



## 2일차: REST API 기초

# 목 차

- 1일차 복습
- REST API 개념
- Flask에서 API 만들기: To-Do CRUD API
- 템플릿 엔진 Jinja2 추가: 조건문, 반복문, 레이아웃 템플릿

**1 일차 복습**

# 기본 라우팅 코드

라우팅

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

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

```
def home():
```

```
    return "홈 화면입니다!"
```

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

```
def hello():
```

```
    return "안녕하세요, Flask!"
```

클라이언트에서  
보이는 결과

# 동적 라우팅 (Dynamic Routing) 소개

## [실습 3] 동적 라우팅 API 만들기

- 기본 라우팅에서 “변수”를 사용해 값을 동적으로 처리
- 즉, URL에 따라 다른 응답이 가능함

```
@app.route("/user/<name>")  
def greet(name):  
    return f"{name}님, 환영합니다!"
```

```
(venv) └─rubykim@Rubyui-MacBookPro ~/Desktop  
└─$ curl http://127.0.0.1:5000/user/oz-be14  
oz-be14님 , 환영합니다 !%  
(venv) └─rubykim@Rubyui-MacBookPro ~/Desktop  
└─$ curl http://127.0.0.1:5000/user/flask  
flask님 , 환영합니다 !%
```

# 변수 바인딩 예시

## [실습 4] 변수 바인딩 해보기

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/hello")
def hello():
    return render_template("hello.html", name="OZ")

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

main.py

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello Page</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

templates/hello.html

# 정적 (Static) 파일 & 모듈화

```
project/  
| app.py  
├ templates/  
|   ├── index.html  
|   └── greet.html  
└ static/  
    ├── style.css  
    └── logo.png
```

# REST API 개념



# API (Application Programming Interface)

- 프로그램끼리 대화할 수 있도록 만든 “접점”
- 예시
  - 사람 눈에 보이는 웹페이지
  - API가 주는 데이터 (JSON)

```
<h1>오늘 날씨는 맑음</h1>
```

```
{  
  "city": "Seoul",  
  "weather": "sunny",  
  "temp": 25  
}
```

# REST (REpresentational State Transfer)

- 웹에서 자원(Resource)을 일관성 있는 규칙으로 다루기 위한 아키텍처 스타일
- 쉽게 말하면, API를 “표준 규칙”에 맞게 짜는 방법
  - 자원을 URL로 표현: /users, /posts/1
  - 동작을 HTTP 메서드로 표현: GET, POST, PUT, DELETE
  - 즉, 주소(무엇을 할지) + 메서드(어떻게 할지) 조합

# REST API

- 일관성 있는 규칙에 맞춰서 작성된 API
- 목적: 사람이 보는게 아니라, **프로그램/앱/프론트엔드**가 쓰는 인터페이스

# JSON (JavaScript Object Notation)

- JSON: 데이터 표준 포맷
- key-value 구조 (Python dict과 비슷)
- 언어에 상관없이 거의 모든 시스템이 사용

```
{  
  "name": "OZ",  
  "camp": "BE14",  
  "status": "student"  
}
```

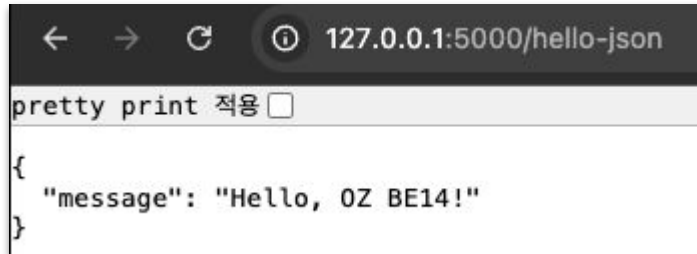
# Flask에서 JSON 응답하기

```
from flask import Flask

app = Flask(__name__)

@app.route("/hello-json")
def hello_json():
    return {"message": "Hello, OZ BE14!"}

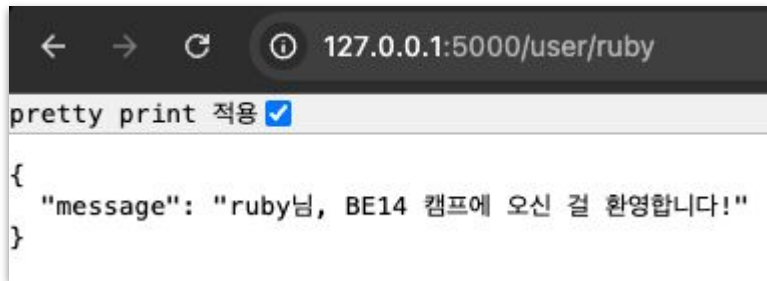
if __name__ == "__main__":
    app.run(debug=True)
```



