Flask로 만드는 웹 서버

> 작성자: 유승연

1. 개요

> 본 문서는 서울고 기계공학부 동아리에서의 교육 및 실습 목적으로 작성되었습니다.

해당 문서에서는 다음과 같은 내용을 서술하고 있습니다.

- RESTful API 기초
- JSON
- Flask 기초
- SQL을 활용한 DB 기초
- SocketIO 기초
- 약간의 HTML, CSS, JS

2. RESTful API

RESTful API는 두 컴퓨터 시스템이 인터넷을 통해 정보를 안전하게 교환하기 위해 사용하는 인터페이스입니다. 대부분의 애플리케이션은 다양한 일을 수행하기 위해 다른 여러 애플리케이션들과 통신해야 합니다. RESTful API는 안전하고 신뢰할 수 있으며 효율적인 소프트웨어 통신 표준을 따르므로 이러한 정보 교환을 지원합니다.

아래부터는 RESTful API를 짧게 REST라고 부르겠습니다.

2.1. REST 요청에 필요한 것들

REST 요청에서의 구성요소를 서술하였습니다.

2.1.1. URL

요청을 경로를 결정하는 구성요소입니다.

참고로, URL의 구조는 다음과 같다. 아래는 URL의 모든 요소를 나타내어 보았다.

```
https://www.seoul-meta.com:4000/directory/page?param=true#1
```

- `https`: Protocol
- `www`: sub domain
- `seoul-meta.com`: domain, 서브 도메인과 도메인 자리에는 IP 번호가 올 수 있다.
- `4000`: port, 도메인 옆에 콜론(:)을 적고 포트 번호를 적는다. 포트 번호는 (0 ~ 99999)까지 있다.
- `directory`: sub folder
- `page`: slug
- `?<key>=<value>`: parameter, 링크 뒤에 물음표(?)가 붙고 그 뒤에 `키=값`의 형태로 붙는 것이 특징이다. 구글 검색으로 예를 들자면 구글은 `https://www.google.com/search?q=python` 다음과 같이 python을 검색하게 된다면 이러한 링크가 만들어진다. search 뒤로 q값에 python이 들어간 모습을 볼 수 있다.

- `#<anchor>`: anchor, 같은 페이지 내에서 특정 위치로 이동할 때 쓰인다. 약간의 책갈피 느낌이다. 예를 들어 나무위키에서는 각 문단에 anchor를 넣어 원하는 문단으로 바로가게 할 수 있는 기능이 있다.

2.1.2. Data

요청에 들어갈 데이터입니다.
데이터를 넣지 않은 상태로 요청을 할 수도 있습니다.

2.1.3. HTTP method

HTTP method는 서버가 받은 데이터를 가지고 처리해야 할 작업을 명시합니다.
대표적인 4가지 메소드는 다음과 같습니다.

2.1.3.1. GET

서버의 지정된 URL에 있는 리소스에 액세스합니다.
일부는 요청에 따라 데이터를 필터링하는 작업을 요청할 수도 있습니다.

예) 학생 리스트 요청, 학생 리스트 중 특정 반 리스트만 요청, 서버에 있는 파일 액세스

2.1.3.2. POST

이 요청은 데이터와 함께 요청됩니다.
서버가 새로운 리소스를 만들게 요청합니다.

예) 새로운 학생 등록, 댓글 달기

2.1.3.3. PUT

이 요청은 데이터와 함께 요청됩니다.
서버가 기존 리소스를 수정하도록 요청합니다.

예) 학생 정보 변경, 댓글 수정

2.1.3.4. DELETE

서버가 기존 리소스를 제거하도록 요청합니다.

예) 학생 삭제, 댓글 삭제

2.1.4. HTTP header

요청 헤더는 클라이언트와 서버 간에 교환되는 메타데이터입니다.
예를 들어, 요청 헤더는 요청 및 응답의 형식을 나타내고 요청 상태 등에 대한 정보를 제공합니다.

2.1.4.1. Metadata

Metadata는 데이터에 대한 설명을 하는 데이터입니다.

2.2. REST 응답에 들어오는 것

REST 응답에서의 구성요소를 서술하였습니다.

2.2.1. Status code

상태 코드는 클라이언트와 서버간 요청을 할 때에 나오는 대부분의 상황을 3자릿수의 코드로 정의하였습니다.

- 200번대: 성공
- 400번대: 실패
  Status code 404는 리소스를 찾을 수 없을 때 나옵니다. (Page Not Found)

더 자세한 상태 코드는 아래 웹사이트에 정리되어 있습니다.
https://developer.mozilla.org/ko/docs/Web/HTTP/Status

2.2.2. Message

응답 본문(Body)에는 응답에 대한 리소스가 들어옵니다.
문자열 형태로 전송되며, 대표적인 포맷은 XML/JSON 입니다.
그렇기 때문에 이런 메시지를 파싱하여 우리가 가공할 수 있는 형태로 만듭니다.
XML은 개틀딱 포맷이여서, 우리는 MZ한 JSON을 쓸 겁니다.

2.2.2.1. JSON

JSON은 아래와 같이 여러 데이터 타입를 저장할 수 있습니다

2.2.2.1.1. Boolean

참/거짓, 1/0 ON/OFF 등등 편한대로 생각하면 된다.

아래와 같이 작성된다. 파이썬은 True, False지만 여기서는 true, false로 소문자로 쓴다.
소문자로 쓴다.
소문자로 쓴다.

```
true #참
false #거짓
```

2.2.2.1.2. Integer & Floating Point Number

정수와 부동 소수점이다. 부동 소수점 그냥 4.0같이 점(.)을 포함한 모든 수이다.

아래와 같이 작성된다.

```
1
3.0
```

2.2.2.1.3. String

문자열이다.

아래와 같이 작성된다.

```
"string"
'string'
```

2.2.2.1.4. List

데이터를 나열하는 리스트이다. 리스트에는 다른 데이터 타입도 들어갈 수 있고, 리스트 본인도 리스트 안에 들어갈 수 있다.

아래와 같이 작성된다.

```
[true, 2, 3.0, "4", [5], {"7": 8}]
```

2.2.2.1.5. Dictionary

key와 value로 구분된다. 사전처럼 생각하면 key는 단어이고, value는 단어에 대한 뜻이다.
key는 문자열로 쓰이고, value에는 리스트같이 다른 모든 데이터 타입이 들어갈 수 있다.

아래와 같이 작성된다.

```
{"유승연": "Piop2", "박현우": "mbp16", "1": 2, "another": {"hi": [1, 2, true]}}
```

2.2.3. Header

응답에 대한 헤더 또는 메타데이터도 포함됩니다. 응답에 대한 추가 컨텍스트를 제공하고 서버, 인코딩, 날짜 및 콘텐츠 유형과 같은 정보를 포함합니다.

3. Flask

플라스크(Flask)는 파이썬으로 작성된 마이크로 웹 프레임워크이다. 그냥 쉽게 말해서, 간단하게 만들 수 있는 웹 서버 개발 도구라는 뜻이다.
Flask를 이용해서 RESTful API를 작성할 수 있는 것 뿐만 아니라, 웹 사이트 구동도 할 수 있어서 하나의 프로그램으로 API도 만들고 웹 구동도 할 수 있다. 그냥 웹 사이트 띄우는 거랑 기능 구현이랑 같이 할 수 있다는 뜻이다.

3.1. 시작하기 앞서서 해야하는 것들

Flask는 파이썬에 기본적으로 장착되어 있는 것이 아니라, 다른 사람들이 만든 것들이여서 인터넷에 있는 Flask를 다운받아주어야 한다.

윈도우는 CMD로, 맥은 Terminal을 켜준다. 파이썬에서 기본적으로 제공하는 pip라는 툴을 이용해서 외부 패키지를 다운받아준다. (맥에서는 pip3을 사용해주세요.)

```
pip list
```

다운받은 패키지를 확인한다. Flask가 없으면 아래 항목들을 계속 진행한다.

```
pip install -U pip
```

일단 pip를 업데이트해준다.

```
pip install Flask
```

Flask를 깔아준다.

