Fast API

Fast API

- Fast API
  - ✓ 빠르고 Python 표준 타입 힌트에 기초한 Python3.6+의 API를 빌드하기 위한 웹 프레임워크
  - ✓ NodeJS 및 Go와 대등할 정도로 매우 높은 성능을 가지는데 사용 가능한 가장 빠른 Python 프레임워크 중 하나
  - ✓ 약 200%에서 300%까지 기능 개발 속도 증가
  - ✓ 사람(개발자)에 의한 에러 약 40% 감소
  - ✓ 쉽게 사용하고 배우도록 설계
  - ✓ 적은 문서 읽기 시간
  - ✓ API에 대한 개방형 표준 기반: Open API(이전에 Swagger로 알려졌던) 및 JSON 스키마

Fast API

- Fast API Application 개발 환경 생성 및 실행
  - ✓ 가상 환경 생성 및 활성화
    - mkdir todos
    - cd todos
    - python3 -m venv venv
    - source venv/bin/activate(venv₩Scripts₩activate)
  - ✓ 의존성 라이브러리 설치
    - fastapi
    - uvicorn

      pip install fastapi uvicorn

Fast API

- Fast API Application 개발 환경 생성 및 실행
  - ✓ 서버 역할을 수행할 main.py 파일을 디렉토리에 생성하고 작성

```
from fastapi import FastAPI

app = FastAPI()

#기본 요청이 오면 아래 함수를 수행해서 결과를 리턴
@app.get("/")
async def welcome() -> dict:
    return {
        "message": "손으로 코딩하고 뇌로 컴파일하면 눈으로 디버깅한다."
    }
```

Fast API

- Fast API Application 개발 환경 생성 및 실행
  - ✓ 실행 명령 - uvicorn file:instance --port 포트번호 --reload
    - file: FastAPI 인스턴스가 존재하는 파이썬 파일 경로
    - instance: FastAPI 인스턴스 이름
    - --port: 애플리케이션에 접속할 포트 번호 설정
    - --reload: 파일의 내용이 변경될 때 자동으로 재시작하는 옵션으로 생략 가능
    - --host: 외부에서 접속 가능한 IP로 실행하는 옵션으로 0.0.0.0 이면 어떤 컴퓨터에서 실행해도 IP로 접속 가능
  - ✓ 실행
    - uvicorn api:app --port 8000 --reload

```
INFO:     Will watch for changes in these directories: ['/Users/adam/Documents/lecture/python/2.Python_Web/test/fastapi/todos']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [28543] using StatReload
INFO:     Started server process [28545]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Fast API

- Fast API Application 개발 환경 생성 및 실행
  - ✓ 확인 – 브라우저에서 127.0.0.1:8000

Fast API

- Fast API Application 개발 환경 생성 및 실행
  - ✓ 외부에서 접속 가능하도록 실행
    - uvicorn main:app --host 0.0.0.0 --port 8000 --reload

Fast API

- Fast API Application 개발 환경 생성 및 실행
  - ✓ CORS 설정

```
from fastapi.middleware.cors import CORSMiddleware

#허용할 URL을 list로 작성
origins = [
    "http://localhost:3000",
]

#app이 처리하기 전에 미들웨어에 CORS 사용 여부를 설정
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

요청 처리

- Routing
  - ✓ 클라이언트가 서버로 보내는 HTTP 요청을 처리하는 프로세스
  - ✓ HTTP 요청(Request)이 지정한 Router로 전송되면 미리 정의된 코드가 해당 요청을 처리해서 결과를 반환(Response)
  - ✓ 요청 메서드
    - GET: 리소스 조회에 사용되며 데이터는 쿼리 스트링을 이용해서 전달, 데이터 조회에 POST를 사용할 수 있지만 Caching 때문에 GET을 사용하는 것이 유리하지만 서버에게 전달하는 데이터의 크기에 제한이 있고 데이터가 외부로 노출됨
    - POST: 리소스 삽입에 사용되면 메시지 바디를 통해서 서버로 요청을 전달해서 처리
    - PUT: 리소스 대체에 사용되는데 없으면 생성
    - PATCH: 리소스의 부분 변경
    - DELETE: 리소스 삭제

요청 처리

- FastAPI Routing
  - ✓ Router는 HTTP 요청 메서드의 요청을 수락하고 선택적으로 인수를 받을 수 있도록 정의
  - ✓ 요청이 특정 Router로 전달되면 애플리케이션은 Route Handler가 요청을 처리하기 전에 해당 요청을 처리할 수 있는 Router가 정의되어 있는지 확인한 후 Route Handler를 호출해서 처리하며 Router가 정의되어 있지 않다면 404 Error
  - ✓ FastAPI 인스턴스는 단일 경로만 처리 가능

요청 처리

- APIRouter
  - ✓ 다중 라우팅을 위한 경로 처리 클래스
  - ✓ 애플리케이션 라우팅 과 로직을 독립적으로 구성하고 모듈화 가능
  - ✓ APIRouter 클래스를 이용해서 라우팅을 하고 이를 메인 파일에 import 해서 여러 요청을 처리

요청 처리

- APIRouter
  - ✓ 여러 요청 처리
    - 요청을 처리할 수 있는 todo.py 파일을 생성하고 작성

```
from fastapi import APIRouter
#라우터 객체 생성
todo_router = APIRouter()

#데이터를 저장할 list 생성
todo_list = []
```

요청 처리

- APIRouter
  - ✓ 여러 요청 처리
    - 요청을 처리할 수 있는 todo.py 파일을 생성하고 작성

```
#todo 요청을 post 방식으로 요청한 경우 처리
@todo_router.post("/todo")
#매개변수를 받아서 todo_list에 삽입
async def add_todo(todo: dict) -> dict:
    todo_list.append(todo)
    return {
        "message": "Todo added successfully."
    }

#todo 요청을 get 방식으로 요청한 경우 처리
@todo_router.get("/todo")
async def retrieve_todo() -> dict:
    return {
        "todos": todo_list
    }
```

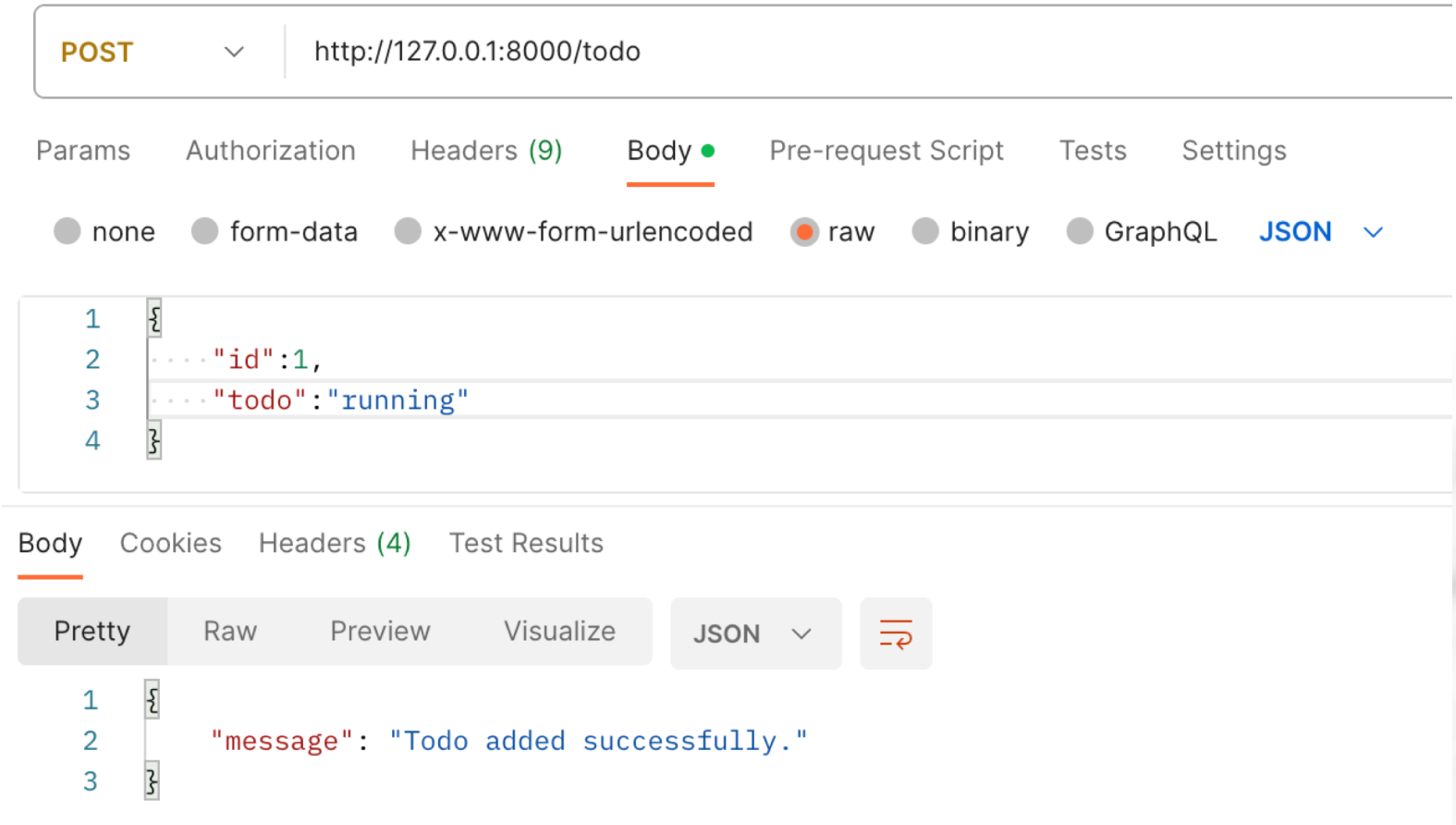
요청 처리

- APIRouter
  - ✓ 여러 요청 처리
    - main.py 파일에 라우터를 등록

```
from fastapi import FastAPI
from todo import todo_router

app = FastAPI()

#기본 요청이 오면 아래 함수를 수행해서 결과를 리턴
@app.get("/")
async def welcome() -> dict:
    return {
        "message": "손으로 코딩하고 뇌로 컴파일하면 눈으로 디버깅한다."
    }

app.include_router(todo_router)
```

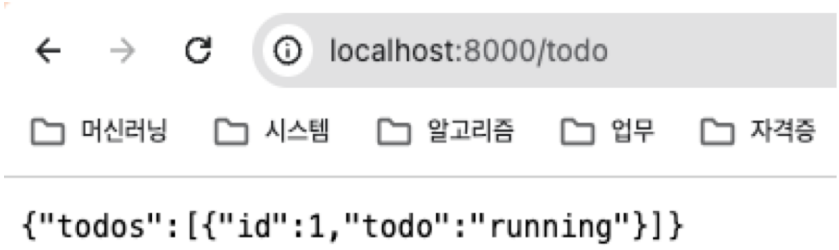
요청 처리

- APIRouter
  - ✓ 여러 요청 처리
    - postman을 이용해서 데이터 삽입 확인



요청 처리

- ❖ APIRouter
  - ✓ 여러 요청 처리
    - ● 브라우저에서 데이터 조회 확인



요청 처리

- ❖ Request Body
  - ✓ 개요
    - FastAPI에서는 정의된 모양의 데이터만 전송되도록 요청 바디를 설정할 수 있음
    - 요청 데이터가 적절한지 확인하고 악의적인 공격의 위험을 줄일 수 있음
    - FastAPI에서는 Model 클래스를 이용해서 검증을 할 수 있는데 이 클래스는 pydantic 패키지의 BaseModel 클래스의 하위 클래스로 생성
    - 현재는 POST 요청의 매개변수가 dict로 되어있는데 빈 객체를 전송해도 에러가 발생하지 않고 데이터가 삽입됨

요청 처리

- Request Body
  - ✓ 수정
    - 요청 바디를 처리할 모델을 생성 – models.py

```
from pydantic import BaseModel

class Todo(BaseModel):
    #필수
    id: int
    #선택
    item: str | None = None
```

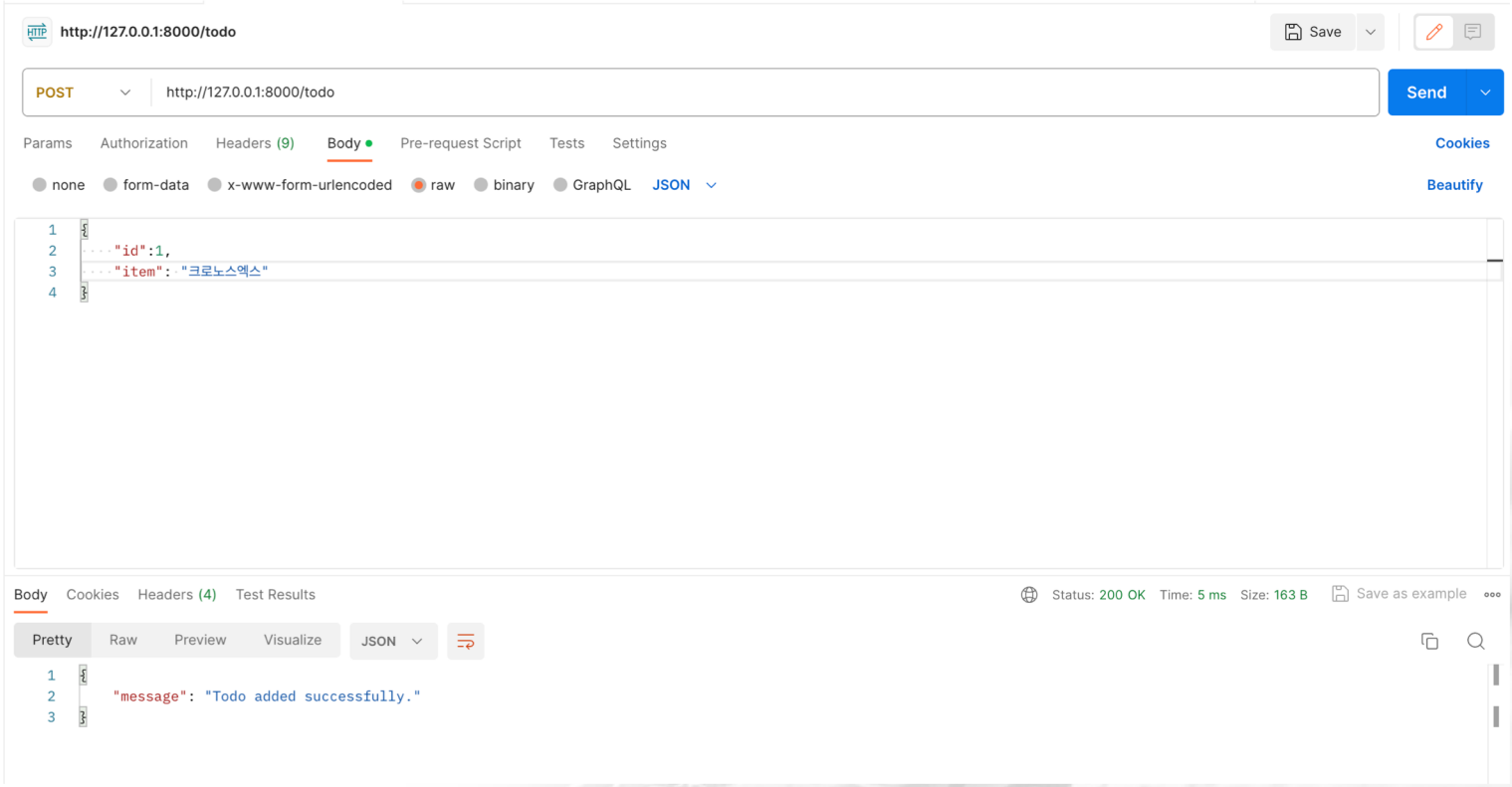
요청 처리

- Request Body
  - ✓ 수정
    - POST 요청 처리 메서드를 수정 – todo.py

```
from models import Todo

#todo 요청을 post 방식으로 요청한 경우 처리
@todo_router.post("/todo")
#매개변수를 받아서 todo_list에 삽입
async def add_todo(todo: Todo) -> dict:
    todo_list.append(todo)
    return {
        "message": "Todo added successfully."
    }
```

