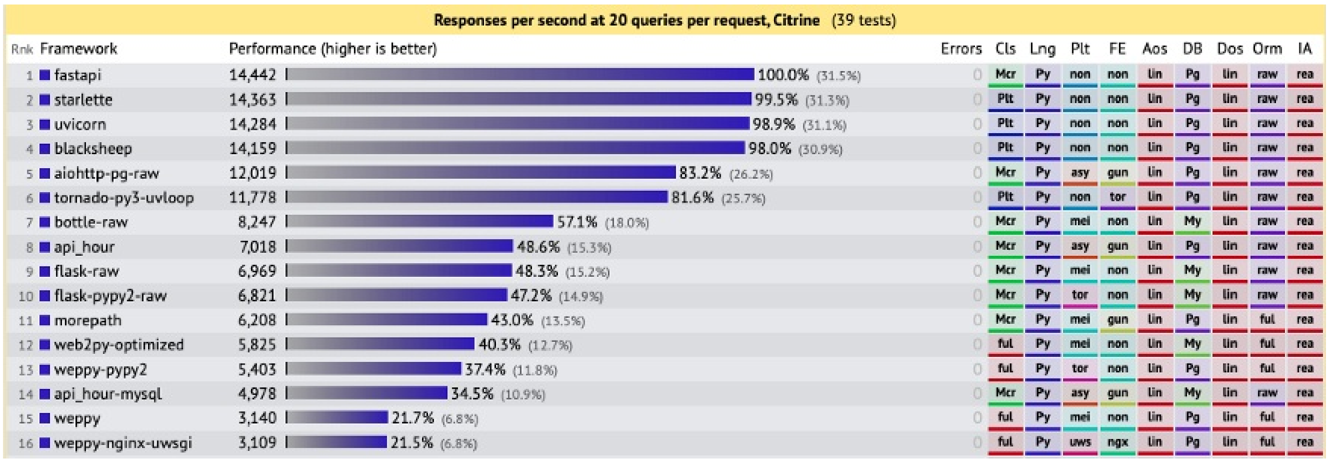
Why FastAPI?

Fast = Better Performance



Responses per second at 20 queries per request, Citrine (39 tests)

Rnk	Framework	Performance (higher is better)			Errors	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA
1	fastapi	14,442	100.0%	(31.5%)	0	Mcr	Py	non	non	lin	Pg	lin	raw	rea
2	starlette	14,363	99.5%	(31.3%)	0	Plt	Py	non	non	lin	Pg	lin	raw	rea
3	uvicorn	14,284	98.9%	(31.1%)	0	Plt	Py	non	non	lin	Pg	lin	raw	rea
4	blacksheep	14,159	98.0%	(30.9%)	0	Plt	Py	non	non	lin	Pg	lin	raw	rea
5	aiohttp-pg-raw	12,019	83.2%	(26.2%)	0	Mcr	Py	asy	gun	lin	Pg	lin	raw	rea
6	tornado-py3-uvloop	11,778	81.6%	(25.7%)	0	Plt	Py	non	tor	lin	Pg	lin	raw	rea
7	bottle-raw	8,247	57.1%	(18.0%)	0	Mcr	Py	mei	non	lin	My	lin	raw	rea
8	api_hour	7,018	48.6%	(15.3%)	0	Mcr	Py	asy	gun	lin	Pg	lin	raw	rea
9	flask-raw	6,969	48.3%	(15.2%)	0	Mcr	Py	mei	non	lin	My	lin	raw	rea
10	flask-pypy2-raw	6,821	47.2%	(14.9%)	0	Mcr	Py	tor	non	lin	My	lin	raw	rea
11	morepath	6,208	43.0%	(13.5%)	0	Mcr	Py	mei	gun	lin	Pg	lin	ful	rea
12	web2py-optimized	5,825	40.3%	(12.7%)	0	ful	Py	mei	non	lin	My	lin	ful	rea
13	weppy-pypy2	5,403	37.4%	(11.8%)	0	ful	Py	tor	non	lin	Pg	lin	ful	rea
14	api_hour-mysql	4,978	34.5%	(10.9%)	0	Mcr	Py	asy	gun	lin	My	lin	raw	rea
15	weppy	3,140	21.7%	(6.8%)	0	ful	Py	mei	non	lin	Pg	lin	ful	rea
16	weppy-nginx-uwsgi	3,109	21.5%	(6.8%)	0	ful	Py	uws	ngx	lin	Pg	lin	ful	rea

Simple to Write = Less Errors

FastAPI is simple to write, Below is the hello world example in FastAPI,

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def index():
    return {"Hello": "World"}
```

The simplicity is comparable to Flask, Flask being a microframework is the simplest to write.

```python
from flask import Flask, jsonify

app=Flask(__name__)

@app.route('/', methods=['GET'])
def index():
    return jsonify({"Hello": "World"})
```

Built-in Documentation = Better Collaboration

Documentation Demo

Pydantic Validations = Easy Validations

Validations are supported in FastAPI through Pydantic, Its simple.

