

Event Planner

Modularization

❖ 이벤트 플래너 애플리케이션 구조화

planner/

main.py

database/

__init__.py

connection.py

routes/

__init__.py

events.py

users.py

models/

__init__.py

events.py

users.py

Modularization

❖ 프로젝트 구조

- ✓ planner 디렉토리 생성 및 경로 이동
 - mkdir planner && cd planner
- ✓ planner 디렉토리에 main.py 파일 생성
- ✓ planner 디렉토리에 database, routes, models 디렉토리 생성
- ✓ database, routes, models 디렉토리에 __init__.py 파일 생성
- ✓ routes, models 디렉토리에 users.py 와 events.py 파일 생성

Modularization

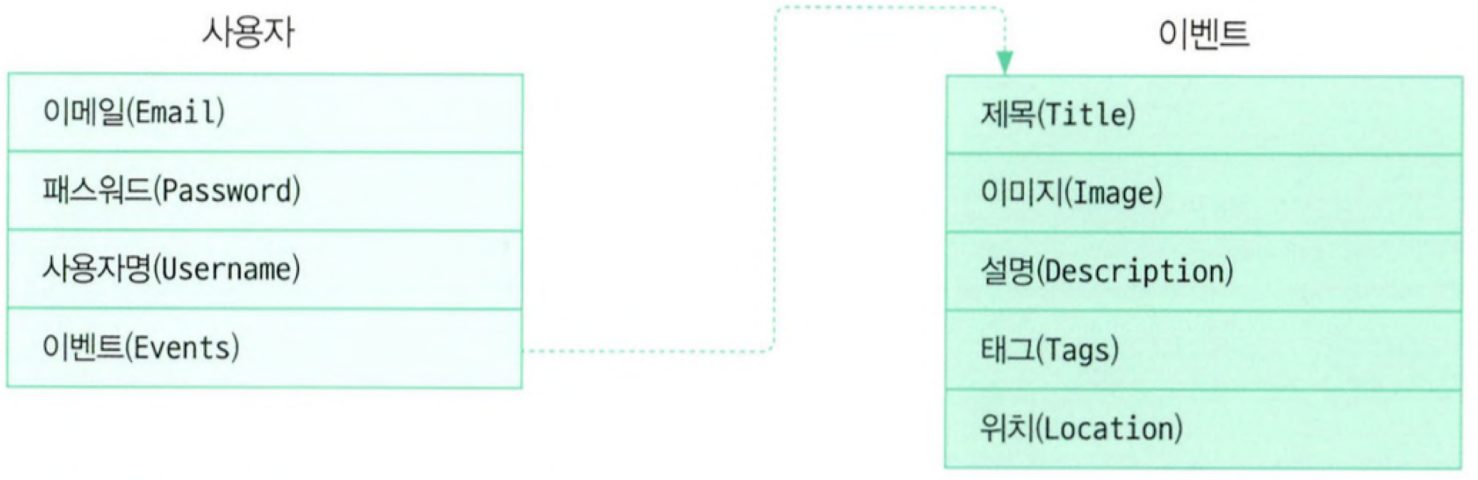
❖ 개발 환경 설정

- ✓ planner 디렉토리에 가상 환경을 활성화
 - `python3 -m venv venv`
 - `source venv/bin/activate`(Windows - `venv\Scripts\activate`)
- ✓ 패키지 설치 및 의존성 파일 생성
 - `pip install fastapi uvicorn "pydantic[email]"`
 - `pip freeze > requirements.txt`

Modularization

❖ 모델 구현

- ✓ 한 명의 사용자는 여러 개의 이벤트를 가질 수 있고 하나의 이벤트는 한 명의 사용자 만이 소유



Modularization

❖ 모델 구현

✓ events.py 파일에 Event 모델을 정의

● 모델 구조

- ◆ id: 자동 생성되는 고유 식별자
- ◆ title: 이벤트 타이틀
- ◆ image: 이벤트 이미지 배너의 링크
- ◆ description: 이벤트 설명
- ◆ tags: 그룹화를 위한 이벤트 태그
- ◆ location: 이벤트 위치

Modularization

❖ 모델 구현

- ✓ events.py 파일에 Event 모델을 정의

- 코드

```
from typing import List
from pydantic import BaseModel
```

```
class Event(BaseModel):
    id: int
    title: str
    image: str
    description: str
    tags: List[str]
    location: str
```

Modularization

❖ 모델 구현

- ✓ events.py 파일에 Event 모델을 정의

- 코드

```
model_config = {  
    "json_schema_extra" : {  
        "example": {  
            "title": "FastAPI Book Launch",  
            "image": "https://linktomyimage.com/image.png",  
            "description": "We will be discussing the contents of the FastAPI book  
in this event.Ensure to come with your own copy to win gifts!",  
            "tags": ["python", "fastapi", "book", "launch"],  
            "location": "Google Meet"  
        }  
    }  
}
```

Modularization

❖ 모델 구현

- ✓ users.py 파일에 User 모델을 정의
 - 모델 구조
 - ◆ email: 사용자이메일
 - ◆ password: 사용자패스워드
 - ◆ events: 해당 사용자가 생성한 이벤트

Modularization

❖ 모델 구현

- ✓ users.py 파일에 User 모델을 정의

```
from pydantic import BaseModel, EmailStr
from typing import Optional, List
from models.events import Event
```

```
class User(BaseModel):
    email: EmailStr
    password: str
    events: Optional[List[Event]]
```

```
model_config = {
    "json_schema_extra": {
        "example": {
            "email": "fastapi@packt.com",
            "password": "strong!!!",
            "events": Optional[List[Event]]
        }
    }
}
```

Modularization

❖ 모델 구현

- ✓ users.py 파일에 User 모델을 정의

```
class UserSignIn(BaseModel):  
    email: EmailStr  
    password: str  
  
model_config = {  
    "json_schema_extra": {  
        "example": {  
            "email": "fastapi@packt.com",  
            "password": "strong!!!"  
        }  
    }  
}
```

Modularization

- ❖ Router 구현
 - ✓ Router 시스템 설계



Modularization

❖ Router 구현

✓ user Router 구현

- routes/users.py 파일에 라우팅 코드 작성

```
from fastapi import APIRouter, HTTPException, status  
from models.users import User, UserSignIn
```

```
user_router = APIRouter(  
    tags=["User"],  
)
```

```
users = {}
```

Modularization

❖ Router 구현

✓ user Router 구현

- routes/users.py 파일에 라우팅 코드 작성

```
#signup 요청이 post 방식으로 전송된 경우 처리
@user_router.post("/signup")
async def sign_user_up(data: User) -> dict:
    if data.email in users:
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail="존재하는 이메일"
        )

    users[data.email] = data

    return {
        "message": "유저 등록 성공"
    }
```

Modularization

❖ Router 구현

✓ user Router 구현

- routes/users.py 파일에 라우팅 코드 작성

#signin 요청을 post 방식으로 전송한 경우

@user_router.post("/signin")

async def sign_user_in(user: UserSignIn) -> dict:

 #email이 존재하지 않는다면

 if user.email not in users:

 raise HTTPException(

 status_code=status.HTTP_404_NOT_FOUND,

 detail="없는 유저"

)

 if users[user.email].password != user.password:

 raise HTTPException(

 status_code=status.HTTP_403_FORBIDDEN,

 detail="비밀번호 틀림"

)

 return {

 "message": "로그인 성공"

 }

Modularization

❖ Router 구현

✓ user Router 구현

- 실행 파일인 main.py 파일에 라우터를 등록하고 실행 코드를 작성

```
from fastapi import FastAPI
from routes.users import user_router
```

```
import uvicorn
```

```
app = FastAPI()
```

```
# 라우터 등록
```

```
# 처리하는 경로 앞에 /user를 추가
```

```
app.include_router(user_router, prefix="/user")
```

```
#8000번 포트로 시작
```

```
if __name__ == '__main__':
```

```
    uvicorn.run("main:app", host="0.0.0.0", port=8080, reload=True)
```

Modularization

- ❖ Router 구현
 - ✓ user Router 구현
 - 애플리케이션 시작
 - ◆ python main.py



Modularization

❖ Router 구현

- ✓ user Router 구현
 - signup 확인

The screenshot displays a REST client interface with the following details:

- Request:**
 - Method: POST
 - URL: http://127.0.0.1:8080/user/signup
 - Body (JSON):

```
1 {  
2   "email": "itstudy@kakao.com",  
3   "password": "1234",  
4   "events": []  
5 }
```
- Response:**
 - Status: 200 OK
 - Time: 6 ms
 - Size: 159 B
 - Body (JSON):

```
1 {  
2   "message": "유저 등록 성공"  
3 }
```

Modularization

❖ Router 구현

✓ user Router 구현

● signup 확인

The screenshot displays a REST client interface with the following details:

- Request:**
 - Method: POST
 - URL: http://127.0.0.1:8080/user/signup
 - Body (JSON):

```
1 {
2   "email": "itstudy@kakao.com",
3   "password": "5678",
4   "events": []
5 }
```
- Response:**
 - Status: 409 Conflict
 - Time: 6 ms
 - Size: 166 B
 - Body (JSON):

```
1 {
2   "detail": "존재하는 이메일"
3 }
```

Modularization

❖ Router 구현

- ✓ user Router 구현
 - signin 확인

The screenshot displays a REST client interface with the following details:

- Method:** POST
- URL:** http://127.0.0.1:8080/user/signin
- Body (Request):**

```
1 {  
2   "email": "itstudy@kakao.com",  
3   "password": "1234"  
4 }
```
- Status:** 200 OK
- Time:** 4 ms
- Size:** 155 B
- Body (Response):**

```
1 {  
2   "message": "로그인 성공"  
3 }
```

Modularization

❖ Router 구현

- ✓ user Router 구현
 - signin 확인

The screenshot displays a REST client interface with a POST request to `http://127.0.0.1:8080/user/signin`. The request body is a JSON object containing email and password. The response status is 403 Forbidden, and the response body contains a detail message.