3.2. 프로젝트 구조

```
/project
        /static
        /templates
        app.py
```

project 폴더 안에 있는 Flask 파일 구조를 적어보았다.

- app.py: Flask 코드를 적는다
- templates: HTML 파일이 들어가는 폴더이다
- static: CSS, JS 파일과 다른 이미지 등등 리소스 파일들이 들어가는 폴더이다.

위 구조대로 폴더를 일단 만들자.

3.3. 기본 코드

가장 기본이 되는 코드는 아래와 같다.

```
from flask import Flask
app = Flask(__name__)

if __name__ == "__main__":
    app.run()
```

코드를 천천히 뜯어보자.

```
from flask import Flask
```

flask 패키지에서 Flask 클래스를 불러와준다. 가장 기초가 되는 클래스이다.

```
app = Flask(__name__)
```

app 변수에 Flask 앱을 생성해준다. __name__을 써주는 거는 설명하기 복잡해지니까 일단 외우자.

```
if __name__ == "__main__":
    app.run()
```

run() 메소드로 앱을 실행시켜준다. 여기서 if문은 C/C++언어에서 main 함수와 같은 의미이다.
이 프로그램을 실행했을 때, 아래 코드를 실행한다는 조건문이다.

기본적으로 5000번 포트에서 열린다.
`http://127.0.0.1:5000/`에 들어가면 켜진 서버를 볼 수 있다.

```
if __name__ == "__main__":
    app.run(debug=True)
```

개발중이라면, debug를 활성화해서 에러 내용을 바로 볼 수 있게 할 수 있다.

3.4. 라우팅

웹사이트에 URL 경로를 하나 만들어 보자

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello_world():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

```
@app.route("/")
```

위 구문은 `http://127.0.0.1:5000/` 같은 웹사이트에 접속했을 때, 아래 함수를 실행하겠다는 구문이다.

```
@app.route("/hello")
```

이 구문은 `http://127.0.0.1:5000/hello` 으로 접속했을 때, 아래 함수를 실행한다는 구문이다.

```
def hello_world():
    return "Hello World!"
```

그리고 위 루트에 접속하였을 때, 실행되는 함수이다. 이 함수는 `"Hello World"` 문자열을 리턴하기 때문에 서버를 실행하고 `http://127.0.0.1:5000/` 으로 접속한다면, 웹사이트에 Hello World라고 적힐 것이다.
그리고 프로젝트를 하다보면 여러 루트를 생성하면서, 여러 함수들도 만들어지게 될 텐데 그럴때마다 다른 함수로 이름을 붙여야한다. 파이썬에서는 같은 이름으로 함수를 여러 개를 정의할 수 없기 때문이다.

이렇게 간단하게 웹사이트의 루트를 정의하는 것을 배웠다.

3.5. URL 변수 규칙 만들기

url을 받아서 유동적인 웹사이트를 만들어보자.

```
from flask import Flask
app = Flask(__name__)

@app.roite("/hi/<name>")
def say_hi(name):
```

```
    return "Hello %s" % name

@app.route("/cookie/<int:count>")
def count_cookies(count):
    return "Here are %d cookies"

if __name__ == "__main__":
    app.run()
```

```
@app.roite("/hi/<name>")
def say_hi(name):
    return "Hello %s" % name
```

위 코드는 `http://127.0.0.1/hi/` 다음에 오는 값을 `name` 변수로 받아서 Hello `name`을 출력하는 함수이다. 그렇기 때문에 `http://127.0.0.1/hi/piop2`를 입력하면 화면에 Hello piop2가 출력이 될 것이다. 다른 값을 넣어서 확인해보자.

근데 여기에 들어오는 모든 변수들은 문자열 형태로 들어오게 되는데, 이것을 정수나 부동 소수점의 형태로 바꾸고 싶을 때가 있을 것이다. 이럴 때에는 아래와 같이 작성해주면 된다.

```
@app.route("/cookie/<int:count>")
def count_cookies(count):
    return "Here are %d cookies" % count
```

위 코드는 `http://127.0.0.1/cookie/` 다음에 오는 값을 정수의 형태로 바꾼 후에 count 변수에 값을 저장하고, Here are `count` cookies를 출력한다.

이렇게 링크 규칙을 만들어서 변수로 저장할 수가 있는데, <타입:변수이름> 형태로 적어야한다. 다음 줄에는 가능한 타입들이다.

- string: 아무것도 적지 않으면 적용되는 기본 타입이다. 문자열의 형태로 저장된다.
- int: 정수의 형태로 저장된다.
- float: 부동 소수점의 형태로 저장된다.
- path: 슬래시(/)가 포함된 문자열의 형태로 저장된다. 즉, 라우트 URL 뒤에 있는 모든 경로가 문자열로 저장된다는 의미이다.
- uuid: UUID 문자열의 형태로 저장된다. 몰라도됨

이렇게 여러 형태로 저장할 수 있다.

3.6. URL 빌드

다음으로는 특정 라우트의 URL을 빌드하는 함수를 배워보자. 예제 코드는 다음과 같다.

```
from flask import Flask, url_for

app = Flask(__name__)

@app.route('/')
def index():
    return 'index'
```

```
@app.route('/login')
def login():
    return 'login'

@app.route('/user/<username>')
def profile(username):
    return f'{username}\'s profile'

with app.test_request_context():
    print(url_for('index'))
    print(url_for('login'))
    print(url_for('login', next='/'))
    print(url_for('profile', username='John Doe'))
```

위 3개의 라우트 정의에 대해서는 **3.4. 라우팅**과 **3.5. URL 변수 규칙 만들기**를 보고오자.

```
url_for(end_point, value1, value2, ...)
```

`url_for()`함수는 첫 번째로 `end_point`를 받고, 선택에 따라 `value`값을 받는다. `value`값이 없어도 작동한다.
`end_point`는 URL을 빌드할 라우트를 적는 곳이다. 문자열 형태로 넣어야되고, 링크가 아닌 라우트를 정의한 함수의 이름을 넣어야한다.
`value`값은 여러개가 들어갈 수 있다. `value`값은 문자열이여야 하고, 적을 때에는 **key=value**와 같은 형태로 적어줘야 한다.

자세한 내용은 아래 코드를 보자.

```
with app.test_request_context():
    print(url_for('index'))
    print(url_for('login'))
    print(url_for('login', next='/'))
    print(url_for('profile', username='John Doe'))
```

아래는 프로그램을 실행했을 때의 출력이다.

```
/
/login
/login?next=/
/user/John%20Doe
```

첫 줄의 `with app.tetst_requests_contents()` 구문은 flask 앱 테스트 할 때 쓰는 구문이다. 위에 설명했던 `if __name__=="__main__"`과 비슷하게 생각해도 된다.

어쨋든 이제 코드와 출력을 비교해보면서 배워보자.
`index()`함수는 라우트를 `/`로 정의했으니, /가 나온다. 그리고 `login`도 동일한 원리로 나오는 것을 볼 수 있다.

그리고 3번째에는 `login`을 호출 후에 next라는 매개변수를 /값을 넣어서 집어넣었는데, 그 결과 물음표(?) 뒤에 `next=/`라는 URL 매개변수가 붙은 상태로 출력이 되었다.

하지만 4번째를 보면 username매개변수를 넣었는데도 불구하고 아까 3번째와 다른 결과가 나왔다. 그 이유는 `profile()`함수가 정의될 때를 자세히보면 알 수 있다. `profile()`함수는 매개변수로 username을 받고 있기 때문에 URL 매개변수가 아닌 곳으로 입력이 되었다.

3.7. HTTP 메소드

> 이 문단의 코드들은 직접 실행하지 말고, 읽으면서 공부하세요.

