FastAPI Seminar

Week 1: 입문자를 위한 FastAPI

By: 이민규

출석 체크

https://areyouhere.today/

Table of Contents

1. HTTP Request 를 받아보자!

1. HTTP Request 를 받아보자!

천 리 길도 한 걸음부터

```python
from fastapi import FastAPI, status

app = FastAPI()

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

- `FastAPI()`로 ASGI application 만들기
- `@app.get()`으로 GET 엔드포인트 추가하기
- 엔드포인트 구현하기

1. HTTP Request 를 받아보자!

HTTP Request 는 어떻게 생겼나

```
POST /api/v1/items/ HTTP/1.1
Host: sns.wafflestudio.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) ...
Accept: application/json
Accept-Encoding: gzip, deflate
Content-Length: 29
Content-Type: application/json; charset=utf-8

{"message": "Hello World 😄"}
```

- HTTP Request Message 는 US-ASCII 로 인코딩되어 있다.
- Request Line, Headers, Body 로 구성되어 있다.

1. HTTP Request 를 받아보자!

HTTP Method

GET /brands/10/items HTTP/1.1

- METHOD = Request Line 의 첫 번째 Token
- METHOD 는 클라이언트의 요청 목적
  - 리소스 검색, 추가, 변경, etc.
  - TMI) HTTP/1.1 의 Semantic 은 RFC 7231 에서 확인할 수 있다.

1. HTTP Request 를 받아보자!

HTTP Method

GET /brands/10/items HTTP/1.1

- HTTP Method 에는 어떤 토큰도 들어갈 수 있다.
- 하지만, 대부분의 경우에 다음 메서드 중 하나를 사용하게 된다.
  - GET: 리소스 요청
  - POST: 리소스 생성
  - PUT: 리소스 생성또는 덮어씌움
  - PATCH: 리소스 부분 수정
  - DELETE: 리소스 삭제
  - OPTIONS

1. HTTP Request 를 받아보자!

HTTP Method

```
@app.get("/brands/10/items/{item_id}", status_code=status.HTTP_200_OK)
@app.put("/brands/10/items/{item_id}", status_code=status.HTTP_200_OK)
@app.delete("/brands/10/items/{item_id}", status_code=status.HTTP_204_NO_CONTENT)
@app.post("/brands/10/items/", status_code=status.HTTP_201_CREATED)
@app.patch("/brands/10/items/{item_id}", status_code=status.HTTP_200_OK)
```

- 자주 사용하는 메서드들은 별도의 데코레이터 팩토리로 구현되어 있다. (Path Operation Decorator)

```
@app.api_route("/brands/10/items/{items_id}", methods=["GET", "PUT", "HOIZZA"])
```

- 사용자 정의 HTTP Method 를 사용하거나, 여러 Method 를 허용하려고 할 때에는 범용 데코레이터 팩토리인 `app.api_route`를 사용한다.

1. HTTP Request 를 받아보자!

Path

GET /brands/10/items HTTP/1.1

- Path = 계층 구조 리소스 식별자
- Slash 로 구분되는 segment 들로 이루어져있음
- 각 segment 마다 ";"로 구분되는 "key=value" 형태의 파라미터가 존재할 수 있는데, 잘 쓰이지 않음
- 리소스를 표현하는 데이터 중 계층 구조로 표현하기 쉬운 경우, path로 전달하는 것이 바람직함

1. HTTP Request 를 받아보자!

Path

```python
@app.get("/brands/{brand_id}/items", status_code=status.HTTP_200_OK)
def list_brand_items(brand_id: int):
  ...
```

- 정적인 문자열뿐만 아니라 동적인 문자열을 받는 것도 가능
- 이때, 타입 힌트를 통해 원시 타입으로의 단순한 형변환은 자동으로 지원