Request:

```
POST http://127.0.0.1:8080/user/signin
```

Body:

```
{  1  "email": "itstudy@kakao.com",
  2
  3  "password": "5678"
  4}
```

Response:

```
{  1  "detail": "비밀번호 틀림"
  2
  3}
```

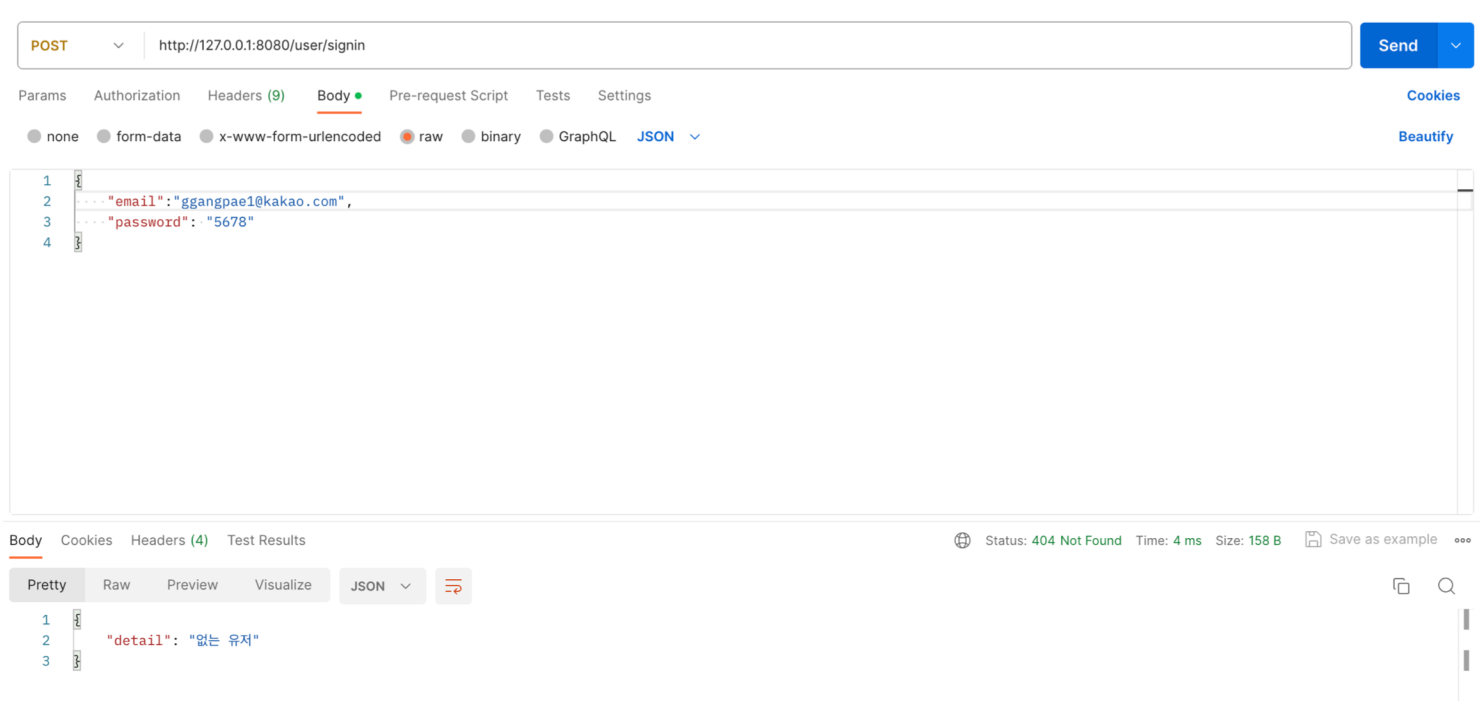
Status: 403 Forbidden **Time:** 4 ms **Size:** 164 B

Modularization

❖ Router 구현

✓ user Router 구현

● signin 확인



Modularization

❖ Router 구현

✓ event Router 구현

- routes/events.py 파일에 라우팅 코드 작성

```
from typing import List
```

```
from fastapi import APIRouter, Body, HTTPException, status  
from models.events import Event
```

```
event_router = APIRouter(  
    tags=["Events"]  
)
```

```
events = []
```

Modularization

❖ Router 구현

✓ event Router 구현

- routes/events.py 파일에 라우팅 코드 작성

#모든 이벤트 추출

```
@event_router.get("/", response_model=List[Event])
async def retrieve_all_events() -> List[Event]:
    return events
```

#id에 해당하는 이벤트 추출

```
@event_router.get("/{id}", response_model=Event)
async def retrieve_event(id: int) -> Event:
    for event in events:
        if event.id == id:
            return event
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="일치하는 이벤트가 없습니다."
    )
```

Modularization

❖ Router 구현

✓ event Router 구현

- routes/events.py 파일에 라우팅 코드 작성

#이벤트 생성

@event_router.post("/new")

async def create_event(body: Event = Body(...)) -> dict:

events.append(body)

return {

"message": "이벤트 생성 성공"

}

Modularization

❖ Router 구현

✓ event Router 구현

- routes/events.py 파일에 라우팅 코드 작성

```
@event_router.delete("/{id}")
async def delete_event(id: int) -> dict:
    for event in events:
        if event.id == id:
            events.remove(event)
            return {
                "message": "이벤트 삭제 성공"
            }

    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="해당하는 이벤트가 없음"
    )

@event_router.delete("/")
async def delete_all_events() -> dict:
    events.clear()
    return {
        "message": "이벤트 전체 삭제"
    }
```

Modularization

❖ Router 구현

✓ event Router 구현

- main.py 파일에 event_router 추가

```
from fastapi import FastAPI
from routes.users import user_router
from routes.events import event_router
```

```
import uvicorn
```

```
app = FastAPI()
```

```
# 라우터 등록
```

```
# 처리하는 경로 앞에 /user를 추가
```

```
app.include_router(user_router, prefix="/user")
```

```
app.include_router(event_router, prefix="/event")
```

```
#8000번 포트로 시작
```

```
if __name__ == '__main__':
```

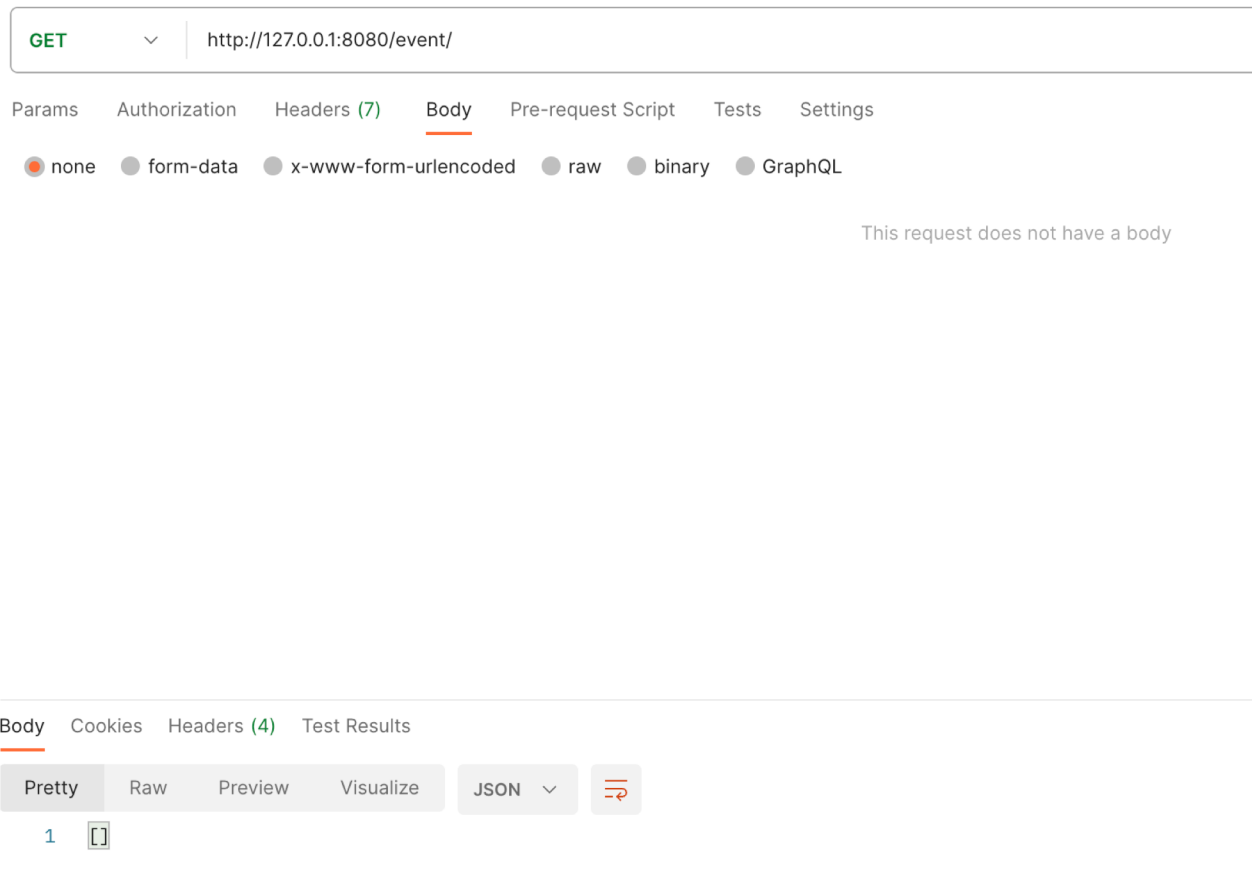
```
    uvicorn.run("main:app", host="0.0.0.0", port=8080, reload=True)
```

Modularization

❖ Router 구현

✓ event Router 구현

● 확인 – 모두 가져오기



Modularization

❖ Router 구현

✓ event Router 구현

● 확인 – 데이터 삽입

The screenshot displays a REST client interface with a POST request to `http://127.0.0.1:8080/event/new`. The request body is a JSON object with the following fields: `id` (1), `title` ("Fast API"), `image` (a URL to a teal logo), `description` ("빠르게 REST API 서버를 만들기 위한 라이브러리"), `tags` (["python", "fastapi", "rest-api"]), and `location` ("Google Meet"). The response status is 200 OK, and the body contains a JSON object with a `message` field: "이벤트 생성 성공".

```
POST http://127.0.0.1:8080/event/new

{
  "id": 1,
  "title": "Fast API",
  "image": "https://fastapi.tiangolo.com/img/logo-margin/logo-teal.png",
  "description": "빠르게 REST API 서버를 만들기 위한 라이브러리",
  "tags": ["python", "fastapi", "rest-api"],
  "location": "Google Meet"
}
```

Body Cookies Headers (4) Test Results

Status: 200 OK Time: 5 ms Size: 162 B Save as example

```
1
2 "message": "이벤트 생성 성공"
3
```

Modularization

❖ Router 구현

✓ event Router 구현

● 확인 - 데이터 삽입

The screenshot displays a REST client interface with a POST request to `http://127.0.0.1:8080/event/new`. The request body is a JSON object containing event details. The response shows a 200 OK status with a success message.