# Flask에서 JSON 응답하기

[실습 1] 예시를 참고하여 아래의 엔드포인트를 만들어보세요.

- /user/<name> 주소로 접속하면 JSON으로 인사메시지 반환



A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:5000/user/ruby`. Below the address bar, there is a checkbox labeled "pretty print 적용" which is checked. The main content area displays a JSON object: 

```
{  "message": "ruby님, BE14 캠프에 오신 걸 환영합니다!"}
```



A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:5000/user/오즈`. Below the address bar, there is a checkbox labeled "pretty print 적용" which is checked. The main content area displays a JSON object: 

```
{  "message": "오즈님, BE14 캠프에 오신 걸 환영합니다!"}
```

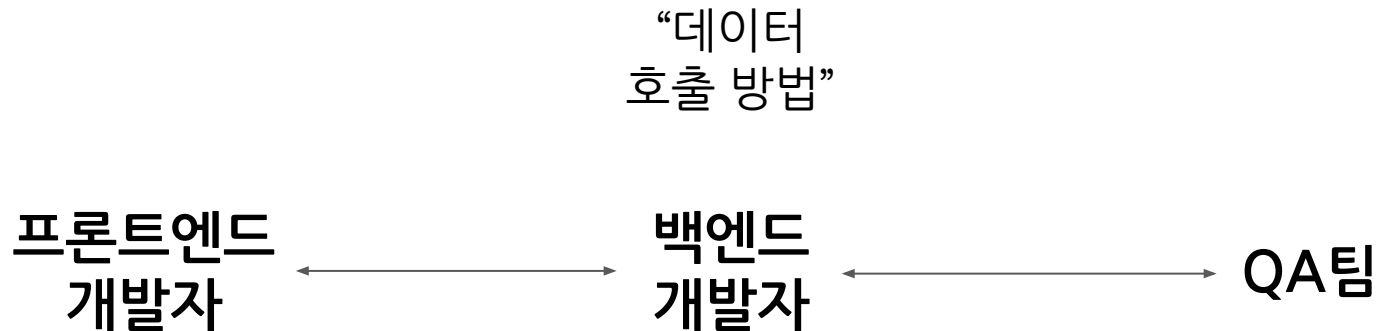
- 힌트: 1일차 동적 라우팅
- 만약에 한글이 깨져서 나온다면 pretty print를 적용하세요

# API 문서화

- API (Application Programming Interface)를 사용하는 방법과 규칙을 설명하는 문서
- 엔드포인트, 매개변수, 응답 구조, 인증 방법 등을 포함

엔드 포인트	메서드	설명	요청 매개 변수	예시	응답 예제
/users/{id}	POST	해당 id의 사용자 정보 업데이트	Header <ul style="list-style-type: none"><li>• Authorization</li><li>• Access-Token</li></ul> Body <pre>{   "id": INT,   "name": STRING,   "age": INT }</pre>	URL: /users/1  Header ...  Body ...	<ul style="list-style-type: none"><li>• 200<ul style="list-style-type: none"><li>◦ 사용자 정보가 업데이트 되었습니다.</li></ul></li><li>• 400<ul style="list-style-type: none"><li>◦ 잘못된 요청입니다.</li></ul></li></ul> ...

