

FastAPI

1. FastAPI

- > 비교적 최근에 등장한 Web Framework
- > Python 3.6부터 지원
- > Type Annotation 방식
- > 풍부한 자유도
- > 자동 스웨거(Swagger) 지원
 - => 포스트맨을 안써도 볼 수 있다.
- > 큰 커뮤니티가 있지만 아직은 작은 생태계이다.

2. Uvicorn

- > ASGI(Asynchronous Server Gateway Interface)서버
- > **비동기 처리**
- > 멀티스레드 방식보다 더욱 빠른 속도를 보장
- > Swagger와 같은 API 문서 자동화 기능을 제공
- > API 개발 시 생산성을 높일 수 있다.
- > 실행 방법
 - => uvicorn (파일명):app --reload

3. Flask vs FastAPI

	Flask	FastAPI
성능	가벼우면서도 빠르다	높은 부하 상황에서 더 효율적으로 작동한다.
타입 힌트 및 자동 문서화	-> 타입 힌트 지원 X -> API 문서화를 위해 별도의 라이브러리나 주석 기반의 문서화가 필요	-> Python 3.6+의 타입 힌트를 지원 -> 자동으로 API 문서화를 생성하는 기능을 제공 -> Swagger 같은 것들을 이용하여 간단하게 API 문서를 생성할 수 있다.
비동기 지원	-> 기본적으로 지원 X -> Gevent 또는 asyncio와 같은 외부 라이브러리를 사용하여 비동기 처리를 구현할 수는 있다.	-> 비동기 지원이 내장되어 있다. -> <code>async/await</code> 를 사용하여 비동기 코드를 작성가능하다.
코드의 간결성	-> 매우 간단하고 직관적인 API를 제공하여 초보자에게 적합 -> 큰 규모의 애플리케이션에는 복잡성이 발생할 수 있다.	-> 타입 힌트와 자동 문서화를 사용할 수 있다. -> 이로 인해 큰 규모의 애플리케이션을 더 쉽게 유지 및 보수할 수 있다.
생산성	-> 매우 유연하며, 초기 단계부터 빠르게 시작할 수 있도록 도와준다.	-> 코드 자동 완성과 타입 힌트를 사용하여 개발 생산성을 높여준다.

Table1. Flask vs FastAPI

4. 실전편

4-1. module import

```
from fastapi import FastAPI
import torch
import torch.nn as nn
from fastapi import Request
```

Figure 1. module import

4-2. FastAPI 객체 생성

```
app = FastAPI()
```

Figure 2. FastAPI 객체 생성

4-3. FastAPI를 이용한 GET

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

@app.get("/math/sum")
def math_sum(num1: int, num2: int):
    return {"result": num1 + num2}
```

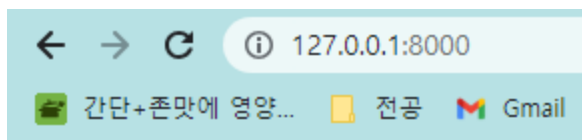
Figure 3. FastAPI를 이용한 GET

```
(venv) PS C:\KDT_안영준\Flask\FastAPI> uvicorn app:app --reload
INFO:     Will watch for changes in these directories: ['C:\KDT_
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C
INFO:     Started reloader process [2744] using WatchFiles
```

Figure 4. uvicorn을 이용해서 실행

-> FastAPI 객체를 생성한 다음 router를 생성했다면 Flask가 아닌 uvicorn을 이용해서 실행해야 한다.

1> 첫번째 router의 get 실행("/")