받는 HTTP 메소드에 따라 다르게 반응하는 기능을 만들어 볼 것이다. HTTP 메소드에 관한 내용은 **2.1.3. HTTP method**를 참고하자. 예제 코드는 다음과 같다.

```
from flask import Flask, requests, abort
app = Flask(__name__)

@app.route("/hello", method="GET")
def hello():
    if requests.method == "GET":
        return "Hello"

@app.route("/user", method=["POST", "GET"])
def user():
    user_name = request.arg.get("userName")

    if requests.method == "GET":
        if get_user(user_name) is None:
            abort(400)
        return get_user(user_name)
    elif requests.method == "POST":
        if get_user(user_name) is not None:
            abort(400)
        make_user(user_name)

if __name__ == "__main__":
    app.run()
```

아래와 같은 RESTful API 서버를 만든다 가정하자.

- `/hello`에 GET 요청을 하면 Hello가 나오게 한다.
- `/user`에 GET 요청을 하면 기존에 있는 유저를 가져오고, 그 정보를 보낸다.
- `/user`에 POST 요청을 하면 새로운 유저를 추가하고, 이미 있는 유저라면 에러(400)가 나게 한다.

```
from flask import request, abort
```

URL의 parameter를 받기 위해 request를 가져오고, 에러를 일으키기 위해 abort함수를 가져왔다.
URL에 관한 내용은 **2.1.1. URL**, HTTP status code에 관한 내용은 **2.2.1. Status code**을 찾아보자.

```
@app.route("/hello", method="GET")
```

`/hello`라는 라우트를 생성해주고, 이 링크로 받을 메소드를 GET이라고 예약해주었다.

```
def hello():
    if request.method == "GET":
        return "Hello"
```

그리고 requests.method를 통해 받은 것의 메소드를 확인해주는 작업을 "GET"으로 하였다. 그리고 GET으로 받게 된다면 Hello를 리턴하도록 하였다.

다음은 유저에 관한 기능을 만들 것이다.

```
@app.route("/user", method=["POST", "GET"])
```

GET과 POST 요청을 받기 때문에 2가지 메소드를 예약해두었다.

```
def get_user(name: str) -> Optional[str]:
    """get user data"""
    ...

def make_user(name: str) -> None:
    """make new user"""
    ...
```

간단한 2개의 유저 관련 함수를 만들었다고 가정하자.
`get_user`는 name 매개변수를 문자열을 받아서 기존에 존재했던 유저의 데이터를 리턴하거나, 유저가 존재하지 않으면 None을 리턴한다.
`make_user`는 name 매개변수를 문자열로 받아서 새로운 유저를 생성한다.

```
def user():
    user_name = request.arg.get("userName")

    if request.method == "GET":
        if get_user(user_name) is None:
            abort(400)
        return get_user(user_name)
    elif request.method == "POST":
        if get_user(user_name) is not None:
            abort(400)
        make_user(user_name)
```

위 함수를 하나 하나 뜯어보자.

```
user_name = request.arg.get("userName")
```

`requests.arg.get`은 값을 넣어서 요청이 들어온 링크의 URL 매개변수에서 원하는 값을 받을 수 있다.
하지만, 이 함수는 URL에 없는 매개변수를 받으려고 할 때 KeyError가 발생한다. 이런 에러의 발생은 아래와 같이 해결 할 수 있을 것이다.

```
try:
    user_name = request.arg.get("userName")
except KeyError:
    abort(400)
```

userName 매개변수를 제출하지 않았을 때에는 `abort`함수를 이용해서 status 400 Bad requests 에러가 나도록 하였다.

또는, 값을 넣지 않으면 기본 값을 설정해서 에러가 발생하지 않고 유저 데이터가 반환되게 할 수 있을 것이다.

```
try:
    user_name = request.arg.get("userName")
except KeyError:
    user_name =  "유승연"
```

이렇게 에러를 처리해서 기본 유저명을 설정하는 것도 좋지만, 아래에 더 좋은 방법이 있다.

```
user_name = request.arg.get("userName", "유승연")
```

`requests.arg.get`함수의 2번째 매개변수를 이용하면 가져올 값과 없을 때 사용하는 기본값을 설정할 수 있다. 전에 썼던 코드보다 3줄이나 줄어든 형태이다.

그럼 이제 처리 부분 코드를 보자.

```
if request.method == "GET":
    if get_user(user_name) is None:
        abort(400)
    return get_user(user_name)
elif request.method == "POST":
    if get_user(user_name) is not None:
        abort(400)
    make_user(user_name)
```

GET으로 들어온다면 user_name을 확인한 후에 `get_user`로 존재하는 유저인지 확인할 것이다. None이 들어온다면 존재하지 않는 유저이니 `abort`함수로 에러 코드를 전송하자. 그렇지 않으면 유저 데이터를 반환한다.

POST를 받게 된다면, 이미 존재하는 유저인지 확인하고 이미 존재한다면 에러 코드를 전송한다. 그렇지 않다면 `make_user`함수로 새로운 유저를 생성한다.

3.7.1. 코드를 더욱 가독성있게 만들기

이제 위에 썼던 코드를 좀 더 읽기 쉽게 바꾸어 보겠다.

```
from flask import Flask, request, abort
app = Flask(__name__)

@app.get("/hello")
def hello():
    if request.method == "GET":
        return "Hello"

@app.get("/user")
def user_get():
    if get_user(request.arg.get("userName")) is None:
        abort(400)
    return get_user(user_name)

@app.post("/user")
def user_make():
```

```
    if get_user(request.arg.get("userName")) is not None:
        abort(400)
    make_user(user_name)

if __name__ == "__main__":
    app.run()
```

`route`함수를 사용하지 않고, 오직 HTTP method에만 집중한 코드 구조라고 볼 수 있다.
여기에는 get과 post만 나왔지만, get post put delete도 다 가능하다,

다음 문단에서는 html를 불러와서 사용자의 화면에 렌더링하는 방법을 살펴보겠다.

3.8. 요청 데이터 받기

url의 매개변수를 가져오는 것이 아닌, 요청 데이터라는 것도 볼 수 있다.
요청 데이터는 JSON으로 주고 받기 때문에 여러 데이터 형식을 주고 받을 수 있다는 장점이 있다.

```
from flask import Flask, request, abort

app = Flask(__name__)

@app.get("/hi")
def say_hi():
    if not request.form:
        abort(400)
    name = request.form["name"]
    return "Hello %s" % name

if __name__ == "__main__":
    app.run()
```

하나씩 살펴보자.

```
from flask import Flask, request, abort
```

일단 요청받은 데이터를 받기 위해 request, 오류를 일으키기 위해 abort를 가져왔다.

```
@app.get("/hi")
def say_hi():
```

`http://127.0.0.1:5000/hi`라는 링크로 get 요청이 들어왔을 때, `say_hi` 함수를 실행시킬 것이다.

```
if not request.form:
    abort(400)
```

`request.form`은 요청받은 데이터를 가져오게 해준다. 데이터가 없을 때에는 빈 딕셔너리({})를 받게 되니 not을 사용해 데이터를 받지 못한 경우에는 `abort`를 이용해 에러 코드 400을 일으키자.

```
name = request.form["name"]
return "Hello %s" % name
```

이제 name변수에 name 데이터를 가져와서 반환해준다.
정말 간단하다.

3.8.1. requests로 요청보내기

파이썬의 requests 모듈은 여러 http요청을 위한 패키지이다. 이걸 이용하면 요청을 보낼 때, 데이터를 같이 보낼 수 있다.

일단 requests 모듈이 없다면 설치해주자

```
pip install requests
```

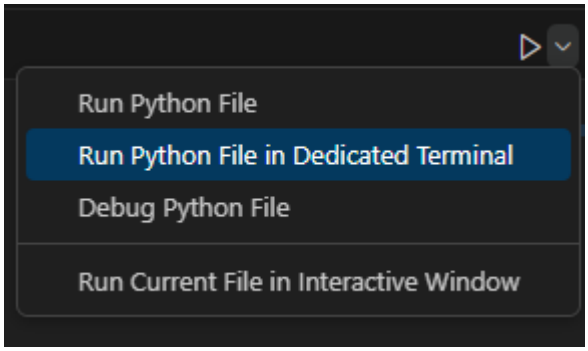
그리고 테스트 진행을 위해 스크립트를 하나 만들어서 실행할 것이다. 새 파이썬 파일을 만들어주자.