# API 문서화: 문서화의 필요성



- 효율적인 커뮤니케이션
- 유지보수 용이성
- 오류 감소



# API 문서화: 좋은 API 문서화 요소

- 명확한 설명
  - 각 엔드포인트의 목적과 사용법을 명확히 기술
  - 복잡한 개념을 이해하기 쉽게 작성
- 간결한 설명
  - 직관적인 문구 사용
  - 불필요한 정보는 제거 / 핵심 정보만 제공

# API 문서화: 좋은 API 문서화 요소

- API 설계 원칙: **일관성**, 엔드포인트 설계, HTTP 상태 코드
  - API 사용자가 엔드포인트와 데이터 구조를 예측 가능하게 사용하도록 구성

```
GET /users           // 사용자 목록 조회
GET /users/123       // 특정 사용자 조회
POST /users          // 사용자 생성
DELETE /users/123    // 사용자 삭제
```

```
GET /getUsers        // 사용자를 조회
GET /fetchUser/123   // 특정 사용자 조회
POST /newUser        // 사용자 생성
DELETE /deleteUser/123 // 사용자 삭제
```



# API 문서화: 좋은 API 문서화 요소

- API 설계 원칙: 일관성, **엔드포인트 설계**, HTTP 상태 코드
  - 리소스를 명확히 나타내야 하며, 동작이 아닌 데이터를 중심으로 설계하도록 구성
  - 설계 원칙
    - 복수형 사용: **/users** vs /user
    - 계층적 구조: 관계가 있는 리소스를 논리적으로 표현

// 사용자 123의 주문 목록 조회

GET /users/123/orders



GET /ordersByUserId?userId=123

# API 문서화: 좋은 API 문서화 요소

- API 설계 원칙: 일관성, **엔드포인트 설계**, HTTP 상태 코드
  - 리소스를 명확히 나타내야 하며, 동작이 아닌 데이터를 중심으로 설계하도록 구성
  - 설계 원칙
    - 필터링, 정렬, 페이징 지원 (= 성능 최적화)

`GET /products?category=shirts` // 필터링

`GET /products?sort=price&order=asc` // 정렬

`GET /products?page=2&limit=20` // 페이징

# API 문서화: 좋은 API 문서화 요소

- API 설계 원칙: 일관성, 엔드포인트 설계, HTTP 상태 코드
  - 상태 코드를 통해 요청의 결과를 명확히 전달

<https://developer.mozilla.org/ko/docs/Web/HTTP/Status>

클래스	설명	대표 상태 코드
2xx	요청이 성공적으로 처리됨	200 OK 201 Created 204 No Content
3xx	Redirection. 요청 완료를 위해 추가 작업이 필요함	301 Moved Permanently 302 Found 304 Not Modified
4xx	Client Error 클라이언트의 잘못된 요청으로 서버가 처리할 수 없음	400 Bad Request 401 Unauthorized 403 Forbidden 404 Not Found
5xx	Server Error 서버가 요청을 처리하던 도중 오류 발생	500 Internal Server Error 502 Bad Gateway 503 Service Unavailable 504 Gateway Timeout

# API 문서화: 좋은 API 문서 예시

- [Github Docs](#)
- [네이버 Developers 서비스 API](#)
- [카카오 REST API](#)
- [Twilio Docs](#)
- [Dropbox HTTP documentation](#)