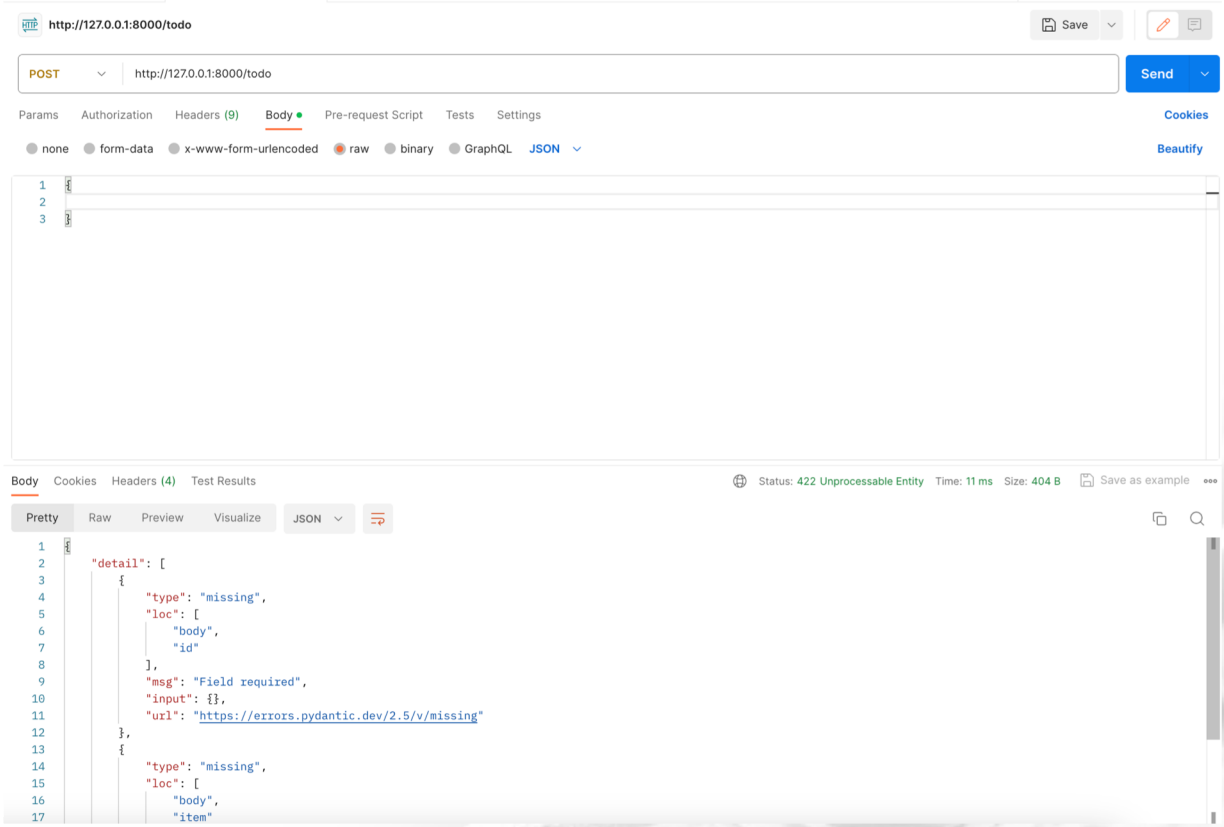
요청 처리

- ❖ Request Body
  - ✓ 확인
    - ● 정상적인 객체 전송



요청 처리

- ❖ Request Body
  - ✓ 확인
    - ● 빈 객체 전송 - 에러



요청 처리

- Parameter
  - ✓ 경로 매개변수
    - 경로 부분에 매개변수를 추가해서 사용하는 방식
    - 매개변수로 사용하고자 하는 부분에 { } 와 함께 사용할 변수 이름을 지정해서 사용

```
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

    - 경로 매개변수 와 고정 경로를 가지는 형태가 존재하는 경우 상수를 처리하는 코드가 앞에 위치해야 함

```
@app.get("/users/me")
async def read_user_me():
    return {"user_id": "the current user"}


@app.get("/users/{user_id}")
async def read_user(user_id: str):
    return {"user_id": user_id}
```

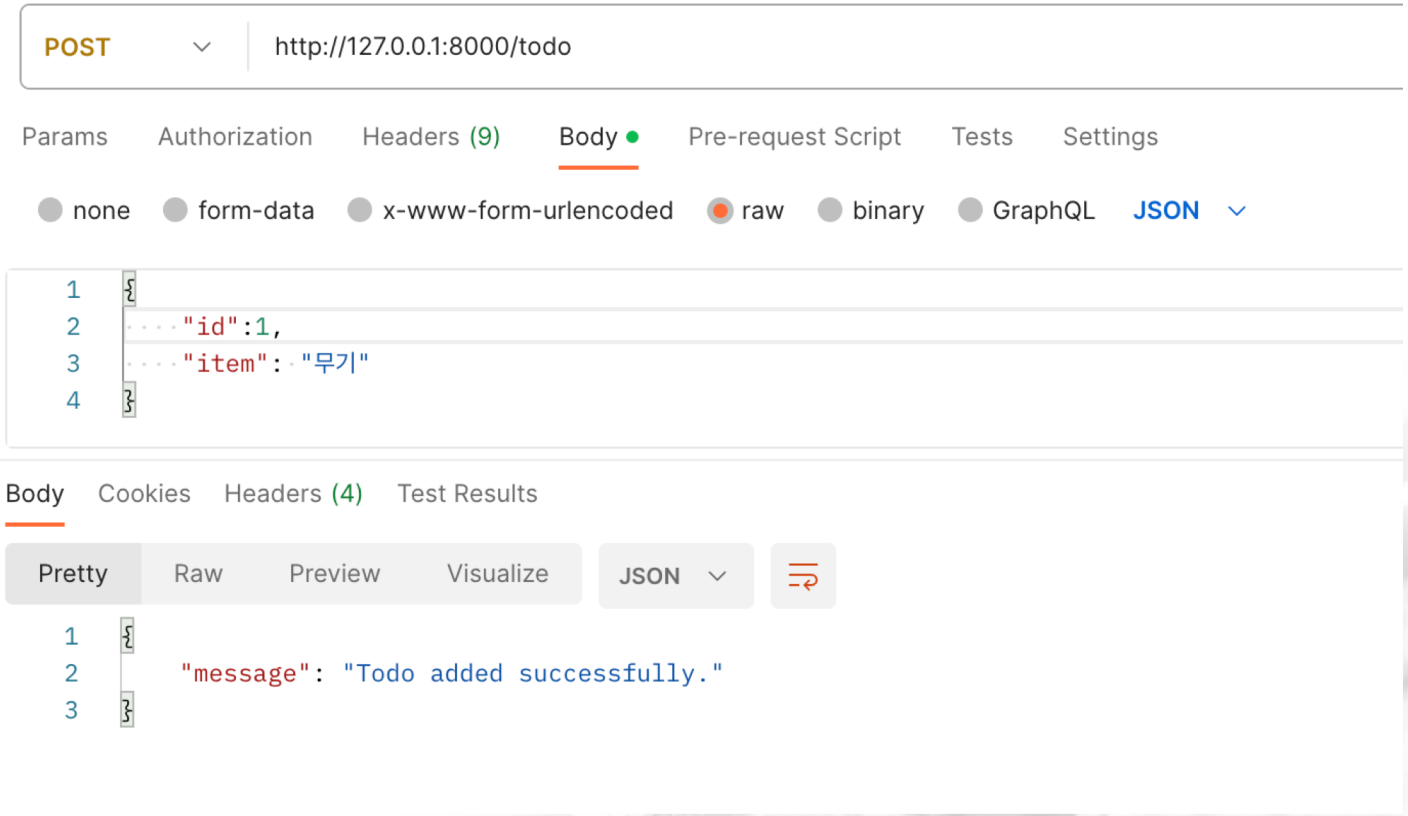
요청 처리

- Parameter
  - ✓ 경로 매개변수
    - todo.py 파일에 아이디를 받아서 리턴하는 요청 처리 메서드를 추가