아래 코드는 get 요청을 보내는 코드이다.

```
import requests
data = {"name": "piop2"}
response = requests.get("http://127.0.0.1:5000/hi", data=data)
print(response.text)
```

코드를 살펴보면 딕셔너리 형태의 data에 name을 넣어서 `requests.get`으로 요청을 보내는 모습이다. 그리고 받은 요청에 대한 메시지를 `response.text`로 출력을 하였다.

flask앱과 요청 스크립트를 순서대로 실행하자. 앱이 먼저 실행되어야지 요청을 받을 수 있다.
직접 테스트해보자.



Visual Studio Code에서는 2개 이상의 실행부터는 위와 같은 버튼을 눌러야지, 분할시켜서 실행시킬 수 있다. 참고하자.

3.9. 템플릿 띄워보기

이번에는 html파일을 띄워볼 것이다.
완성된 코드는 다음과 같다.

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route("/meta")
```

```
def meta():
    return render_template("meta.html")

if __name__ == "__main__":
    app.run()
```

일단, 아래 HTML파일을 `/templates` 폴더에 `meta.html`로 저장하자. html에 대한 설명은 지금은 생략하도록하겠다. 일단 코드를 복사한 후에 파일을 만들어 보자.

```
<!doctype html>
<html lang="ko">
    <head>
        <meta charset="utf-8">
        <title>Hello Flask</title>
        <style>
            html, body {
                margin: 0;
                height: 100%;
                overflow: hidden;
            }
            iframe {
                overflow: show;
            }
        </style>
    </head>
    <body>
        <iframe src="https://seoulmeta2023.github.io/" width="100%" height="100%"/>
    </body>
</html>
```

완성된 파일 구조는 아래와 같다.

```
/project
        /static
        /templates
                meta.html
        app.py
```

```
from flask import render_template
```

flask 패키지에서 새롭게 render_template이라는 함수를 가져왔다. 이 함수는 `/templates`폴더 안에 있는 html 파일을 웹사이트에 보여주는 역할을 한다.

```
@app.route("/meta")
def meta():
    return render_template("meta.html")
```

`http://127.0.0.1:5000/meta/`에 접속하면 `meta.html`을 띄워보고 싶다.
return 자리에 render_template함수를 쓴 다음에 "meta.html"을 매개변수에 넣자. 그러면 자동으로 flask앱이

`/templates`폴더 안에 있는 `meta.html`을 찾아서 띄울 것이다.
확인해보자.

3.10. 여러 기기에서 접속

이제 만든 flask앱을 개발 컴퓨터가 아닌 다른 기기에서도 접속시켜보자. 호스트만 변경해주면 된다.

```
...
if __name__=="__main__":
    app.run(host="0.0.0.0")
```

호스트 주소를 0.0.0.0으로 바꾸면 된다. 0.0.0.0은 다른 기기에서 접속을 할 수 있게하는 특별한 주소이다.

이제 실행시켜보자.
터미널에 `http://127.0.0.0:5000`도 나오지만 밑에 주소가 하나 더 나올 것이다. 이제 같은 와이파이 내에서 그 주소로 접속하면 어떤 기기든지 내 flask앱에 접속해서 볼 수 있을 것이다.

직접 해보자.

3.11. URL 디자인에 관하여

4. SQL

SQL은 데이터베이스에서 데이터를 추출하고 조작하는 데에 사용하는 데이터 처리 언어이다. 액셀같은 형태의 데이터베이스에서 백엔드 서버에서 원하는 데이터를 빠르게 필터링해서 가져오고 싶을 때 종종 쓰인다.

하나의 데이터 베이스 안에서 여러 개의 테이블을 만들 수 있다. 테이블을 하나를 만들었을 때에는 엑셀과 비슷하게 생성된다. 우리는 각 열(여기서는 칼럼이라고 부른다)의 데이터 이름과 타입을 지정해주고, 그에 맞게 계속 데이터를 추가하는 구조이다.

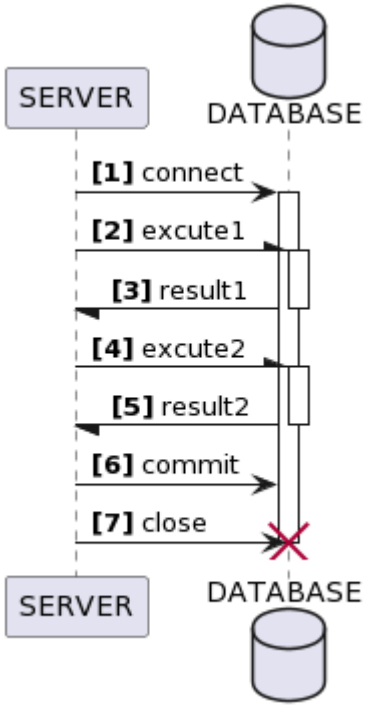
4.1. SQLite

> SQLite의 문법은 일단 외우고, **4.2. 파이썬에서 사용하기**를 배울 때 연습해보자.

SQL로 데이터 베이스를 관리할 수 있는 방법은 여러가지인데 우리는 그 중에서 SQLite를 배울 것이다

4.1.1. DataBase의 작동 루틴

데이터 베이스는 아래와 같은 작동 루틴을 가진다



- **connect**: 데이터 베이스를 연다
- **execute**: SQL의 문법을 사용해서 데이터 베이스에 명령을 실행한다.
- **commit**: 변경된 사항을 저장한다.
- **close**: 데이터 베이스를 닫는다.

commit을 하지 않고 close를 하게 된다면, 저장이 되지 않는다는 것을 꼭 기억하자.

4.1.2. Data Type

일단 각 칼럼에 들어갈 수 있는 데이터 타입을 알아보자.

- **NULL**: 파이썬의 None과 같이 아무것도 없는 값을 말한다.
- **INTEGER**: 정수, 0~8비트 정도의 수를 저장할 수 있다.
- **REAL**. 파이썬의 float와 같은 실수이다.
- **TEXT**. 크기를 알 수 없는 다량의 문자열을 저장할 때 사용한다.
- **BLOB**. Binary Large Object의 약자로, '큰 이진 파일'이다. 이미지와 같은 다양한 파일들을 저장할 수 있다.

4.1.3. 테이블 만들기

한 데이터 베이스에 테이블은 여러 개 만들 수 있다. 엑셀에서 원하는 만큼 시트를 생성할 수 있는 것처럼 말이다.

이제부터 데이터베이스에 학생들의 출석부를 만들어보자. 학생마다 아래와 같은 정보를 저장할 것이다.

- id: 학번
- name: 이름
- phone_number: 전화번호
- birth_date: 생년월일

```
CREATE TABLE student_list (id INTEGER, name TEXT, phone_number TEXT, birth_date TEXT);
```

4.1.3.1. 데이터 기본값 설정하기

출석부에서 새로운 학생을 추가할 때, 학생의 이름을 같이 넣지 않았다면 "이름 없음"으로 추가되게 해보자.

```
CREATE TABLE student_list (id INTEGER, name TEXT DEFAULT "이름 없음", phone_number TEXT, birth_da
```

기본값이 존재하길 원하는 값 뒤에 DEFAULT를 붙인 후에 그 뒤에 기본값이 될 값을 넣어주면 된다.

4.1.4. 폴더 삭제하기

DROP 구문으로는 폴더에 해당하는 테이블 또는 데이터베이스를 삭제할 수 있다.

4.1.4.1. 테이블 삭제하기

```
DROP TABLE student_list;
```

간단하게 테이블의 이름을 넣어줌으로써 `student_list` 테이블을 통째로 삭제할 수 있다.

4.1.4.2. 데이터베이스 삭제하기

```
DROP DATABASE student_db;
```

똑같이 데이터베이스의 이름을 넣어서 `student_db`를 삭제할 수 있었다.
이 구문은 보통 기존에 있는 데이터베이스를 초기화할 때 사용한다.

4.1.5. 데이터 추가하기

INSERT INTO를 사용하면 원하는 테이블에 데이터를 추가할 수 있다.