```
{"index": "Hello FastAPI"}
```

Figure 5. 첫번째 get 실행 결과

2> 두번째 router의 get 실행("/math/sum")

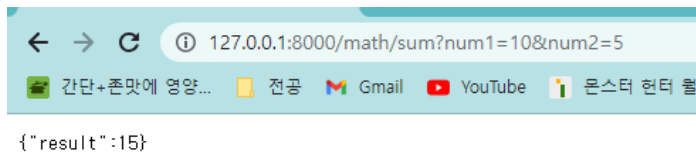


Figure 6. 두번째 router의 get 실행 결과

-> url 뒤에 ?과 함께 각 매개변수에 넣을 숫자들을 &를 이용해 각각 대입해주면 된다.

-> 최종적인 url의 형태로는

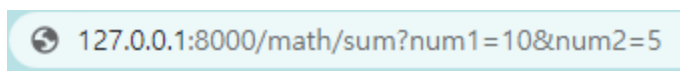


Figure 7. url 형태

-> url + router + ? + 매개변수=넣을 값 + & + 매개변수2=넣을 값2

4-4. FastAPI를 이용한 POST

```
class Item(BaseModel):
    data: list

app = FastAPI()

model = nn.Linear(3, 1)
model.load_state_dict(torch.load('model.pth'))
model.eval()

@app.post('/predict')
def predict(item: Item):
    data = item.data
    x_test = torch.FloatTensor(data)
    y_pred = model(x_test)
    return {"pred": y_pred.tolist()}
```

Figure1. FastAPI를 이용한 POST

-> BaseModel

=> BaseModel은 Pydantic 라이브러리에서 제공하는 클래스로, 데이터의 유효성 검사와 직렬화를 지원하는 기반 모델

=> BaseModel 클래스를 상속하고 클래스의 속성들을 타입 힌트로 정의하면, Pydantic은 이를 기반으로 데이터 모델을 자동으로 생성 아래와 같은 주요 기능들을 제공한다.

※※ Pydantic이 제공하는 기능들

1> 데이터 유효성 검사

-> BaseModel을 상속한 데이터 모델은 선언된 타입 힌트를 기준으로 입력 데이터의 유효성을 검사한다.

-> 예를 들어, 숫자 타입으로 선언된 속성에 문자열이 들어오면 유효성 검사를 통해 오류가 발생한다.

2> 데이터 직렬화

-> BaseModel을 통해 정의된 데이터 모델은 JSON 등 다양한 형식으로 직렬화할 수 있다.

-> Python 객체를 JSON 형식으로 변환하거나, JSON 데이터를 Python 객체로 변환하는 작업을 자동으로 수행할 수 있다.

3> 디폴트 값 설정

-> BaseModel의 속성에는 디폴트 값도 지정할 수 있으며, 입력 데이터에 해당 속성 값이 없는 경우 자동으로 디폴트 값이 할당된다.

4> 중첩 모델

-> BaseModel을 상속한 모델 내에 다른 BaseModel을 속성으로 사용할 수 있다.

=> 복잡한 데이터 구조를 간단하게 표현할 수 있다.

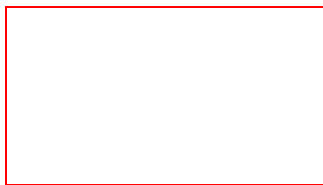
5> 허용되지 않는 필드 제한

-> 입력 데이터에 정의되지 않은 필드가 있을 경우, BaseModel은 해당 필드를 걸러내거나 오류를 발생시킨다.

4-5. 코드 분석

```
class Item(BaseModel):  
    data: list  
  
app = FastAPI()  
  
model = nn.Linear(3, 1)  
model.load_state_dict(torch.load('model.pth'))  
model.eval()  
  