```
#id를 받아서 하나의 데이터를 리턴하는 요청을 처리
@todo_router.get("/todo/{todo_id}")
async def get_single_toto(todo_id:int) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            return{
                "todo":todo
            }
    return{
        "message": "존재하지 않는 ID"
    }
```

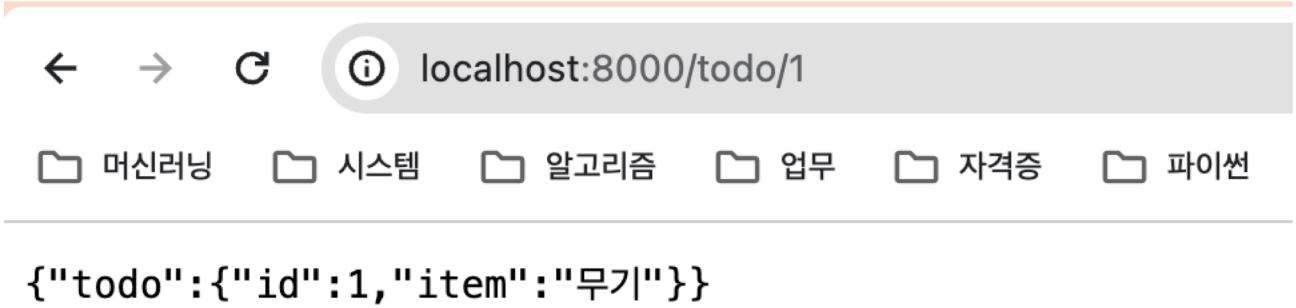
요청 처리

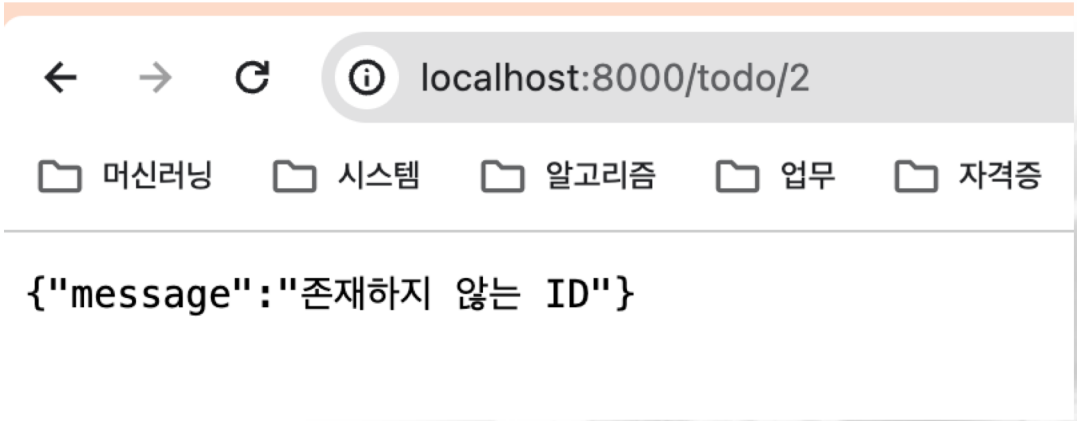
- ❖ Parameter
  - ✓ 경로 매개변수 적용
    - ● 확인



요청 처리

- ❖ Parameter
  - ✓ 경로 매개변수 적용
    - ● 확인





요청 처리

- Parameter
  - ✓ 경로 매개변수
    - Path
      - FastAPI가 제공하는 클래스로 Router 함수 안에 있는 다른 인수 와 경로 매개변수를 구분하는 역할
      - Swagger 와 ReDoc 등으로 OpenAPI 기반 문서를 자동 생성할 때 Router 관련 정보를 함께 문서화하도록 함
      - Path(..., kwargs)
        - 첫 번째 인수로 None 이나 ... 을 사용할 수 있는데 첫 번째 인수가 ... 이면 경로 매개변수가 필수

요청 처리

- Parameter
  - ✓ 경로 매개변수
    - todo.py 파일에서 상세보기 처리 메서드를 수정

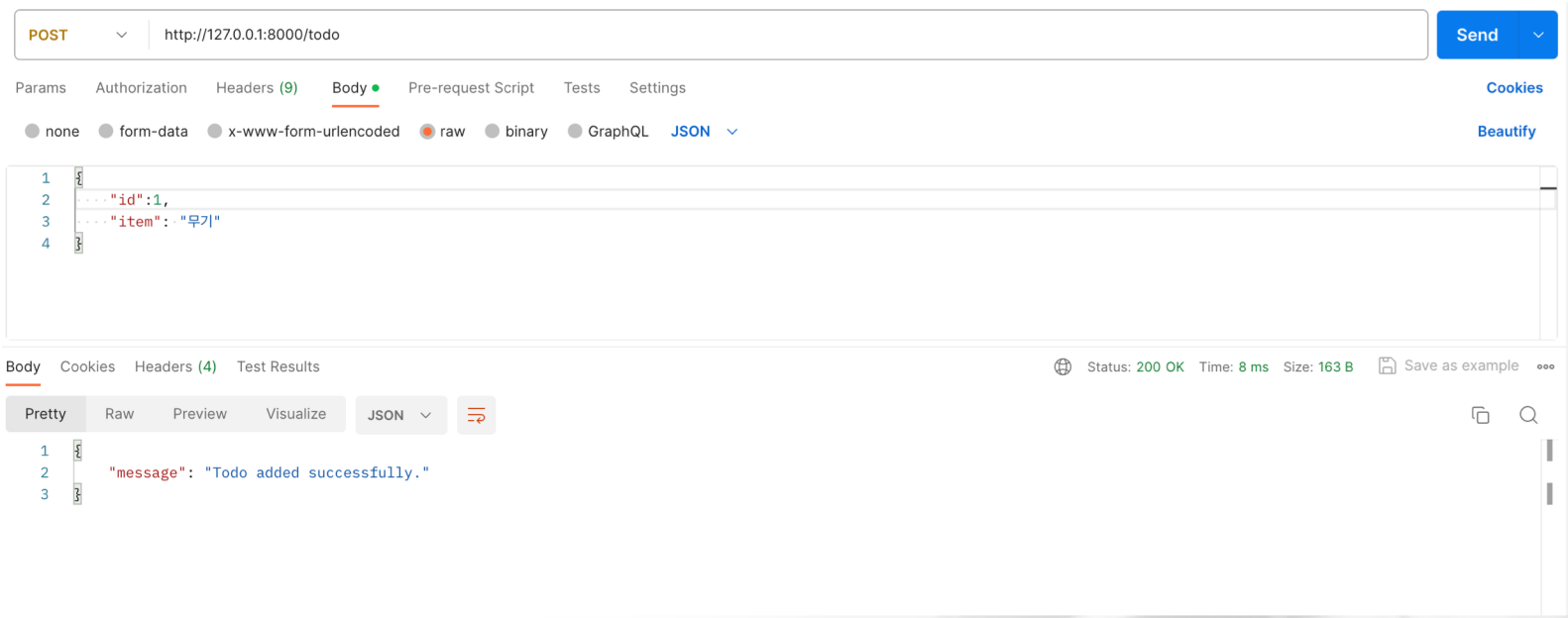
```
#id를 받아서 하나의 데이터를 리턴하는 요청을 처리
from fastapi import Path
@todo_router.get("/todo/{todo_id}")
async def get_single_todo(todo_id:int = Path(..., title="ID를 가지고 하나의 데이터를 찾아오는 요청")) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            return{
                "todo":todo
            }
    return{
        "message": "존재하지 않는 ID"
    }
```

요청 처리

- Parameter
  - ✓ 쿼리 매개변수
    - 쿼리는 URL에서 ? 다음에 나오고 & 으로 구분되는 키-값 쌍의 집합
    - http://127.0.0.1:8000/items?skip=0&limit=10 에서 쿼리 매개변수
      - skip: 0
      - limit: 10
    - URL의 일부이므로 문자열이지만 Python 타입과 함께 선언할 경우 해당 타입으로 변환되고 이에 대해 검증

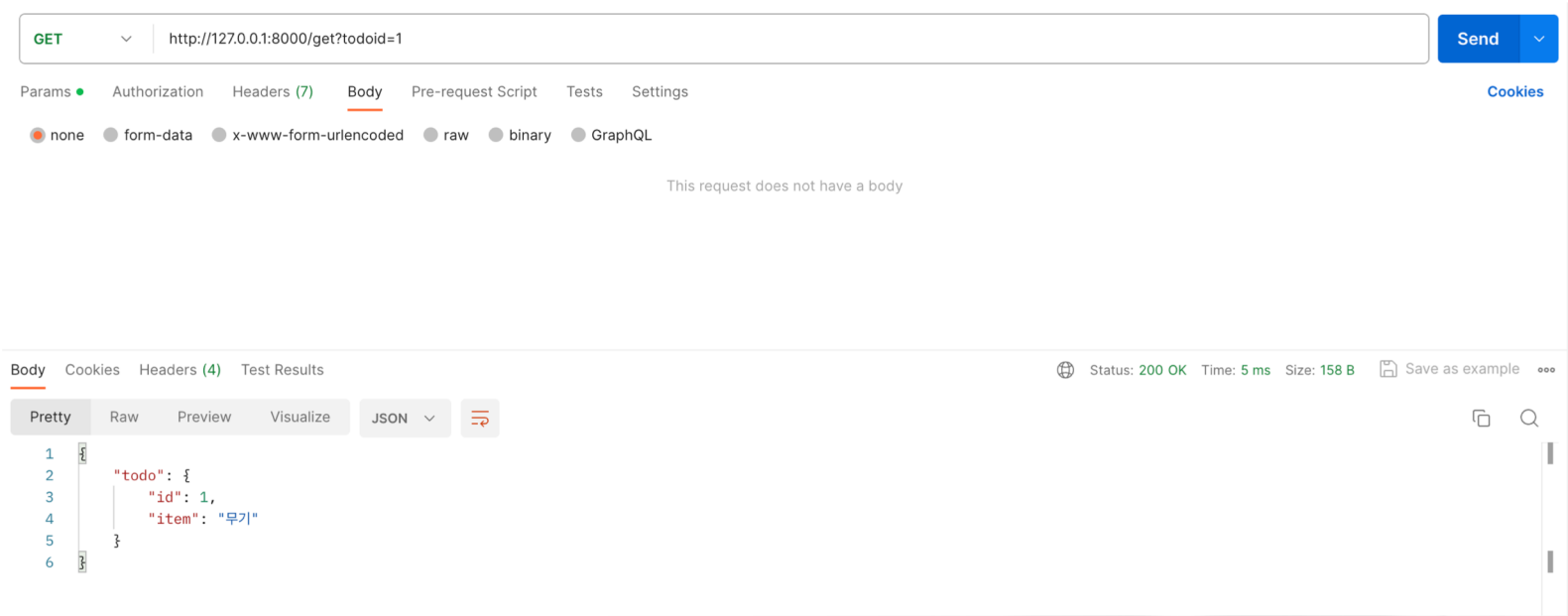
요청 처리

- ❖ Parameter
  - ✓ 쿼리 매개변수
    - ● 확인



요청 처리

- ❖ Parameter
  - ✓ 쿼리 매개변수
    - ● 확인



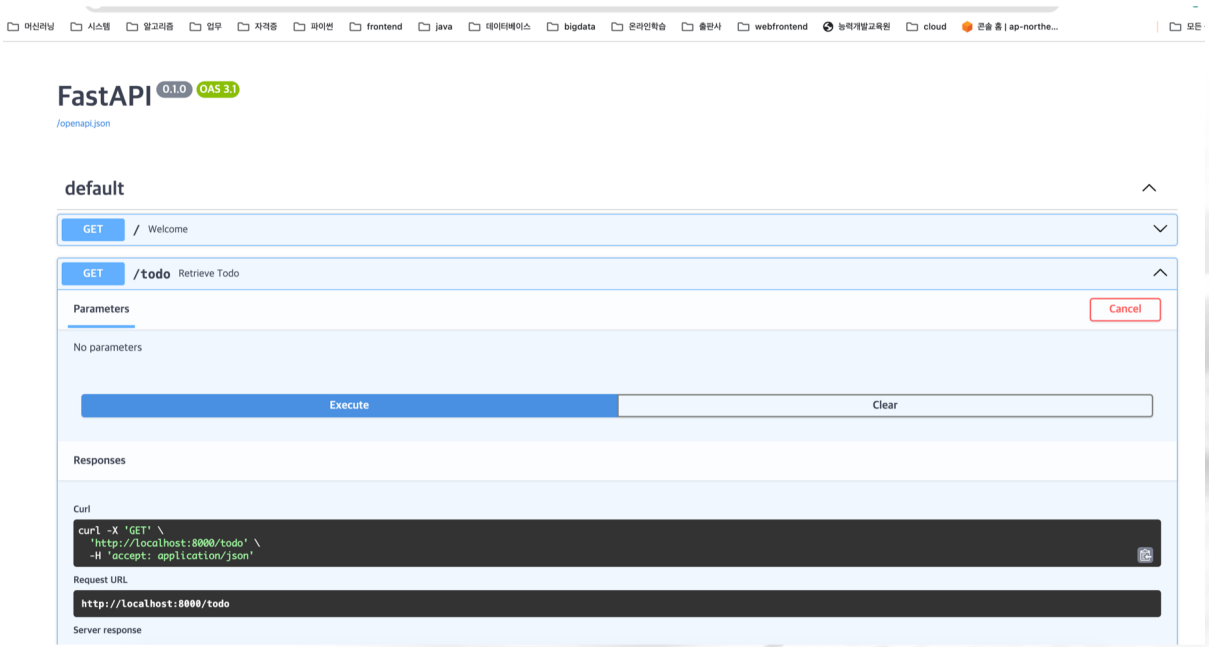
요청 처리

- Parameter
  - ✓ 쿼리 매개변수
    - todo.py 파일에서 상세보기 처리 메서드를 추가

```
@todo_router.get("/get")
async def get_todo(todoid:int = 0) -> dict:
    for todo in todo_list:
        if todo.id == todoid:
            return{
                "todo":todo
            }
    return{
        "message": "존재하지 않는 ID"
    }
```

요청 처리

- 자동 문서화
  - ✓ FastAPI는 모델의 JSON 스키마 정의를 생성하고 라우트, 요청 바디의 유형, 경로 및 쿼리 매개변수, 응답 모델 등으로 자동으로 문서화
  - ✓ Swagger
    - http://127.0.0.1:8000/docs 로 접속
    - 사용자가 실행할 수 있는 문서를 제공하여 API를 테스트 할 수 있도록 도와줌



요청 처리

- ❖ 자동 문서화
  - ✓ ReDoc
    - 127.0.0.1/redoc 에 접속
    - 데이터를 샘플의 형태를 제공하고자 할 때는 모델 클래스 안에 Config 클래스로 정의
    - Todo 클래스 수정

```
from pydantic import BaseModel

class Todo(BaseModel):
    #필수
    id: int
    #선택
    item: str | None=None
    model_config = {
        "json_schema_extra" : {
            "example":{
                "id":1,
                "item": "Sample Item"
            }
        }
    }
```

요청 처리

- 데이터 수정 처리
  - ✓ 데이터 수정을 위한 모델을 models.py에 추가

```
class TodoItem(BaseModel):
    item: str

    model_config = {
        "json_schema_extra" : {
            "example":{
                "item": "Update Item"
            }
        }
    }
```

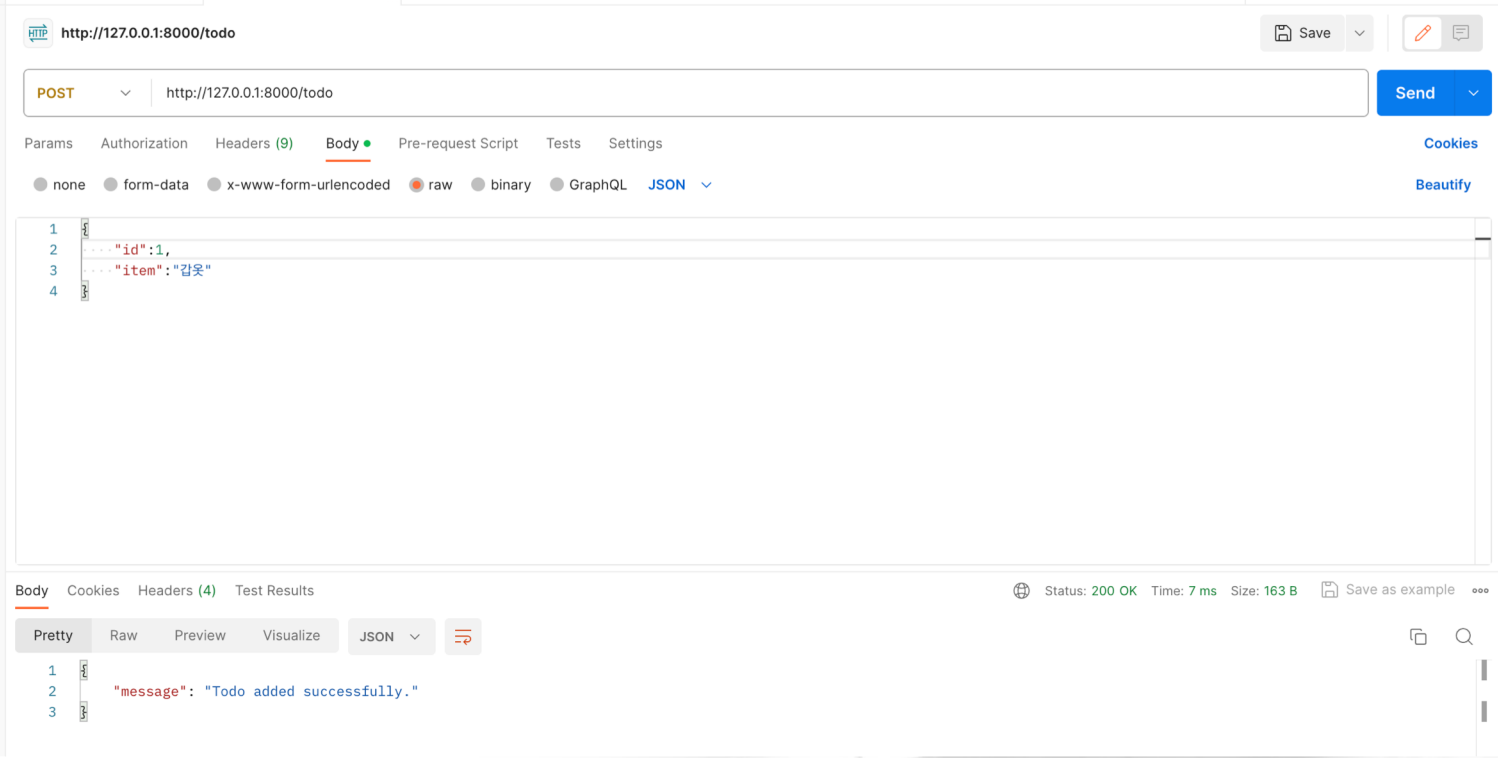
요청 처리

- 데이터 수정 처리
  - ✓ 데이터 수정을 처리하기 위한 함수를 todo.py에 추가