```
INSERT INTO student_list VALUES (20715, "유승연", "010-5039-8849", "2006-02-20");
```

`student_list`의 칼럼 순서대로 id, name, phone_number, birth_date 순으로 데이터를 추가해주었다.

```
INSERT INTO student_list VALUES (20715, "유승연", "010-5039-8849", "2006-02-20"), (00000, "유승현
```

여러개도 추가할 수 있다.

또는, 원하는 칼럼의 데이터만 넣어줄 수 있다.

```
INSERT INTO student_list (id, phone_number) VALUES (20715, "010-5039-8847");
```

id와 phone_number만 저장했다. 이렇게 된다면 name은 기본값인 "홍길동"이 들어가고 birth_date는 기본값이 주어지지 않았기 때문에 NULL값이 들어간다.

4.1.6. 데이터 조회

SELECT와 조건문을 이용하면 여러 작업을 수행할 수 있다.

기본적인 조회 방법은 아래와 같다.

```
SELECT id FROM student_list;
```

위 코드는 `student_list`의 모든 id를 조회하였다. 이렇게 된다면 데이터의 id만 나오게 된다.

```
SELECT * FROM student_list;
```

별(*)을 사용해서 모든 데이터를 가져올 수 있다.

4.1.6.1. 조건 조회

WHERE로 조건문을 추가할 수 있다.

들어가기 전에 기본적인 야라 연산자를 공부해보자.

- = : 같다
- != : 같지않다
- > : 크다
- >= : 크거나 같다
- < : 작다
- <= : 작거나 같다
- and : 양 옆의 조건이 같다.
- or : 양 옆의 조건 중, 하나라도 같다.

```
SELECT * FROM student_list WHERE id = 20715;
```

`student_list`의 모든 데이터 중에서 id가 20715인 데이터만 가져왔다.

4.1.6.2. 범위 조회

BETWEEN ... AND 문을 이용해서 ~이상~이하의 범위 조회를 할 수 있다.

```
SELECT * FROM student_list WHERE id BETWEEN 20700 AND 20899
```

id가 20700 ~ 20899 즉, 2학년 7반과 2학년 8반의 모든 학생들을 선택하였다.

4.1.6.3. 정렬 조회

ORDER BY 구문으로 오름차순/내림차순 조회를 할 수 있다.

```
SELECT * FROM student_list ORDER BY birth_date ASC;
SELECT * FROM student_list ORDER BY birth_date DESC;
```

학생들의 생일을 오름차순(ASC), 내림차순(DESC)으로 정렬해 주었다.

4.1.6.4. 제한 조회

LIMIT을 이용하면 검색된 결과들의 개수를 제한할 수 있다. 기본적으로 정렬 조회가 되었을 때에만 사용할 수 있다. 제한을 정하거나, 원하는 구간의 데이터만 조회할 수 있다.

```
SELECT * FROM student_list ORDER BY birth_date ASC LIMIT 5;
SELECT * FROM student_list ORDER BY birth_date ASC LIMIT 3, 8;
```

첫 구문은 오름차순 데이터중에서 5개까지만 가져온다.
아래 구문은 오름차순된 데이터에서 3 ~ 8번까지의 데이터만 가져온다.

4.1.6.5. 유사 조회

데이터의 문자열에서 유사한 부분이 있는 데이터만 가져올 수 있다.

```
SELECT * FROM student_list WHERE name LIKE "김%"
```

김으로 시작하는 모든 학생들의 이름을 가져온다.
이렇게 %는 어떠한 수 많은 문자가 그 곳에 들어갈 수 있음을 뜻한다.

- 김%: 김으로 시작하는~
- %김: 김으로 끝나는~
- %김%: 중간에 김이 있는~

그리고 하나의 문자만도 찾을 수 있다.

- 김_: 김으로 시작하고, 뒤에 한 글자만 오는 ~
- _김: 김으로 끝나고, 앞에 한 글자만 오는 ~
- _김_: 중간에 김이 있고 양 옆에 한 글자만 있는 ~

4.1.7. 데이터 업데이트

UPDATE를 사용하여 원하는 데이터를 업데이트할 수 있다.

학생들의 모든 학번을 20000로 초기화한다고 가정해보자.

```
UPDATE student_list SET id = 20000;
```

이렇게 된다면 `student_list`의 모든 데이터의 id가 20000이 된다.

그렇다면, 이제 id가 20715인 학생의 전화번호를 010-1234-5678로 바꿔보자.

```
UPDATE student_list SET phone_number = "010-1234-5678" WHERE id = 20715;
```

뒤에 WHERE 조건문을 추가해서 id가 20715인 학생의 전화번호를 010-1234-5678로 바꿔주었다. 여기서는 학번이 고유하여 한 명만 선택되지만, WHERE는 해당 조건문을 만족하는 모든 데이터를 가져온다는 것을 잊지 말자.

4.1.8. 데이터 삭제하기

모든 `student_list`의 데이터를 삭제해보자.

```
DELETE FROM student_list;
```

DROP가 다른 점은 DROP은 테이블을 삭제하는 것이고, DELETE는 테이블의 데이터를 삭제하는 것이다. 위 구문은 테이블의 모든 구문을 삭제한다.

```
DELETE FROM student_list WHERE id = 20715;
```

WHERE 조건문을 추가하여 id가 20715인 학생의 데이터를 삭제하였다.

4.2. 파이썬에서 사용하기

파이썬에서는 sqlite3이라는 이름으로 기본적으로 sqlite를 제공하고 있다.

```
import sqlite3
```

위 구문으로 sqlite3을 가져올 수 있다.

4.2.1. 데이터베이스 열기

데이터베이스를 열어보자.

```
import sqlite3

con = sqlite3.connect("student.sqlite3")
```

`student.sqlite3` 데이터베이스 파일을 열었다. 만약 존재하지 않는 파일을 열었다면, 해당하는 이름의 파일을 만들어준다. 걱정하지 말자.

```
con = sqlite3.connect(":memory:")
```

경로를 넣는 곳에 :memory:를 넣는다면, 컴퓨터의 렘에 데이터베이스를 연다. 컴퓨터가 꺼지면 데이터베이스는 삭제된다.

```
con.close()
```

`close()`로 열었던 데이터베이스의 연결을 닫을 수 있다.

```
con.commit()
```

그리고 잊으면 안되는 `commit()`으로 변경사항을 저장할 수 있다.

4.2.2. 커서

데이터베이스에 접근하게 해주는 커서를 얻어보자.

```
import sqlite3

con = sqlite3.connect("student.sqlite3")
cur = con.cursor()
```

`cursor()`함수를 통해서 커서를 얻을 수 있다. 커서는 데이터베이스에 SQL문을 실행시켜주도록 접근할 수 있게 해준다.

4.2.3. SQL 구문을 사용해보기

배운 SQL문을 직접 실행해보자.

```
import sqlite3

con = sqlite3.connect("student.sqlite3")
cur = con.cursor()

cur.execute("CREATE TABLE student_list (id INTEGER, name TEXT, phone_number TEXT, birth_date TEXT
```

커서의 `execute()`함수로 SQL문을 실행할 수 있다.

4.2.3.1. SQL 구문 포맷팅

```
student_id = 20715
student_name = "유승연"
student_phone_number = "010-5039-8847"
student_birth_date = "2006-02-20"

cur.execute("INSERT INTO student_list VALUES (?, ?, ?, ?);",
            (student_id, student_name, student_phone_number, student_birth_date)
)

cur.execute("INSERT INTO student_list VALUES (:id, :name, :phone_number, :birth_date);",
            {
                "id": student_id,
                "name": student_name,
                "phone_number": student_phone_number,
                "birth_date": student_birth_date
            }
)
```

위 2개의 execute 구문 다 같은 뜻이다.
물음표(?)를 넣고 뒤에 순서대로 변수를 넣을 수 있다. 또는, `:변수`형태로 이름을 넣고 뒤에 딕셔너리를 넣어서 이름마다의 변수를 넣어줄 수 있다.
간단하게 물음표(?)를 넣는 방식이 좋겠지만, 한 구문에 많은 변수가 들어가는 경우에는 가독성이 떨어질 수 있다. 2가지 표현 방식을 적절히 사용하자.

4.2.3.2. 값 가져오기

```
import sqlite3
```

```
con = sqlite3.connect("student.sqlite3")
cur = con.cursor()

cur.execute("SELECT * FROM student_list;")
```

SELECT같은 데이터를 가져오는 구문을 했을 때에는 값을 가져오는 2가지의 방법이 있다.