Request:

- Method: POST
- URL: `http://127.0.0.1:8080/event/new`
- Body (JSON):

```
1 {
2   "id": 2,
3   "title": "Spring Boot",
4   "image": "https://velog.velcdn.com/images/falling_star3/post/7eef0696-76c6-4dcb-858b-f91ef597eddc/%EC%8A%A4%ED%94%84%EB%A7%81%EB%B6%80%ED%8A%B8.JPG",
5   "description": "Java Application Framework",
6   "tags": ["java", "rest api"],
7   "location": "zoom"
8 }
```

Response:

- Status: 200 OK
- Time: 4 ms
- Size: 162 B
- Body (JSON):

```
1 {
2   "message": "이벤트 생성 성공"
3 }
```

Modularization

❖ Router 구현

✓ event Router 구현

● 확인 – 전체 데이터 가져오기

The screenshot shows a REST client interface. At the top, a GET request is configured for the URL `http://127.0.0.1:8080/event/`. The 'Body' tab is selected, showing the message 'This request does not have a body'. Below the request configuration, the response is displayed in the 'Body' tab. The response is a JSON array with two objects, each representing an event. The first object has an 'id' of 1, a 'title' of 'Fast API', an 'image' URL, a 'description', and a list of tags. The second object has an 'id' of 2 and a 'title' of 'Spring Boot'.

```
1 {
2   {
3     "id": 1,
4     "title": "Fast API",
5     "image": "https://fastapi.tiangolo.com/img/logo-margin/logo-teal.png",
6     "description": "빠르게 REST API 서버를 만들기 위한 라이브러리",
7     "tags": [
8       "python",
9       "fastapi",
10      "rest api"
11    ],
12     "location": "Google Meet"
13   },
14   {
15     "id": 2,
16     "title": "Spring Boot",
```

Modularization

❖ Router 구현

✓ event Router 구현

● 확인 – 데이터 가져오기

The screenshot shows a REST client interface with a GET request to `http://127.0.0.1:8080/event/1`. The request is sent, and the response is displayed in the 'Body' tab. The response is a JSON object with the following structure:

```
1 {
2   "id": 1,
3   "title": "Fast API",
4   "image": "https://fastapi.tiangolo.com/img/logo-margin/logo-teal.png",
5   "description": "빠르게 REST API 서버를 만들기 위한 라이브러리",
6   "tags": [
7     "python",
8     "fastapi",
9     "rest api"
10  ],
11   "location": "Google Meet"
12 }
```

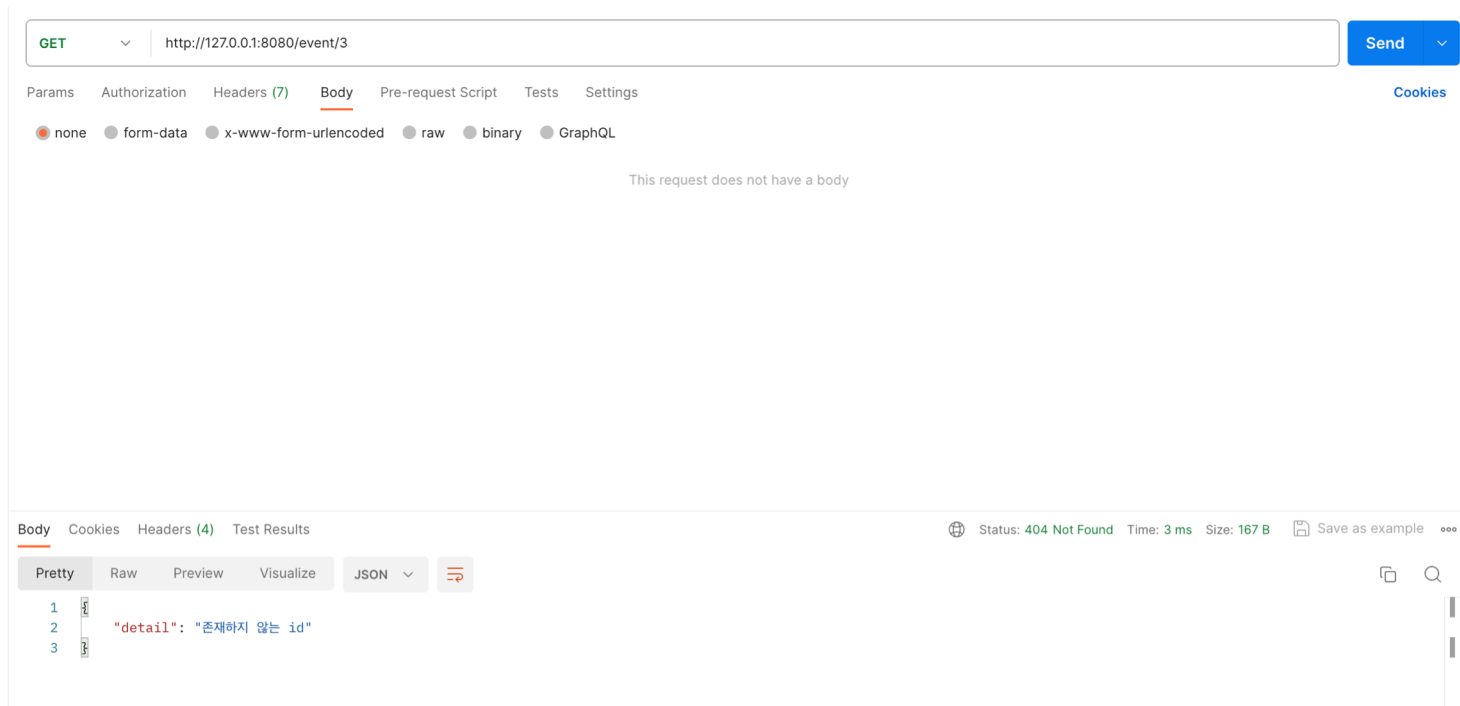
The status bar at the bottom indicates: Status: 200 OK, Time: 4 ms, Size: 364 B. There are also options to 'Save as example' and a search icon.