```python
@app.get("/brands/{brand_id}/items", status_code=status.HTTP_200_OK)
def list_brand_items(brand_id: int = Path(...)):
  ...
```

- `Path` 라는 함수를 기본값으로 전달해서 추가적인 정보를 전달할 수 있음
- 이를 통해 간단한 validation이나, API 문서를 보강하는 등의 작업을 할 수 있음

1. HTTP Request 를 받아보자!

Query String

GET /items?min_price=1000&max_price=5000

- Query String = 리소스 필터링, 정렬, 페이징 등 계층 구조가 아닌 리소스 식별자
- Path 끝에서 "?" 뒤에서부터, "#" 앞 또는 URL의 끝 부분까지
- "key=value" 형태의 값을 "&"로 구분하여 전달하는 것이 사실상 표준
- 몇몇 특수문자는 사용을 지양하거나 인코딩을 필요로함
  - "#" : Query String의 끝을 나타내는 문자
  - "&", "=" : key, value 구분자
  - "/" : Path 와 헷갈릴 수 있으므로
  - key 에서는 사용을 지양
  - value 로 전달이 불가피하다면 인코딩을 적용
    (Percent Encoding, Base62, URL-Safe Base64, etc.)

1. HTTP Request 를 받아보자!

Query String

```python
@app.get("/items", status_code=status.HTTP_200_OK)
def list_items(min_price: int = Query(0), max_price: int = Query(99999)):
  ...
```

- 엔드포인트의 파라미터 중 Path 가 아니면 자동으로 Query String 으로 추론됨
- `Path` 와 마찬가지로 `Query` 함수를 이용해서 추가적인 기능을 제공할 수 있음
- 파라미터를 옵셔널하게 만들려면, 기본값을 전달하거나 타입을 `Optional` 로 지정
- 파라미터를 필수로 만들려면, 기본값을 전달하지 않거나, `Query(...)` 으로 지정

1. HTTP Request 를 받아보자!

Header

```
POST /api/v1/items/ HTTP/1.1
Host: sns.wafflestudio.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) ...
Accept: application/json
Accept-Encoding: gzip, deflate
Content-Length: 29
Content-Type: application/json; charset=utf-8

{"message": "Hello World 😄"}
```

1. HTTP Request 를 받아보자!

Header

```
POST /api/v1/items/ HTTP/1.1
Host: sns.wafflestudio.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) ...
Accept: application/json
Accept-Encoding: gzip, deflate
Content-Length: 29
Content-Type: application/json; charset=utf-8

{"message": "Hello World 😄"}
```

- Header = 요청의 메타데이터
- Request Line 다음부터, Body 전까지
- "key: value" 형태의 줄들로 이루어져 있음
- Header 와 Body 사이에는 빈 줄(CRLF)이 존재해야 함

1. HTTP Request 를 받아보자!

Header

```python
@app.post("/items", status_code=status.HTTP_201_CREATED)
def create_item(item: Item, user_agent: str = Header(None)):
  ...
```

- Path, Query 와 달리 반드시 Header 함수를 이용해서 명시적으로 지정해야 함
- Query 와 마찬가지로, 기본값이나 옵셔널 여부를 지정할 수 있음

1. HTTP Request 를 받아보자!

특수한 Request Header 들

- 일부 헤더들은 브라우저에 의해 자동으로 전달되며, 특별한 의미를 가지고 있음
  - `Host`: 요청을 받는 서버의 호스트 정보
  - `Referer`: 이전 페이지의 URL 정보
  - `User-Agent`: 클라이언트의 소프트웨어 정보
  - `Content-Type`: Body 의 타입을 지정
  - `Content-Length`: Body 의 길이를 지정
  - `Accept`: 클라이언트가 받아들일 수 있는 타입
  - `Accept-Encoding`: 클라이언트가 받아들일 수 있는 압축 방식
  - `Authorization`: 클라이언트의 인증 정보
  - `Cookie`: 클라이언트의 세션 정보
  - ...

1. HTTP Request 를 받아보자!

Body

```
POST /api/v1/items/ HTTP/1.1
Host: sns.wafflestudio.com
...
Content-Type: application/json; charset=utf-8
Content-Length: 29

{"message": "Hello World 😄"}
```

- Body = Header 의 마지막 빈 줄 다음부터 끝까지
- Body 가 존재하는 경우에는 컨텐츠의 메타데이터를 Header 에서 전달해야 함
    - Content-Length, Content-Type, Transfer-Encoding 등

1. HTTP Request 를 받아보자!

Body (feat. Pydantic)