```
from models import TodoItem
@todo_router.put("/todo/{todo_id}")
async def update_todo(todo_data: TodoItem, todo_id:int = Path(..., title="ID에 해당하는 데이터 수정")) -> dict:
    for todo in todo_list:
        if todo.id == todo.id:
            todo.item = todo_data.item
            return{
                "message":"수정 성공"
            }
    return{"message": "수정 실패"}
```

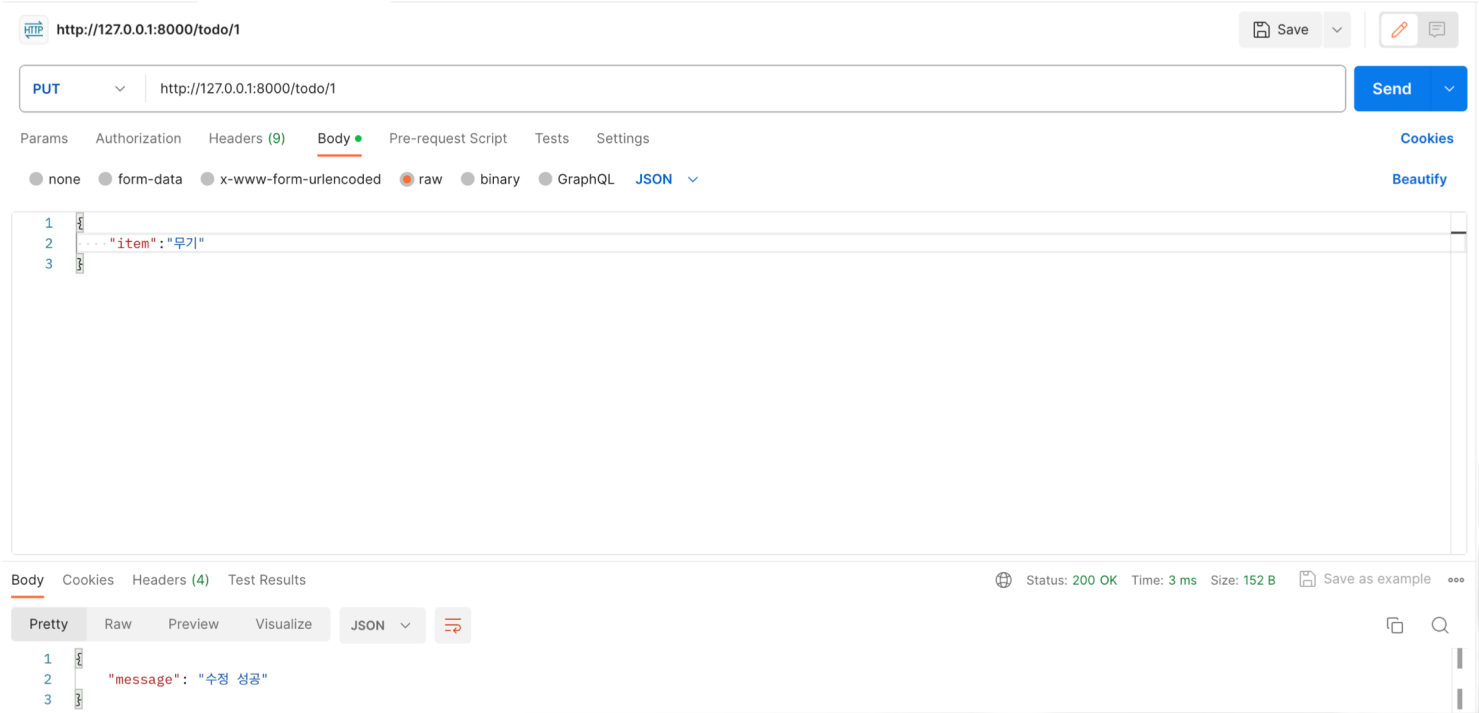
요청 처리

- 데이터 수정 처리
  - ✓ 확인 – 한 개의 데이터를 추가하고 수정



요청 처리

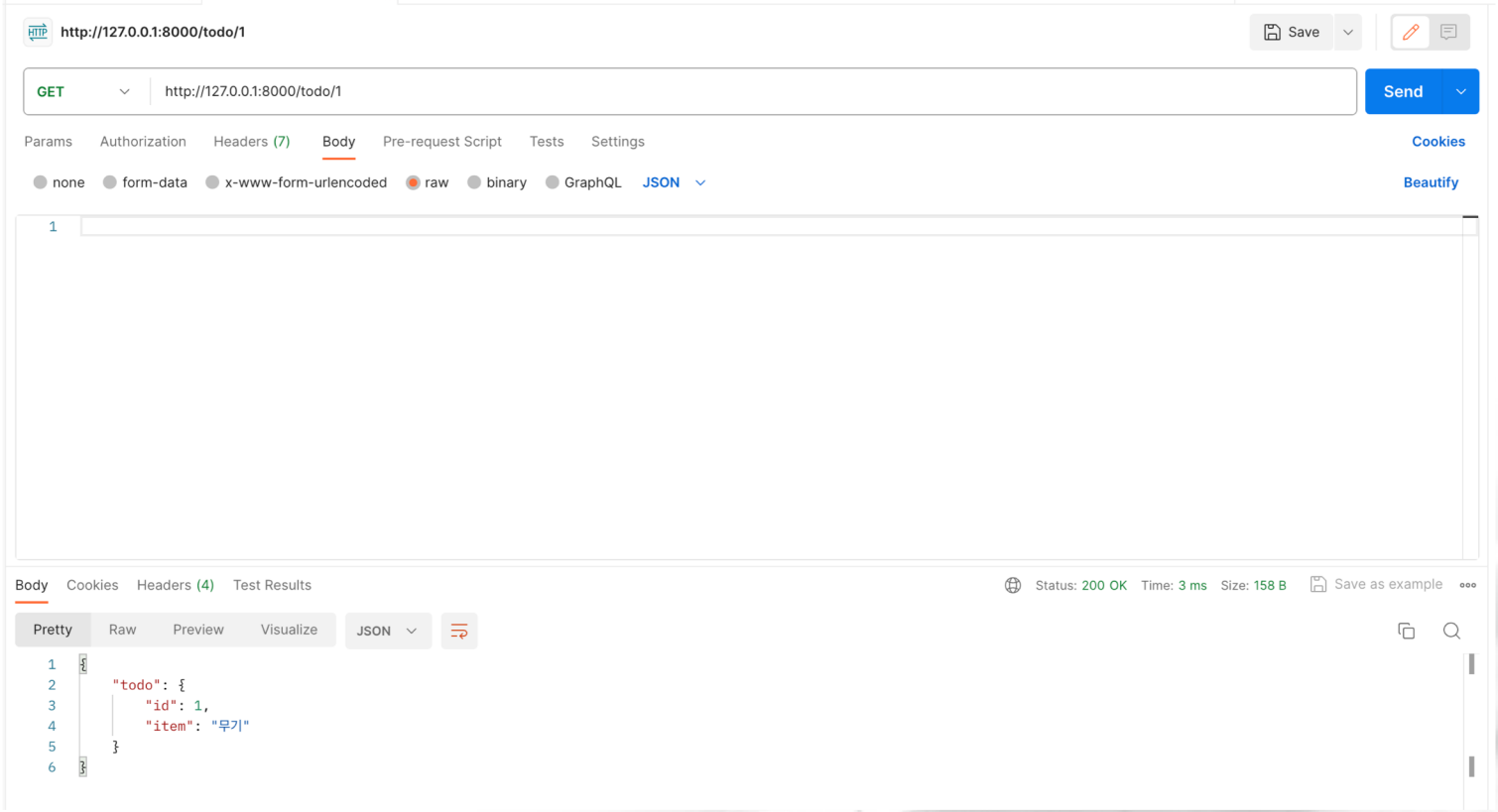
- 데이터 수정 처리
  - ✓ 확인 – 한 개의 데이터를 추가하고 수정



요청 처리

❖ 데이터 수정 처리

✓ 확인 – 한 개의 데이터를 추가하고 수정



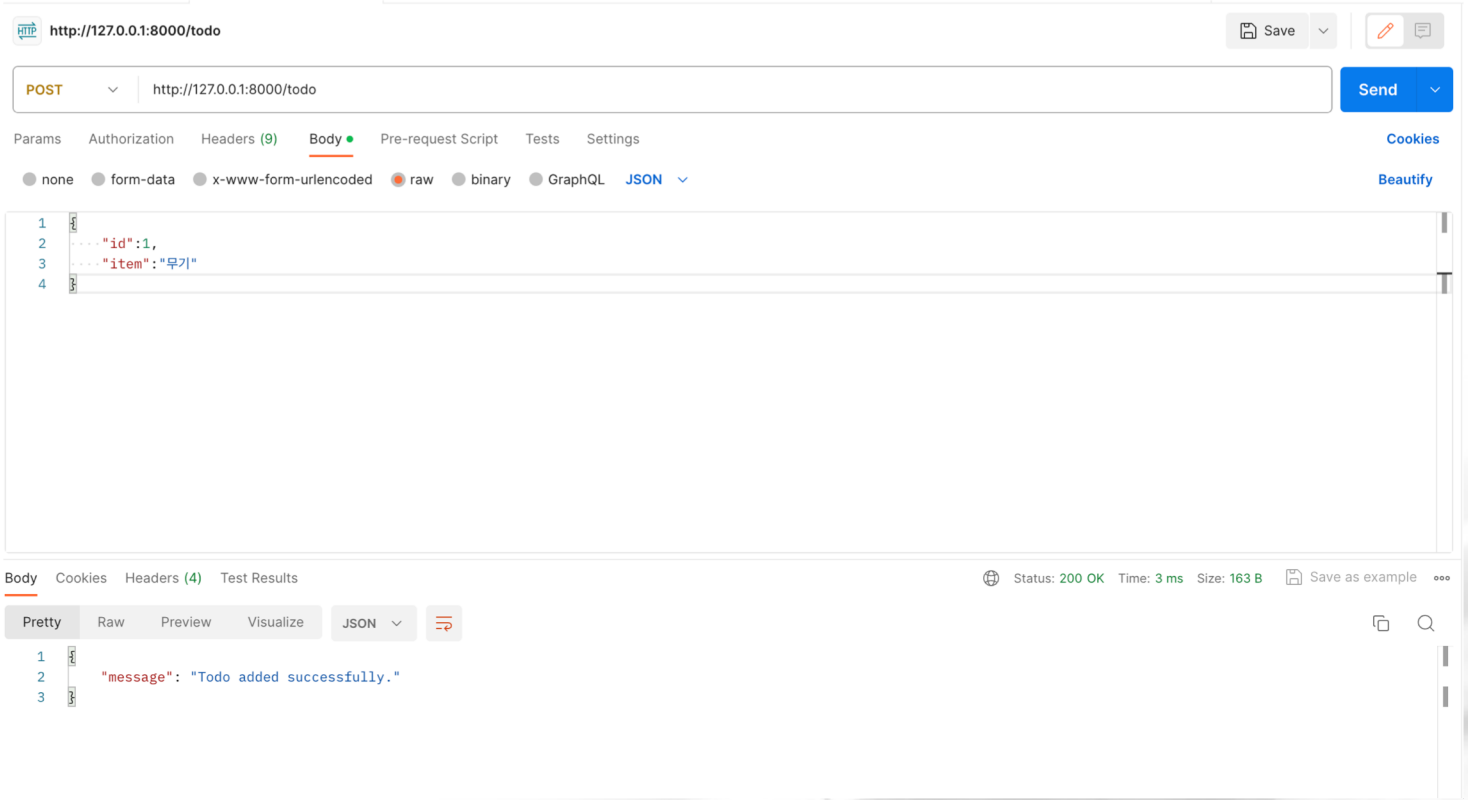
요청 처리

- 데이터 삭제 처리
  - ✓ todo.py 파일에 id를 받아서 데이터를 삭제하는 요청을 처리하는 함수를 추가