```
cur.fetchone()
```

하나만 가져와도, 여러 개를 가져와도, 오직 하나의 값만 나오게 한다.
여러 개의 값을 가져온다면, 그 값들 중에서 하나씩 가져온다.
가져올 값이 없다면 None이 나온다. 참고하자.

```
cur.fetchmany()
```

보통 여러 개의 값을 가져올 때, 사용한다. 가져온 모든 값들을 리스트에 묶어서 나오게 한다.
가져올 값이 없다면 빈 리스트([])가 나온다.

`fetchmany()`의 경우 for 반복문과 같이 쓸 수 있다. 리스트를 반환하기 때문이다.

```
for data in cur.fetchmany():
    ...
```

```
import sqlite3

con = sqlite3.connect("student.sqlite3")
cur = con.cursor()

cur.execute("SELECT * FROM student_list;")
students = cur.fetchmany()
```

`fetchone()`과 `fetchmany()`는 SELECT같은 값을 가져오는 구문이 실행되고 나서 바로 하단 줄에 작성해야한다. 주의하자.

4.2.4. 잊지말아야 되는 것

```
con.close()
```

다 쓰고 난 후에는 데이터베이스 파일을 꼭 닫아주도록 하자.

5. Web Socket

웹소켓은 하나의 TCP 접속에 정보 통신 채널을 제공하는 컴퓨터 통신 방법을 말한다.
TCP 통신은 HTTP통신과 다르게 요청을 보내고 연결을 끊지 않고, 계속 연결을 유지하여 서버와 통신한다. 대표적으로 게임에서 이런 TCP 통신을 한다.

실시간으로 정보를 주고 받아야 되는 상황에서는 HTTP 요청을 여러번 보내는 것이 아닌, 웹소켓을 이용하도록 하자.

5.1. SocketIO

> SocketIO를 설명하는 문단에서는 Python과 JS로 작성하는 방법을 각각 설명합니다.
> SocketIO에서의 개념을 먼저 설명하고, 하위 문단에 각 언어들의 작성 방식을 설명합니다.
> JS는 바닐라JS를 기준으로, 웹에서의 클라이언트 구현만을 설명합니다.

> 이 문단의 코드들은 직접 실행하지 말고, 읽으면서 공부하세요.

SocketIO는 웹 어플리케이션을 위한 웹소켓 기반 이벤트 기반 라이브러리이다.
방금 말했다시피 이벤트 기반으로 돌아간다. 서로가 서로의 이벤트를 발생시키면서 정보를 주고 받는 형식이다.

SocketIO는 하나의 개념으로 여러 언어에서 실행시킬 수 있다.

- Server: JS, Java, Python, Go, Rust
- Client: JS, Java, C++, Swift, Dart, Python, .Net, Rust, Kotlin

Web Socket이라는 프로토콜을 사용하기 때문에, 클라이언트는 `ws://127.0.0.1/`와 같은 ws프로토콜로 접속해야한다.

5.1.1. 시작하기 앞서서 해야하는 것들

필요한 라이브러리를 설치해보도록 하자.

JS

JavaScript는 다운받지않고 링크를 통해 라이브러리를 가져올 수 있다.

```
import { io } from "https://cdn.socket.io/4.6.0/socket.io.esm.min.js";
```

Python

pip를 이용해서 라이브러리를 다운받자.

```
pip install python-socketio
```

5.1.2. Server / Client 만들기

서버와 클라이언트는 서로 통신한다.서버는 서버의 데이터를 이용해서 유저들에게 서비스를 제공하고, 클라이언트를 통해 유저들은 서비스를 제공받는다.

1개의 서버에 여러개의 클라이언트가 연결된 형태를 생각하면 된다.

JS

클라이언트를 정의하는 법은 다음과 같다.

```
import { io } from "https://cdn.socket.io/4.6.0/socket.io.esm.min.js";
const socket = io("ws://127.0.0.1/")
```

Python

서버를 정의하는 법은 다음과 같다.

```
import socketio

sio = socketio.Server()
app = socketio.WSGIApp(server)
```

위와 같이 간단하게 만들 수 있다.

그리고 클라이언트는 다음과 같이 정의한다.

```
import socketio

sio = socketio.SimpleClient()
```

그리고 다음과 같이 클라이언트를 서버에 연결할 수 있다.

```
sio.connect("ws://127.0.0.1/")
```

5.1.3. event 처리

SocketIO의 핵심이 되는 개념인 event에 대해 알아보자.
게임 개발을 하면 많이 접할 수 있는 이벤트는 쉽게 말하면 프로그램에 의해 감지되고 처리될 수 있는 동작이나 사건을 말한다.

서버는 클라이언트에 데이터만 보내지 않고, 이벤트와 데이터를 함께 보낸다. 그렇게 각각 발생한 것들에 대해 핸들링하도록 한다. 이 문단에서는 일단 이벤트가 왔을 때, 처리하는 함수를 만드는 방법을 알아보겠다.

서버와 클라이언트의 이벤트 처리 코드는 똑같다.

JS

```
import { io } from "https://cdn.socket.io/4.6.0/socket.io.esm.min.js";
const socket = io("ws://127.0.0.1/")

socket.on("my_event", (data) => {
    ...
})
```

`socket.on`이라는 함수에 이벤트 이름과 이벤트 처리 함수를 넣는다.

Python

아래는 서버 코드이다.

```
import socketio

sio = socketio.Server()
```

```
app = socketio.WSGIApp(server)

@sio.on("my_event")
def on_my_event(sid, data):
    ...

@sio.event
def my_event(sid, data):
    ...
```

아래는 클라이언트 코드이다.

```
import socketio

sio = socketio.SimpleClient()

@sio.on("my_event")
def on_my_event(data):
    ...

@sio.event
def my_event(data):
    ...
```

위 2개의 이벤트 처리 함수는 같은 my_event라는 이름의 이벤트를 처리한다. `sio.on`을 사용하면 이벤트 이름을 매개변수로 받고, `sio.event`는 밑에 있는 함수의 이름을 이벤트 이름으로 설정한다. 2가지 모두 허용되지만 앞으로 이벤트 핸들러를 만들때에는 `sio.on`만 적도록 하겠다.

서버는 sid와 data를 받고, 클라이언트는 data만 받는다. sid는 클라이언트를 구별하게 하는 고유번호이고, data는 상대 소켓으로부터 받는 데이터이다.
서버에서는 각 클라이언트를 관리하기 위해 받는 데이터와 송신자의 id를 받고, 클라이언트는 서버에서 받는 데이터만 처리하면 되기 때문에 데이터만 받게 된다.

하위 문단에서는 기본적으로 지원해주는 이벤트를 알아보도록 하겠다.

5.1.3.1. connect / disconnect

- connect: 서버에서는 클라이언트가, 클라이언트는 서버에 연결되었을 때 발생하는 이벤트이다.
  - 서버에서는 클라이언트가 들어올 때 sid, environ, auth를 받는다.
  - 클라이언트에서는 아무 데이터도 받지 않는다.
- disconnect: 서버에서는 클라이언트가, 클라이언트는 서버에 연결이 끊겼을 때 발생하는 이벤트이다.
  - 서버에서는 클라이언트가 들어올 때 sid만 받는다.
  - 클라이언트에서는 아무 데이터도 받지 않는다.

JS

```
socket.on("connect", () => {
    ...
})

socket.on("disconnect", () => {
```

```
    ...
})
```

Python

서버에서는 아래와 같이 사용한다.