```python
class Item(BaseModel):
  name: str
  price: float
  brand_id: int

@app.post("/items", status_code=status.HTTP_201_CREATED)
def create_item(item: Item):
  ...
```

- Body 는 Pydantic 을 통해 파싱됨
- Pydantic 은 타입 힌트에 기반한 JSON 직렬화/역직렬화 라이브러리
- Nested Model, Optional Field, Default Value, Validation 등을 지원

```http
POST /items HTTP/1.1
Content-Type: application/json

{"name": "iPhone", "price": 1000, "brand_id": 1}
```

1. HTTP Request 를 받아보자!

Body (feat. Pydantic)

```python
class Item(BaseModel):
  name: str = Field(alias="item_name", max_length=100)
  price: float
  brand_id: int

  model_config = ConfigDict(
    str_to_lower = True,
    str_max_length = 100,
    extra = "ignore",
    faux_immutable = True,
  )
```

- `Field` 를 이용해서 필드별 옵션을 지정할 수 있음
- model_config 를 통해 모델의 옵션을 지정할 수 있음
  - v1 에서는 `Config` 클래스를 이용해서 모델 전체의 옵션을 지정할 수 있었음
  - fastapi 버전에 따라 Pydantic 버전도 달라지므로, 사용하는 버전에 유의할 것

1. HTTP Request 를 받아보자!

야생의 Request

```python
@app.post("/items", status_code=status.HTTP_201_CREATED)
async def create_item(request: Request):
  print(request.method)
  print(request.url)
  print(request.headers)
  print(request.query_params)
  print(await request.body())
  ...
```

- 흔하진 않지만, 다양한 이유로 `Request` 객체가 직접 필요할 수 있음
- `Request` 객체에는 거의 날 것의 Request Message 를 포함해서 다양한 메타데이터가 포함되어 있음

```
POST
http://localhost:8000/items?some_key=some_value
Headers({'host': 'localhost:8000', 'user-agent': 'curl/8.7.1', 'accept': '*/*', 'some-header':
some_key=some_value
b'{"some_json": "some_value"}'
```

2. HTTP Response 를 보내보자!

2. HTTP Response 를 보내보자!

HTTP Response 는 어떻게 생겼나

```
1 HTTP/1.1 200 OK
2 Content-Type: application/json
3 Content-Length: 26
4
5 {"message": "Hello World"}
```

- 마찬가지로 US-ASCII 로 인코딩되어 있다.
- Status Line, Headers, Body 로 구성되어 있다.
- 첫 번째 줄을 제외하면 Request Message 와 크게 다르지 않다.

2. HTTP Response 를 보내보자!

Response Body

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 26

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

- Request Body 와 크게 다르지 않음

2. HTTP Response 를 보내보자!

Response Body

```python
class Item(BaseModel):
  name: str
  price: float
  brand_id: int

@app.get("/item")
def get_item() -> Item:
  return Item(name="iPhone", price=1000, brand_id=1)
```

- Request Body 가 알아서 역직렬화가 되었듯, Response Body 도 자동으로 직렬화가 됨

```python
class Items(BaseModel):
  items: List[Item]

@app.get("/items")
def get_items() -> Items:
  return Items(items=[Item(name="iPhone", price=1000, brand_id=1)])
```

- Model 을 중첩해서 깊은 구조의 JSON 을 반환할 수 있음 (Nested Model)

2. HTTP Response 를 보내보자!

Headers

```
1 HTTP/1.1 200 OK
2 Content-Type: application/json
3 Content-Length: 26
4
5 {"message": "Hello World"}
```

- Request Header 와 크게 다르지 않음

2. HTTP Response 를 보내보자!

Headers

```python
from fastapi.responses import JSONResponse

@app.get("/item")
def get_item():
  return JSONResponse(content={"message": "Hello World"}, headers={"X-Custom-Header": "Custom V
```

- `fastapi.responses` 모듈을 이용해서 Response Body 와 Header 를 동시에 지정할 수 있음
- `headers` 는 `dict` 형태로 전달

2. HTTP Response 를 보내보자!

Status Code

HTTP/1.1 200 OK

- Status Code = Status Line 의 두 번째 Token
- Status Code 뒤에는 Status Message 가 붙지만, 실제로는 큰 의미 없음
- Status Code 는 서버의 응답 상태를 나타내며, 첫 번째 숫자에 따라 다음과 같이 구분됨
  - 1xx: Informational
  - 2xx: Successful
  - 3xx: Redirection
  - 4xx: Client Error
  - 5xx: Server Error
  - RFC 7231 또는 MDN 참조

2. HTTP Response 를 보내보자!

Status Code