```
@todo_router.delete("/todo/{todo_id}")
async def delete_todo(todo_id:int) -> dict:
    for index in range(len(todo_list)):
        todo = todo_list[index]
        if todo.id == todo.id:
            todo_list.pop(index)
            return{
                "message":"삭제 성공"
            }
    return{"message": "삭제 실패"}
```

요청 처리

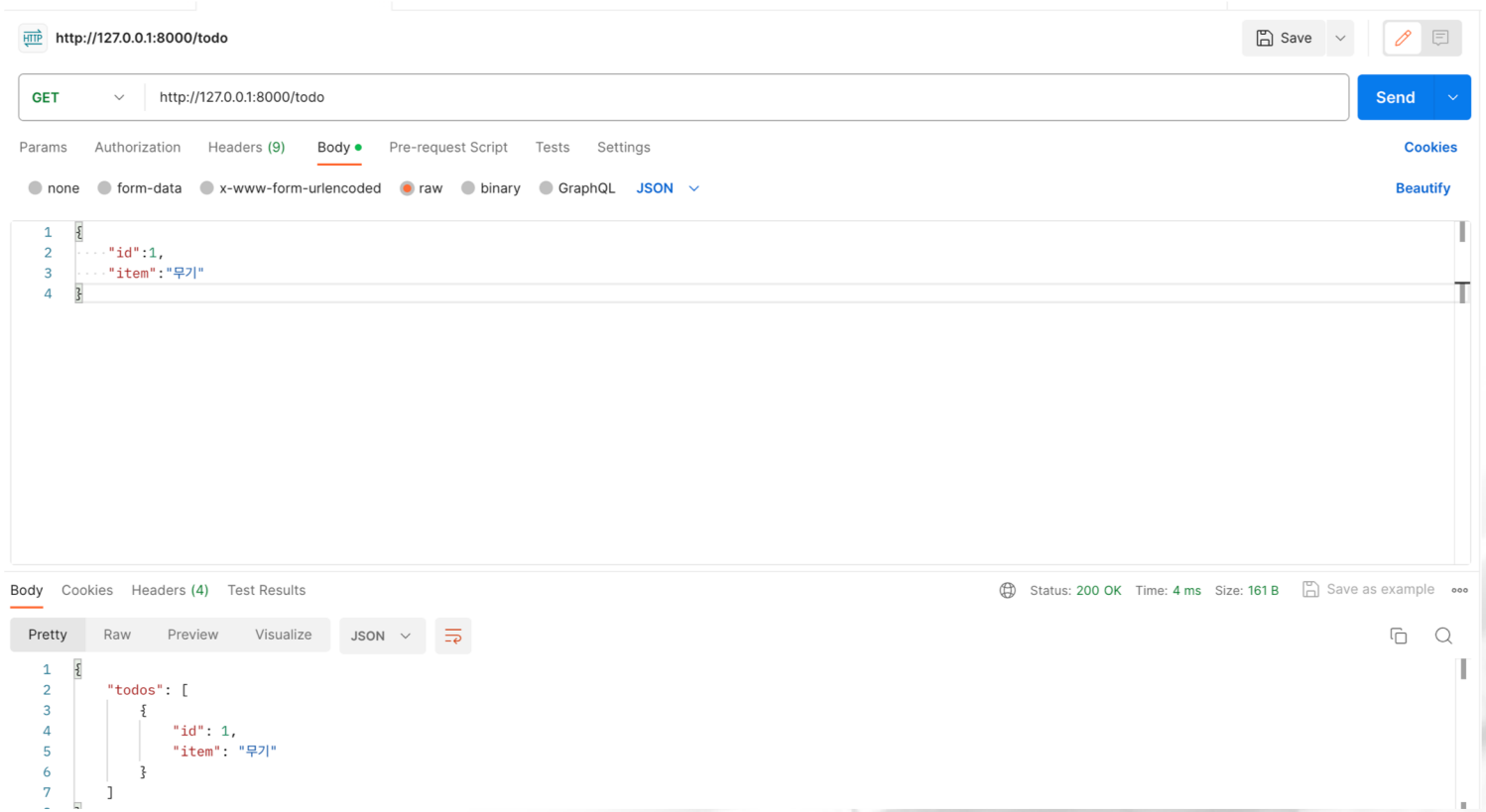
- 데이터 삭제 처리
  - ✓ 확인 – 데이터 추가



요청 처리

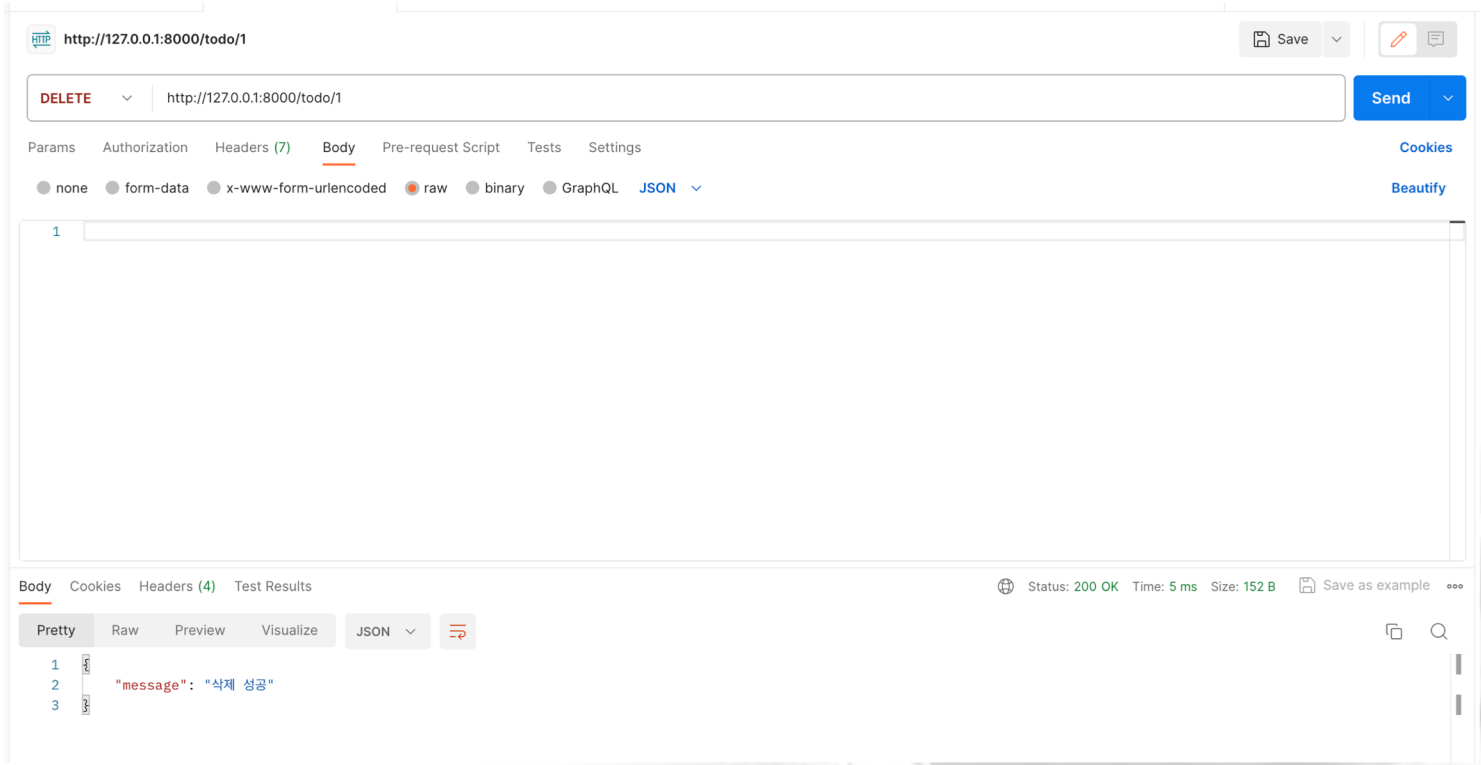
❖ 데이터 삭제 처리

✓ 확인 – 데이터 확인



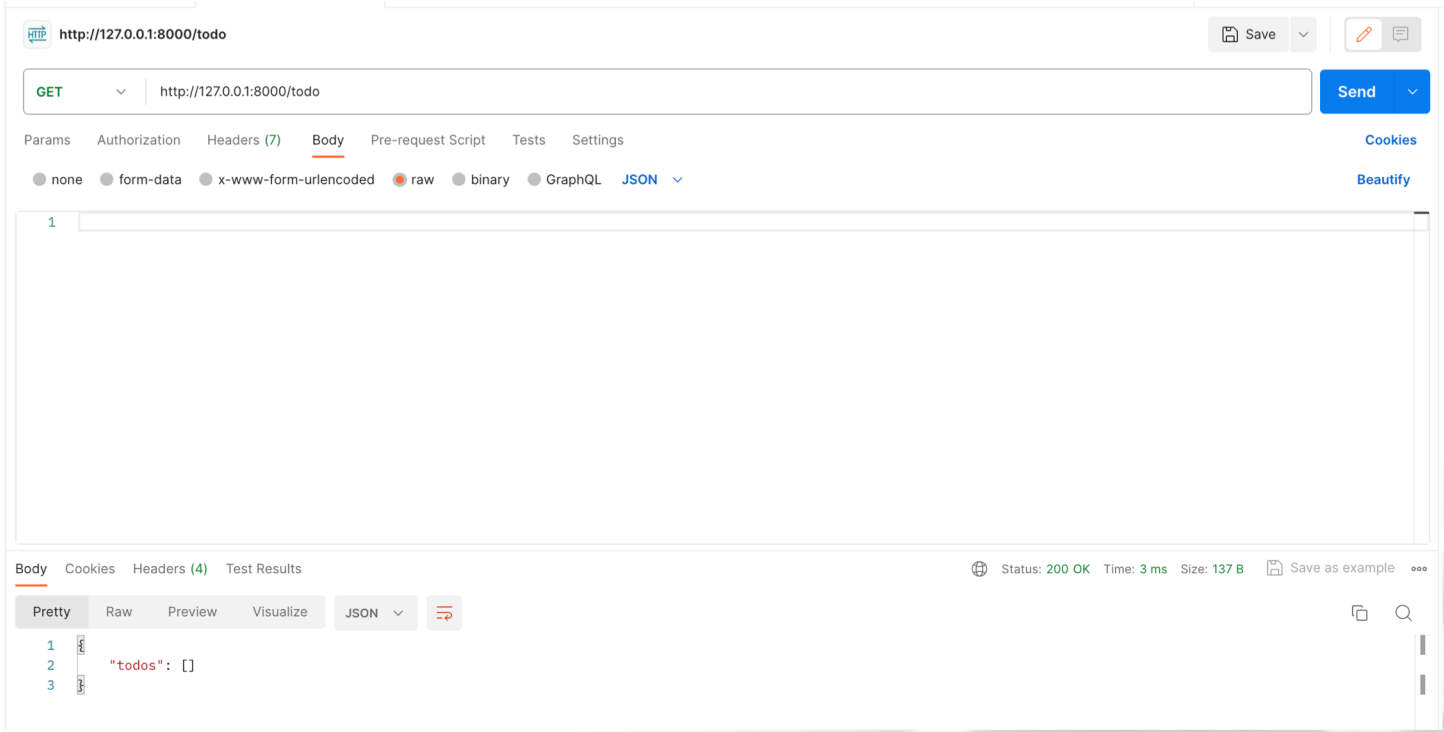
요청 처리

- 데이터 삭제 처리
  - ✓ 확인 – 데이터 삭제



요청 처리

- 데이터 삭제 처리
  - ✓ 확인 – 데이터 확인



요청 처리

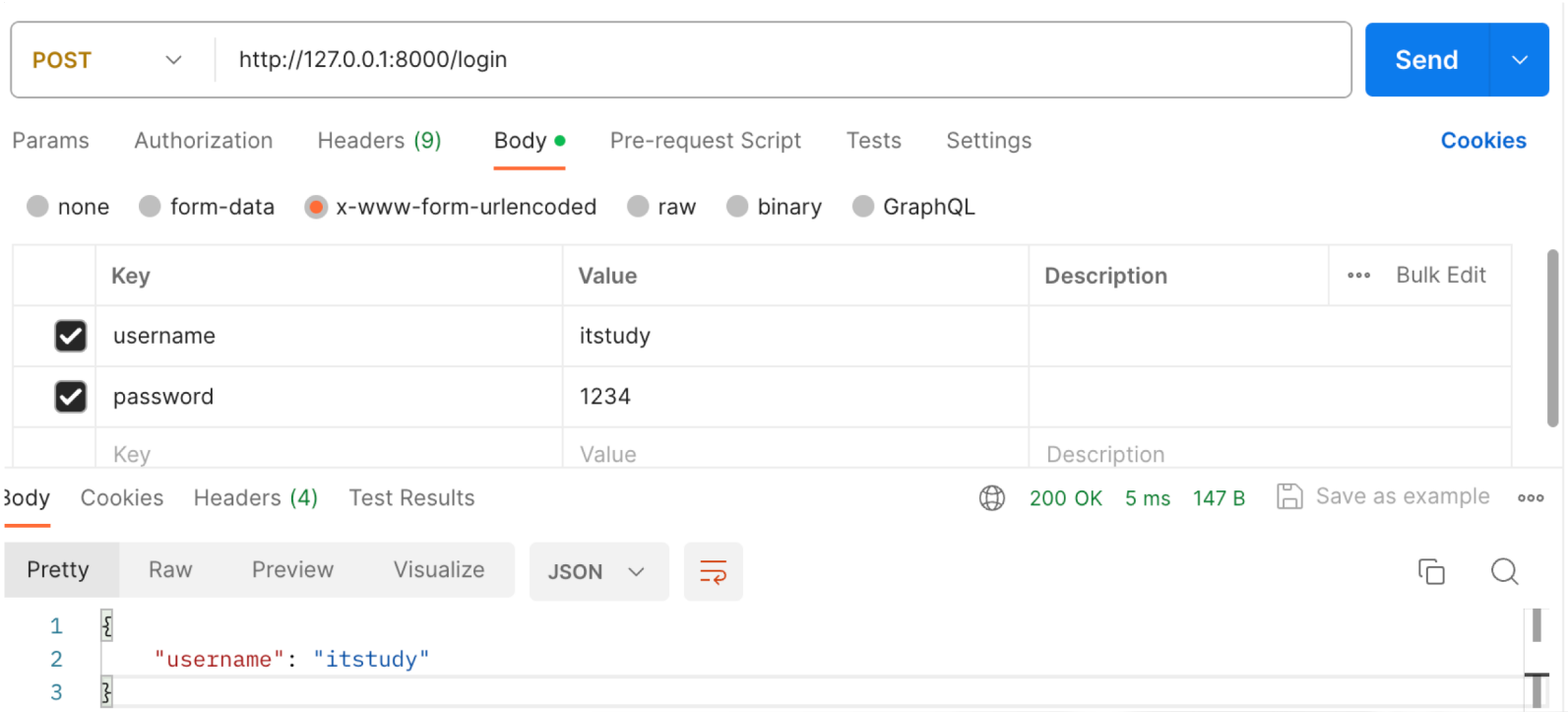
- 폼 데이터 처리
  - ✓ 패키지 설치: pip install python-multipart
  - ✓ Form에서 입력받아 전송한 데이터를 처리하기 위한 함수를 todo.py에 추가