```
@sio.on("connect")
def on_connect(sid, environ, auth):
    ...

@sio.on("disconnect")
def on_disconnect(sid):
    ...
```

클라이언트에서는 아래와 같이 사용한다.

```
@sio.on("connect")
def on_connect():
    ...

@sio.on("disconnect")
def on_disconnect():
    ...
```

5.1.4. event 보내기

이벤트 리스닝을 배웠으니, 이 문단에서는 이벤트와 데이터를 보내는 방법을 배워보자.

emit이라는 메소드는 상대에게 이벤트와 데이터를 보낸다.

아래에서는 my_event 이벤트에 hi라는 문자열을 보내보자.

JS

```
socket.emit("my_event", "hi")
```

Python

서버에서는 아래와 같이 보낸다.

```
sio.emit("my_event", "hi", to=sid)
```

서버에서는 다수의 클라이언트가 접속하기 때문에 to매개변수에 클라이언트의 sid를 넣어줘야한다.

아래는 클라이언트에서 서버로 보내는 방법이다.

```
sio.emit("my_event", "hi")
```

5.1.5. room으로 여러 client 관리

카톡에서 일부 사람들을 모아 단톡방을 만들듯이, 이곳에는 room이라는 시스템으로 일부 클라이언트들을 묶어서 관리할 수 있게 해준다.
클라이언트를 어떠한 room에 enter와 leave를 해줌으로써 유동적인 관리가 가능하다.

room은 서버에서만 관리하는 시스템이기 때문에, 클라이언트에서는 다루지 않는다.

5.1.5.1. room 관리

클라이언트를 room에 넣고 빼는 함수를 알아보겠다.

Python

```
sio.enter_room(sid, "my_room")

sio.leave_room(sid, "my_room")
```

room에 넣기 위해서는 클라이언트의 sid와 room의 이름, 이렇게 2가지가 필요하다. room은 따로 만들 필요없이 원하는 이름을 만들어서 넣으면 된다. 해당 코드에서는 my_room이라는 room에 클라이언트를 넣었다.

5.1.5.2. room 단위로 보내기

room단위로 클라이언트들에게 메시지를 보낼 수도 있다.

Python

```
sio.emit("my_event", "hi", room="my_room")
```

my_room에 있는 모든 클라이언트들에게 my_event로 hi라는 문자열을 보냈다.

또는 한 클라이언트만 제외시킬 수 있다. 이 기능이 있는 이유는 카톡을 생각해보면 편하게 이해할 수 있다. 메시지를 내가 보내면 그 채팅방에 있는 모두에게 다시 메시지를 발송할텐데, 나에게도 보낸다면 나는 메시지를 보낼때마다 나의 메시지를 또 받아야하는 불편한 상황이 나올 수 있다.

```
sio.emit("my_event", "hi", room="my_room", skip_sid=sid)
```

해당하는 클라이언트의 sid를 넣어주어야 한다.

5.2. Direct Message를 만들어보자

다이렉트 메시지 서비스를 만들어보자.

5.2.1. 목표 정하기

- 메시지를 보내면 나머지 유저들에게 메시지를 보낸다.
- 유저가 참가하면 참가했다는 메시지를 보낸다.
- 나갔던 유저가 들어오면 이전 대화내용이 복구되었으면 좋겠다.
- 오직 웹소켓의 연습용으로 만드는 것이니, 로그인 시스템 구현을 하지 않을 것이다. 닉네임을 표시하는 것은 과감하게 버리겠다.

5.2.2. 생각해보기

- 메시지를 보내면 받아야하니 메시지용 이벤트를 만들어서 받으면 되겠다.
- 다수에게 메시지를 보내는 것이니 room을 만들어서 한 번에 전송하면 되겠다.
- 그렇다면 클라이언트에 접속했을 때, room에도 들어가게 해야겠다.
- 참가 메시지와 같은 시스템 메시지는 일반 메시지와 구별이 되도록 데이터를 보내야겠다.
- 이전 대화 내용을 복구하려면 일단 클라이언트에서 보낸 메시지를 다 저장한 후에 다른 클라이언트가 접속했을 때, 지금까지의 메시지를 다 보내야겠다.

5.2.3. 준비하기

파일의 구조는 다음과 같다.

```
project/
        static/
                index.css
                index.js
        templates/
                index.html
        app.py
```

완성된 코드는 다음과 같다. 일단 붙여넣자.

app.py

```
from flask import Flask, render_template
from socketio import Server, WSGIApp

sio = Server()
app = Flask(__name__)
app.wsgi_app = WSGIApp(sio, app.wsgi_app)

messages = [{"message": "server started", "isSystem": True}]

@app.route("/")
def index():
    return render_template("index.html")

@sio.event
def connect(sid, environ, auth) -> None:
    """when client connected"""
    print("CONNECT - ", sid)
    sio.enter_room(sid, "DM")
    for msg_data in messages:
        sio.emit(
            "message",
            msg_data,
            to=sid
        )
    total = len(sio.manager.rooms["/"]["DM"])
    sio.emit(
        "message",
```

```
        {"message": f"you came into the room<br/>( total {total} )", "isSystem": True},
        to=sid,
    )
    msg_data = {
        "message": f"someone came into the room<br/>( total {total} )",
        "isSystem": True,
    }
    messages.append(msg_data)
    sio.emit(
        "message",
        msg_data,
        room="DM",
        skip_sid=sid,
    )
    sio.emit("enable_message", to=sid)

@sio.event
def disconnect(sid) -> None:
    """when client disconnected"""
    print("DISCONNECT - ", sid)
    sio.leave_room(sio, "DM")
    total = len(sio.manager.rooms["/"]["DM"]) - 1
    msg_data = {"message": f"someone left room<br/>( total {total} )", "isSystem": True}
    messages.append(msg_data)
    sio.emit(
        "message",
        msg_data,
        room="DM",
    )

@sio.event
def message(sid, data):
    """when client send message"""
    msg_data = {"message": data, "isSystem": False}
    messages.append(msg_data)
    sio.emit("message", msg_data, room="DM", skip_sid=sid)

if __name__ == "__main__":
    app.run()
```

index.html

```
<!DOCTYPE html>
<html lang="ko">
    <head>
        <title>Direct Message Client</title>
        <link rel="stylesheet" href="{{url_for('static', filename='index.css')}}"/>
    </head>
    <body>
        <div id="dm-box">
            <div id="msg-container">
                <div class="message msg-left"></div>
            </div>
            <label id="input-label">
                <input
```

```
                        id="input-box"
                        type="text"
                        placeholder="연결중..."
                        onkeyup="sendMessage()"
                        disabled/>
                </label>
            </div>
            <script src="{{url_for('static', filename='index.js')}}"></script>
        </body>
</html>
```

index.css

```
body {
    position: relative;
    background-color: #3a3a3a;
    width: 100%;
    margin: 0;
    font-size: medium;
}

#dm-box {
    position: absolute;
    display: grid;
    grid-template-rows: 90% 10%;
    left: 50%;
    transform: translateX(-50%);
    margin-top: 1em;
    width: 22em;
    height: 34em;
    border-radius: 0.5em;
    background-color: white;
}

#input-label {
    position: relative;
    width: 100%;
    height: 100%;
    background-color: inherit;
    border-radius: inherit;
    &>#input-box {
        position: absolute;
        left: 50%;
        top: 50%;
        transform: translate(-50%, -50%);
        background-color: rgb(225, 225, 225);
        border-radius: 1em;
        margin: unset;
        border: unset;
        padding: 0 5%;
        width: 85%;
        height: 80%;
    }
}
```

```scss
#msg-container {
    position: relative;
    border-radius: 1em;
    display: flex;
    flex-direction: column;
    width: 100%;
    height: 100%;
    overflow: auto;
    &::-webkit-scrollbar {
        width: 6px;
    }
    &::-webkit-scrollbar-thumb {
        background-color: #2f3542;
        border-radius: 1rem;
    }
    &::-webkit-scrollbar-track {
        background-color: rgba(128, 128, 128, 0.75);
        border-radius: 1rem;
    }
}

