corrected issues  
3. Unit Test: pytest and pytest-cov -> shows tests executed and result. Takes files with test\_\*.py or \*\_test.py as input. Also provides coverage report. When executing in Gitlab, you should o/p xml. In local, better to output in html format.   
4. API Test using Postman collection: newman (npm package). Input will be postman\_collection.json which contains the API calls and response mapping tests. You get test report  
5. Run: uvicorn  
  
You can push the dockerimage to AWS ECR  
In the build stage, you can build the docker image.  
In the push stage, you can push the image to ECR and other devops process can trigger another pipeline which creates Kubernetes deployment using this image in ECR. This has advantages of maintainability and better control over scaling and deployment across multiple envs.   
  
For this to happen, you need to install AWS cli in the before\_script and also perform aws ecr get-login bla bla to login to ECR. In the Push stage, you can push the image to ECR.   
  
Using docker to run the app locally

|  |
| --- |
| FROM python:3.10-slim WORKDIR /app RUN pip install pip install -r requirements.txt COPY .. EXPOSE 8000 CMD [“uvicorn”, “app.main:app”,”—host”,”0.0.0.0”,”—port”,”8000” |

1. Go to docker website, create login and install desktop app  
2. Login to Desktop app  
3. Build docker image using   
 docker build -t givesomeappname .  
4. Look at the available images. You can see a new image in Docker app as well  
 docker images  
5. Create a tag if needed for the image  
 docker tag imageidseenabove imagename:tagname  
6. Create a container and run  
 docker run -d -p 8000:8000 --name containername image-name  
 Here, port mapping is done. 8000 or uvicorn is mapped to 8000 of docker  
7. After above step, you can go to localhost on browser and you can see the output  
8. View container  
 docker ps (add -a if you want to see all the containers even if its stopped)  
9. To login to Container  
 docker exec -it containeridOrcontainername /bin/bash  
10. View logs of container  
 docker logs container  
11. If you want to Stop the container  
 docker stop containerid  
12. If you want to Start the container  
 docker start containerid  
13. If you want to remove the container  
 docker rm containerid  
  
Making application publicly accessible  
You can push this image to a repository like docker hub, ECR.   
In EC2, Install Docker. You can pull the image from ECR.  
Now create and run the container container  
docker run -d -p 80:8000 containername imagename

Interacting with Databases  
SQLAlchemy, SQLite3, Alembic, Postgres, MySQL  
  
SQLAlchemy is used to create database like sqlite, create session with a DB.  
For this, you need 3 things from SqlAlchemy – create\_engine, sessionmaker, declarative\_base.   
  
In database.py file, You will use DB url in create\_engine instance, use create\_engine instance in sessionmaker instance.   
In models.py, you will define schema of table by inheriting Base class (from database package)   
In main.py, you will use engine instance. Also, get\_db method uses sessionmaker instance.  
  
def get\_db():  
 db = sessionmakerinstance()  
 try:  
 yield db  
 finally:  
 db.close()  
  
You need to inject this method as dependency to every API method that uses database using Annotated function from typing package. For each method, db is the parametername and value is this db\_dependency.   
  
In the API method,   
for get, perform db.query(modelclassname).filter(columnname=incomingvalue)   
for post, it will be db.add(incomingdict) and db.commit()  
for put,   
 get the existing record by db.query(modelclassname).filter(columnname=incomingvalue). Once you fetch, assign incomingdict values to existing modelclassname data by equating individual keys and then db.add(modelclassname) and db.commit()  
for delete, similar to get by id but in the end add .delete() and then db.commit()

SQLite3 is a software You can install on your system by downloading, unzipping and pasting the contents in c drive (C:/here). You need to add env variable for path.   
You will be able to use sqlite3 command in terminal and can access the db by running regular sql queries.   
  
Alembic is a library used because everytime you launch the app containing sqlite db, db gets erased. Also, if you want to alter the table schema, say you want to add another column, you cant do to existing table and if you create table again, data will be lost. Alembic helps here. You need to pip install this.   
Run alembic init alembicenvinstancename -> alembicenvinstancename folder will be created with env.py  
alembic revision -m “added a new column” -> once you do this a new file gets created inside versions folder. Also, a revision id will be created. Alter env.py and revision file according to your changes.   
Run alembic upgrade revisionid -> this will update the existing table without losing data.  
  
If you want to use Postgres DB, above approach used for sqlite can be followed. Just that Postgres is a Prod Database which can handle huge data and also is scalable. You can run it ona different server and port. Sqlite is running on the same server as app within app instance. Preferred only for non-critical and less data load. To use above for Postgres, you will also need to pip install psycopg2-binary.   
  
mySQL creating table command is slightly different to PostresDB. Additionally, you need to install pymysql library.   
  
Authentication  
  
Hash Password: When user signs up, you can collect username and password from user in the backend API. PIP Install Passlib and bcrypt. Encrypt user’s password using Passlib and bcrypt to generate hash password. This can be stored in Table. Algorithm used in bcrypt hashing is unknown to everyone. When user logs in, user’s password and the hash password fetched from table for the user is used to compare in bcrypt’s method to check if password matches.  
  
JWT token is Json web token. When client requests for a service, it needs to provide username and password. Providing these details every time is a risky thing. Hence client gets a token using username and password. Client keeps this in the session and uses this token for all further requests.   
JWT includes 3 parts - Header, Payload and Signature.   
Header consists of {‘alg’:’algorithmusedlikeHS256’, ‘typ’:’jwt’}.  
Payload is the actual data encoded using b64. Ex: {‘username’:’pk’}  
Signature is created by combining encoded header and payload encoded with secret key available on server.   
This JWT is sent to client when client requests for Token. When client makes further request using this token, server decodes the jwt token, uses secret key to decode the signature and fetches the payload. It does these steps to make sure the token is not altered. Also, It may use this payload data like username to go server’s db data and understand what services an be allowed to the user and then proceeds to actually serve the request.   
The secret key used in server can be anything. You can also generate using openssl rand command.   
For encoding and decoding jwt, jose library is needed.

Miscellaneous

* When you create a collection in Postman, for each request, under tests section you can write the code for test. When you run the API by pressing Get/P/P/D button, the test also gets executed and you can see the test result in the console. You can create a new environment and add env variables. In the collection you can select the environment.   
  When you export the collection, you get collection json which can be used for newman in CICD. Same json can be imported in postman as well.