@app.post('/predict')  
def predict(item: Item):  
    data = item.data  
    x_test = torch.FloatTensor(data)  
    y_pred = model(x_test)  
    return {"pred": y_pred.tolist()}
```

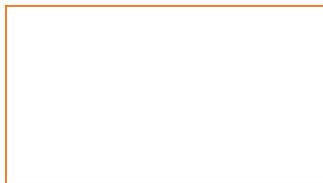
Figure 2. POST 코드



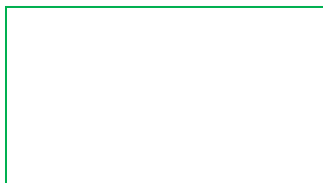
-> Item이라는 Pydantic의 BaseModel을 정의.

-> 모델은 하나의 속성 data를 가진다.

=> data는 리스트 형태의 데이터를 나타내고 전송된 데이터의 유효성을 검사하는데 사용된다.

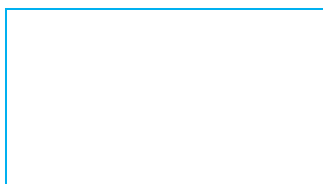


-> FastAPI 객체를 생성하여 애플리케이션을 설정한다.



-> 모델을 불러오기 전에 같은 형태의 모델을 미리 정의를 한다.

-> 미리 학습된 가중치(pth)를 로드하고 model.eval()를 통해 모델을 평가 모드로 설정한다.



-> item: Item은 BaseModel의 Item을 item으로 받는다는 의미이다.

-> predict함수로 POST 요청으로 받은 데이터를 처리하여 예측결과를 반환한다.

-> item.data를 통해 POST 요청으로 전송된 데이터를 가져온다.

5. Swagger

-> Swagger는 RESTful API를 설계, 문서화 및 시각화하는 도구로, API 개발자와 사용자들이 API의 기능과 사용법을 쉽게 이해할 수 있도록 도와준다.

-> OpenAPI Specification(OAS)를 사용하여 API에 대한 표준화된 문서를 생성

=> OAS는 API의 메타데이터를 정의하는 JSON 또는 YAML 형식의 스펙이며, 이를 사용하여 API의 자동 문서화와 클라이언트 코드 생성 등을 지원

-> Swagger UI는 Swagger 문서를 시각적으로 보여주는 웹 인터페이스로, API의 엔드포인트, 매개 변수, 응답 형식 등을 자동으로 생성하여 사용자가 API를 탐색하고 테스트할 수 있도록 제공

-> Swagger UI를 통해 API를 사용하는 데 필요한 정보를 쉽게 얻을 수 있으며, API 개발자가 별도의 문서를 작성하지 않더라도 자동으로 API 문서를 생성하여 제공한다.

5-1. Swagger의 주요 기능

-> API 자동 문서화: Swagger는 API 코드를 분석하여 자동으로 API에 대한 상세한 문서를 생성합니다.

-> 인터랙티브 API 테스트: Swagger UI를 통해 API 엔드포인트를 직접 테스트할 수 있습니다.

-> API 클라이언트 코드 생성: Swagger 스펙을 기반으로 클라이언트 코드를 자동으로 생성하여 API를 사용하는 데 도움을 줍니다.

-> 다양한 플랫폼 지원: Swagger를 사용하여 다양한 플랫폼과 언어에서 API를 문서화하고 사용할 수 있습니다.

5-2. Swagger를 적용해주는 경로인 docs

```
(venv) PS C:\KDT_안영준\Flask\FastAPI> uvicorn api:app --reload
INFO:      Will watch for changes in these directories: ['C:\KDT_안영준\Flask\FastAPI']
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Figure 1. url 예시