.message {
    display: flex;
    margin: 2% 1%;
    &.msg-left {
        flex-direction: row;
        left: 0;
    }
    &.msg-right {
        flex-direction: row-reverse;
        right: 0;
        &>div {
            background-color: #dcdcdc;
        }
    }
    &:last-child {
        margin-bottom: 2%;
    }
    &.msg-right>div, &.msg-left>div {
        white-space: normal;
        width: fit-content;
        max-width: 75%;
        padding: 3%;
        border: #dcdcdc solid 1px;
        border-radius: 0.5rem;
    }
    &.system {
        height: fit-content;
        justify-content: center;
        background-color: rgba(169, 169, 169);
        opacity: 0.6;
        font-size: small;
        border-radius: 1rem;
        &>div {
            text-align: center;
        }
```

```
    }
}
```

index.js

```
import { io } from "https://cdn.socket.io/4.3.2/socket.io.esm.min.js";
const socket = io("ws://127.0.0.1:5000")
let latestText = ""

socket.on('disconnect', function(){
    const inputBox = document.getElementById("input-box")
    inputBox.disabled = true
    latestText = inputBox.value
    inputBox.value = ""
    inputBox.placeholder = "연결중..."
})

socket.on("enable_message", () =>{
    const container = document.getElementById("msg-container")
    const inputBox = document.getElementById("input-box")
    inputBox.disabled = false
    inputBox.value = latestText
    inputBox.placeholder = "메시지 보내기..."
    scrollToEnd(container)
})

socket.on("message", (data) => {
    const container = document.getElementById("msg-container")
    let taskScroll = false;
    if (container.offsetHeight + container.scrollTop === container.scrollHeight) {
        taskScroll = true
    }
    addMessageElement(container, data["message"], "left", data["isSystem"])
    if (taskScroll) {
        scrollToEnd(container)
    }
})

function sendMessage() {
    if(window.event.keyCode===13){
        const inputBox = document.getElementById("input-box")
        const message = inputBox.value
        if (!message) {
            return null
        }
        inputBox.value = ""
        const container = document.getElementById("msg-container")
        addMessageElement(container, message, "right")
        scrollToEnd(container)
        socket.emit("message", message)
    }
    return null
}

function addMessageElement(container, message, direction="right", is_system=false) {
```

```
    const msgContainer = document.createElement("div")
    msgContainer.className += "message "
    if (is_system) {
        msgContainer.className += "system"
    }
    else {
        msgContainer.className += `msg-${direction}`
    }
    const msgBox = document.createElement("div")
    msgBox.innerHTML = message
    msgContainer.appendChild(msgBox)
    container.appendChild(msgContainer)
}

function scrollToEnd(container) {
    container.scrollTop = container.scrollHeight
}
```

5.2.4. 메시지

메시지는 딕셔너리 형태로 정의되어 일반 메시지와 시스템 메시지를 구분하도록 구조를 디자인하였다.

```
{
    "message": "blablabla",
    "isSystem": false
}
```

message에는 메시지 내용이 들어가고, isSystem에서 true/false로 시스템 메시지인지를 구분하게 하였다.

5.2.5. 이벤트

서버 기준으로 작성되었습니다.

4.2.5.1. connect

app.py에서 connect 이벤트입니다.

```
@sio.event
def connect(sid, environ, auth) -> None:
    """when client connected"""
    print("CONNECT - ", sid)
    sio.enter_room(sid, "DM")
    for msg_data in messages:
        sio.emit(
            "message",
            msg_data,
            to=sid
        )
    total = len(sio.manager.rooms["/"]["DM"])
    sio.emit(
        "message",
        {"message": f"you came into the room<br/>( total {total} )", "isSystem": True},
        to=sid,
```

```
    )
    msg_data = {
        "message": f"someone came into the room<br/>( total {total} )",
        "isSystem": True,
    }
    messages.append(msg_data)
    sio.emit(
        "message",
        msg_data,
        room="DM",
        skip_sid=sid,
    )
    sio.emit("enable_message", to=sid)
```

최초로 클라이언트가 접속하면, 채팅을 칠 수 없게 index.html에서 막아놨다. 그리고 enable_message가 이벤트가 오면 index.js에서 채팅을 칠 수 있게 하였다.
왜냐하면, 기존 메시지가 복구되는 중에 채팅을 칠 수 있기 때문이다. 채팅 기록이 꼬일 수 있기 때문에 미리 막아놨다가 로딩이 끝나면 채팅을 풀었다.

```
sio.enter_room(sid, "DM")
```

최초로 클라이언트가 접속을 하면, DM room에 클라이언트를 넣었다.

```
for msg_data in messages:
    sio.emit(
        "message",
        msg_data,
        to=sid
    )
```

그리고 지금까지의 메시지들을 클라이언트에게 모두 보내주었다.

```
total = len(sio.manager.rooms["/"]["DM"])
sio.emit(
    "message",
    {"message": f"you came into the room<br/>( total {total} )", "isSystem": True},
    to=sid,
)
msg_data = {
    "message": f"someone came into the room<br/>( total {total} )",
    "isSystem": True,
}
messages.append(msg_data)
```

이 부분은 입장 메시지를 클라이언트와 다른 클라이언트들에게 다르게 보이기 위해서 작성한 부분이다.
클라이언트 자신한테는 "내가 접속했어요"라고 알려주고, 다른 사람들에게는 "누군가가 접속했어요"라고 띄운다.

```
sio.emit(
    "message",
    msg_data,
```

```
    room="DM",
    skip_sid=sid,
)
```

이 부분은 다른 사람들에게 메시지를 보내는 부분이다. DM room에 한 번에 메시지를 보내지만, 클라이언트 자신에게는 "누군가가 접속했어요"라고 띄우지 않기 위해 skip_sid에 해당 클라이언트 sid를 추가해주었다.

```
sio.emit("enable_message", to=sid)
```

그리고 작업이 다 끝난 시점에서 enable_message 이벤트를 보내주었다.

4.2.5.2. message

app.py에서 message 이벤트입니다.

```
@sio.event
def message(sid, data):
    """when client send message"""
    msg_data = {"message": data, "isSystem": False}
    messages.append(msg_data)
    sio.emit("message", msg_data, room="DM", skip_sid=sid)
```

```
msg_data = {"message": data, "isSystem": False}
```

클라이언트에게 메시지를 받으면 우리가 정한 구조대로 메시지를 저장한다.

```
messages.append(msg_data)
```

그리고 메시지 기록에 해당 메시지를 추가한다.

```
sio.emit("message", msg_data, room="DM", skip_sid=sid)
```

마지막으로 보낸 클라이언트를 제외한 다른 모든 클라이언트들에게(DM room에 있는 클라이언트에게) 이 메시지를 보낸다. 이렇게 된다면 다른 클라이언트들은 message 이벤트로 메시지 데이터를 받게 될 것이고, 미리 작성된 코드에 따라 화면에 보이도록 작동할 것이다.

4.2.5.3. disconnect

app.py에서 disconnect 이벤트입니다.

```
@sio.event
def disconnect(sid) -> None:
    """when client disconnected"""
    print("DISCONNECT - ", sid)
    sio.leave_room(sio, "DM")
    total = len(sio.manager.rooms["/"]["DM"]) - 1
    msg_data = {"message": f"someone left room<br/>( total {total} )", "isSystem": True}
    messages.append(msg_data)
```

```
    sio.emit(
        "message",
        msg_data,
        room="DM",
    )
```

```
sio.leave_room(sio, "DM")
```

나간 클라이언트를 DM room에서 나가게 한다.

```
msg_data = {"message": f"someone left room<br/>( total {total} )", "isSystem": True}
messages.append(msg_data)
sio.emit(
    "message",
    msg_data,
    room="DM",
)
```

그 후에, 남은 클라이언트들에게 시스템 메시지로 "누군가가 나갔어요"라고 메시지를 보낸다.

5.2.5. 음

app.py만 설명했지만, 다른 코드 설명이 필요하시다면 따로 말씀해주세요.