```python
from fastapi import status

@app.get("/items/{item_id}", status_code=status.HTTP_200_OK)
def read_item(item_id: int):
  return get_item_by_id(item_id)
```

- 엔드포인트의 데코레이터에 `status_code` 를 지정할 수 있음
- 정수로 지정할 수도 있지만, `fastapi.status` 를 이용해서 더 명시적으로 지정할 수 있음
- 기본값은 200 OK

2. HTTP Response 를 보내보자!

Status Code

```python
from fastapi import status
from fastapi.responses import JSONResponse

@app.get("/items/{item_id}")
def read_item(item_id: int):
  return JSONResponse(content=..., status_code=status.HTTP_200_OK)
```

- 반환값이 여러 개이고 각각 다른 상태 코드를 지정해야 할 때에는 `fastapi.responses` 모듈을 이용
- 하지만 이 방법은 API 문서에 자동으로 반영되지 않음

2. HTTP Response 를 보내보자!

Redirect

```python
from fastapi.responses import RedirectResponse

@app.get("/redirect")
def redirect():
  return RedirectResponse(url="/items")
```

- `RedirectResponse` 를 이용해서 리다이렉트 응답을 보낼 수 있음

2. HTTP Response 를 보내보자!

Response Object

```python
from fastapi.responses import Response

@app.get("/items/{item_id}")
def read_item(item_id: int):
  response = Response(content="Hello World", status_code=200)
  response.headers["X-Custom-Header"] = "Custom Value"
  return response
```

- Response 객체를 이용해서 직접 Response 를 생성할 수 있음
- 주입받은 Response 객체는 반환을 생략할 수 있음

3. 의존성 주입

3. 의존성 주입

배경 지식 - 의존성 주입이란?

- 의존성 주입(Dependency Injection)은 객체 지향 프로그래밍에서 사용되는 디자인 패턴
- 객체의 생성을 외부에 위임하여, 객체 간의 의존 관계를 느슨하게 만든다.
- 인터페이스에 의존하는 설계로 객체 간의 결합도를 낮추어 유지보수성, 테스트 용이성을 높인다.

3. 의존성 주입

배경 지식 - 의존성 주입이란?

- FastAPI 에서 의존성이란 다음과 같이 정의할 수 있다.
  (⚠ 아래 정의는 공식 문서에 있는 건 아니고 공식 문서를 토대로 한 세미나장 뇌피셜)

```
depandable ::= primitive_dependency | callable_dependency
primitive_dependency ::= Request | Response | Path | Query | ...
callable_dependency ::= Callable[...dependable, Any]
```

- `dependable` 을 통해 의존성을 주입받을 수 있음
- `primitive_dependency` 는 FastAPI 에서 제공하는 기본적인 의존성 (앞서 배운 것들이 대표적)
- `callable_dependency` 는 사용자 정의 의존성을 주입받을 수 있음

3. 의존성 주입

primitive dependency

```python
from fastapi import FastAPI, Request, Path, Query, Header

@app.get("/items/{item_id}")
def read_item(
  request: Request,
  item_id: int = Path(...),
  min_price: int = Query(0),
  user_agent: str = Header(None),
):
  ...
```

- 앞서 배운 `Request`, `Path`, `Query`, `Header` 등 FastAPI 에서 제공하는 기본적인 의존성을 주입
- 이 의존성들은 별도로 정의해줄 필요 없이 바로 사용 가능

3. 의존성 주입

Callable Dependency

```python
from fastapi import Depends

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

@app.get("/items")
def read_items(db: DBSession = Depends(get_db)):
  ...
```

- Depends 를 이용해서 명시적으로 의존성을 주입받을 함수나 객체를 지정할 수 있음
- Depends 안에 인자가 없다면 타입 자체가 dependable 해야함
    - 즉, 생성자의 모든 인자가 dependable 해야함 (혹은 인자가 없거나)

---

3. 의존성 주입

Sub Dependency