Modularization

❖ Router 구현

✓ event Router 구현

● 확인 – 데이터 가져오기

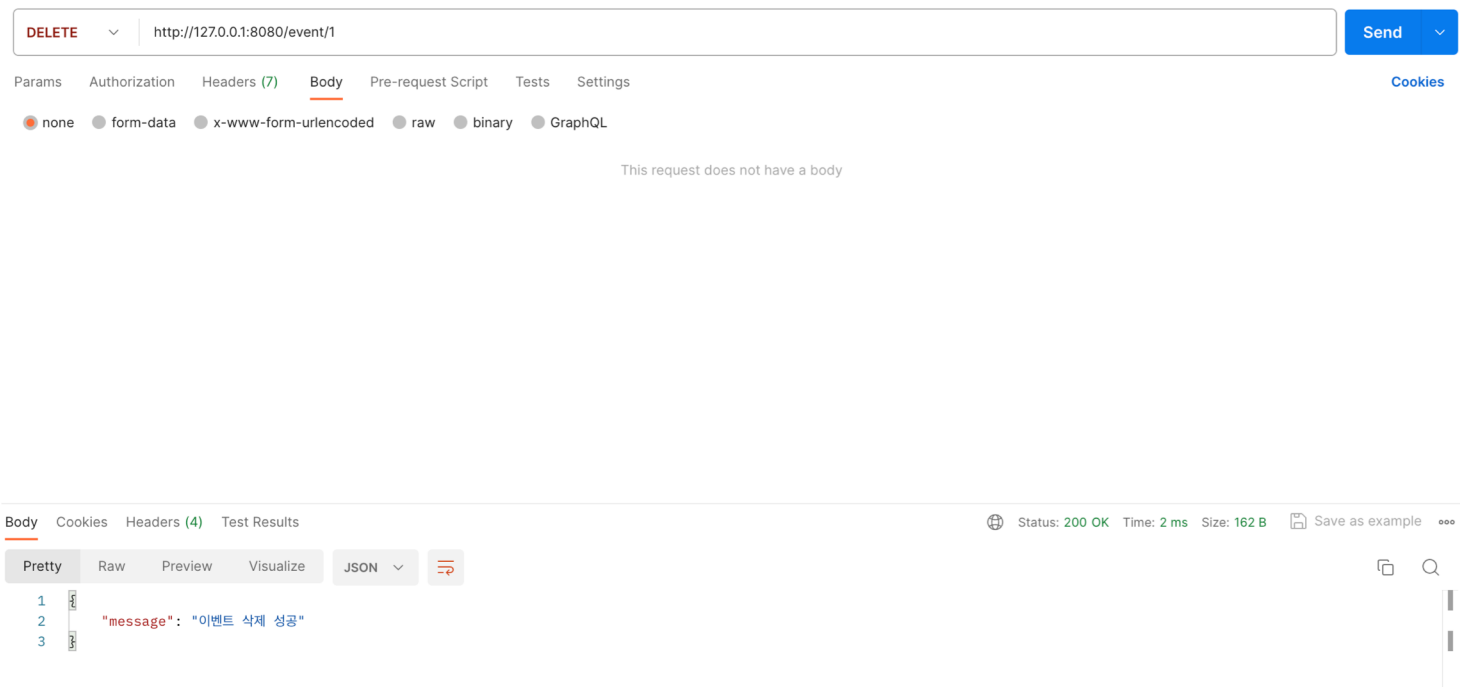


Modularization

❖ Router 구현

✓ event Router 구현

● 확인 – 데이터 삭제

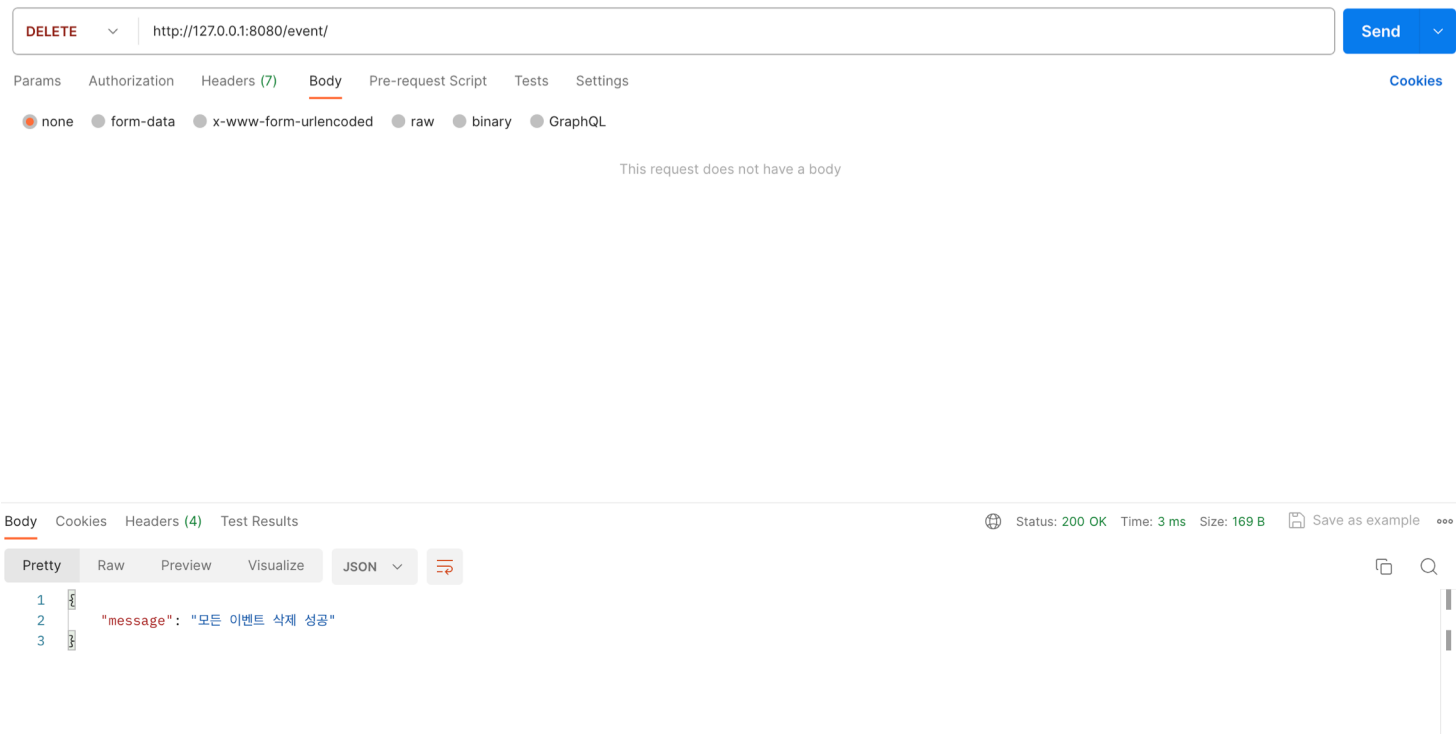


Modularization

❖ Router 구현

✓ event Router 구현

● 확인 – 모든 데이터 삭제



Database Use

❖ SQL 모델 설정

- ✓ 관계형 데이터베이스 와 Fast API Application을 연동하려면 SQLAlchemy 패키지를 설치해야 함
- ✓ SQLAlchemy 패키지는 pydantic 과 SQLAlchemy를 기반으로 함
- ✓ 패키지 설치: `pip install sqlalchemy`

Database Use

❖ SQL 모델 설정

✓ 테이블 생성 방법

- SQLAlchemy을 상속받는 모델 클래스를 생성하고 table 옵션에 True를 설정
- 변수 이름 과 자료형만 설정하면 기본 필드로 설정됨
- 필드의 특성을 지정하고자 하는 경우는 Field 함수를 이용

```
class Event(SQLModel, table=True):  
    id: Optional[int] = Field(default=None, primary_key=True)  
    title: str  
    image: str  
    description: str  
    tags: List[str]  
    location: str
```

Database Use

❖ SQL 모델 설정

✓ 행 생성 방법

● 테이블의 인스턴스를 생성

```
new_event = Event(title="Book Launch",  
                  image="src/fastapi.png",  
                  description="The book launch event will  
be held at Packt HQ, Packt city",  
                  location="Google Meet",  
                  tags=["packt", "book"])
```

Database Use

❖ SQL 모델 설정

✓ 관계형 데이터베이스 연결

- SQLAlchemy에서는 SQLAlchemy 엔진을 사용해서 데이터베이스를 연결
- SQLAlchemy 엔진은 `create_engine()` 메서드를 사용해서 만들며 SQLAlchemy 라이브러리에서 임포트
- `create_engine()` 메서드는 데이터베이스 URL을 인수로 사용하는데 데이터베이스 URL은 데이터베이스 종류마다 다름
- `create_engine()`은 `echo`를 선택적 인수로 지정할 수 있는데 `True`로 설정하면 실행된 SQL 명령을 출력
- `create_engine()` 메서드만으로는 데이터베이스를 생성할 수 없는데 파일을 만들 수 없는데 `SQLModel.metadata.create_all(engine)` 메서드를 사용해서 `create_engine()` 메서드의 인스턴스를 호출하면 데이터베이스를 생성할 수 있음

Database Use

❖ SQL 모델 설정

✓ 세션을 이용한 데이터베이스 트랜잭션 수행

- 세션 객체는 코드와 데이터베이스 사이에서 이루어지는 처리를 관리하며 주로 특정 처리를 데이터베이스에 적용하기 위해 사용
 - Session 클래스는 SQL 엔진의 인스턴스를 인수로 사용
 - Document: <https://docs.sqlalchemy.org/en/20/orm/session.html>
 - Session 클래스의 메서드
 - ◆ commit (): 현재 세션에 있는 트랜잭션을 완료
 - ◆ add(): 모델을 인수로 받아서 데이터를 추가
 - ◆ get(): 데이터베이스에서 단일 로우를 추출하는데 모델과 문서 ID를 인수로 사용
- ```
with Session(engine) as session :
 session.add(new_event)
 session.commit()
```

# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ MySQL 사용을 위한 패키지 설치: `pip install pymysql`



# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ models/users.py 파일의 User 클래스 정의를 수정

```
from sqlmodel import SQLModel, Field

class User(SQLModel, table=True):
 email: str = Field(primary_key=True)
 password: str
```

```
model_config = {
 "json_schema_extra": {
 "example": {
 "email": "fastapi@packt.com",
 "password": "strong!!!"
 }
 }
}
```

# Database Use

## ❖ Event 모델을 MySQL에 연결해서 사용

- ✓ 데이터베이스 접속 정보를 저장할 connection.py 파일을 생성하고 MySQL 접속 정보를 작성

```
from sqlalchemy import SQLModel, Session, create_engine
from models.events import Event
```

```
database_connection_string = "mysql+pymysql://root:wnddkd@localhost:3306/itstudy"
engine_url = create_engine(database_connection_string,
 echo=True, pool_recycle=500)
```

```
def conn():
 SQLModel.metadata.create_all(engine_url)
```

```
def get_session():
 with Session(engine_url) as session:
 yield session
```

# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ main.py 파일에 데이터베이스 접속을 위한 코드 추가

```
from fastapi import FastAPI
from routes.users import user_router
from models.users import User, UserSignIn
from routes.events import event_router
import uvicorn
```

```
from database.connection import conn
```

```
from contextlib import asynccontextmanager
@asynccontextmanager
async def lifespan(app: FastAPI):
 # Load the ML model
 print("lifespan start")
 conn()
 yield
```

# Database Use

## ❖ Event 모델을 MySQL에 연결해서 사용

- ✓ main.py 파일에 데이터베이스 접속을 위한 코드 추가

```
app = FastAPI(lifespan=lifespan)
```

```
라우터 등록
```

```
처리하는 경로 앞에 /user를 추가
```

```
app.include_router(user_router, prefix="/user")
```

```
처리하는 경로 앞에 /event를 추가
```

```
app.include_router(event_router, prefix="/event")
```

```
#8000번 포트로 시작
```

```
if __name__ == '__main__':
```

```
 uvicorn.run("main:app", host="0.0.0.0", port=8080, reload=True)
```

# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ 애플리케이션을 실행(python main.py)해서 로그 확인

```
CREATE TABLE user (
 email VARCHAR(255) NOT NULL,
 password VARCHAR(255) NOT NULL,
 PRIMARY KEY (email)
)
```

# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ User 작업

- routes/users.py 파일을 수정해서 회원 가입 과 로그인을 처리

```
from fastapi import APIRouter, HTTPException, status, Depends
from database.connection import get_session
from sqlmodel import select
from models.users import User
```

```
user_router = APIRouter(
 tags=["User"],
)
```

```
users = {}
```

# Database Use

## ❖ Event 모델을 MySQL에 연결해서 사용

### ✓ User 작업

- routes/users.py 파일을 수정해서 회원 가입 과 로그인을 처리

#signup 요청이 post 방식으로 전송된 경우 처리

@user\_router.post("/signup")

async def sign\_user\_up(data: User, session=Depends(get\_session)) -> dict:

statement = select(User)

users = session.exec(statement).all()

for user in users:

if data.email == user.email:

raise HTTPException(

status\_code=status.HTTP\_409\_CONFLICT,

detail="존재하는 이메일"

)

session.add(data)

session.commit()

session.refresh(data)

print(data)

return {

"message": "유저 등록 성공"

}

# Database Use

## ❖ Event 모델을 MySQL에 연결해서 사용

### ✓ User 작업

- routes/users.py 파일을 수정해서 회원 가입 과 로그인을 처리

#signin 요청을 post 방식으로 전송한 경우

@user\_router.post("/signin")

async def sign\_user\_in(data: User, session=Depends(get\_session)) -> dict:

statement = select(User)

users = session.exec(statement).all()

#email이 존재하지 않는다면

for user in users:

if user.email == data.email:

if user.password == data.password:

return {

"message": "로그인 성공"

}

else:

raise HTTPException(

status\_code=status.HTTP\_403\_FORBIDDEN,

detail="비밀번호 틀림"

)

# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ User 작업

- routes/users.py 파일을 수정해서 회원 가입 과 로그인을 처리

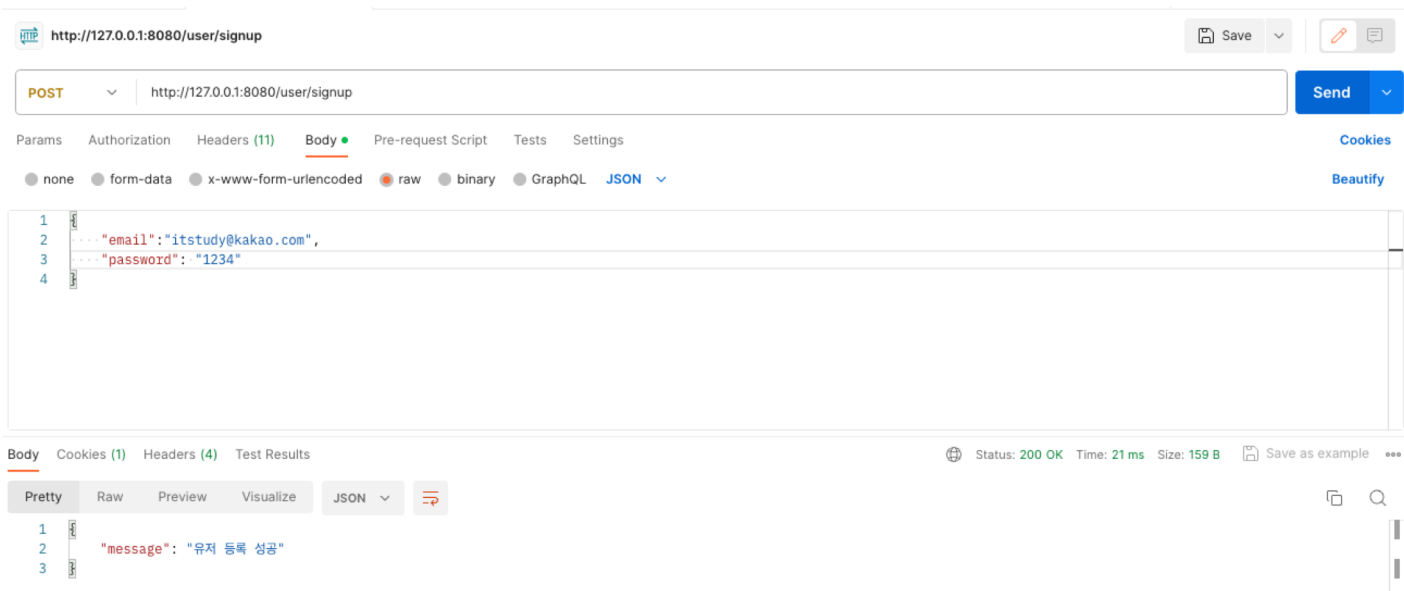
```
raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail="없는 유저"
)
```

# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

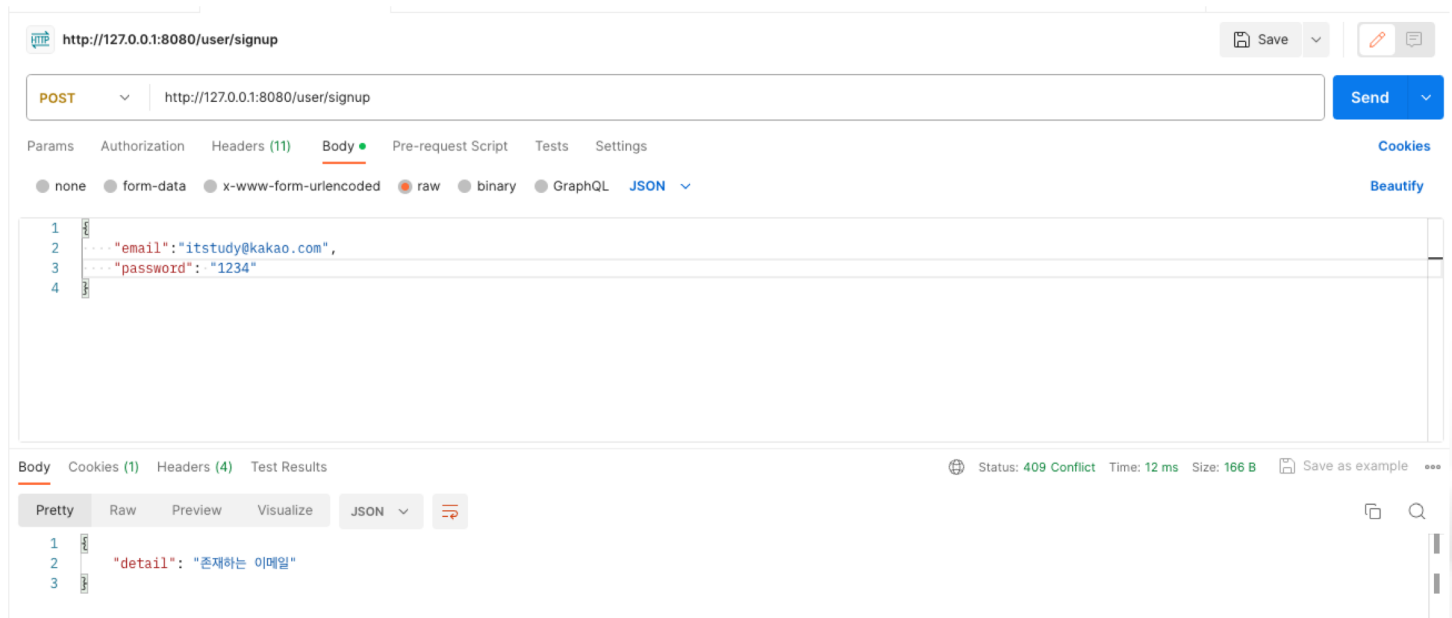
✓ User 작업

● 회원 가입 확인



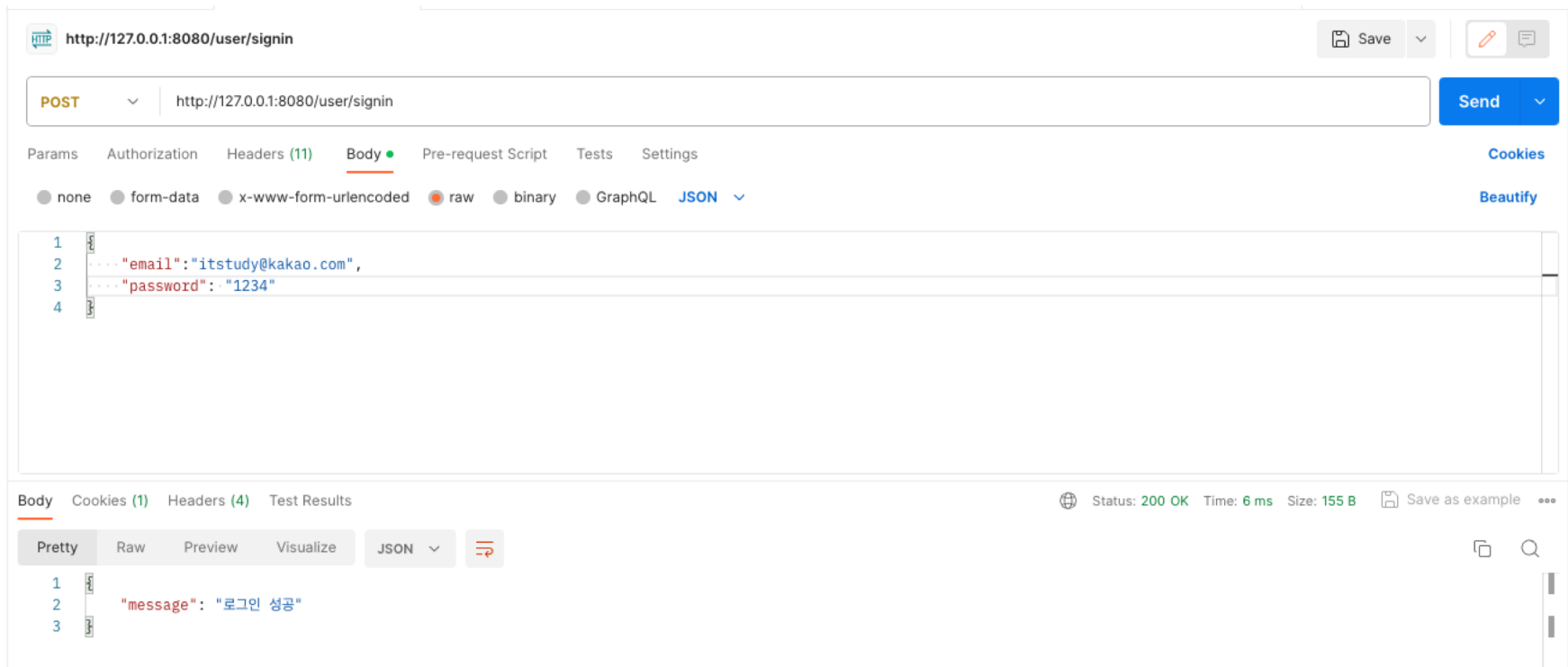
# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ User 작업
    - 회원 가입 확인



# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ User 작업
    - 로그인 확인



# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ User 작업
    - 로그인 확인

The screenshot shows a REST client interface with the following details:

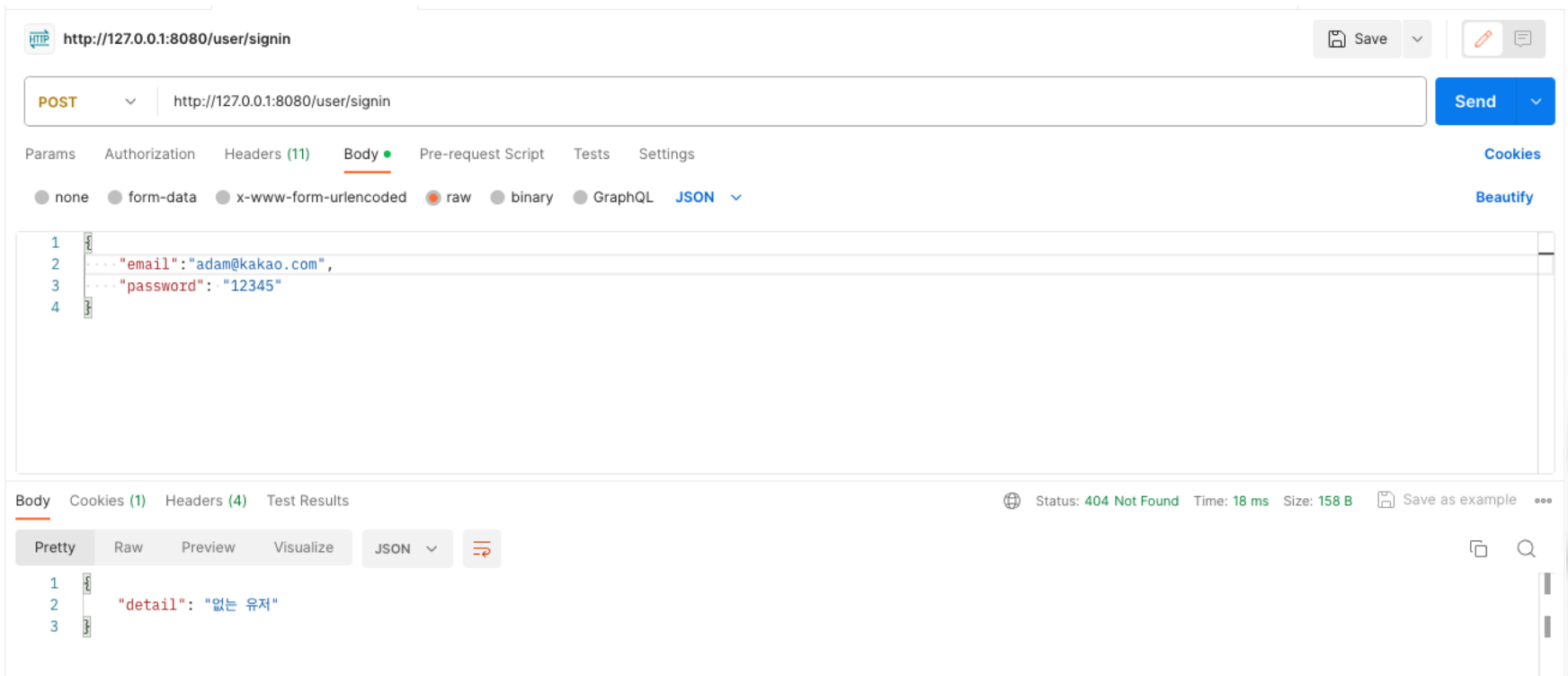
- URL:** http://127.0.0.1:8080/user/signin
- Method:** POST
- Body (JSON):**

```
1 {
2 "email": "itstudy@kakao.com",
3 "password": "12345"
4 }
```
- Response:**
  - Status:** 403 Forbidden
  - Time:** 13 ms
  - Size:** 164 B
  - Body (JSON):**

```
1 {
2 "detail": "비밀번호 틀림"
3 }
```

# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ User 작업
    - 로그인 확인



HTTP client interface showing a POST request to `http://127.0.0.1:8080/user/signin`. The request body is a JSON object:

```
{ "email": "adam@kakao.com", "password": "12345"}
```

The response status is **404 Not Found**. The response body is a JSON object:

```
{ "detail": "없는 유저"}
```

# Database Use

## ❖ Event 모델을 MySQL에 연결해서 사용

- ✓ models/events.py 파일의 Event 클래스 정의를 수정하고 수정을 위한 클래스 추가

```
from sqlalchemy import JSON, SQLAlchemy, Field, Column, ForeignKey
from typing import Optional, List
```

```
class Event(SQLAlchemy, table=True):
 id: int = Field(default=None, primary_key=True)
 title: str
 image: str
 description: str
 tags: List[str] = Field(sa_column=Column(JSON))
 location: str
 email: str = Column(ForeignKey("user.email"), nullable=False)
```

# Database Use

## ❖ Event 모델을 MySQL에 연결해서 사용

- ✓ models/events.py 파일의 Event 클래스 정의를 수정하고 수정을 위한 클래스 추가

```
class EventUpdate(SQLModel):
 title: Optional[str] = Field(None)
 image: Optional[str] = Field(None)
 description: Optional[str] = Field(None)
 tags: Optional[List[str]] = Field(None)
 location: Optional[str] = Field(None)
```

# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

- routes/events.py 파일의 import 수정

```
from typing import List
```

```
from fastapi import APIRouter, Body, HTTPException, status, Depends
```

```
from database.connection import get_session
```

```
from models.events import Event, EventUpdate
```

# Database Use

## ❖ Event 모델을 MySQL에 연결해서 사용

### ✓ 이벤트 작업

#### ● 이벤트 추가

#### ◆ routes/events.py 파일의 이벤트 추가 처리 수정

#새로운 이벤트 등록

@event\_router.post("/new")

async def create\_event(new\_event: Event, session=Depends(get\_session)) -> dict:

    session.add(new\_event)

    session.commit()

    session.refresh(new\_event)

    print(new\_event)

    return {

        "message": "이벤트 생성 성공"

    }

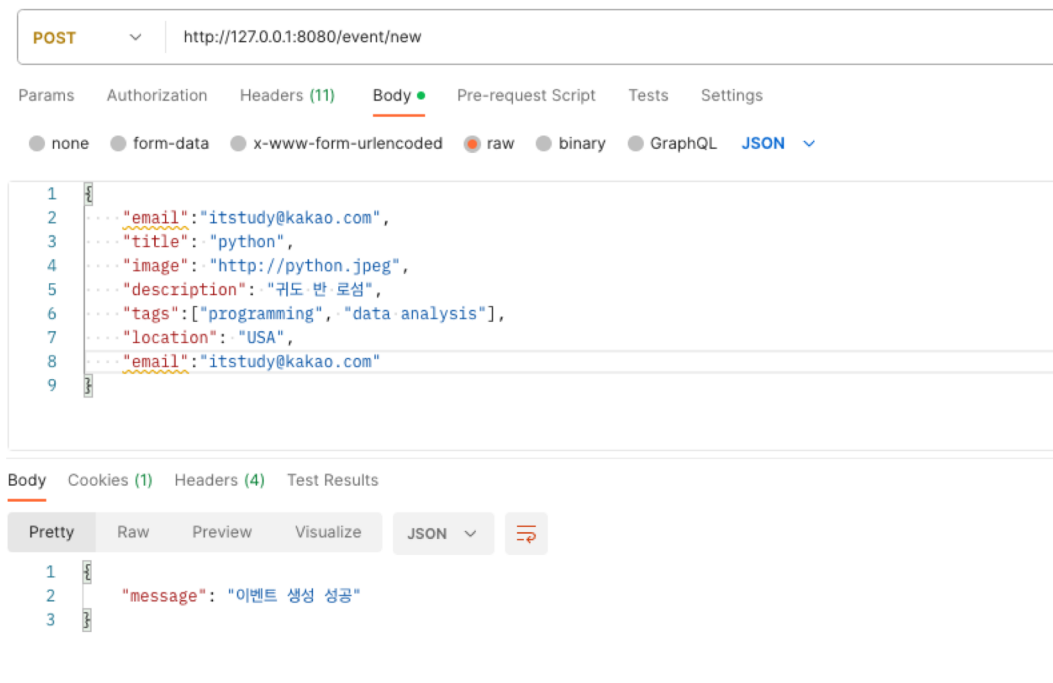
# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