# API 문서화: 문서화 도구 추천

## [실습 2] API 문서 세팅하기 (Postman, Swagger)

- Postman
  - 협업 기능에 효과적
  - 자동화 API 테스트 기능 제공

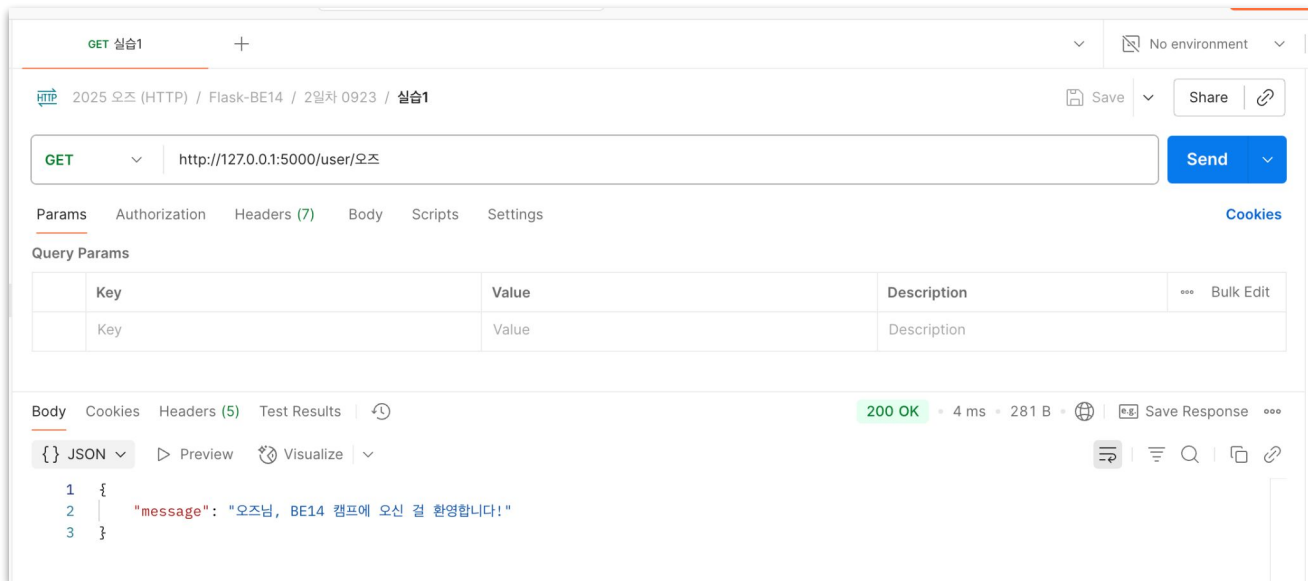


<https://www.postman.com/>

# API 문서화: 문서화 도구 추천

## [실습 2] API 문서 세팅하기 (Postman, Swagger)

- Postman





# API 문서화: 문서화 도구 추천

## [실습 2] API 문서 세팅하기 (Postman, Swagger)

- Swagger
  - YAML 또는 JSON로 작성
  - 폭넓은 커뮤니티 및 다양한 언어 지원



<https://swagger.io/>

# API 문서화: 문서화 도구 추천

## [실습 2] API 문서 세팅하기 (Postman, Swagger)

- Swagger

```
from flask import Flask, jsonify
from flasgger import Swagger

app = Flask(__name__)
swagger = Swagger(app)

@app.route('/hello')
def hello():
    """
    Hello API
    ---
    responses:
      200:
        description: 성공 응답
        schema:
          type: object
          properties:
            message:
              type: string
              example: "Hello, OZ BE14!"
    """
    return jsonify({"message": "Hello, OZ BE14!"})

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

점심 & 쉬는 시간

# Flask에서 API 만들기:

## To-Do CRUD API

# CRUD 개념

- Create 생성
- Read 조회
- Update 수정
- Delete 삭제

API는 결국 데이터에 대해 이 4가지 동작을 함

# CRUD <-> HTTP 메서드

동작	메서드	예시 엔드포인트
Create	POST	/todos
Read (전체)	GET	/todos
Read (특정)	GET	/todos/<id>
Update	PUT (또는 PATCH)	/todos/<id>
Delete	DELETE	/todos/<id>

# Flask CRUD API 만들기

[실습 3] CRUD <-> HTTP 메서드 표에서 아래 API 구현하기

- [GET] /todos
- [POST] /todos (Body JSON)
- [PUT] /todos/<int:todo\_id> (Body JSON)
- [DELETE] /todos/<int:todo\_id>

# Flask CRUD API 만들기

[실습 3] CRUD <-> HTTP 메서드 표에서 아래 API 구현하기

- [GET] /todos

```
@app.route("/todos", methods=["GET"])  
def get_todos():  
    return jsonify(todos)
```