```python
from fastapi import Depends

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

def get_current_user(db: DBSession = Depends(get_db)):
  user = db.query(User).filter(User.id == 1).first()
  return user

@app.get("/items")
def read_items(user: User = Depends(get_current_user)):
  ...
```

- 의존성은 여러 번 중첩해서 사용할 수 있음
- 이를 통해 의존성을 계층적으로 관리할 수 있고, 재사용성을 높일 수 있음

3. 의존성 주입

path operation decorator 에서의 의존성 주입

```python
from fastapi import Depends

@app.get("/some_api", dependencies=[Depends(get_current_user)])
def some_api_required_auth():
  ...
```

- 직접 유저 객체를 사용하진 않지만 인증이 필요한 API가 있을 수 있음
- 이렇듯 간접적으로 사용되는 의존성은 경로 데이터에 제공함으로써 복잡성을 줄일 수 있음

3. 의존성 주입

Global Dependency

```
from fastapi import FastAPI, Depends

app = FastAPI(dependencies=[Depends(get_current_user)])
```

- `FastAPI` 객체를 생성할 때, `dependencies` 를 지정해서 모든 엔드포인트에 의존성을 주입할 수 있음
- 혹은, 나중에 배울 `Router` 를 이용해서 특정 라우터에만 의존성을 주입할 수도 있음

3. 의존성 주입

Dependency Generator

```python
from fastapi import Depends

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

- 앞선 예시에서 `get_db` 함수는 제너레이터 함수로 구현되어 있음
- 이를 통해, 의존성을 생성하고 제거하는 과정을 명시적으로 제어할 수 있음
- 이는 FastAPI 내부적으로 컨텍스트 매니저를 사용하여 구현됨

3. 의존성 주입

Dependency Generator

```python
def get_A():
  try:
    yield make_A()
  finally:
    ... # clean up A after return response

def get_B(a: A = Depends(get_A)):
  try:
    yield make_B(a)
  finally:
  ... # clean up B after clean up A

def get_C(b: B = Depends(get_B)):
  try:
    yield make_C(b)
  finally:
    ... # clean up C after clean up B

@app.get("/items")
def read_items(c: C = Depends(get_C)):
  ...
```

4. 에러 핸들링

4. 에러 핸들링

에러란?

- 프로그램은 언제나 오류가 발생할 수 있다.
- 서버에서 발생하는 오류가 발생하는 원인은 크게 두 가지로 나뉜다.
    - 클라이언트의 잘못된 요청 (Client Error)
    - 서버의 잘못된 처리 (Server Error)

4. 에러 핸들링

Client Error

- 클라이언트의 잘못된 요청에 대한 응답은 4xx 응답 코드를 사용한다.
- 잘못된 요청의 예는 다음과 같다.
    - 존재하지 않는 경로 (404 Not Found)
    - 잘못된 요청 형식 (400 Bad Request)
    - 인증이 필요한 요청 (401 Unauthorized)
    - 권한이 없는 요청 (403 Forbidden)

4. 에러 핸들링

Server Error

- 서버의 잘못된 처리에 대한 응답은 5xx 응답 코드를 사용한다.
- 서버 오류는 일시적인 오류일 수도 있고, 영구적인 오류일 수도 있다.
- 일시적인 오류라면 `Retry-After` 헤더를 통해 재시도를 권장할 수 있다.
- 서버 오류의 예는 다음과 같다.
    - 서버 내부 오류 (500 Internal Server Error)
    - 서버가 처리할 수 없는 요청 (501 Not Implemented)
    - 서버가 과부하 상태 (503 Service Unavailable)

4. 에러 핸들링

HTTPException

```python
from fastapi import HTTPException

@app.get("/items/{item_id}")
def read_item(item_id: int):
  if item_id == 0:
    raise HTTPException(status_code=404, detail="Item not found")
  return {"item_id": item_id}
```

- HTTPException 를 이용해서 명시적으로 에러를 발생시킬 수 있다.
- 직접 Response 를 반환하지 않는 것이 좋다.
  - 예외를 던지면 FastAPI 내장 에러 핸들러가 이를 처리해주기 때문

4. 에러 핸들링

Exception Handler

```python
@app.exception_handler(MyException)
async def http_exception_handler(request, exc):
  return JSONResponse(
    status_code=500,
    content={"detail": exc.detail},
  )
```

- `exception_handler` 를 이용해서 특정 예외나 상탯 코드에 대해 내장 에러 핸들러를 오버라이드할 수 있다.
- 해당 데코레이터 팩토리에는 하나의 예외 타입만 지정할 수 있으므로, 여러 개의 예외를 처리하려면
  - 여러 예외 핸들러를 등록하거나,
  - 부모 클래스로 묶어서 처리할 수 있다.