```
from fastapi import Form
from typing import Annotated

@todo_router.post("/login/")
async def login(username: Annotated[str, Form()], password: Annotated[str, Form()]):
    print(username)
    print(password)
    return {"username": username}
```

요청 처리

❖ 폼 데이터 처리

✓ 확인



요청 처리

- 파일 처리
  - ✓ Body 및 Form 과 동일한 방식으로 파일의 매개변수를 생성
  - ✓ 파일들은 폼 데이터의 형태로 업로드
  - ✓ 처리 방법
    - 경로 작동 함수의 매개변수를 bytes 로 선언
      - FastAPI는 파일을 읽고 bytes 형태의 내용을 전달하는 것이 가능
      - 전체 내용이 메모리에 저장된다는 것을 의미
      - 작은 크기의 파일들에 적합
    - UploadFile 을 사용
      - 스풀 파일을 사용
      - 최대 크기 제한까지만 메모리에 저장되며 이를 초과하는 경우 디스크에 저장
      - 이미지, 동영상, 큰 이진 코드와 같은 대용량 파일들을 많은 메모리를 소모하지 않고 처리하기에 적합
      - 업로드 된 파일의 메타데이터를 얻을 수 있음
      - file-like async 인터페이스를 갖고 있음
      - file-like object를 필요로 하는 다른 라이브러리에 직접적으로 전달할 수 있는 파이썬 SpooledTemporaryFile 객체를 반환

요청 처리

- ❖ 파일 처리
  - ✓ UploadFile
    - 속성
      - filename: 문자열(str)로 된 업로드 된 파일의 파일 이름
      - content_type: 문자열(str)로 된 파일 형식(MIME type / media type)
      - file: SpooledTemporaryFile(파일형태의 객체)로 다른 라이브러리에 직접적으로 전달할 수 있는 실질적인 파이썬 파일
    - async 메서드: 내부적인 SpooledTemporaryFile 을 사용하여 해당하는 파일 메서드를 호출하는데 상기 모든 메소드들이 async 메소드이기 때문에 await를 사용해야 함
      - ◆ write(data): data(str 또는 bytes)를 파일에 작성
      - ◆ read(size): 파일의 바이트 및 글자의 size(int) 만큼 읽음
      - ◆ seek(offset): 파일 내 offset(int) 위치의 바이트로 이동
      - ◆ close(): 파일을 닫음
    - async 경로 작동 함수의 내부에서 다음과 같은 방식으로 내용을 가져올 수 있음

      contents = await myfile.read()
    - 일반적인 def 경로 작동 함수의 경우

      contents = myfile.file.read()

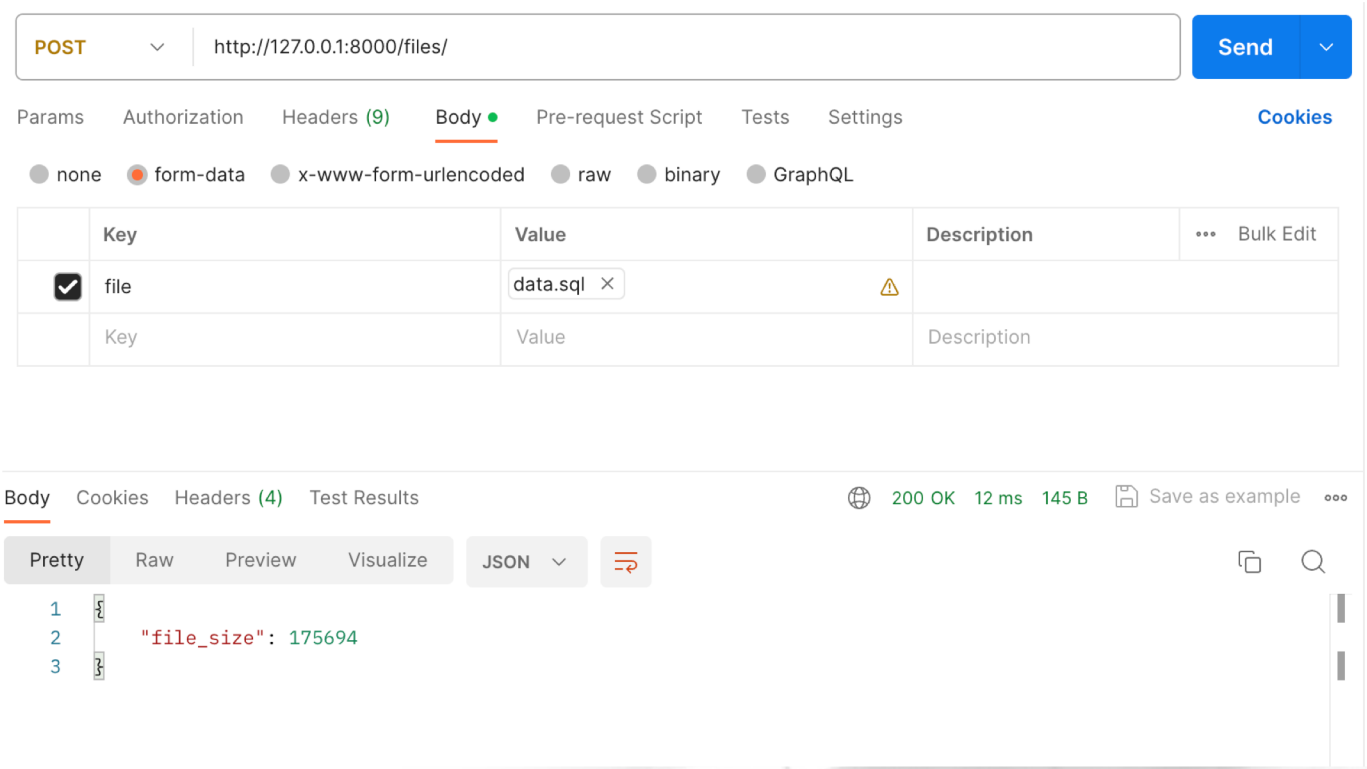
요청 처리

- ❖ 파일 처리
  - ✓ 파일 업로드 처리
    - todo.py 파일에 처리 메서드 추가

```
from fastapi import File, UploadFile

@todo_router.post("/files/")
async def create_file(file: bytes = File()):
    return {"file_size": len(file)}

@todo_router.post("/uploadfile/")
async def create_upload_file(file: UploadFile):
    return {"filename": file.filename}

from typing import List
@todo_router.post("/uploadfiles/")
async def create_upload_files(files: List[UploadFile]):
    return {"filenames": [file.filename for file in files]}
```

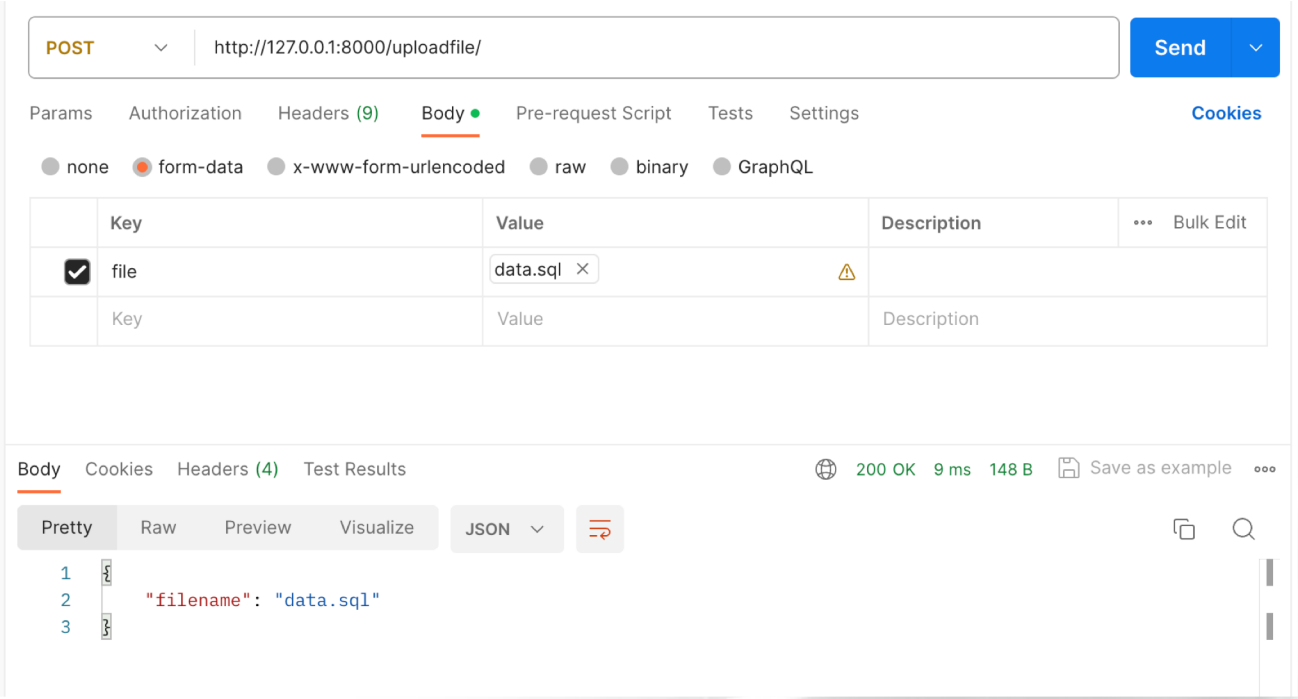
요청 처리

- ❖ 파일 처리
  - ✓ 파일 업로드 처리
    - ● 확인



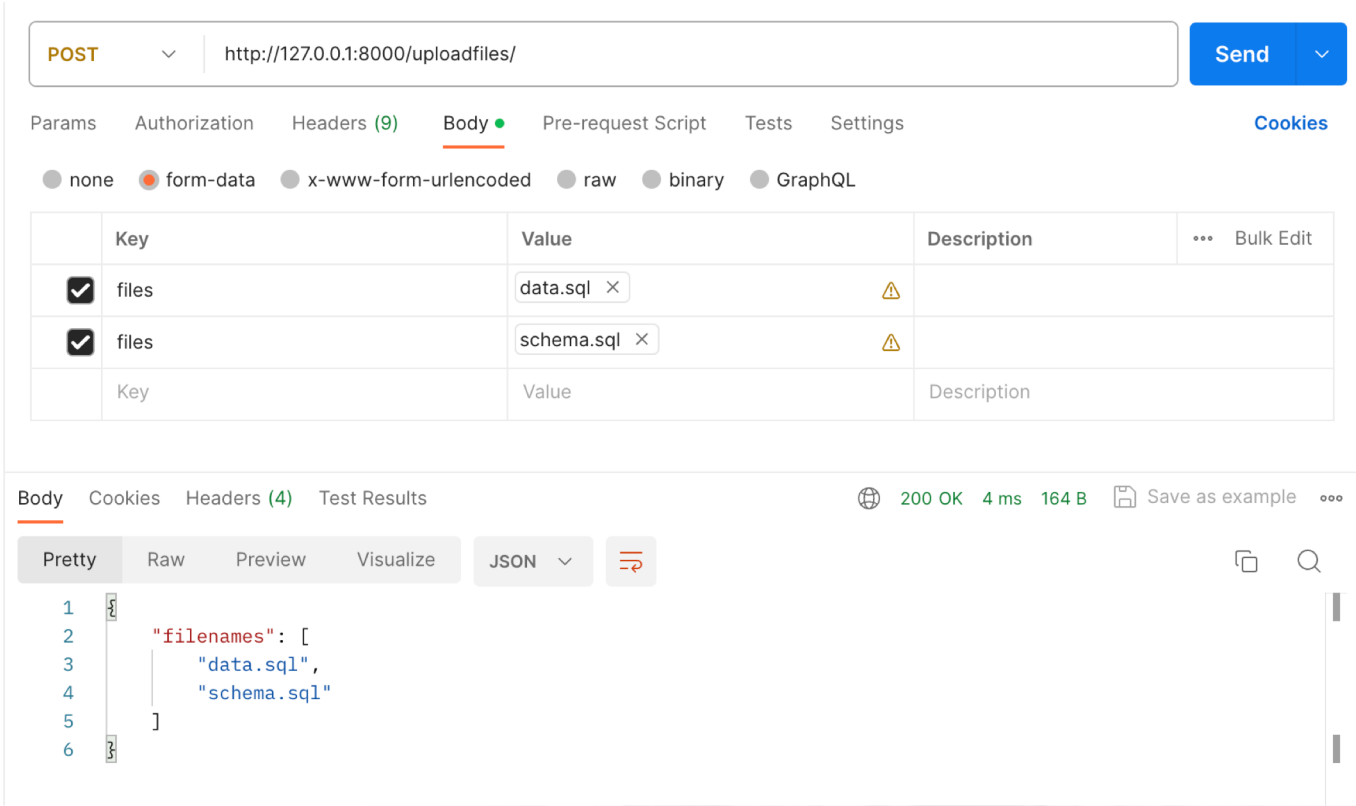
요청 처리

- ❖ 파일 처리
  - ✓ 파일 업로드 처리
    - ● 확인



요청 처리

- ❖ 파일 처리
  - ✓ 파일 업로드 처리
    - ● 확인



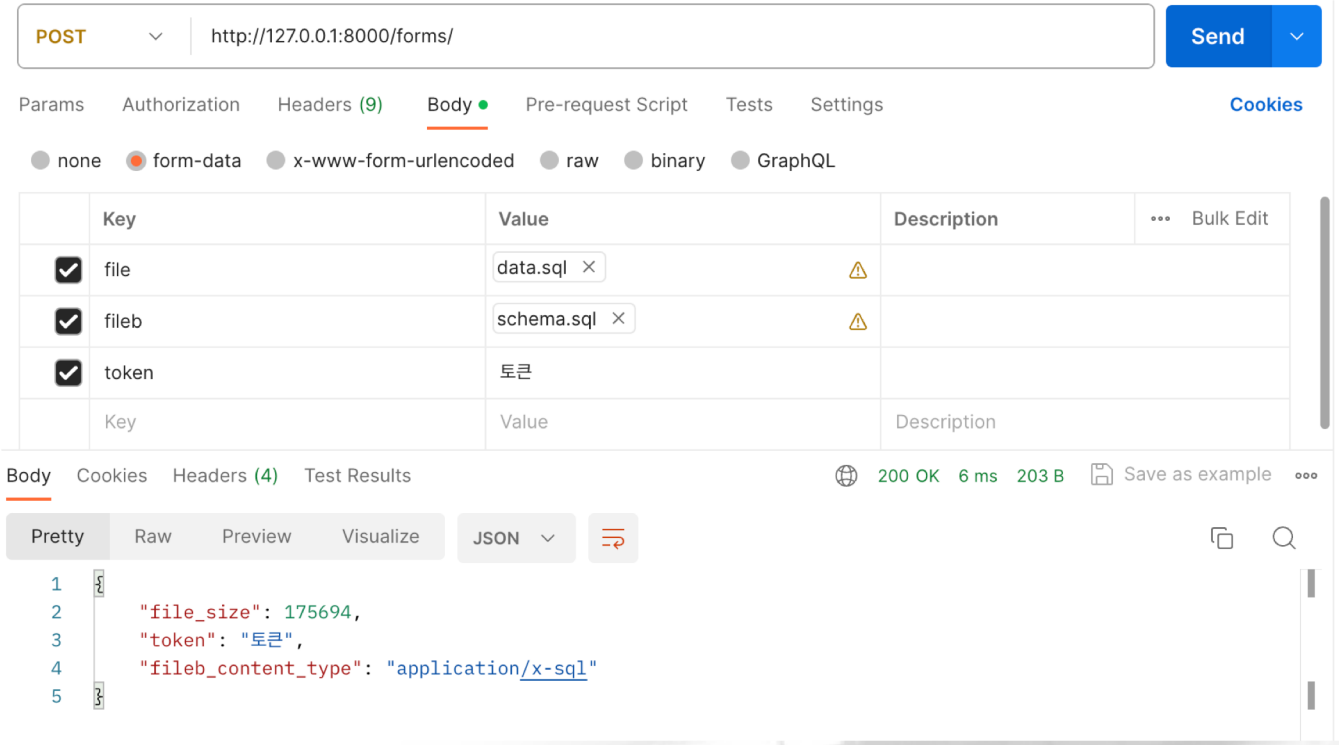
요청 처리

- 파일 처리
  - ✓ 폼 과 파일 처리
    - todo.py 파일에 처리 메서드 추가