```python
class StudentSchema(BaseModel):
    fullname: str = Field(...)
    email: EmailStr = Field(...)
    course_of_study: str = Field(...)
    year: int = Field(..., gt=0, lt=9)
    gpa: float = Field(..., le=4.0)
    gpa: float = Field(..., le=4.0)

    class Config:
        schema_extra = {
            "example": {
                "fullname": "John Doe",
                "email": "jdoe@x.edu.ng",
                "course_of_study": "Water resources engineering",
                "year": 2,
                "gpa": "3.0",
            }
        }
```

With this we can achieve validations on all the given input parameters to Student. Let us go through the quick Demo

One pain point to this is adding custom error messages are hard with this approach.

Asynchronous = Faster IO

With FastAPI the we can use the python keywords for asynchronous requests, response with async, await. This increases the speed of requests significantly, Consider the below example where we are getting top reddit comments. Since the `get_reddit_top` function can run in parallel in ***async*** mode, the request is much more faster than sequential mode.

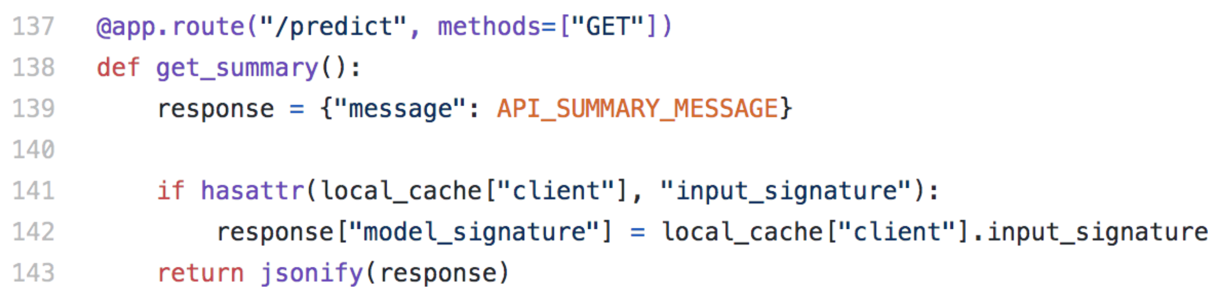
```python
@app.get("/reddit", tags=['Asynchronous Data Fetch'])
async def get_reddit_data_api() -> dict:
    start_time: float = time.time()
    client: ClientSession = aiohttp.ClientSession()
    data: dict = {}

    await asyncio.gather(
        get_reddit_top('python', client, data),
        get_reddit_top('programming', client, data),
        get_reddit_top('compsci', client, data),
    )
    await client.close()

    print("Got reddit data in ---" + str(time.time() - start_time) + "seconds ---")
    return data
```

Adopting FastAPI for backend

Easy to Switch

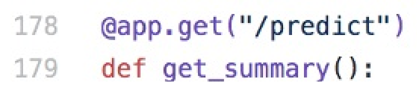
Ease of adopting FastAPI if using Flask already,

Flask code

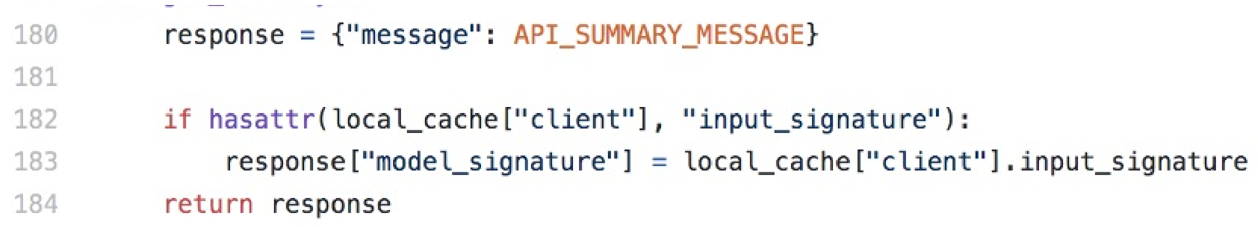


```python
@app.route("/predict", methods=["GET"])
def get_summary():
    response = {"message": API_SUMMARY_MESSAGE}

    if hasattr(local_cache["client"], "input_signature"):
        response["model_signature"] = local_cache["client"].input_signature
    return jsonify(response)
```

FastAPI code



```python
@app.get("/predict")
def get_summary():
```



```
    response = {"message": API_SUMMARY_MESSAGE}

    if hasattr(local_cache["client"], "input_signature"):
        response["model_signature"] = local_cache["client"].input_signature
    return response
```

Backend Features

- **Database Support**
  - **Supports SQL** - SQLAlchemy ORM
  - **Supports NoSQL** - Use standard packages like Flask does
- **Security**
  - Oauth2, basic auth, API, JWT etc.
- **Testing**
  - Easy Testing using TestClient

```
client = TestClient(app)
  def test_read_main():
      response = client.get("/")
      assert response.status_code == 200
      assert response.json() == {"msg": "Hello World"}
```

- **Central Exception Handling**

  Adding user defined exceptions with `@app.exception_handler`

```
@app.exception_handler(SomeException)
  async def http_exception_handler(request: Request, exc: SomeException) -> PlainTextResp
onse:
      return PlainTextResponse(str(exc.detail), status_code=exc.status_code)

  async def request_exception_handler(request: Request, exc: SomeOtherException) -> Plain
TextResponse:
  return PlainTextResponse(str(exc.detail),status_code=exc.status_code)

  app.add_exception_handler(exc_class_or_status_code=SomeOtherException,
  handler=request_exception_handler)
```

- **Dependency Injection**
- **Easy Deployment**

Drawbacks

- No support for user defined validation error messages
- Can become crowded if a standard not followed properly
- Inferior admin page than Django
- Performance gain might not be a requirement for Large Scale Applications
- Although improved flexibility increases development speed, may increase maintanance effort