5. 서버를 띄워보자!

5. 서버를 띄워보자!

uvicorn

- FastAPI 는 ASGI 애플리케이션이다.
- **uvicorn**: ASGI 프로토콜을 지원하는 가장 대표적인 애플리케이션 서버

5. 서버를 띄워보자!

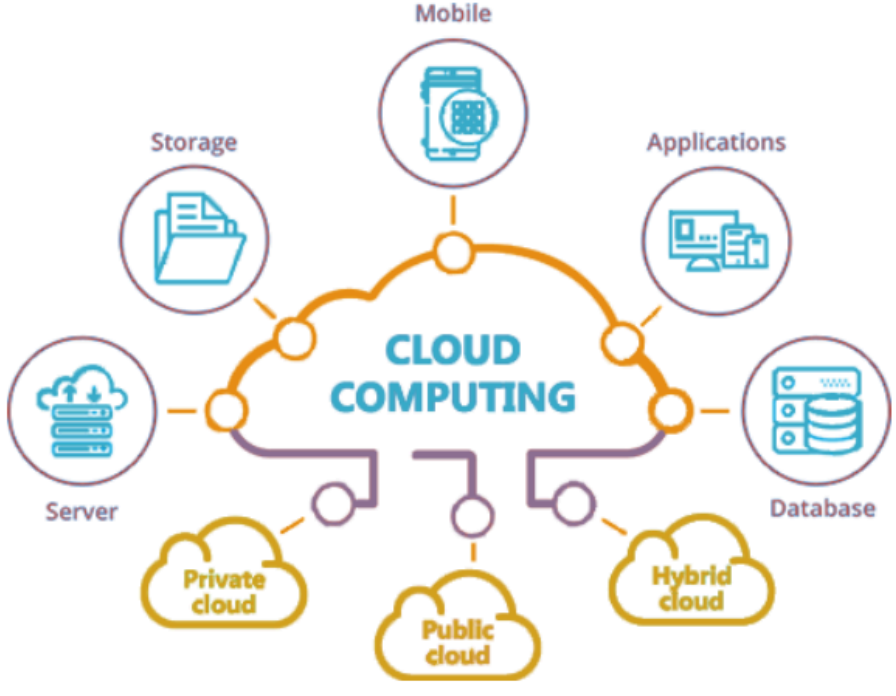
uvicorn

```
# uvicorn <경로>:<ASGI app 변수명> [OPTIONS]
uvicorn src.main:app --reload
```

- 쉘에서 `uvicorn` 명령어를 통해 FastAPI 애플리케이션을 실행할 수 있다.
- 몇 가지 유용한 옵션들
  - `--reload` 옵션을 통해 코드 변경 시 자동으로 서버를 재시작할 수 있다. (Hot Reloading)
  - `--host`, `--port` 옵션을 통해 호스트와 포트를 지정할 수 있다.

5. 서버를 띄워보자!

Cloud Computing



- 만약 내 컴퓨터에 서버를 띄운다면, 외부에서 접속하기 위해 컴퓨터를 24시간 켜두어야 한다.
- 전용 서버를 구축하자니, 초기 비용이 너무 많이 들고 Scaling 이 어렵다.
- 이에 대한 대안이 클라우드 서비스 (Off-Premises)

5. 서버를 띄워보자!

AWS EC2

Amazon EC2

- AWS 는 가장 대표적인 클라우드 서비스 제공 업체
- EC2 는 AWS 의 가상 서버 서비스
- EC2 인스턴스를 생성해서 가상 서버를 띄우고, FastAPI 애플리케이션을 배포할 수 있다.

5. 서버를 띄워보자!(실습)

Q&A