```
@todo_router.post("/forms/")
async def create_file(
    file: bytes = File(), fileb: UploadFile = File(), token: str = Form()):
    return {
        "file_size": len(file),
        "token": token,
        "fileb_content_type": fileb.content_type,
    }
```

요청 처리

- ❖ 파일 처리
  - ✓ 폼 과 파일 처리
    - ● 확인



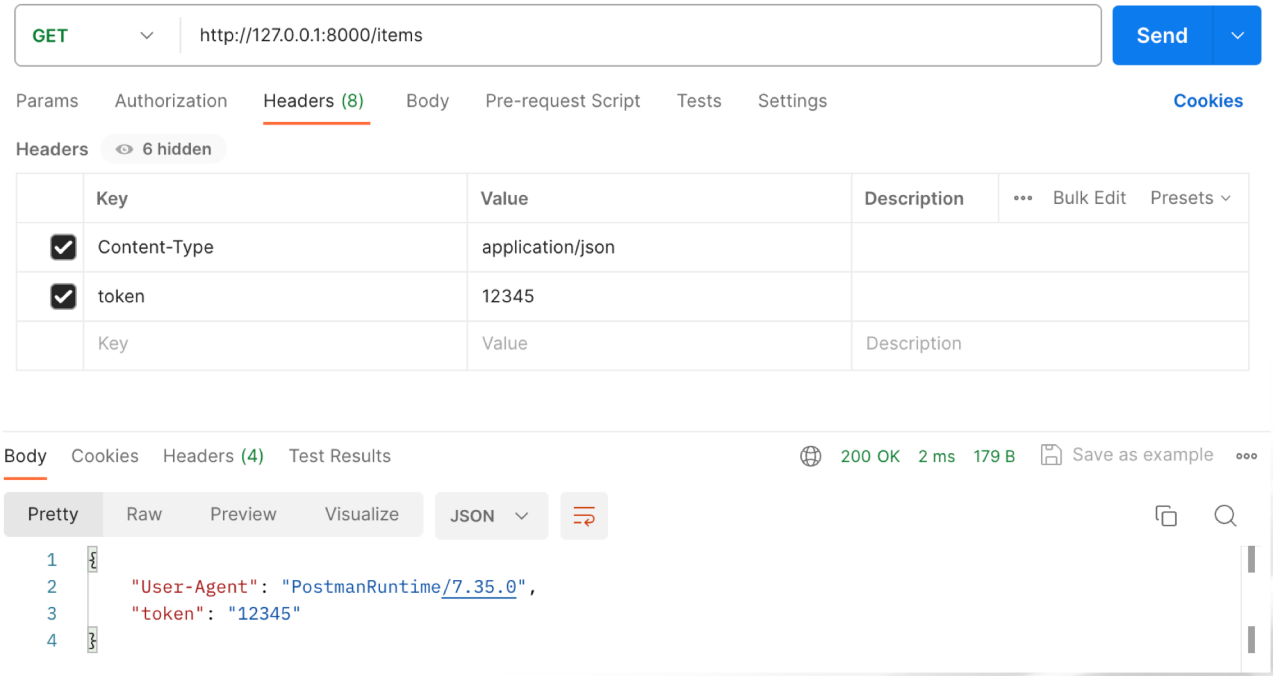
요청 처리

- 헤더 처리
  - ✓ 헤더 처리를 함수를 todo.py 파일에 추가

```
from typing import Union
from fastapi import Header

@todo_router.get("/items/")
async def read_items(user_agent: Union[str, None] = Header(default=None),
                     token: Union[str, None] = Header(default=None)):
    return {"User-Agent": user_agent, "token": token}
```

요청 처리

- ❖ 헤더 처리
  - ✓ 확인


GET
http://127.0.0.1:8000/items
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
Headers
6 hidden
Key
Value
Description
Bulk Edit
Presets
Content-Type
application/json
token
12345
Key
Value
Description
Body
Cookies
Headers (4)
Test Results
200 OK
2 ms
179 B
Save as example
Pretty
Raw
Preview
Visualize
JSON
{
"User-Agent": "PostmanRuntime/7.35.0",
"token": "12345"
}

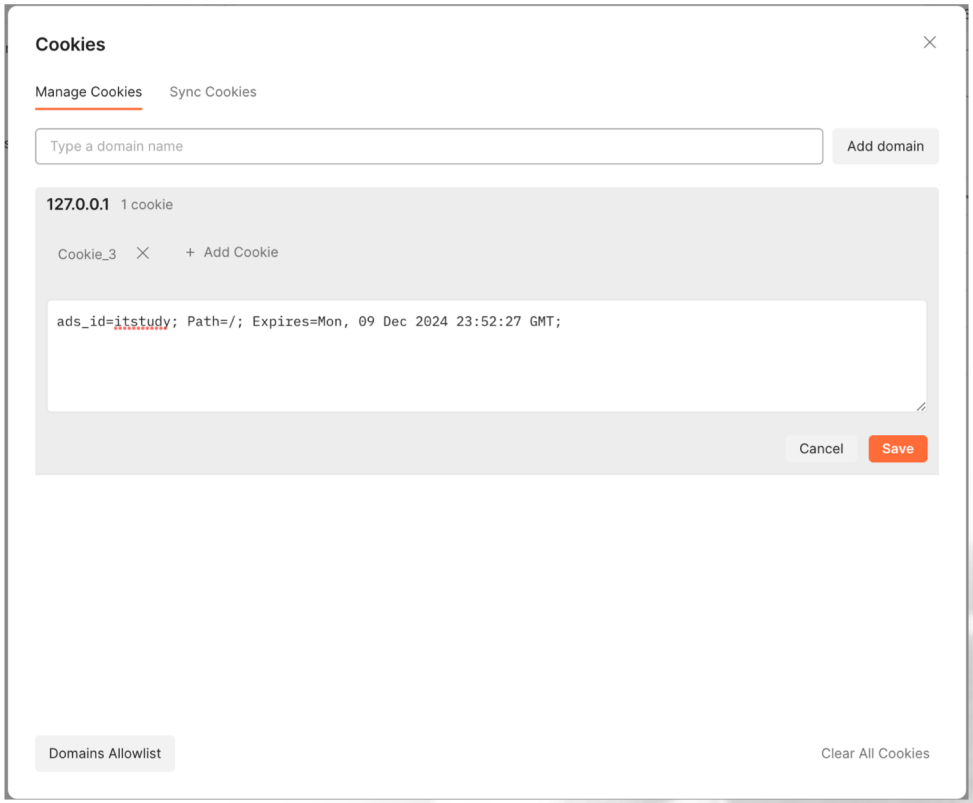
요청 처리

- 쿠키 처리
  - ✓ 쿠키 처리를 함수를 todo.py 파일에 추가

```
from fastapi import Cookie

@todo_router.get("/headers/")
async def read_headers(ads_id: Union[str, None] = Cookie(default = None)):
    return {"ads_id": ads_id}
```

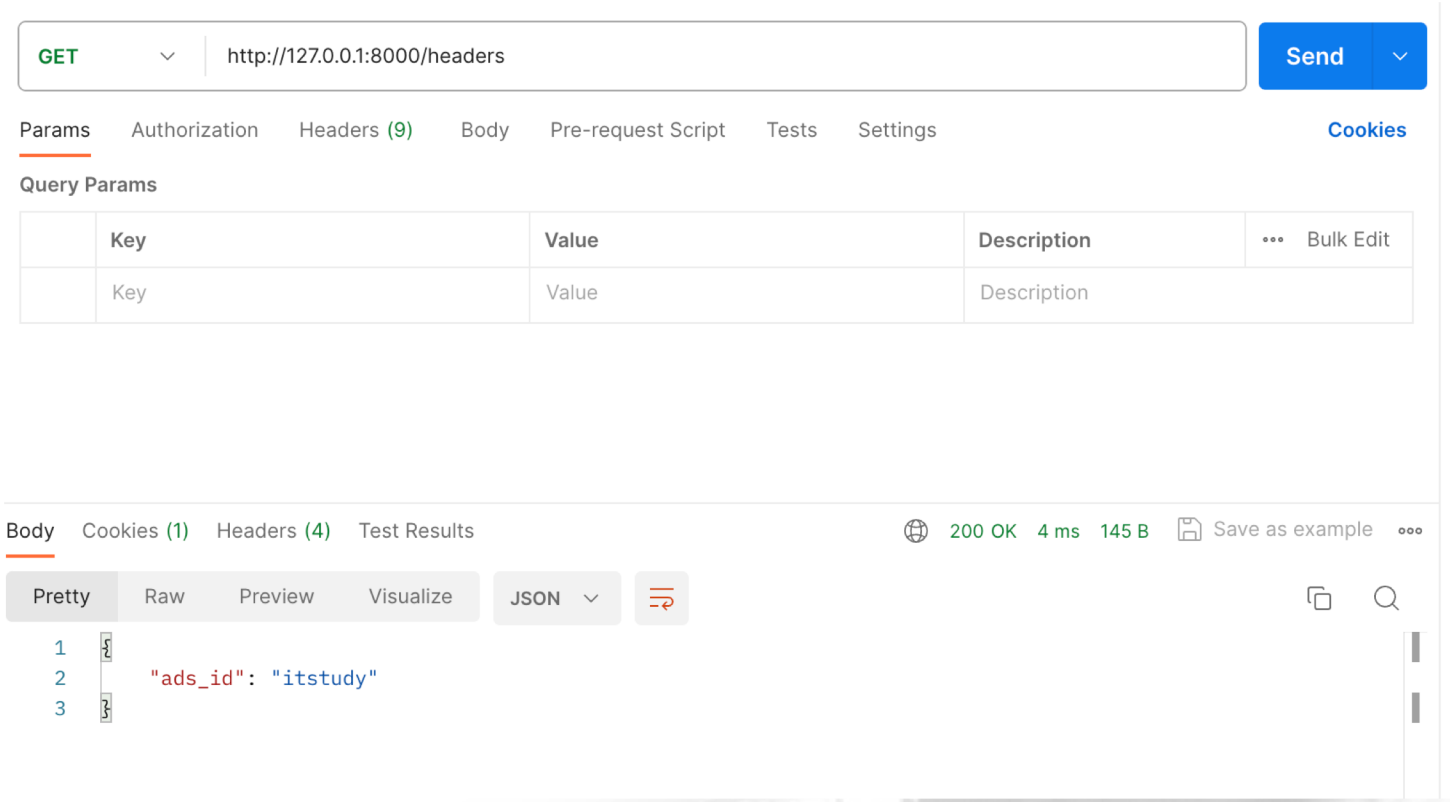
요청 처리

- ❖ 쿠키 처리
  - ✓ 확인


Cookies
Manage Cookies
Sync Cookies
Type a domain name
Add domain
127.0.0.1 1 cookie
Cookie_3
+ Add Cookie
ads_id=itstudy; Path=/; Expires=Mon, 09 Dec 2024 23:52:27 GMT;
Cancel
Save
Domains Allowlist
Clear All Cookies

요청 처리

- ❖ 쿠키 처리
  - ✓ 확인


GET
http://127.0.0.1:8000/headers
Send
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Key
Value
Description
Body
Cookies (1)
Headers (4)
Test Results
200 OK
4 ms
145 B
Save as example
Pretty
Raw
Preview
Visualize
JSON
1 {
2 "ads_id": "itstudy"
3 }

요청 처리

- Response
  - ✓ Response는 API 처리 과정의 한 부분으로 HTTP 메서드를 통해 API와 상호 작용하며 API로부터 받은 결과를 의미
  - ✓ API 응답은 보통 JSON 또는 XML 형식이지만 문서 형식으로 전달되기도 하며 header 와 Body로 구성
  - ✓ Response Header
    - 응답 헤더는 요청 상태 및 응답 바디 전달을 안내하는 정보로 구성
    - 응답 헤더의 대표적인 것으로는 Content-Type이 있으며 반환하는 콘텐츠 유형이 무엇인지 클라이언트에게 알려주는 역할
  - ✓ Response Body
    - 응답 바디는 서버가 클라이언트에게 반환하는 데이터
    - 응답 바디의 형식은 Content-Type 헤더에 의해 결정되며 대표적인 예로 application/json 이 있음

요청 처리

- Response
  - ✓ Status Code
    - 서버가 반환한 응답에 포함되는 짧은 고유 코드로 클라이언트가 보낸 요청의 상태를 의미
    - 상태 코드는 크게 다섯 개의 그룹으로 분류할 수 있으며 각 그룹은 다음과 같은 상태를 의미
      - 1XX: 요청을 받음
      - 2XX: 요청을 성공적으로 처리
      - 3XX: 요청을 리다이렉트
      - 4XX: 클라이언트 측에 오류 발생
      - 5XX: 서버 측에 오류 발생

요청 처리

- Response
  - ✓ 응답 모델
    - 데이터 중에서 특정한 형태의 데이터만 반환하게 하고자 하는 경우 Fast API에서는 원하는 필드만 반환하도록 모델을 생성할 수 있음
    - Router 함수의 매개변수로 response_model에 정의만 추가하면 됨

요청 처리

- Response
  - ✓ 응답 모델 수정
    - model.py 파일에 모델 클래스 추가