● 이벤트 추가

◆ 확인



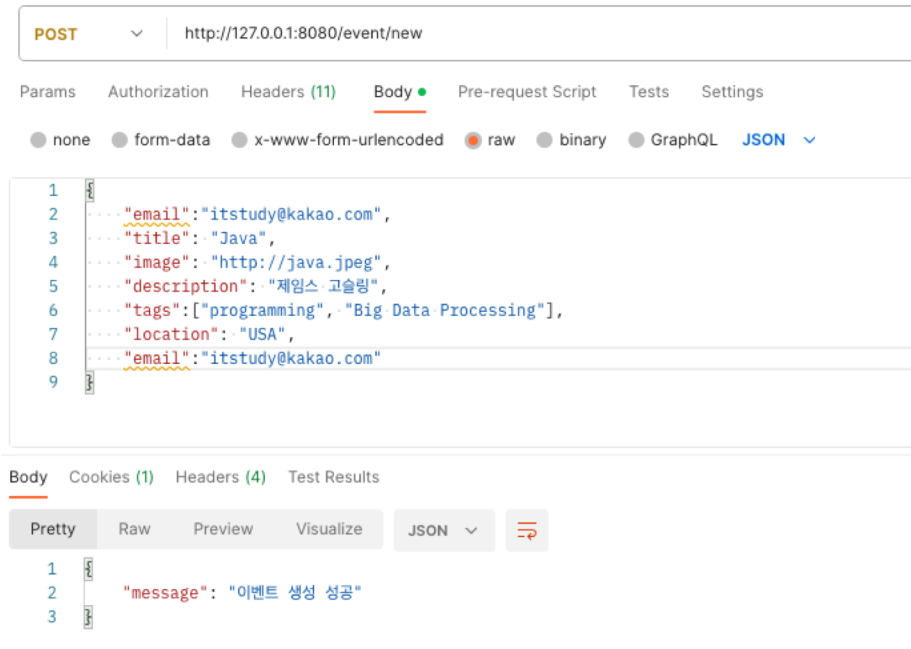
# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

● 이벤트 추가

◆ 확인



# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

● 이벤트 추가

◆ 확인

|   |  id ▼ |  title ▼ |  image ▼                                           |  description ▼ |  tags ▼ |  location ▼ |  email ▼ |
|---|----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| 1 | 1                                                                                      | python                                                                                    |  <a href="http://python.org">http://python.org</a> | 귀도 반 로섬                                                                                          | ["programmin                                                                               | USA                                                                                            | itstudy@kakao.                                                                              |
| 2 | 2                                                                                      | Java                                                                                      |  <a href="http://java.java">http://java.java</a>   | 제임스 고슬링                                                                                          | ["programmin                                                                               | USA                                                                                            | itstudy@kakao.                                                                              |

# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

● 이벤트 조회

◆ routes/events.py 파일의 이벤트 조회 처리 수정

```
from sqlmodel import select
```

```
#모든 이벤트 가져오기
```

```
@event_router.get("/", response_model=List[Event])
```

```
async def retrieve_all_events(session=Depends(get_session)) -> List[Event]:
```

```
 statement = select(Event)
```

```
 events = session.exec(statement).all()
```

```
 return events
```

# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

● 이벤트 조회

◆ routes/events.py 파일의 이벤트 조회 처리 수정

#id에 해당하는 이벤트만 가져오기

@event\_router.get("/{id}", response\_model=Event)

async def retrieve\_event(id: int, session=Depends(get\_session)) -> Event:

event = session.get(Event, id)

if event:

return event

raise HTTPException(

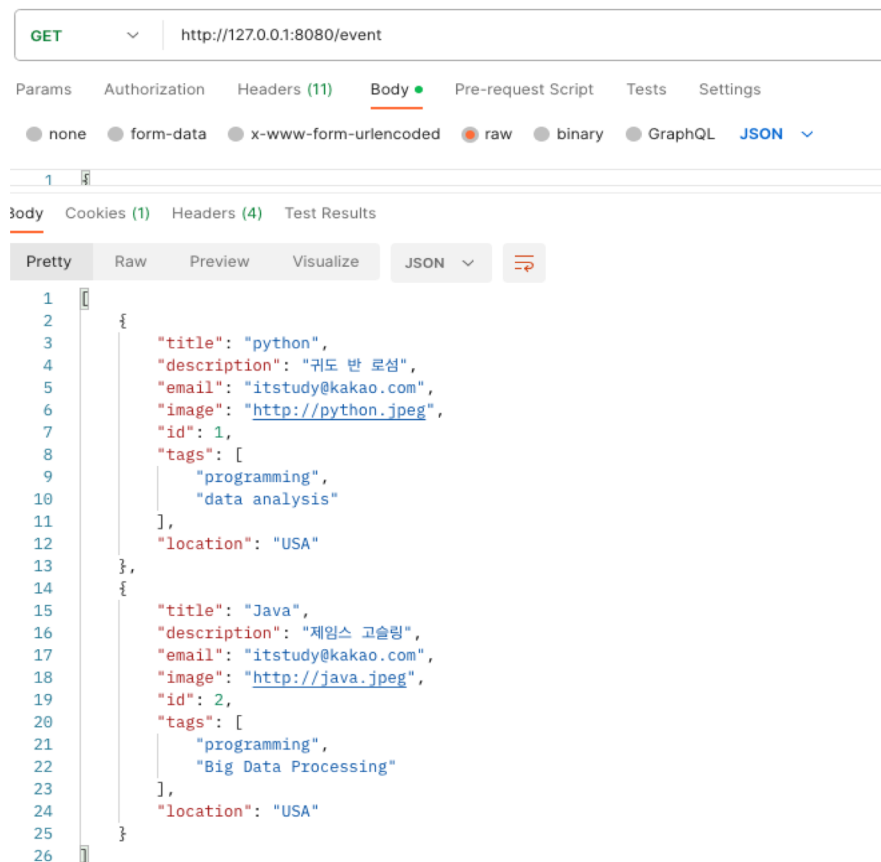
status\_code=status.HTTP\_404\_NOT\_FOUND,

detail="존재하지 않는 id"

)

# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ 이벤트 작업
    - 이벤트 조회
      - ◆ 전체 데이터 가져오기 확인

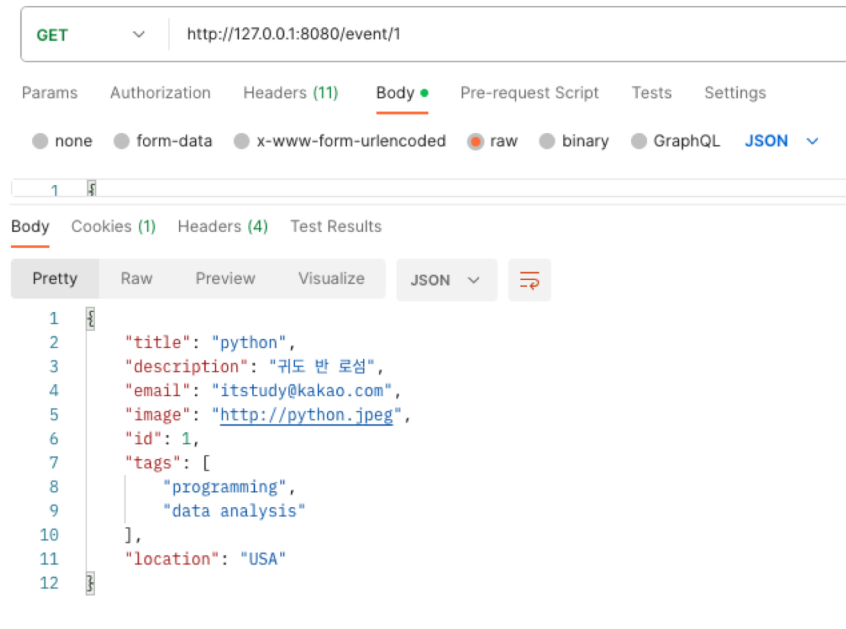


The screenshot shows a REST client interface with a GET request to `http://127.0.0.1:8080/event`. The response is displayed in JSON format, showing a list of two events. The first event is for 'python' and the second is for 'Java'.

```
1 [
2 {
3 "title": "python",
4 "description": "귀도 반 로섬",
5 "email": "itstudy@kakao.com",
6 "image": "http://python.jpeg",
7 "id": 1,
8 "tags": [
9 "programming",
10 "data analysis"
11],
12 "location": "USA"
13 },
14 {
15 "title": "Java",
16 "description": "제임스 고슬링",
17 "email": "itstudy@kakao.com",
18 "image": "http://java.jpeg",
19 "id": 2,
20 "tags": [
21 "programming",
22 "Big Data Processing"
23],
24 "location": "USA"
25 }
26]
```

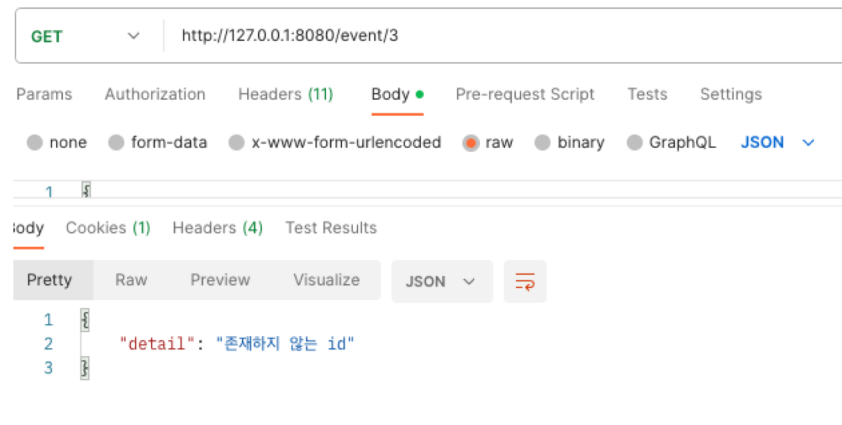
# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ 이벤트 작업
    - 이벤트 조회
      - ◆ id에 해당하는 데이터 가져오기 확인



# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ 이벤트 작업
    - 이벤트 조회
      - ◆ id에 해당하는 데이터 가져오기 확인



# Database Use

## ❖ Event 모델을 MySQL에 연결해서 사용

### ✓ 이벤트 작업

#### ● 이벤트 수정

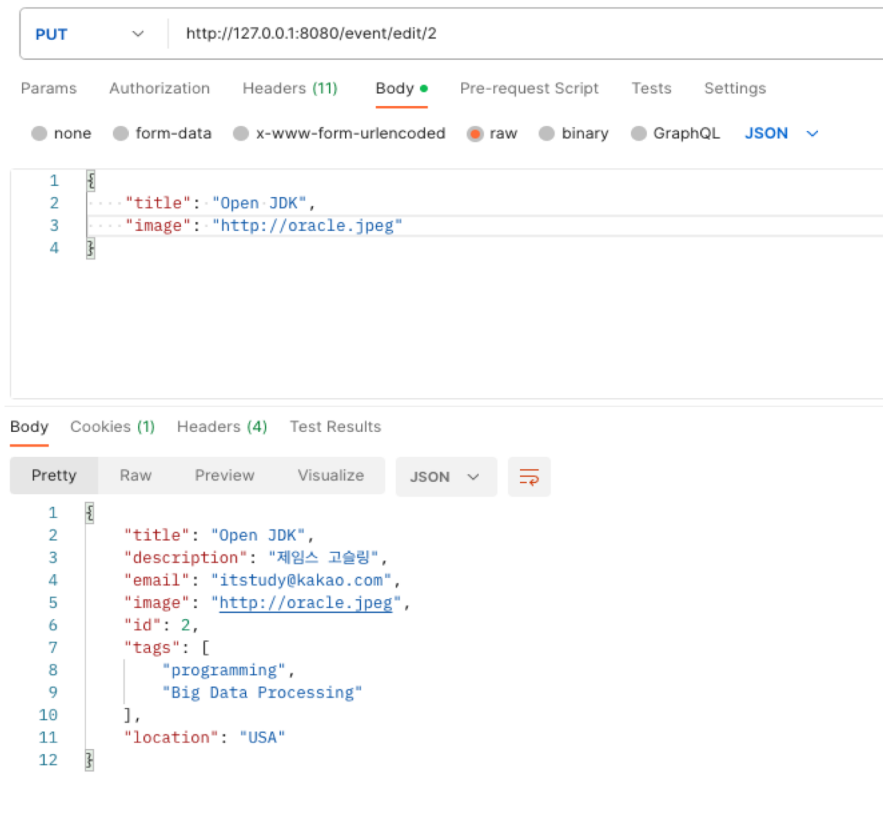
#### ◆ routes/events.py 파일의 이벤트 수정 처리 추가

```
@event_router.put("/edit/{id}", response_model=Event)
async def update_event(id: int,
 new_data:EventUpdate,
 session=Depends(get_session)) -> dict:
 event = session.get(Event, id)
 if event:
 event_data = new_data.dict(exclude_unset=True)
 for key, value in event_data.items():
 if value != None:
 setattr(event, key, value)
 session.add(event)
 session.commit()
 session.refresh(event)

 return event
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail="존재하지 않는 id"
)
```

# Database Use

- ❖ Event 모델을 MySQL에 연결해서 사용
  - ✓ 이벤트 작업
    - 이벤트 수정
      - ◆ 확인



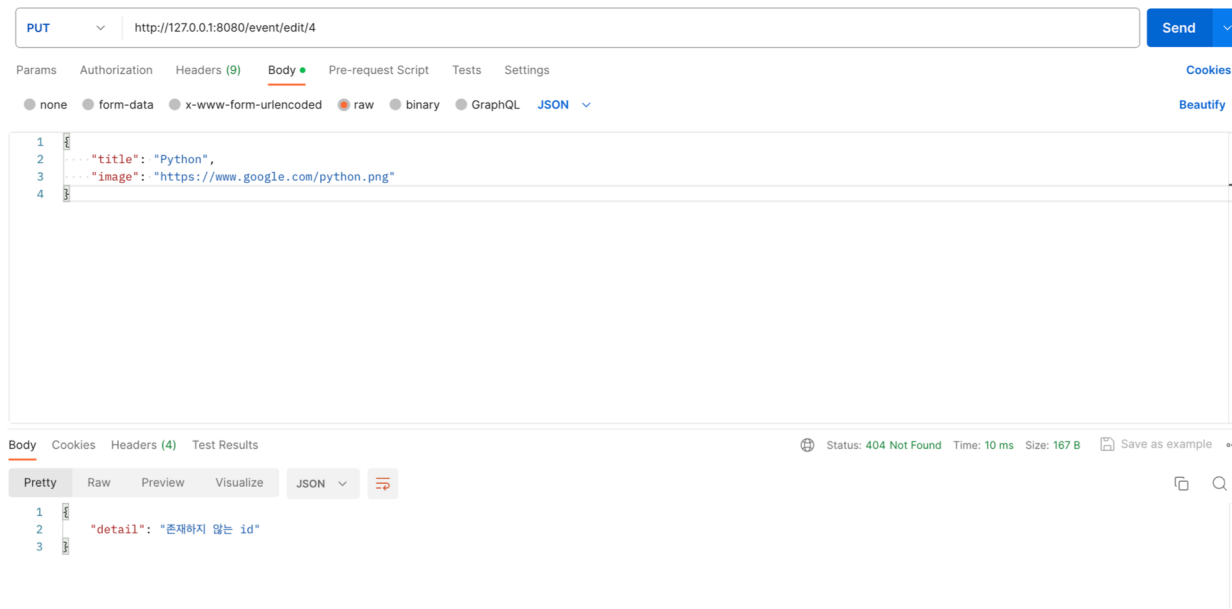
# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

● 이벤트 수정

◆ 확인



# Database Use

## ❖ Event 모델을 MySQL에 연결해서 사용

### ✓ 이벤트 작업

#### ● 이벤트 삭제

#### ◆ routes/events.py 파일의 이벤트 삭제 처리 수정

```
@event_router.delete("/{id}")
async def delete_event(id: int, session=Depends(get_session)) -> dict:
 event = session.get(Event, id)
 if event:
 session.delete(event)
 session.commit()
 return {
 "message": "이벤트 삭제 성공"
 }

 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail="존재하지 않는 id"
)
```

# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

● 이벤트 삭제

◆ routes/events.py 파일의 이벤트 삭제 처리 수정

```
@event_router.delete("/")
async def delete_all_events(session=Depends(get_session)) -> dict:
 statement = select(Event)
 events = session.exec(statement).all()
 for event in events:
 session.delete(event)
 session.commit()
 return {
 "message": "모든 이벤트 삭제 성공"
 }
```

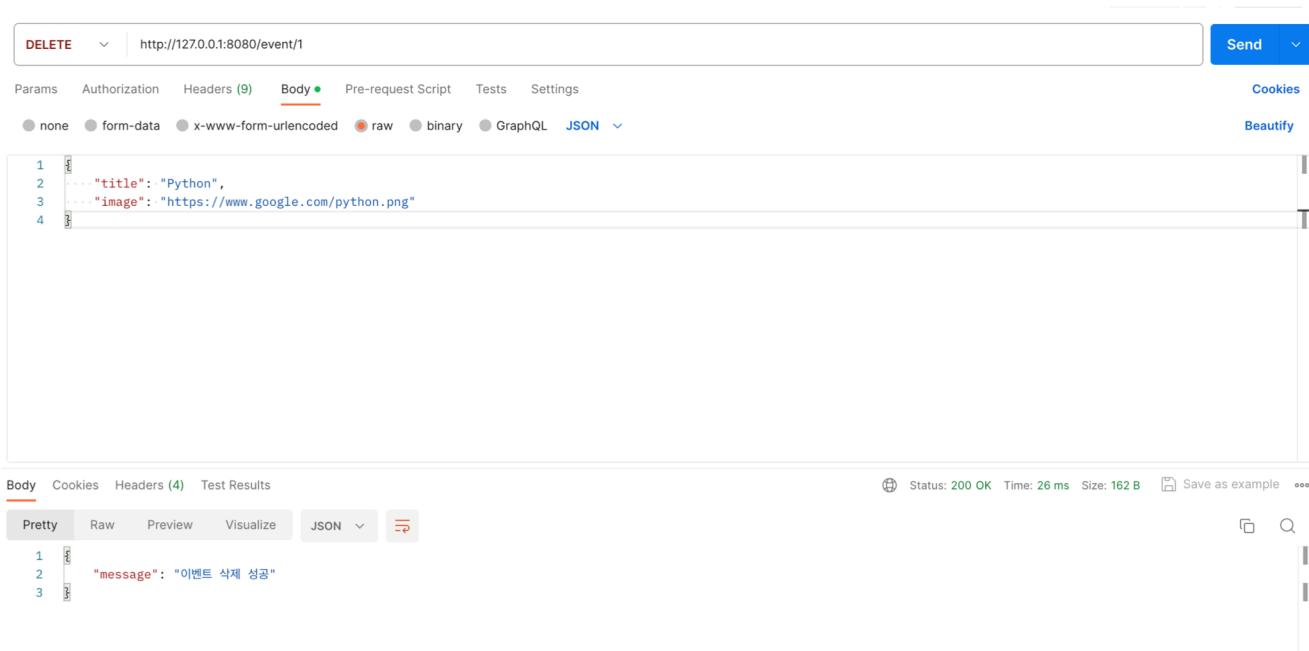
# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

● 이벤트 삭제

◆ 확인



# Database Use

❖ Event 모델을 MySQL에 연결해서 사용

✓ 이벤트 작업

● 이벤트 삭제

◆ 확인