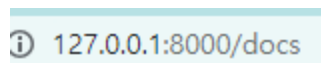


Figure 2. docs를 추가한 url

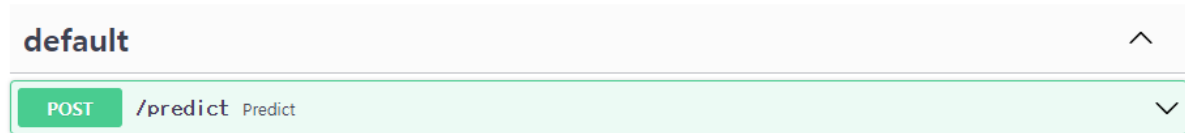


Figure 3. url + docs의 내부

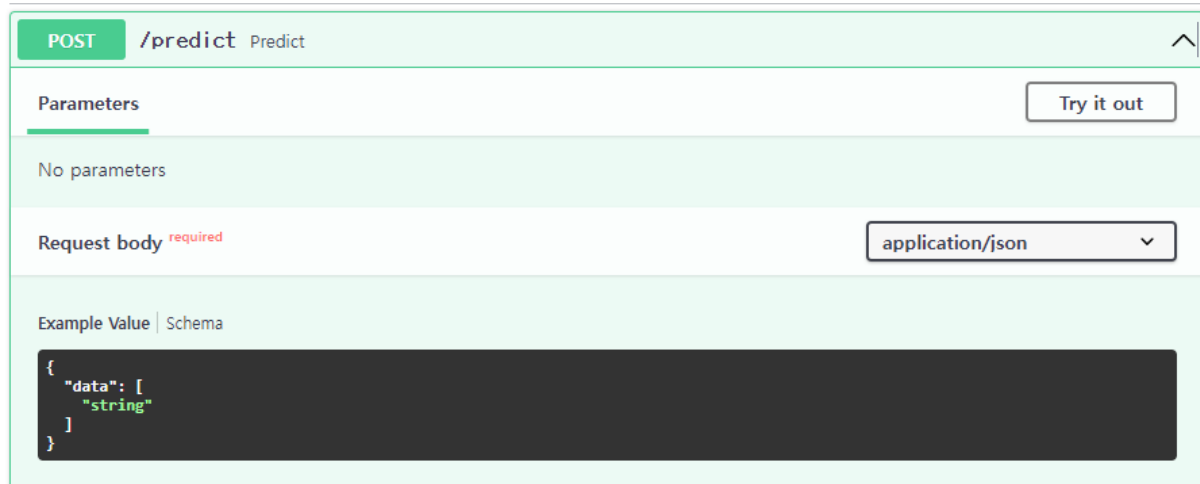


Figure 4. 펼쳤을 때의 내부

-> Try it out을 누르면 창이 열리게 되면서 data를 입력할 수 있다.

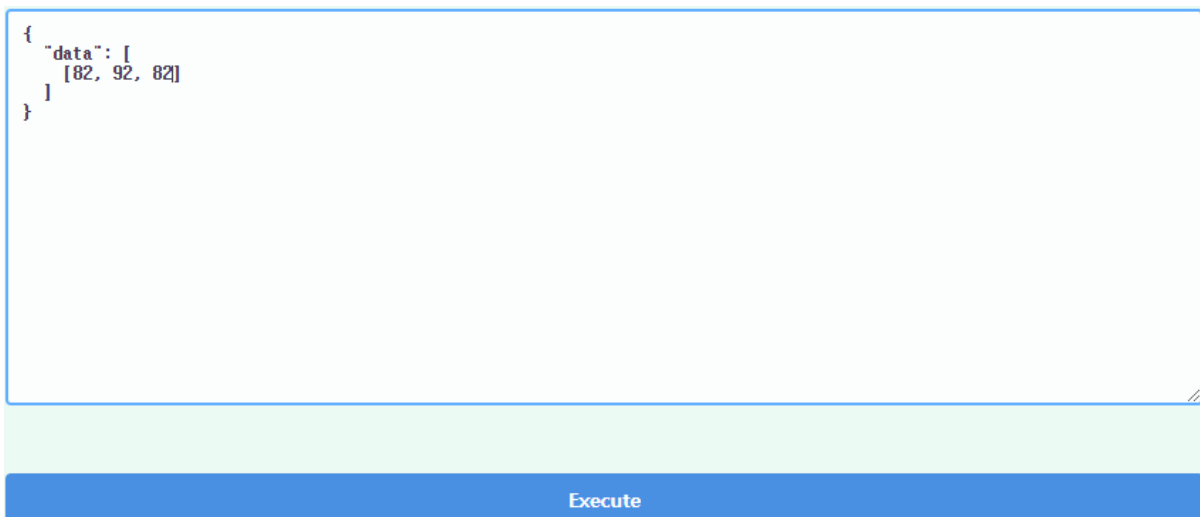


Figure 5. 확인해 볼 data를 형식에 맞춰 입력한 후 Execute를 누른다.

Code	Details
200	<div><div>Response body</div><div><pre>{ "pred": [[171.2141571044922]] }</pre></div><div> Download</div></div>

Figure 6. execute의 결과

-> 형식에 맞춰서 넣게 되면 예측값이 나오게 된다.