```
from typing import List
class TodoItems(BaseModel):
    todos: List[TodoItem]

    model_config = {
        "json_schema_extra" : {
            "example":[{
                "item": "Item1!"
            }, {
                "item": "Item2!"
            }]
        }
    }
```

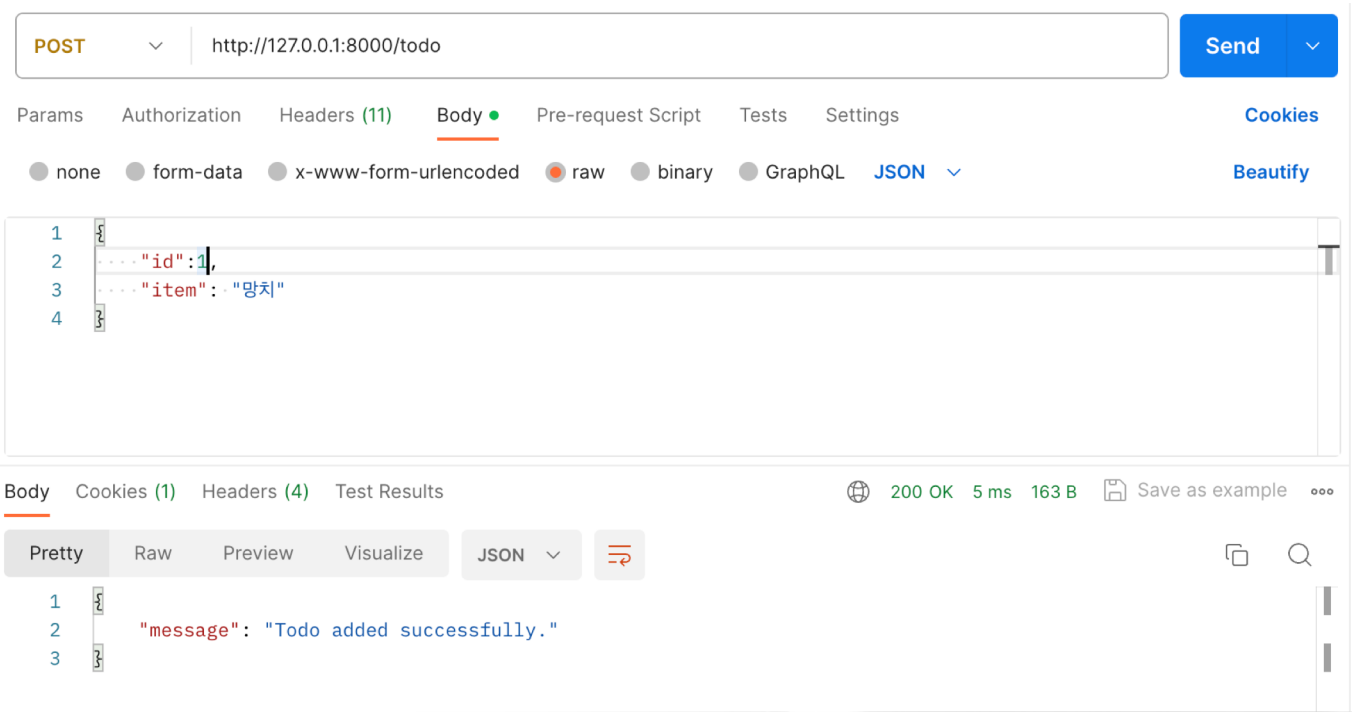
요청 처리

- Response
  - ✓ 응답 모델 수정
    - todo.py 파일에서 todo 의 GET 요청을 처리하는 함수를 수정

```
from model import TodoItems
#todo 요청을 get 방식으로 요청한 경우 처리
@todo_router.get("/todo", response_model=TodoItems)
async def retrieve_todo() -> dict:
    return {
        "todos": todo_list
    }
```

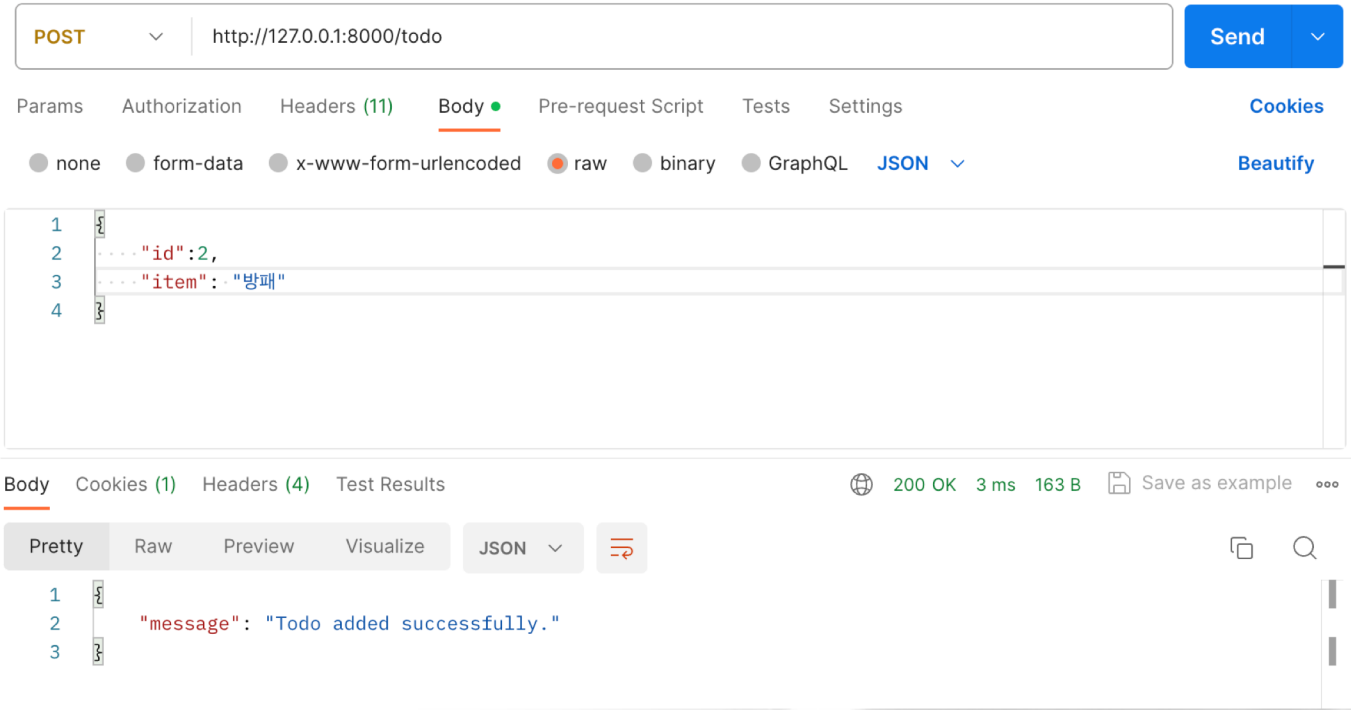
요청 처리

- ❖ Response
  - ✓ 응답 모델 수정
    - ● 확인



요청 처리

- ❖ Response
  - ✓ 응답 모델 수정
    - ● 확인

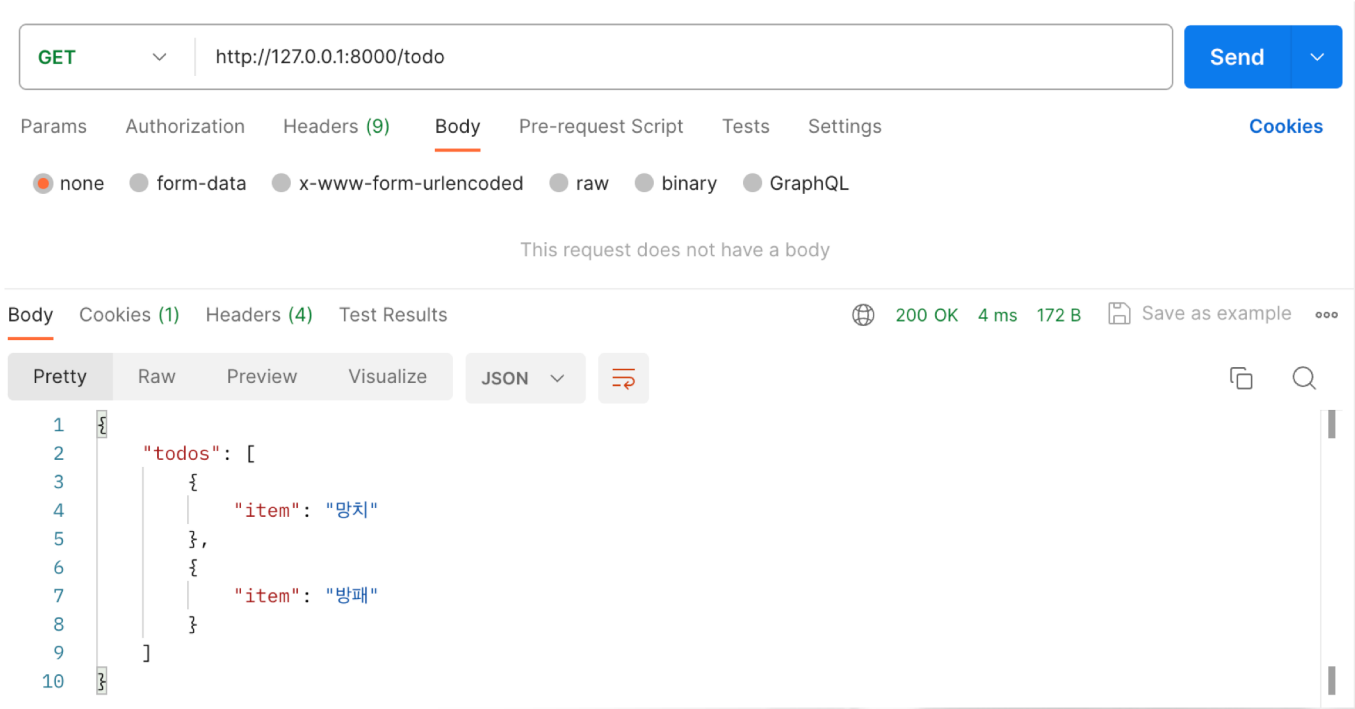


요청 처리

❖ Response

✓ 응답 모델 수정

● 확인



요청 처리

- Error Handling
  - ✓ 오류는 HTTPException 클래스를 사용해서 처리
  - ✓ HTTPException 클래스의 인스턴스를 생성할 때 넘겨주는 매개변수
    - status_code: 예외 처리 시 반환할 상태 코드
    - detail: 클라이언트에게 전달한 메시지
    - headers: 헤더를 요구하는 응답을 위한 선택적 인수

요청 처리

- Error Handling
  - ✓ 에러 처리
    - todo.py 파일 수정

```
from fastapi import APIRouter, HTTPException, status
#라우터 객체 생성
todo_router = APIRouter()

#데이터를 저장할 list 생성
todo_list = []
```

요청 처리

- Error Handling
  - ✓ 에러 처리
    - todo.py 파일 수정

```
from model import Todo
#todo 요청을 post 방식으로 요청한 경우 처리
@todo_router.post("/todo", status_code=201)
#매개변수를 받아서 todo_list에 삽입
async def add_todo(todo: Todo) -> dict:
    todo_list.append(todo)
    return {
        "message": "Todo added successfully."
    }
```

요청 처리

- Error Handling
  - ✓ 에러 처리
    - todo.py 파일 수정

```
from model import TodoItems
#todo 요청을 get 방식으로 요청한 경우 처리
@todo_router.get("/todo", response_model=TodoItems)
async def retrieve_todo() -> dict:
    return {
        "todos": todo_list
    }
```

요청 처리

- ❖ Error Handling
  - ✓ 에러 처리
    - ● todo.py 파일 수정

```
#id를 받아서 하나의 데이터를 리턴하는 요청을 처리
from fastapi import Path
@todo_router.get("/todo/{todo_id}")
async def get_single_toto(todo_id:int = Path(..., title="ID를 가지고 하나의 데이터를 찾아오는 요청")) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            return{
                "todo":todo
            }
    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                        detail = "존재하지 않는 아이디")
```

요청 처리

- ❖ Error Handling
  - ✓ 에러 처리
    - ● todo.py 파일 수정

```
from model import TodoItem
@todo_router.put("/todo/{todo_id}")
async def update_todo(todo_data: TodoItem, todo_id:int = Path(..., title="ID에 해당
하는 데이터 수정")) -> dict:
    for todo in todo_list:
        if todo.id == todo.id:
            todo.item = todo_data.item
            return{
                "message":"수정 성공"
            }
    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                        detail = "존재하지 않는 아이디")
```

요청 처리

- Error Handling
  - ✓ 에러 처리
    - todo.py 파일 수정

```
@todo_router.delete("/todo/{todo_id}")
async def delete_todo(todo_id:int) -> dict:
    for index in range(len(todo_list)):
        todo = todo_list[index]
        if todo.id == todo.id:
            todo_list.pop(index)
            return{
                "message":"삭제 성공"
            }
    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                        detail = "존재하지 않는 아이디")
```