```
@app.route("/todos/<int:todo_id>", methods=["GET"])  
def get_todo(todo_id):  
    task = todos.get(todo_id)  
    if not task:  
        return jsonify({"error": "Todo not found"}), 404  
    return jsonify({todo_id: task})
```



# Flask CRUD API 만들기

[실습 3] CRUD <-> HTTP 메서드 표에서 아래 API 구현하기

- [POST] /todos (Body JSON)

```
@app.route("/todos", methods=["POST"])
def create_todo():
    data = request.get_json()
    new_id = max(todos.keys()) + 1 if todos else 1
    todos[new_id] = data["task"]
    return jsonify({new_id: todos[new_id]}), 201
```

# Flask CRUD API 만들기

[실습 3] CRUD <-> HTTP 메서드 표에서 아래 API 구현하기

- [PUT] /todos/<int:todo\_id> (Body JSON)

```
@app.route("/todos/<int:todo_id>", methods=["PUT"])
def update_todo(todo_id):
    if todo_id not in todos:
        return jsonify({"error": "Todo not found"}), 404
    data = request.get_json()
    todos[todo_id] = data["task"]
    return jsonify({todo_id: todos[todo_id]})
```

# Flask CRUD API 만들기

[실습 3] CRUD <-> HTTP 메서드 표에서 아래 API 구현하기

- [DELETE] /todos/<int:todo\_id>

```
@app.route("/todos/<int:todo_id>", methods=["DELETE"])
def delete_todo(todo_id):
    if todo_id not in todos:
        return jsonify({"error": "Todo not found"}), 404
    deleted = todos.pop(todo_id)
    return jsonify({"deleted": deleted})
```

템플릿 엔진 Jinja2 추가

# [실습 4] 실습 3에 Jinja2 적용하기

조건문 if & 반복문 for

```
{% if todos %}  
  <p>할 일이 있습니다!</p>  
{% else %}  
  <p>할 일이 없습니다.</p>  
{% endif %}
```

```
<ul>  
{% for task in todos %}  
  <li>{{ loop.index }}. {{ task }}</li>  
{% endfor %}  
</ul>
```

# [실습 4] 실습 3에 Jinja2 적용하기

레이아웃 상속: `extends` / `block`

- 여러 페이지에서 공통으로 반복되는 HTML (헤더, 푸터, 메뉴 등)을 따로 빼내는 방법
- 공통 뼈대 (`base.html`)을 만들고, 각 페이지가 그 뼈대를 **extends**
  - 공통 뼈대에서 각 페이지를 불러올 때는 **block**
- 유지보수가 쉬워지고, 협업할 때도 깔끔해짐

# [실습 4] 실습 3에 Jinja2 적용하기

레이아웃 상속: extends / block

```
project/  
| app.py  
└─ templates/  
    ├── base.html      ← 공통 레이아웃  
    ├── index.html     ← base 상속  
    └─ todos.html      ← base 상속
```

# [실습 4] 실습 3에 Jinja2 적용하기

레이아웃 상속: extends / block

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}기본 제목{% endblock %}</title>
</head>
<body>
  <header>
    <h1>📌 To-Do 앱</h1>
    <hr>
  </header>

  <!-- 각 페이지별 내용이 들어갈 자리 -->
  {% block content %}{% endblock %}

  <footer>
    <hr>
    <p>© 2025 OZ BE14</p>
  </footer>
</body>
</html>
```

templates/base.html



# [실습 4] 실습 3에 Jinja2 적용하기

레이아웃 상속: extends / block

```
{% extends "base.html" %}

{% block title %}메인 페이지{% endblock %}

{% block content %}
    <form action="/add" method="post">
        <input type="text" name="task" placeholder="할 일 입력">
        <button type="submit">추가</button>
    </form>
{% endblock %}
```

templates/index.html

## [실습 4] 실습 3에 Jinja2 적용하기

레이아웃 상속: extends / block

```
{% extends "base.html" %}

{% block title %}할 일 목록{% endblock %}

{% block content %}
<h2>할 일 목록</h2>
<ul>
    {% for id, task in todos.items() %}
        <li>{{ id }}. {{ task }}</li>
    {% endfor %}
</ul>
{% endblock %}
```

templates/todos.html

**숙제: 실습 복습하기**

QnA